User-defined Functions in MATLAB

Gerald W. Recktenwald
Department of Mechanical Engineering
Portland State University
gerry@pdx.edu

Overview

Topics covered in these slides

- The basic function m-file
- Local functions, a.k.a. subfunctions
- Nested functions
- Anonymous functions

Basic m-file Functions

Basic *function m-files* have these properties:

1. The function is defined in a separate m-file.
2. Function name is the same as the name of the m-file.
3. The function can have many input parameters and many output parameters.
   - *input parameters* take on the values given by the calling function.
   - *output parameters* have values that are returned to the calling function.
4. Variables in the function have their own workspace (memory).
   - Variables in the function are separate from variables in other functions and in the command window environment, even if those variables have the same names.
   - Values are only shared via input and output parameters.

Function m-files

Syntax:

The first line of a function m-file has the form:

```
function [outArgs] = funName(inArgs)
```

outArgs are enclosed in `[ ]`

- *outArgs* is a comma-separated list of variable names
- `[ ]` is optional if there is only one parameter
- functions with no *outArgs* are legal

inArgs are enclosed in `( )`

- *inArgs* is a comma-separated list of variable names
- functions with no *inArgs* are legal

Basic m-file Function: Example

The `quadraticRoots` function is contained in `quadraticRoots.m` and can be called from the command line, or another m-file.

```
>> c1 = 1;
>> c2 = 5;
>> c3 = 2;
>> r = quadraticRoots(c1,c2,c3);
r =
   -0.4384   -4.5616
```

quadraticRoots.m

```
function x = quadraticRoots(a,b,c)
% quadraticRoots  Compute the two roots
%                 of the quadratic equation
%
% Input:  a,b,c = (scalar) coefficients of
%                 the quadratic equation
%                   a*x^2 + b*x + c = 0
%
% Output:  x = vector of the two roots
%
d = sqrt( b^2 - 4*a*c);
x1 = (-b + d)/(2*a);
x2 = (-b - d)/(2*a);
x = [x1, x2];
end
```

Function Input and Output (1)

Examples: Demonstrate use of I/O arguments

- `twosum.m` — two inputs, no output
- `threesum.m` — three inputs, one output
- `addmult.m` — two inputs, two outputs

Function Input and Output (2)

twosum.m

```
function twosum(x,y)
% twosum  Add two matrices
%         and print the result
x+y
end
```

threesum.m

```
function s = threesum(x,y,z)
% threesum  Add three variables
%           and return the result
s = x+y+z;
end
```

addmult.m

```
function [s,p] = addmult(x,y)
% addmult  Compute sum and product
%          of two matrices
s = x+y;
p = x*y;
end
```

Function Input and Output Examples (3)

Example: Experiments with `twosum`

```
>> twosum(2,2)
ans =
      4

>> x = [1 2];  y = [3 4];
>> twosum(x,y)
ans =
     4     6
```

Note: The result of the addition inside `twosum` is exposed because the `x+y` expression does not end in a semicolon. (What if it did?)

Function Input and Output Examples (4)

```
>> A = [1 2; 3 4];  B = [5 6; 7 8];
>> twosum(A,B);
ans =
     6     8
    10    12

>> twosum('one','two')
ans =
   227   229   212
```

Note: The strange results produced by `twosum('one','two')` are obtained by adding the numbers associated with the ASCII character codes for each of the letters in 'one' and 'two'.

Try `double('one')` and `double('one') + double('two')`.

Function Input and Output Examples (5)

Example: Experiments with `twosum`:

```
>> clear
>> x = 4; y = -2;
>> twosum(1,2)
ans =
     3

>> x+y
ans =
     2

>> disp([x y])
     4    -2

>> who

Your variables are:

ans       x         y
```

In this example, the `x` and `y` variables defined in the workspace are distinct from the `x` and `y` variables defined in `twosum`. The `x` and `y` in `twosum` are *local* to `twosum`.

Function Input and Output Examples (6)

Example: Experiments with `threesum`:

```
>> a = threesum(1,2,3)
a =
    6

>> threesum(4,5,6)
ans =
   15

>> b = threesum(7,8,9);
```

Note: The last statement produces no output because the assignment expression ends with a semicolon. The value of 24 is stored in `b`.

Function Input and Output Examples (7)

Example: Experiments with `addmult`

```
>> [a,b] = addmult(3,2)
a =
    5
b =
    6

>> addmult(3,2)
ans =
    5

>> v = addmult(3,2)
v =
    5
```

Note: `addmult` *requires* two return variables. Calling `addmult` with no return variables or with one return variable causes undesired behavior.

