Notes on functions in Matlab

By Gilberto E. Urroz, August 2004

Pre-defined functions

Matlab has a variety of pre-defined functions readily available for use by simply using their names and providing the right number and type of arguments. For example, the natural logarithm function (log) is readily available:

```
» log(-5.2)

ans =

   1.6487 + 3.1416i
```

Help is also readily available for these functions, for example:

```
EDU» help log

 LOG    Natural logarithm.
    LOG(X) is the natural logarithm of the elements of X.
    Complex results are produced if X is not positive.

    See also LOG2, LOG10, EXP, LOGM.

 Overloaded methods
    help sym/log.m
```

If a function name is not available in Matlab, you will get a reply similar to this:

```
EDU» help myFunction

myFunction.m not found.
```

Notice that *help* searches the Matlab installation for a file named *myFunction.m* to try to obtain the information requested, and then reports that such file was not found. Most functions in Matlab will have an *m* file associated with their names, where information about their operation is available.

Functions defined by m files

User-defined functions (i.e., those no pre-programmed in Matlab) can be defined by using an *m* file. These files are simply text files whose name end with the suffix *.m*. Those *m* files that a user may create are typically stored in the *work* directory. This directory is the default working directory of a Matlab installation.

Functions defined by *m* files start with the line

```
function [assignment_variable(s)] = function_name(arguments)
```

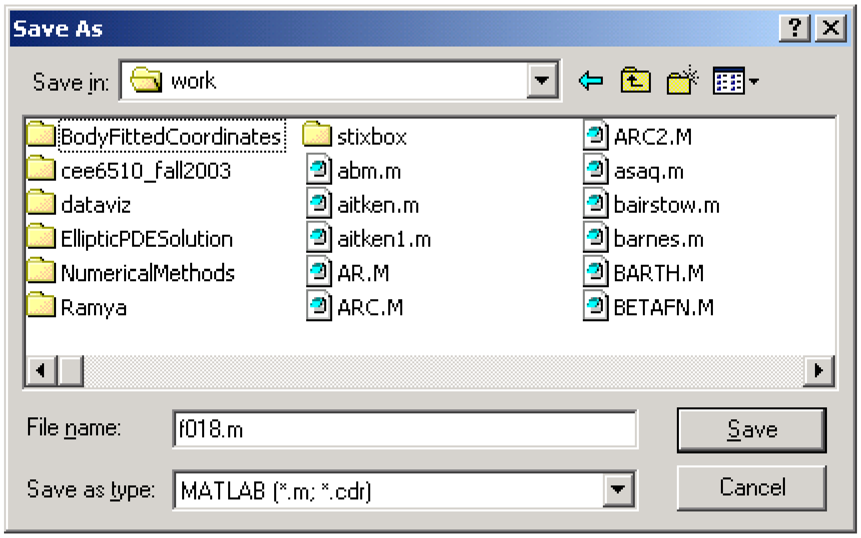
Here, `function` is a required Matlab particle to identify the file as a function, *`assignment_variable(s)`* is an optional dummy variable, *`function_name`* is the name given to the function, and *`arguments`* are optional values passed on to the function from the main Matlab interface or from within another function.

If one or more *assignment_variables* are present, then their names must be assigned a value within the function. Consider for example the function:

```
function [x] = f018(r)
% This is an example function.
% I have no idea what r or x are.
% Just take my word for it.
x = r.*sin(r);
```

The line `function [x] = f018(r)` helps us identify the assignment variable as *x*, the function name as *f018*, and one argument as *r*. The next three lines are simply comment lines describing the function. The fourth line is the line where a value is assigned to the assignment variable *x*. Notice that, since we don't know whether *r*, the argument, will be a scalar, a vector, or a matrix, it is safer to use term-by-term multiplication (.*) in the assignment line `x = r.*sin(r);`. It is recommended that you use semi-colons (;) to end every executable line in the function, whose value you don't want showing in the Matlab interface when the function is invoked.

