# Armstrong State University
# Engineering Studies
# MATLAB Marina – Functions Primer

**Prerequisites**
The Functions Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, and iteration. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, and MATLAB Marina Iteration module.

**Learning Objectives**
1. Be able to write MATLAB functions to solve problems.
2. Be able to write a test program to verify the correct operation of a function.
3. Be able to use built-in and user written MATLAB functions to solve problems.

**Terms**
function, function definition, parameter (argument), return variable, invoke (call), scope, optional argument

**MATLAB Functions, Keywords, and Operators**
function, nargin, nargout, global

**Functions**
As opposed to having a large program (script file) with many lines of code, programs are typically broken up into parts. Functions are pieces of code designed to perform a specific task or operation as part of a larger program. Functions allow one to perform the same operation on different data (apply the same code to different data). Functions should perform a single task and should be placed in a separate script file.

The general form of a function is shown in Figure 1. The first line of the function is the function definition. This is followed by the function comment header, and then the code implementing the function operation.

```
function [output arguments] = functionName( input arguments)
% function comment header
   function implementation
end
```

Figure 1, General Form of Function

Figure 2a shows the implementation of a function to compute the diameter of a circle given the circle radius.

```
function diameter = circleDiameter(radius)
% -----------------------------------------------------
% circleDiameter.m
% -----------------------------------------------------
% circleDiameter computes diameter of circle
% -----------------------------------------------------
% Syntax: diameter = circleDiameter(radius)
%  radius is radius of circle in meters
%  diameter is diameter of circle in meters
% -----------------------------------------------------
% Examples:
% d = circlediameter(1.0); will return d = 2.0
% -----------------------------------------------------
% Notes: units of length are assumed to be meters
% -----------------------------------------------------

diameter = 2.0 * radius;
end
```

Figure 2a, circleDiameter Function

The function definition starts with the keyword `function` followed by a list of output arguments enclosed in square brackets (return variables), an equal sign, the function name, and a list of input arguments enclosed in parentheses. The square brackets enclosing the output variables are not needed if only one value will be returned. The function comment header specifies the function name, the operation the function performs, the syntax for how to use (invoke or call) the function, and sometimes examples of the functions use.

The function implementation contains the MATLAB statements to perform the operation described by the function definition. In effect, the function implementation is a program that uses the function input arguments. The return variables must be assigned the appropriate values before the end of the function. Functions are terminated with an `end` statement, but this is not required unless the file containing the function contains a nested function. For files that contain nested functions, all functions in the file must be terminated with an `end` statement.

The script file defining the function must have the same name as the function name. For programs to access the function, the file containing the function must be in a folder in MATLAB's search path (generally one should have user created functions in the same folder as the main or test program).

Functions are passed data via their input arguments, operate on the data, and return a result to the calling program via the output arguments. Functions cannot be run like programs; instead, a main program calls the function. Functions are used in a program by invoking them by name, providing a list of variables matching the function input argument list, and assigning results that the function returns to program variables for future use. Figure 2b shows a program that uses the circleDiameter function to compute and display a circle's diameter after reading in the radius.

```
clear;
clc;
close all;

radiusOfCircle = input('Enter radius of circle in meters: ');
diameterOfCircle = circleDiameter(radiusOfCircle);

disp('Circle has diameter');
disp(diameterOfCircle);
```

Figure 2b, Program to Determine Diameter of Circle

The program of Figure 2b reads in the circle radius from the user, calls the `circleDiameter` function passing it the user-input radius value, assigns the diameter that the function returns to the program variable named diameterOfCircle, and displays the diameter. Notice that the function does not read in the radius nor does it display the diameter. The radius is passed to the function via the function input argument and the diameter is displayed by the calling program.

Functions should not read in data from the user (using input) or display data (using display, plot, etc) unless that is the functions only task. Data needed by the function should be passed in via the function input arguments and results should be returned and used/displayed in the calling program.

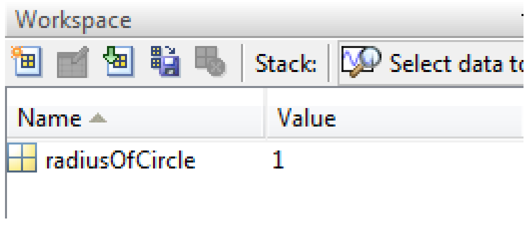
Built in MATLAB functions are used by providing values to the function and assigning the result the function returned to a variable. User created MATLAB functions are invoked (called) in the same manner as built in MATLAB functions.