Summary of Input and Output Parameters

- Values are communicated through input arguments and output arguments.
- Variables defined inside a function are *local* to that function. Local variables are invisible to other functions and to the command environment.
- The number of return variables should match the number of output variables provided by the function. This can be relaxed by testing for the number of return variables with `nargout`.

Variable Number of Input and Output Parameters

Variable Input and Output Arguments (1)

Each function has internal variables, *nargin* and *nargout*.

Use the value of `nargin` at the beginning of a function to find out how many input arguments were supplied.

Use the value of `nargout` at the end of a function to find out how many input arguments are expected.

Usefulness:

- Allows a single function to perform multiple related tasks.
- Allows functions to assume default values for some inputs, thereby simplifying the use of the function for some tasks.

Variable Input and Output Arguments (2)

Consider the built-in `plot` function

	Inside the `plot` function	
	`nargin`	`nargout`
`plot(x,y)`	2	0
`plot(x,y,'s')`	3	0
`plot(x,y,'s--')`	3	0
`plot(x1,y1,'s',x2,y2,'o')`	6	0
`h = plot(x,y)`	2	1

The values of `nargin` and `nargout` are determined when the `plot` function is invoked.

Variable Input and Output Arguments (3)

Draw a circle of radius 1, centered at $(x, y) = (0, 0)$:

```
>> drawCircle(1)
```

Draw a circle of radius 1, centered at $(x, y) = (2, 0)$:

```
>> drawCircle(1,2)
```

Draw a circle of radius 1, centered at $(x, y) = (2, -1)$:

```
>> drawCircle(1,2,-1)
```

```
function drawCircle(r,x0,y0,lineStyle)
% drawCircle  Draw a circle in the (x,y) plane
%
% Synopsis:  drawCircle(r)
%            drawCircle(r,x0)
%            drawCircle(r,x0,y0)
%            drawCircle(r,x0,y0,lineStyle)
%
% Input:  r = radius of the circle
%         x0,y0 = (x,y) coordinates of center of
%             the circle. Default:  x0 = 0,  y0 = 0;
%         lineStyle = (optional) string used
%             to specify line style of the circle.
%             Default: lineStyle = '-'

if nargin<2, x0 = 0;  end
if nargin<3, y0 = 0;  end
if nargin<4, lineStyle = '-';  end

t = linspace(0,2*pi);
x = x0 + r*cos(t);
y = y0 + r*sin(t);
plot(x,y,lineStyle)
end
```

Local Functions

Local Functions in an m-file

MATLAB m-files can contain more than one function.

1. The first function is called the *main function* and has the same name as the m-file.
2. Only the first function is visible to the command window or to functions in other m-files.
3. Other functions in the m-file are called *local functions* or *subfunctions*.
4. Functions in the same m-file can call each other, but the main function should only (usually) be called from outside the m-file.
5. Each function in the m-file must start with "`function`" and have a matching "`end`" statement.
6. Variables in the local and main functions *do not share* the same memory space. In other words, the main function and local functions only communicate via input and output parameters.

Local functions in a single m-file

meanFixNaN.m

```
function ave = meanFixNaN(x,nansub)
% meanFixNaN  Compute mean of a vector with NaNs
%             by removing or replacing NaN

if nargin<2
  y = x( ~isnan(x) );    % Default: Remove NaNs
else
  y = fixnan(x,nansub); % Sub nansub for NaNs
end

ave = mean(y);

end

% ------------------
function y = fixnan(x,nansub)
% fixnan  Replace NaNs with another value

y = x;                      %  Make a copy
y( isnan(y) ) = nansub;     %  Replace NaNs

end
```

function – end defines scope

function – end defines scope

Main function is meanFixNan and is visible to the command line and other functions

