Contents

1 Functions

MATLAB programs are stored as plain text in files having names that end with the extension ".m". These files are called, not surprisingly, m-files. Each m-file contains exactly one MATLAB function. Thus, a collection of MATLAB functions can lead to a large number of relatively small files.

One useful difference between MATLAB and traditional high level languages is that MATLAB functions can be used interactively. In addition to providing the obvious support for interactive calculation, this is a very convenient way to debug functions that are part of a bigger project.

MATLAB functions have two parameter lists, one for input and one for output. This supports one of the cardinal rules of MATLAB programming: don't change the input parameters of a function. This will require you to make some slight adjustments in the way you program. In the end this shift will help you write better MATLAB code.

1.1 Function Definition

The first line of a function m-file must be of the following form.

function [output.parameter.list] = function.name(input.parameter.list)

The first word must always be **function**. Following that, the (optional) output parameters are enclosed in square brackets [ ]. If the function has no `output.parameter.list` the square brackets and the equal sign are omitted.

The `function.name` is a character string that will be used to call the function. The function.name must be the same as the file name (without the ".m") in which the function is stored. In other words the MATLAB function, "func", must be stored in the file, "func.m".

Following the file name is the (optional) `input.parameter.list`. There can exactly be one MATLAB function per m-file.

1.2 Arguments to the Function

MATLAB functions must begin with the keyword "function" MATLAB supports multiple output arguments, listed as shown in square brackets.

- If a function only has a single output argument, then the square brackets are not required.
- If a function does not have any output arguments, then neither the square brackets nor the equals sign that follows are used.
- The name of the function must match the name of the .m file containing the function. (If not, then MATLAB ignores the internal name, which just confuses matters greatly.)
- The output variables list is a comma separated list of variables calculated within the function file and enclosed within square brackets in the function definition. These are the values that the function will return to the MATLAB workspace.
- The input variables list is a comma separated list of variables that need to be passed to the function when you use it.

- `function.name` is the name that you have chosen for the function and MUST be the same as the filename that you use to save the m-file (but it does not include the .m extension)

1.3 Useful Commands

The number of input arguments are in the built-in variable `nargin`.

1.4 Example 1 : Simple Function

```
function addtwo(x,y)
%  addtwo(x,y)  Adds two numbers, vectors, whatever, and
%               print the result = x + y
x+y
```

Exercise: Use the code in example 1 to compute the hypothenus(H) when the inputs are the Adjacent(A) and Opposite(O).

1.5 Comment Statements as Help

The first two lines after the function definition are comment statements. Not only do these statements describe the statements in the file, their position in the file supports the on-line help facility in MATLAB.

If the first line of a MATLAB function definition is immediately followed by non-blank comment statements, then those comment statements are printed to the command window when you type `help functionname`. Try it with the `addtwo` function. MATLAB will print up until a blank line or an executable statement, whichever comes first.

1.6 Example 2: Area of a Trapezoid

Here is another simple function, **traparea.m**, with three input parameters and one output parameter. Since there is only one output parameter the square brackets may be omitted.

```
function area = traparea(a,b,h)
%  traparea(a,b,h)   Computes the area of a trapezoid given
%                    the dimensions a, b and h, where a and b
%                    are the lengths of the parallel sides and
%                    h is the distance between these sides

% Compute the area, but suppress printing of the result
area = 0.5*(a+b)*h;
```

Note that there is a blank line between the comment statements that describe the function and the single comment statement that describes the calculation of area. The comment statement beginning *Compute the area...* will not be printed if you type `help traparea`.

1.7 Example 3: Cartesian Coordinates

Here is another simple function, used to convert Cartesian coordinates to polar coordinates (**cart2plr.m**). Importantly there are two input parameters and two output parameters.

As there are multiple output parameter the square brackets are included to specify the output as a vector.

```
function [r,theta] = cart2plr(x,y)
%   cart2plr  Convert Cartesian coordinates to polar coordinates
%
%   [r,theta] = cart2plr(x,y) computes r and theta with
%
%       r = sqrt(x^2 + y^2);
%       theta = atan2(y,x);
r = sqrt(x^2 + y^2);
theta = atan2(y,x);
```

1.8 Example 4: Normal Distribution

In probability theory, the normal probability distribution is a well-known continuous probability distribution, defined by the formula

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

This function calculates the **density Function** of the Normal Distribution with mean mu and standard deviation sigma at a point z.

```
function f = normdensity(z, mu, sigma);
   f = (1/(sqrt(2*pi)*sigma))*exp(-(z-mu)^2/(2*sigma^2));
end
```

Importantly the standard deviation (sigma) must be a positive non-zero real number. We will update the code to only accept a positive input.

```
function f = normdensity(z, mu, sigma);
if sigma <= 0
   fprintf(Invalid input\n);
   f = NaN;
else
   f = (1/(sqrt(2*pi)*sigma))*exp(-(z-mu)^2/(2*sigma^2));
end
```

1.9 Example 5: Centre of Gravity

We can write a function fo rfinding the centre of gravity of a set of weights, given their locations along a straight line. If you wanted to calculate centres of gravity repeatedly, then you could write a function which would perform the required calculation, and store it for future use.

```
function cg = cofg(weights, locations)
% Some Help Commands here
% Some Help Commands here

cg = weights * locations' / sum(weights);
```

```
cg = cofg([3 1 7],[-2 0 5])
```

1.10 Example 6: Functions with Subfunctions

The first function in an m-file is the primary function, whose name must match the file name.