For each data value needed by the function, there must be a corresponding variable in the input argument list. The arguments passed to the function via the function call can either be literals (constants), variables, or expressions. The values passed to the function are matched left to right. The order in which values are passed is important. The function passes back values via the output argument list and these should be assigned (saved) to variables in the calling program for future use.

**Scope of Variables**

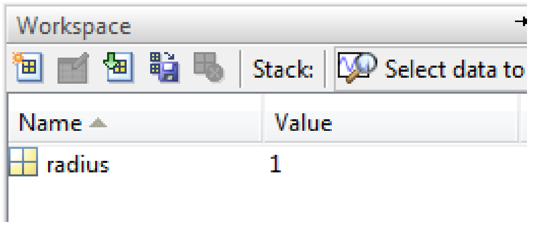
The scope of a variable is the portion of the program where it can be referenced. Variables created in a program or the command window are global variables that can be accessed anywhere. Variables declared inside functions (or in the argument list of functions) are local variables and can only be referenced inside the function. These variables are created when the function is called, exist as long as the function is executing, and are destroyed after the function is completed. Variables with different scope can have the same name, but they are still different variables.

Figures 3a and 3d show the MATLAB workspace before and after `circleDiameter` function is called in the program of Figure 2b. Figures 3b and 3c show the local workspace while the `circleDiameter` function is executing.



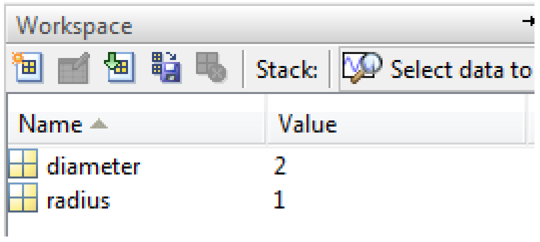
| Name | Value |
|---|---|
| radiusOfCircle | 1 |

Figure 3a, MATLAB Workspace Before `circleDiameter` Function Call



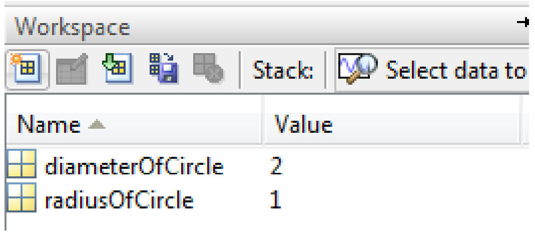
| Name | Value |
|---|---|
| radius | 1 |

Figure 3b, MATLAB Workspace During `circleDiameter` Function Execution



| Name | Value |
|---|---|
| diameter | 2 |
| radius | 1 |

Figure 3c, MATLAB Workspace During `circleDiameter` Function Call



| Name | Value |
|---|---|
| diameterOfCircle | 2 |
| radiusOfCircle | 1 |

Figure 3d, MATLAB Workspace After `circleDiameter` Function Call

Notice that in Figures 3b and 3c, the `circleDiameter` function does not have access to the variables in the main programs workspace; it only has access to the variables in its own local workspace. Notice that in Figure 3d, the main program does not have access to the local variables created as part of the `circleDiameter` function call. Once the function returns its result, its local workspace goes away.

MATLAB passes arguments to functions by value. When a function is called, copies of each argument are passed in to the function to the function parameter variables. Even if that function variable value is modified in the function it has no effect on the original value in the calling program. Naming a main program variable and a function variable the same name does not allow the function access to the main program's variable value. Similarly, if the function changes the value of the variable, the main program's version of that variable is not changed.

**Global Variables**

Variables created in the command window or in a main script show up in the MATLAB workspace are available until cleared. Functions have their own local workspace and cannot access variables in the main MATLAB workspace or in the calling script. Variables created in a function are local to that function and once the function is done they are no longer available. Functions only have access to variables in their parameter list or declared inside the function.

The keyword `global` allows one to create a variable that is accessible in the calling script and in functions that are called from the script. The global variables must be declared both in the calling script and in the function they are to be accessed from. Global variables do not need to be passed in as arguments or returned as results. Global variables are generally frowned upon except in cases where one would otherwise have to duplicate a large quantity of data that would tax the system's memory resources.

**Testing and Using Functions**
Functions should be exhaustively tested before their general use. Functions are typically used over and over and by users other than the function designer. Test programs should invoke the function for all legitimate function arguments and function arguments that may cause errors.