To type this function you should use Matlab's own editor (available by using *File>New>M file* in the *File* menu). Once you finish typing the function in the editor, use the option *File>Save* to save it to a file. By default, the editor will try to save the file to the *work* directory and use the *function_name* as the file name, e.g.,



File name: f018.m

Save as type: MATLAB (*.m; *.cdr)

Press the *Save* button to save the function to a file. The function is now available for use from within Matlab.

For example, the command *help f018* will copy the comment lines immediately following the first line to the Matlab interface:

```
» help f018

  This is an example function.
  I have no idea what r or x are.
  Just take my word for it.
```

A few evaluations of this function are shown next:

```
» f018(2)

ans =

    1.8186

» f018([1 2])

ans =

    0.8415    1.8186

» m = f018([-3 2 5])

m =

    0.4234    1.8186   -4.7946

» R = f018([1.2,2.4;3.2,1.5])

R =

    1.1184    1.6211
   -0.1868    1.4962
```

The first two examples show no assignment made to the value returned by the function. In these cases, Matlab assigns the returned values to the generic variable *ans*. The first example shows a scalar argument, and the function returns a scalar result. The second and third examples show vector arguments with vector results. The fourth example shows a matrix argument and a matrix result. Assignments are made to specific variables in examples 3 and 4.

A function may take no arguments, for example:

```
function [s] = labels()
% This function simply returns a string
s = 'Copyright (C) Gilberto Urroz - 2004';
```

Calls to this function produces the following:

```
» labels

ans =

Copyright (C) Gilberto Urroz - 2004

EDU» notice = labels

notice =

Copyright (C) Gilberto Urroz - 2004
```

A function definition may not have an assignment variable, for example:

```
% Function without assignment variable
r = s^2;
disp(['For s = ',num2str(s),' s-squared is = ',num2str(r)]);
```

A call to this function is shown next:

```
» f019(3)
For s = 3 s-squared is = 9
```

If you attempt to assign function *f019* to a variable, you get the following error message:

```
» s1 = f019(4)
??? Error using ==> f019
Too many output arguments.
```

Finally, here is a function without assignment variable and without arguments:

```
function [] = f020()
% Function without assignment variable and without arguments
fprintf('\n');
fprintf('\n Just a message \n');
```

This is the way this function must be called (it simply returns a couple of printed lines to the Matlab interface) :

```
» f020


 Just a message
```

Functions with more than one argument or output

First, here is an example of a function with two arguments that returns a single output:

```
function [r] = f021(s,t)
% Function with two arguments and one output
r = sin(s).*cos(t);
```

Evaluations of this function are shown below:

```
» f021(3,2)

ans =

   -0.0587

» m = f021([3,5,-2],[1,2,4])

m =

    0.0762    0.3991    0.5944
```

Next, a function of one argument returning two outputs:

```
function [x,y] = f022(theta)
% Function with one argument and two outputs
x = 23*sin(theta); y = 23*cos(theta);
```

Evaluations of this function are shown below:

```
[x1, y1] = f022(pi/3)

x1 =

   19.9186


y1 =

   11.5000

» f022(pi/3)

ans =

   19.9186

» x2 = f022(pi/6)

x2 =

   11.5000
```

Notice that, when no assignment is made to a vector, or when the assignment is made to a scalar variable, only the first value is returned.

Next, a function of two arguments returning two outputs:

```
function [x,y] = f023(r,theta);
% Conversion from polar to Cartesian coordinates
x = r.*cos(theta);
y = r.*sin(theta);
```

Evaluations of this function are shown below:

```
» [x3, y3] = f023(10.,3*pi/5)

x3 =

   -3.0902


y3 =

    9.5106
```

```
» r = [5.0, 3.0, 2.0]; theta = pi*[1/2, 1/3, 1/6];
» [x,y] = f023(r,theta)

x =

    0.0000    1.5000    1.7321


y =

    5.0000    2.5981    1.0000
```

The following is an example of a function that takes as argument a vector of 2 components and returns a vector of two components:

```
function [r] = f024(s)
% r and s are vectors with 2 components
r(1) = sqrt(s(1)^2+s(2)^2);
r(2) = atan(s(2)/s(1));
```

