

# CSC1050 – Hands-On Introduction to MATLAB

Functions and Files

## Section 3.1 Elementary Functions

### Getting Help for Functions:

You can use the `lookfor` command to find functions that are relevant to your application.

For example, type `lookfor imaginary` to get a list of the functions that deal with imaginary numbers. You will see listed:

<code>imag</code>	Complex imaginary part
<code>i</code>	Imaginary unit
<code>j</code>	Imaginary unit

## Some common mathematical functions: Table 3.1–1

**Exponential**  
**exp (x)**

Exponential;  $e^x$

**sqrt (x)**

Square root;  $\sqrt{x}$

**Logarithmic**  
**log (x)**

Natural logarithm;  $\ln x$

**log10 (x)**

Common (base 10) logarithm;  
 $\log x = \log_{10} x$

## Some common mathematical functions (continued)

### Complex

<b>abs (x)</b>	Absolute value.
<b>angle (x)</b>	Angle of a complex number.
<b>conj (x)</b>	Complex conjugate.
<b>imag (x)</b>	Imaginary part of a complex number.
<b>real (x)</b>	Real part of a complex number.

## Some common mathematical functions (continued)

### Numeric

**ceil(x)** Round to nearest integer toward  $\infty$ .

**fix(x)** Round to nearest integer toward zero.

**floor(x)** Round to nearest integer toward  $-\infty$ .

**round(x)** Round toward nearest integer.

**sign(x)** Signum function:  
+1 if  $x > 0$ ; 0 if  $x = 0$ ; -1 if  $x < 0$ .

## Operations on Arrays

MATLAB will treat a variable as an array automatically.  
For example, to compute the square roots of 5, 7, and 15,  
type

```
>>x = [5,7,15];
```

```
>>y = sqrt(x)
```

```
y =
```

```
2.2361
```

```
2.6358
```

```
3.8730
```

## Expressing Function Arguments

We can write  $\sin 2$  in text, but MATLAB requires parentheses surrounding the 2 (which is called the *function argument* or *parameter*).

Thus to evaluate  $\sin 2$  in MATLAB, we type `sin(2)` . The MATLAB function name must be followed by a pair of parentheses that surround the argument.

To express in text the sine of the second element of the array  $x$ , we would type  $\sin[x(2)]$ . However, in MATLAB you cannot use square brackets or braces in this way, and you must type `sin(x(2))` .

## Expressing Function Arguments (continued)

To evaluate  $\sin(x^2 + 5)$ , you type `sin(x.^2 + 5)` .

To evaluate  $\sin(\sqrt{x+1})$ , you type `sin(sqrt(x)+1)` .

Using a function as an argument of another function is called *function composition*. Be sure to check the order of precedence and the number and placement of parentheses when typing such expressions.

Every left-facing parenthesis requires a right-facing mate. However, this condition does not guarantee that the expression is correct!



## Expressing Function Arguments (continued)

Another common mistake involves expressions like  $\sin^2 x$ , which means  $(\sin x)^2$ .

In MATLAB we write this expression as `(sin(x))^2`,  
*not as* `sin^2(x)`, `sin^2x`, `sin(x^2)`, or  
`sin(x)^2`!

## Expressing Function Arguments (continued)

The MATLAB trigonometric functions operate in radian mode. Thus `sin(5)` computes the sine of 5 rad, not the sine of 5°.

To convert between degrees and radians, use the relation

$$q_{\text{radians}} = (\pi / 180) q_{\text{degrees}}$$

## Trigonometric functions: Table 3.1–2

$\cos(x)$	Cosine; $\cos x$ .
$\cot(x)$	Cotangent; $\cot x$ .
$\csc(x)$	Cosecant; $\csc x$ .
$\sec(x)$	Secant; $\sec x$ .
$\sin(x)$	Sine; $\sin x$ .
$\tan(x)$	Tangent; $\tan x$ .

## Inverse Trigonometric functions: Table 3.1–2

<b><code>acos(x)</code></b>	Inverse cosine; arccos $x$ .
<b><code>acot(x)</code></b>	Inverse cotangent; arccot $x$ .
<b><code>acsc(x)</code></b>	Inverse cosecant; arccsc $x$ .
<b><code>asec(x)</code></b>	Inverse secant; arcsec $x$ .
<b><code>asin(x)</code></b>	Inverse sine; arcsin $x$ .
<b><code>atan(x)</code></b>	Inverse tangent; arctan $x$ .
<b><code>atan2(y, x)</code></b>	Four-quadrant inverse tangent.