Consider testing the function of Figure 4a that computes the gravitational force (Newtonian) between two bodies of mass. The gravitational force between two bodies of mass is proportional to their masses and inversely proportional to the square of the distance between them, $F = G\frac{m_1 m_2}{r^2}$ , where G is the universal gravitational constant.

```
function F = gravitationalForce(m1, m2, r)
% -----------------------------------------------------
% gravitationalForce.m
% -----------------------------------------------------
% gravitationalForce computes the force of attraction
% of two bodies of mass on each other
% -----------------------------------------------------
% Syntax: F = gravitationalForce(m1, m2, r)
%  m1 is mass of object one in kg
%  m2 is mass of object two in kg
%  r is distance between centers of mass of objects in meters
%  F is the gravitational force in N
% -----------------------------------------------------

G = 6.67384e-11;  % universal gravitational constant N*m^2/kg^2
F = G*m1*m2/r^2;
end
```

Figure 4a, `gravitationalForce` Function

Since the `gravitationalForce` function consists of straight forward formula, a single test case would verify the correct operation for positive masses and distances. To ensure correct operation for all cases, additional test cases for negative and zero masses and negative and zero distances should also be used. Some of the additional test cases may produce errors and error handling may need to be added. Initial test cases should produce results that are easily verified as correct. For this example, a test case with one of the masses being the earth and the other a person on the earth's surface should produce a force that if divided by the person's mass would give the force of gravity. Figure 4b shows one test program for the `gravitationalForce` function. Figure 4c shows the results of running the test program that calls the `gravitationalForce` function three times and displays each of the three results.

```
clear;
clc;
close all;

% earth and person on earth's surface
% masses in kg and distance in m
massEarth = 5.98e24;
massPerson = 70;
distance = 6.38e6;
force = gravitationalForce(massEarth, massPerson, distance);
disp(force)

% two people one meter apart
massPerson2 = 70;
distance = 1;
force = gravitationalForce(massPerson, massPerson2, distance);
disp(force)

% distance of zero
distance = 0;
force = gravitationalForce(massPerson, massPerson2, distance);
disp(force)
```

Figure 4b, Test Program for `gravitationalForce` Function

```
686.3311

3.2702e-07

Inf
```

Figure 4c, Results of Test Program for `gravitationalForce` Function

The first result is correct as the gravitational force of 686 N when divided by the mass of person yields the force of gravity $g = \frac{F}{m} = \frac{686}{70} = 9.8m/s^2$. The second result makes sense as the gravitational force between two relatively small masses should be small (the result can be verified correct using a calculator). The third result also makes sense as dividing a number by zero should produce either an error or infinity, in this case infinity. Even though the third result makes sense, error handling should be added to this function as an infinite result may produce problems later in a program if this value was used for further calculations.

Notice that the values for the masses and distance are defined in the test program not in the function. The function does not redefine the function input arguments by assignment or reading in a new value. The function does not display the display the computed force; the computed gravitational force is returned to the test program where it is then displayed.

**Functions that Return Multiple Results**

Functions that return more than one result need one return variable for each result in the function definition. The return variables are enclosed in square brackets and separated by commas. Figure 5a shows the implementation of a function `rectToPolar` that converts 2D rectangular coordinates to polar coordinates. Figure 5b shows a test program that verifies the correct operation of the `rectToPolar` function and illustrates its use.

```
function [r, theta] = rectToPolar(x, y)
% -----------------------------------------------------
% rectToPolar.m
% -----------------------------------------------------
% rectToPolar converts two-dimensional rectangular
%   to polar coordinates
% -----------------------------------------------------
% Syntax: [r, theta] = rectToPolar(x, y)
%  x is the x coordinate
%  y is the y coordinate
%  r is the radius
%  theta is the angle in radians (-pi <= theta <= pi)
% -----------------------------------------------------

r = sqrt(x^2 + y^2);
theta = atan2(y, x);
end
```

Figure 5a, `rectToPolar` Function

The `rectToPolar` function requires two input arguments: the x and y values of the rectangular coordinates and two output arguments: the r and theta values of the polar coordinates. To verify the correct operation of the function, values of x and y in the first through fourth quadrants and on each axis should be tested (seven test cases).