Additional subfunctions may follow the primary function in the file. The syntax of subfunctions is the same as for the primary function, except with different function names.

Subfunctions are only accessible from other functions within the same file.

```
function [ result, square, linear ] = quadratic( a, b, c, x )
% QUADRATIC - Calculates the polynomial A x^2 + B x + C
%
% QUADRATIC( A, B, C, X ) calculates the quadratic polynomial:
%       A * X^2 + B * X + C
if( nargin < 4 )
    x = 1;
end
square = a * x * x;
linear = b * x;
constant = c;
result = addThree( square, linear, constant );

function total = addThree( x, y, z )
total = x + y + z;
```

1.11 Example 7: Area Function

The area, A, of a triangle with sides of length a, b and c is given by

$$A = \sqrt{s(sa)(sb)(sc)}$$

where $s = (a + b + c)/2$. Write a MATLAB function that will accept the values a, b and c as inputs and return the value of A as output.

1.12 Example 8: Simulating throws of a pair of dice

This requires random numbers that are integers in the range 1 to 6 which can be produced with `floor(1 + 6*rand)`

```
function [d] = dice(n)
% simulates "n" throws of a pair of dice
% Input: n, the number of throws
% Output: an n times 2 matrix, each
% row referring to one throw.
%
% Useage: T = dice(3)
d = floor(1 + 6*rand(n,2));
%% end of dice
```

1.13 Example 9: Fibonnacci Series

Construct a function that will return the $n-$th Fibonnaci number f_n, where $f_1 = 0$, $f_2 = 1$ and

$$f_n = f_{n-1} + f_{n-2}, n = 3, 4, 5, \ldots$$

```
function f = Fib1(n)
% Returns the nth number in the
% Fibonacci sequence.
F = zeros(1,n);
F(2) = 1;
for i = 3:n
F(i) = F(i-1) + F(i-2);
end
f = F(n);
```

This version makes use of an idea called **recursive programming**the function makes calls to itself.

```
function f = Fib2(n)
% Returns the nth number in the
% Fibonacci sequence.
if n==1
f = 0;
elseif n==2
f = 1;
else
f = Fib2(n-1) + Fib2(n-2);
end
```

1.14 Example 10 : Angle Converter

Write a MATLAB code to convert from degrees to radians. The user should enter the starting and ending points in the table (in degrees), as well as the number of entries desired. The code should then generate the table and write the results to the screen.

```
function angle_table( lo, hi, n )
% angle_table( lo, hi, n )
%
% Outputs a table of angles in degrees and radians.
%
% inputs:
%   lo - lower entry in the table (degrees)
%   hi - upper entry in the table (degrees)
%   n  - number of entries in the table
%

degrees = linspace(lo,hi,n);
radians = degrees * pi/180;

fprintf('Degrees   Radians\n');
for i=1:n
  fprintf('%8.2f %8.2f\n',degrees(i),radians(i) );
end
```

1.15 Example 11 : Newton's Method

Computing $\sqrt{5}$ with Newton's method.

Estimate for square root (x) gets continuously updated.

$$x := \frac{(x + \frac{S}{x})}{2}$$

```
function X=Newton(X0,numIter)
X(1)=X0;
for i=1:numIter
X(i+1) = X(i)/2 + 5/(2*X(i));
end
end
```

In the example above, we change the size of the array. MATLAB doesn't like it and takes a lot of time to do it. The best way is to preallocate zeros in the size of the array.

```
function X=Newton(X0,numIter)
X=zeros(1,numIter+1);
X(1)=X0;
for i=1:numIter
X(i+1) = X(i)/2 + 5/(2*X(i));
end
end
```

2 Plotting in MATLAB

The MATLAB command `plot` generates dots at each (x,y) pair and then connects the dots with a line. To make plot of a function look smoother, evaluate at more points

```
x=linspace(0,4*pi,1000)
plot(x,sin(x))
```

x and y vectors must be same size or else youll get an error.

2.1 3d Plotting in MATLAB

We can plot in 3 dimensions just as easily as in 2 dimensions.

```
time=0:0.001:4*pi;
x=sin(time);
y=cos(time);
z=time;
plot3(x,y,z,'k','LineWidth',2);
zlabel('Time');
```