These are evaluation of this function:

```
» r1 = f024([-3.5, 1.2])

r1 =

    3.7000   -0.3303

EDU» r2 = f024([8.0; -2.0])

r2 =

    8.2462   -0.2450
```

Notice that the argument can be a row vector or a column vector, but the output is, by default, a row vector. If you want to force a column vector, then the function must be modified to read:

```
function [r] = f024(s)
% r and s are column vectors with 2 components
r1 = sqrt(s(1)^2+s(2)^2);
r2 = atan(s(2)/s(1));
r = [r1;r2];
```

The following function takes as argument a column vector of 2 components and returns a matrix:

```
function [J] = f025(s)
% Vector argument that returns a matrix output
J(1,1) = s(1)^2+ s(2)^2;
J(1,2) = s(1)/s(2);
J(2,1) = abs(s(1))+ abs(s(2));
J(2,2) = sqrt(s(1))+ s(2)^(1/3);
```

These are evaluations of this function:

```
»   f025([1,2])

ans =

    5.0000    0.5000
    3.0000    2.2599

» f025([1;2])

ans =

    5.0000    0.5000
    3.0000    2.2599
```

Functions that require constants for their evaluation

Suppose that you want to create a function that will require to calculate the expression

$$f(x,y) = (a*x+b*y)^2,$$

Where *a* and *b* are constants to be defined in the main Matlab interface. The following function was created to calculate *f(x,y)*:

```
function [z] = f026(x,y)
% Function that requires constants
global a; global b;
z = a*x + b*y;
```

Notice that there is a line in the function that makes reference to the constants *a* and *b*, i.e.,

```
global a; global b;
```

This line indicates that the values of *a* and *b*, which should be assigned in the main Matlab interface, are available to be used within function *f026*. Thus, *a* and *b* would be referred to as *global variables*, as opposite to variables *x* and *y*, which are referred to as *local variables* to the function.

An attempt to evaluate the function, as shown next, produces no output:

```
» a = 2; b = 3; f026(2.3,1.2)

ans =

     []
```

The reason for this lack of evaluation of *f026* is that variables *a* and *b* have not been declared as *global* within the main Matlab interface. The following statement may produce a warning message in your Matlab installation:

```
» global a; global b;
Warning: The value of local variables may have been changed to match the
         globals.  Future versions of MATLAB will require that you declare
         a variable to be global before you use that variable.
```

After having declared *a* and *b* as global variables, the evaluation of the function proceeds without a glitch:

```
» a = 2; b = 3; f026(2.3,1.2)

ans =

    8.2000
```

Function evaluation with *feval*

Function *feval* can be used as an alternative way to evaluate a function. For example, instead of using `m = f021([3,5,-2],[1,2,4])` to evaluate function *f021*, one could use:

```
» feval('f021',[3,5-2],[1,2,4])
??? Error using ==> .*
Matrix dimensions must agree.

Error in ==> C:\MATLAB_SE_5.3\bin\f021.m
On line 3  ==> r = sin(s).*cos(t);
```

This application of *feval*, however, was unsuccessful. A way to successfully apply *feval* in this case is to use variables to store the vectors before invoking function *feval*:

```
» r = [3,5,-2]; s = [1,2,4]; feval('f021',r,s)

ans =

    0.0762    0.3991    0.5944
```

The following application of *feval* is allowed because the arguments are scalar values:

```
» feval('f021',2.3,1.2)

ans =

    0.2702
```

Other examples of applications of *feval* follow:

```
» r1 = f024([-3.5,1.2])

r1 =

    3.7000   -0.3303

EDU» r1 = feval('f024',[-3.5,1.2])

r1 =

    3.7000   -0.3303
```

```
» [x1, y1] = feval('f022',pi/3)

x1 =

   19.9186


y1 =

   11.5000
```

Function *f026* still needs *a* and *b* to be declared as global variables:

```
» global a; global b; a = 2; b = 3; feval('f026',2.3,1.2)

ans =

    8.2000
```