Normally the output of statements would be suppressed, but for test programs it is often easier to use the echoed result than a display statement.

```
clear;
clc;
close all;

% first quadrant test
x = 2; y = 3;
[r, th] = rectToPolar(x,y)
% second quadrant test
x = -2; y = 3;
[r, th] = rectToPolar(x,y)
% third quadrant test
x = -2; y = -3;
[r, th] = rectToPolar(x,y)
% fourth quadrant test
x = 2; y = -3;
[r, th] = rectToPolar(x,y)
% test on axes
x = 2; y = 0;
[r, th] = rectToPolar(x,y)
x = 0; y = 3;
[r, th] = rectToPolar(x,y)
x = -2; y = 0;
[r, th] = rectToPolar(x,y)
```

Figure 5b, Test Program for `rectToPolar` Function

**Functions with Optional Arguments**
Default values of function input arguments can be provided by checking for the number of arguments passed in to the function and assigning default values to arguments that did not have a value passed to them. The MATLAB function `nargin` returns the number of arguments passed into a function. This can be used inside a function to specify default values for arguments that are not specified in the function call by a user. Remember that function arguments are matched from left to right, so optional arguments must be the arguments on the right of the parameter list. The MATLAB function `nargout` returns the number of storage variables in the calling statement of the function. This allows a function to omit computing return results that are not needed.

The function `evaluateCosine` of Figure 6a illustrates the use of optional arguments to evaluate the function $x(t) = A\cos(\omega t + \phi)$. The function has six input arguments of which one, fs, is optional. The program of Figure 6b tests the function for both a specified sampling frequency fs and the default sampling frequency.

Notice that `evaluateCosine` function of Figure 6a does not display the `tt` or `xx` vectors nor does it plot them. These vectors are returned to the calling statement where they can be displayed or plotted. The `evaluateCosine` function does not perform any input, i.e. since `tstart`, `tend`, etc are passed in as arguments they should not be read in from the user. These values are set up in the calling program and passed to the function via the function call.

```
function [xx, tt] = evaluateCosine(A, w, phi, tstart, tend, fs)
% -----------------------------------------------------
% evaluateCosine.m
% -----------------------------------------------------
% evaluateCosine evaluates a cosine function over the desired
% time range
% -----------------------------------------------------
% syntax: [xx, tt] = evaluateCosine(A, w, phi, tstart, tend, fs)
%  A is the amplitude of sinusoid
%  w is the frequency of the sinusoid (rad/sec)
%  phi is the phase of the sinusoid (rad)
%  tstart is start time (sec)
%  tend is end time (sec)
%  fs is the sampling frequency in Hz
%  xx is the cosine function values
%  tt is time vector
% -----------------------------------------------------
if (nargin < 6)
    % default sampling frequency of 10 times fmax
    fs = 5*(w/pi);
end
tt = tstart : 1/fs : tend;
xx = A*cos(w*tt + phi);
end
```

Figure 6a, `evaluateCosine` Function

```
% amplitude, frequency, and phase shift
A = 2;
w = 2*pi*100;
phi = pi/2;
% test specifying sampling frequency
[xx1, tt1] = evaluateCosine(A, w, phi, 0.0, 0.025, 25*100);
figure(1)
plot(tt1, xx1), xlabel('t (sec)'), ylabel('x(t), fs = 2.5kHz');
% test using default sampling frequency
[xx2, tt2] = evaluateCosine(A, w, phi, 0.0, 0.025);
figure(2)
plot(tt2, xx2), xlabel('t (sec)'), ylabel('x(t),default fs');
```

Figure 6b, Test Program for `evaluateCosine` Function

**Designing Functions**
Functions allow one to perform the same operation on different data (apply the same code to different data). Remember that when designing functions, someone should be able to use the function without knowing how it is implemented. The user should only need to know what the function does, what arguments must be passed in, and what the function returns. The implementation of the function could be changed and not affect its use if the function signature (function name, parameter list, and return parameters) is not altered.

When designing a new function, it is helpful to ask and answer the questions:

- What should be passed to the function (what data does the function need)? This will determine the function input arguments.
- What operation should the function perform? The function algorithm and implementation will follow from this.
- What should the function return (what is passed back from the function)? This will determine the function output arguments.

It is often useful to generate the algorithm for a specific case of the operation and then once that works generalize it. Add any error handling last after the operation has been tested and determine to be working.

**Build Array Function**
Building an array creates a new collection of data. The data for the collection can come from the user or more typically is read from an external file. The `buildArray` function needs the number of elements in the array and will return a new 1D array. The function will need to allocate space for the array, read in and store the value, and return the array. The `buildArray` function of Figure 7 builds a 1D array (vector) of numbers read from the user.

The `buildArray` function operates as follows:

- When the function is called, the number of elements for the 1D array should be passed to the function.
- Space for a 1D array of that many elements is allocated.
- Each new element is read from the user and stored in the next empty space in the 1D array.
- When the array is full, the resulting 1D array is returned from the function.

