EE 224
User Written Functions

A subprogram is a program that is executed by another program or from within another program It typically has its own private variable and address space but is able to share some variables with the program that called it.

A subprogram may be a *subroutine* or aa *function* aka *method.* A subroutine is activated by an explicit *call* statement and all variables are passed and returned in arguments. For example, a subroutine which does the square root of *x* might be done like this:
`call SquareRoot(x, y)`
Upon return y will be equatl to the square root of x.

A function is similar to a subroutine but it is activated by using its name – there is not call statement. The function typically returns a single variable. A function which does the square root might looklike this:
`y = SquareRoot(x)`
The function takes the square root of the argment x and returns that value to y.

In MATLAB© there are no subroutines – only functions but function in MATLAB© can take multiple arguments and return multiple variables as a result. Arguments and the result may be vectors or matrices.

In older versions of MATLAB©, (prior to R2016a) all functions had to be saves in separate m-files. More recent versions of MATLAB© allow functions to be included in the same m-file as the calling program.

In this class we are going to look at three variations on MATLAB© functions: Primary Functions; Sub-Functions; and Anonymous Functions (aka inline functions).

Primary Functions
A primary function is defined within its own m-file and it may be called from any other MATLAB© m-file or from the command window. As an example, here is a primary function which returns the sum and product of two arguments. The arguments may be vectors, in which case the results will be vectors of the same size.

```
%This function returns the sum and product of
%  two arguments.
function [s p] = SumProd(a, b)
  s = a + b;
  p = a .* b;
end
```

The end statement is optional but makes the function a bit more readable. This primary function must be stored in an m-file which has the same name as the function – in this case `SumProd.m`.

This function can be executed from the command line:

```
>> a = [1 2 3];
>> b = [3 4 5];
>> [c d] = SumProd(a, b)
c =
     4     6     8
d =
     3     8    15
```

It can also be executed from within another m-file using the same syntax.

The comments at the top of the function can be printed using the "help" command:

```
>> help SumProd
 This function returns the sum and product of
   two arguments.
```

Sub-Functions

Sub-functions are the same as primary functions except they are not in a separate m-file – they are included in an existing m-file and are only used by that m-file. Here is an example which finds the roots of the quadratic equation:

```
%QuadExmp.m
a = input('Enter the first coefficient...');
b = input('Enter the second coefficient...');
c = input('Enter the third coefficient...');
[r1 r2] = quad(a, b, c);
disp(r1);
disp(r2);

function [root1 root2] = quad(a, b, c)
discr = sqrt(b^2 - 4*a*c);
root1 = (-b + discr)/(2*a);
root2 = (-b - discr)/(2*a);
end
```

A given program can have as many sub-functions as you care to write. Each is placed at the end of the main code. Sub-functions can call other sub-functions or other primary functions as needed.

Anonymous Functions aka *in-line functions*

An anonymous function is written on one-line and is mostly used to evaluate an expression. The syntax is:

```
f = @(arg list) expression;
```

For example, suppose we want to write a program to evaluate

$y = 3x^4 – 12x^3 + 5x^2 - 34x + 25$

The following m-file program will do this using an anonymous function.

```
%PolyValue
y = @(x)3*x^4 - 12*x^3 + 5*x^2 - 34*x + 25;
x = input('enter x... ');
while(x ~= 0)
    y = 3*x^4 - 12*x^3 + 5*x^2 - 34*x + 25;
    disp(y);
    x = input('enter x... ');
end
```

If we run this program by entering PolyValue from the command line we get the following:

```
>> PolyValue
enter x... 3
  -113
enter x... 4
   -31
enter x... 2
   -71
enter x... 0
```

Functions with an unknown number of arguments

An unusual thing about MATLAB's argument list is that the number of arguments in the function call need not be the same as the number of arguments in the function definition. In general the number of arguments in the function call should be less than or equal to the number of argumetsn in the function definition. MATLAB© defines two variables called `nargin` and `nargout` which the user can use to determine the number of arguments in and out that are present. Using an if statement the person writing the function can determine the response of the number used in the calling program is not the same as those defined in the function.

Here is an example using nargin and nargout:

```
function [x, y] = Example1(a, b)
if(nargin > 2 | nargin < 1)
  disp('Error');
  return;
end
if(nargin == 1)
  b = 0;
end
if(nargout == 1)
  x = a;
else
  x = a;
  y = b;
end
```

Global Variables

Global Variables Functions have access only to those variables in the main program that are in the parameter list. All other main program variables are not defined inside the function. In some cases, especially where constants are used, it is convenient to declare variables as global. Declaring a variable global in a function gives that function access to the variable in the main program. As a matter of style all global variables should be written in all upper-case letters so that they are not inadvertently changed. Here is an example.

In the Command Windows:

```
>> global C;
>> C = exp(1);
>> x = Example3(6);
x = 8.7183
```

In the function definition

```
function x = Example3(a)
global C;   %C comes from the main program
  x = a + C;
end
```

Sample Problems

1. The m-file listed below calls a function which is supposed to count the number of ones in an argument x. Write the function as a sub-function.

```
%CountOnesExmp.m
x = [4 7 1 3 8 4 23 45 67 89];
disp(CountOnes(x));
```

```
Answer
function y = CountOnes(x)
y = 0;
for i = 1:length(x);
    if x(i) == 1
        y = y + 1;
    end
end
end
```

2. Write a primary function to plot data. Your function may contain up to four arguments which are (in order) A) the x-axis vector, B) the y-axis vector, C) the figure number, D) a vector containing up to three strings which will be the title, x-axis label, and y-axis label. The first two arguments are required but the third and fourth arguments are optional. The following are valid calls to the function

```
plotme(x, y);
plotme(x, y, 1);
plotme(x, y, 1, ["my plot" "time in seconds" "voltage"]);
```

Your function should set an axis that goes from the minimum x to the maximum x and the minimum y to the maximum y.

Note that for MATLAB versions greater than R2017a you can create string array using double quotes like this:

```
sArr = ["striing1", "string2", "string3"];
```

You can access individual string inside the string array like this:

```
disp(sArr(2));
```

Solution

```
%plotme.m
%This function creates a new figure and plots the argments
%  It allows the user to add a title and x and y labels.
function plotme(x, y, fNum, labels)
  if(nargin < 2 || nargin > 4)
    disp('error');
    disp('Usage: plotme(x, y)');
    disp('       plotme(x, y, figureNum');
    disp('       plotme(x, y, figureNum, [title xlabel
ylabel])');
    return;
  end
  if nargin == 2
     figure;
  else
      figure(fNum);
  end
  plot(x, y);
  axis([min(x) max(x) min(y) max(y)]);
  if nargin == 4
      xlabel(labels(2));
      ylabel(labels(3));
      title(labels(1));
  end
end
```