Functions whose arguments are functions

Function *feval* is an example of a pre-defined function with one of its argument being another function. More specifically, the first argument of *feval* is the name of a function enclosed between apostrophes. The following is my first attempt to develop a user-defined function that takes a function name as an argument:

```
function [z] = f027(f,x,y)
% variable f in here is the name of a function
z = f(x) + f(y);
```

When I try to evaluate this function in the following example, however, an error is reported:

```

» f027('sin',pi/3,pi/6)
Warning: Subscript indices must be integer values.
> In C:\MATLAB_SE_5.3\bin\f027.m at line 3
Warning: Subscript indices must be integer values.
> In C:\MATLAB_SE_5.3\bin\f027.m at line 3

ans =

   230
```

Interestingly enough, a value is returned (ans = 230), but this value makes no sense whatsoever, and we must ignore it. The error shown above results from the fact that function *f027* has no way to know that *f* is a string, instead it takes *f* to be an array whose indices are *x* and *y*. To avoid this problem, *f027* is modified to read:

```
function [z] = f027(f,x,y)
% variable f in here is the name of a function
z = feval(f,x) + feval(f,y);
```

With this new version of the function, its evaluation proceeds smoothly:

```
» f027('sin',pi/3,pi/6)

ans =

    1.3660
```

Inline functions

So far, all the user-defined functions presented have been contained in an *m* file. There is a way to define what is referred to as an *inline function* by using function *inline*. Not all functions can be defined as inline functions, only those which have a single output and whose expressions contains not constants. Here is an example of an inline function:

```
» fplus = inline('x^2-1')

fplus =

     Inline function:
     fplus(x) = x^2-1
```

Notice that Matlab shows the resulting function expression as *fplus(x)* = *x*^2-*1* having identified *x* as the independent variable in the expression '*x*^2-*1*' used as argument of function *inline*. Evaluations of *fplus* are shown next:

```
» fplus(2)

ans =

     3

EDU» fplus([1 2 3])
??? Error using ==> inline/subsref
Error in inline expression ==> x^2-1
??? Error using ==> ^
Matrix must be square.
```

Notice that if a vector argument is used for *fplus*, the calculation fails. This is so because the power operator (^) applies, without using a dot, only to square matrices to calculate powers of matrices, e.g., $\mathbf{A}^2 = \mathbf{A}\cdot\mathbf{A}$, $\mathbf{A}^3 = \mathbf{A}\cdot\mathbf{A}^2$, etc. An expression such as [1 2 3]^2 needs to be calculated as a term-by-term operation, i.e., as [1 2 3].^2, or it produces an error. Thus, a way to resolve this problem is to redefine *fplus* as:

```
» fplus = inline('x.^2-1')

fplus =

     Inline function:
     fplus(x) = x.^2-1
```

With this re-definition we can now calculate *fplus([1 2 3])* without a problem:

```
» fplus([1 2 3])

ans =

     0     3     8
```

The following is an inline definition for a function of two variables:

```
» ftwo = inline('y^2+x^2')

ftwo =

     Inline function:
     ftwo(x,y) = y^2+x^2
```

Notice that Matlab chooses the arguments as (x,y), and not as (y,x), even though *y* shows up first in the expression. Thus, Matlab preserves alphabetical order of the variables when choosing the independent variables by itself. One can force the order of the variables by declaring so specifically in the call to function *inline*, for example:

```
» fthree = inline('y^2 + x^2','y','x')

fthree =

     Inline function:
     fthree(y,x) = y^2 + x^2
```

The names of inline functions can be used as arguments of functions such as *feval* or the user-defined function *f027* shown earlier. Some examples are shown next:

```
 » feval(fthree,2,-1)

ans =

     5

EDU» feval(fplus,3.2)

ans =

    9.2400
```

Notice that, when using function *feval*, the name of the inline functions is typed without apostrophes. As a matter of fact, using their names with apostrophes produce errors:

```
EDU» feval('fplus',3.2)
??? Cannot find function 'fplus'.

EDU» feval('fthree',2,-1)
??? Cannot find function 'fthree'.
```

These errors result from the fact that, when entered between apostrophes, *feval* seeks a file name for the function. Since *m* files named *fplus.m* and *fthree.m* do not exist, *feval* reports an error. Because functions *fplus* and *fthree* were declared as inline functions, their names are actually variable names available in the workspace, and therefore, must be written without apostrophes as arguments of *feval*.