Note for the `buildArray` function, input is performed. It is acceptable in this case to have user or file input since that is function’s operation. The function should not perform any operations on the data read in except possibly error/validity checking. The data is read in and returned to the calling statement. The code [‘string1’, ‘string2’] concatenates two strings. The `int2str` function converts an integer to a character string.

Error checking and/or default arguments could be added to ensure that the number of elements is greater than or equal to zero or if the function is called with no input arguments an empty 1D array is returned.

```
function result = buildArray(numberElements)
% -----------------------------------------------------
% buildArray.m
% -----------------------------------------------------
% buildArray creates a new 1D array of numbers
%   read from the user
% -----------------------------------------------------
% Syntax: result = buildArray(numberElements)
%  numberElements is the number of elements in the new array
%  result is the new 1D array
% -----------------------------------------------------

% allocate space for new 1D array
result = zeros(1,numberElements);
% read in elements from user and store in array
for k = 1:1:numberElements
    result(k) = input(['Enter element ',int2str(k), ': ']);
end

end
```

Figure 7, `buildArray` Function

**Sum Vector Function**

Summing the elements of a 1D array was covered in the MATLAB Marina For Loops Primer. The program of Figure 8a sums the elements of a vector.

```
v = [17, 32, 13, 42, 63, 57];
% set initial result to zero
result = 0;

% iterate through the elements of the vector
for k = 1:1:length(v)
    % add each element to the previous result
    result = result + v(k);
end

% display result
disp(result);
```

Figure 8a, Program to Sum Elements of a Vector (this is Figure 4b from For Loops Primer)

To convert this program to a `sumVector` function:

- The vector to sum should be passed to the function rather than created in the function as in the program. The function input argument will be the vector.
- The function body will be the same as the programs sum vector operation; initialize the result to zero and perform a running sum.
- The function should not display the resulting sum but rather return the sum to the calling statement. The function return argument will be the sum of the vector.

Figure 8b shows the `sumVector` function. The `sumVector` function should be tested for the same test cases as a sum vector program: vector of length one, vector of length two or greater, and an empty vector.

```matlab
function result = sumVector(x)
% -----------------------------------------------------
% sumVector.m
% -----------------------------------------------------
% sumVector sums the elements of a vector
% -----------------------------------------------------
% Syntax: result = sumVector(x)
%  x is the vector of numbers
%  result is the sum of the elements in the vector
% -----------------------------------------------------
% Examples: r = sumVector([6, 3, 8, 12]); results in r = 29
% -----------------------------------------------------

result = 0.0;
for k = 1:1:length(x)
     result = result + x(k);
end
end
```

Figure 8b, `sumVector` Function

The loop for the `sumVector` function is iterating over the indices of the vector. Error checking could be added to ensure that the number of elements in the vector is greater than zero.

**Search (Find Equal) Vector Function**

Searching involves going through a collection of data to find a match to a search criteria. Common searches are to find elements equal to, less than, or greater than a desired value. The result of the search is an indication that a match was found and may include the location in the collection where the match was found. The `findEqual` function of Figure 9 searches a vector of numbers for all elements that equal a number.

To find all the elements in a vector whose value equals some desired value:

- The function will need the vector to search and the desired value to search for. The function arguments will be the vector and the search value.
- The function will need to return a vector of indices corresponding to the array locations where the value was found. The function output argument will be a vector.
- The function will start with an empty vector of indices.
- Each element of the vector will need to be compared to the desired value and if they match, the location (index) of the match is stored in the result. This will require that the result vector increases in size each time a match is found.
- Once all the elements in the vector have been compared the vector of indices is returned (if no matches are found an empty vector is returned).

```
function result = findEqual(x, value)
% -----------------------------------------------------
% findEqual.m
% -----------------------------------------------------
% findEqual determines the location in the vector of the value
% -----------------------------------------------------
% Syntax: result = findEqual(x, value)
%  x is the vector of numbers
%  value is the number to match
%  result is a vector of indices where matches were found
% -----------------------------------------------------
n = 0;
result = [];
% go though the vector, save indices where matches found
for k = 1:1:length(x)
    if (x(k) == value)
        n = n + 1;
        result(n) = k;
    end
end
end
```

Figure 9, `findEqual` Function

Test cases for the `findEqual` function are a vector and value with no matches, a vector and value with a single match, and a vector and value with more than one match. The `findEqual` function could be modified to search for only the first match. Error checking could be added to ensure that the vector to search has length greater than zero.

Last modified Friday, September 26, 2014