Local function is fixnan and is only visible to functions in meanFixNaN.m

Nested Functions in an m-file

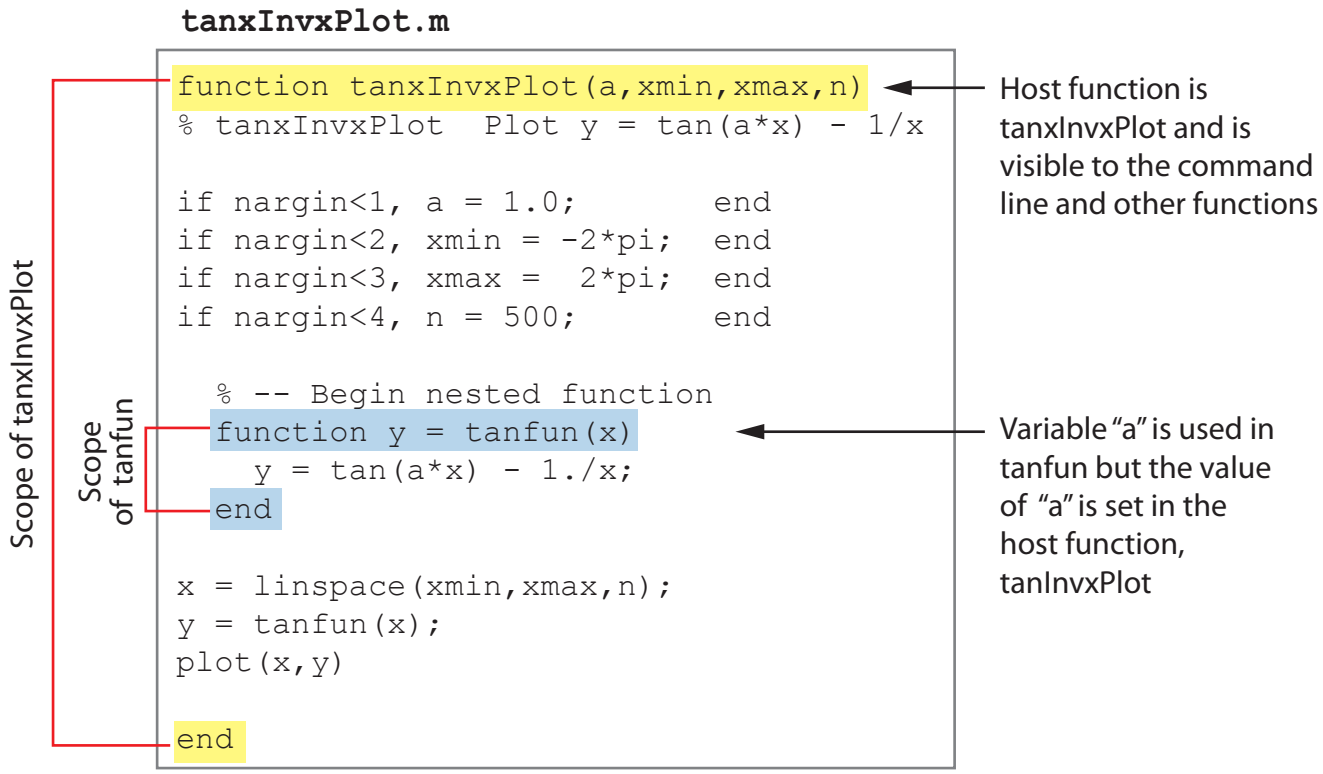
MATLAB functions can contain internal functions that are either *anonymous functions* or *nested functions*.

Nested functions:

1. Begin with a "`function`" statement.
2. End with an "`end`" statement.
3. Share their program variables with the host function they are nested within.
4. Communicate with the host function via input and output variables.
5. Can be passed as arguments to other functions.

Item 5 is the reason why nested functions are important for root-finding and other numerical analysis procedures with MATLAB.

Nested Functions in an m-file



Anonymous Functions

Anonymous functions:

1. Have *only* a one-line executable statement.
2. Do not have either "`function`" or "`end`" statements.
3. Use a special `@(x)` syntax.

Anonymous functions are a handy way to make simple functions from analytical formulas.

Anonymous functions can be used for root-finding and other numerical analysis procedures with MATLAB. However, other than brevity, anonymous functions do not have an advantage over nested functions.

Anonymous Functions in an m-file

tanxInvxPlotAnon.m

```
function tanxInvxPlotAnon(a,xmin,xmax,n)
% tanxInvxPlotAnon  Plot y = tan(a*x) - 1/x

if nargin<1, a = 1.0;       end
if nargin<2, xmin = -2*pi;  end
if nargin<3, xmax =  2*pi;  end
if nargin<4, n = 500;       end

% -- Define anonymous function
tanfun = @(x,a) tan(a*x) - 1./x;

x = linspace(xmin,xmax,n);
y = tanfun(x,a);
plot(x,y)

end
```

Scope of tanxInvxPlotAnon

One-ine scope of tanfun

Main function is tanxInvxPlot and is visible to the command line and other functions

Variable "a" is used in tanfun must be passed in as the second parameter