Once again, inline functions cannot be defined with constants in their expressions. Consider, for example, the following case:

```
» global a; global b; a = 2.3; b = 4.5;
» ffour = inline('a*x+b*y','x','y')

ffour =

     Inline function:
     ffour(x,y) = a*x+b*y

» ffour(2,3)
??? Error using ==> inline/subsref
Error in inline expression ==> a*x+b*y
??? Undefined function or variable 'a'.
```

Even after defining *a* and *b* as global variables, evaluating *ffour* reports an error because Maple offers no way of inserting those global values in the evaluation of *ffour*. An attempt to use *feval* also fails.

```
» global a; global b; a = 2.3; b = 4.5;
EDU» feval(ffour,2,3)
??? Error using ==> inline/feval
Error in inline expression ==> a*x+b*y
??? Undefined function or variable 'a'.
```

Thus, if using constants in the expressions evaluated in a function, use an *m* file and make sure to define the constants as global variables, both in the Matlab interface as in the function file.

Using strings to create an inline function

Since inline functions do not allow the use of constants, one way to create the proper inline function is to replace the values of the constants in the function expression before calling function inline. For example, if we want to define the function *f100(x,y) = a*x+b*y*, with *a = 2* and *b = 3*, we could simply write:

```
» f100 = inline('2*x+3*y','x','y')

f100 =

     Inline function:
     f100(x,y) = 2*x+3*y
```

Suppose, however, that we want to make the substitution of the values of *a* and *b* within a function. We could use the following to create the proper expression:

```
» s = 'a*x+b*y'   % Expression for the function

s =

a*x+b*y
```

```
EDU» a = 2; b = 3;    % loading values of a and b

» s = strrep(s,'a',num2str(a))     % replace value of a in string
expression

s =

2*x+b*y

EDU» s = strrep(s,'b',num2str(b))    % replace value of b in string
expression

s =

2*x+3*y

» f100 = inline(s)

f100 =

     Inline function:
     f100(x,y) = 2*x+3*y
```

This procedure seems complicated, but it could work, within a function, to substitute constants in an expression. Then, the resulting expression can be used to create an inline function that uses those constants. For example, suppose that we want to solve the equation *f101(x)* = *a**x^2+b*x+c* = *0*, for several values of *a*, *b*, and *c*, using function *fzero*. The following function may do the work for us:

```
function [x] = fSQ(a, b, c, x0)
%This function solves the equation a*x^2+b*x+c = 0,
%for several values of a, b, and c, using function fzero.
s = 'a*x^2+b*x+c';
s = strrep(s,'a',num2str(a)); % replace value of a in s
s = strrep(s,'b',num2str(b)); % replace value of b in s
s = strrep(s,'c',num2str(c)); % replace value of c in s
ff = inline(s,'x');           % define local inline function
x  = fzero(ff,x0);            % invoke 'fzero' to solve *ff(x)=0'
```

Some evaluations of this function are shown next:

```
» fSQ(2,3,-1,5)
Zero found in the interval: [-1.4, 9.5255].

ans =

    0.2808

» fSQ(5,-5,0,2.5)
Zero found in the interval: [0.9, 3.6314].

ans =

     1
```

```
» fSQ(-2,2,3,1.2)
Zero found in the interval: [0.432, 1.968].

ans =

    1.8229
```

By the way, function *fzero* takes a function *f(x)* and initial value *x0*, and tries to find a solution to the equation *f(x) = 0* by a numerical approach starting at *x = x0*.

Summary

This document shows a number of examples regarding the use of user-defined functions in Matlab. Both, m-file and inline functions were presented. Some practical issues regarding the use of global variables, and strings to define inline functions were also discussed. The examples shown in this document can be used to developed your own Matlab functions related to numerical methods.