## Hyperbolic functions: Table 3.1–3

<b><math>\cosh(x)</math></b>	Hyperbolic cosine
<b><math>\coth(x)</math></b>	Hyperbolic cotangent.
<b><math>\operatorname{csch}(x)</math></b>	Hyperbolic cosecant
<b><math>\operatorname{sech}(x)</math></b>	Hyperbolic secant
<b><math>\sinh(x)</math></b>	Hyperbolic sine
<b><math>\tanh(x)</math></b>	Hyperbolic tangent

## Inverse Hyperbolic functions: Table 3.1–3

<b><math>\operatorname{acosh}(x)</math></b>	Inverse hyperbolic cosine
<b><math>\operatorname{acoth}(x)</math></b>	Inverse hyperbolic cotangent
<b><math>\operatorname{acsch}(x)</math></b>	Inverse hyperbolic cosecant
<b><math>\operatorname{asech}(x)</math></b>	Inverse hyperbolic secant
<b><math>\operatorname{asinh}(x)</math></b>	Inverse hyperbolic sine
<b><math>\operatorname{atanh}(x)</math></b>	Inverse hyperbolic tangent;

## Section 3.2: User-Defined Functions

The first line in a function file must begin with a *function definition line* that has a list of inputs and outputs. This line distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = name(input variables)
```

Note that the output variables are enclosed in *square brackets*, while the input variables must be enclosed with *parentheses*. The function name (i.e. **name**) should be the same as the file name in which it is saved (with the .m extension).

## User-Defined Functions: Example

```
function z = fun(x,y)
u = 3*x;
z = u + 6*y.^2;
```

Note the use of a semicolon at the end of the lines. This prevents the values of `u` and `z` from being displayed.

Note also the use of the array exponentiation operator (`.^`). This enables the function to accept `y` as an array.



## User-Defined Functions: Example (continued)

Call this function with its output argument:

```
>>z = fun(3,7)
z =
    303
```

The function uses  $x = 3$  and  $y = 7$  to compute  $z$ .

## User-Defined Functions: Example (continued)

Call this function without its output argument and try to access its value. You will see an error message.

```
>>fun(3,7)
```

```
ans =
```

```
    303
```

```
>>z
```

```
??? Undefined function or variable 'z'.
```

## User-Defined Functions: Example (continued)

Assign the output argument to another variable:

```
>>q = fun(3,7)
q =
    303
```

You can suppress the output by putting a semicolon after the function call.

For example, if you type `q = fun(3,7);` the value of `q` will be computed but not displayed (because of the semicolon).

The variables `x` and `y` are *local* to the function `fun`, so unless you pass their values by naming them `x` and `y`, their values will not be available in the workspace outside the function. The variable `u` is also local to the function. For example,

```
>>x = 3;y = 7;  
>>q = fun(x,y) ;  
>>x  
x =  
    3  
>>y  
y =  
    7  
>>u  
??? Undefined function or variable 'u' .
```

Only the order of the arguments is important, not the names of the arguments:

```
>>x = 7;y = 3;  
>>z = fun(y,x)  
z =  
    303
```

The second line is equivalent to `z = fun(3,7)` .

You can use arrays as input arguments:

```
>>r = fun([2:4],[7:9])
```

```
r =
```

```
    300    393    498
```

A function may have more than one output. These are enclosed in square brackets.

For example, the function `circle` computes the area  $A$  and circumference  $C$  of a circle, given its radius as an input argument.

```
function [A, C] = circle(r)
A = pi*r.^2;
C = 2*pi*r;
```

The function is called as follows, if the radius is 4.

```
>>[A, C] = circle(4)
```

```
A =
```

```
50.2655
```

```
C =
```

```
25.1327
```



A function may have no input arguments and no output list.

For example, the function `show_date` computes and stores the date in the variable `today`, and displays the value of `today`.

```
function show_date  
today = date
```

## Examples of Function Definition Lines

1. One input, one output:

```
function [area_square] = square(side)
```

2. Brackets are optional for one input, one output:

```
function area_square = square(side)
```

3. Three inputs, one output:

```
function [volume_box] = box(height,width,length)
```

4. One input, two outputs:

```
function [area_circle,circumf] = circle(radius)
```

5. No named output: **function sqplot(side)**

## Function Example

```
function [dist,vel] = drop(g,v0,t);  
% Computes the distance travelled and the  
% velocity of a dropped object,  
% as functions of g,  
% the initial velocity v0, and  
% the time t.  
vel = g*t + v0;  
dist = 0.5*g*t.^2 + v0*t;
```

## Function Example (continued)

1. The variable names used in the function definition may, but need not, be used when the function is called:

```
>>a = 32.2;  
>>initial_speed = 10;  
>>time = 5;  
>>[feet_dropped,speed] = . . .  
drop(a,initial_speed,time)
```

## Function Example (continued)

2. The input variables need not be assigned values outside the function prior to the function call:

```
[feet_dropped, speed] = drop(32.2, 10, 5)
```

3. The inputs and outputs may be arrays:

```
[feet_dropped, speed] = drop(32.2, 10, [0:1:5])
```

This function call produces the arrays `feet_dropped` and `speed`, each with six values corresponding to the six values of time in the array `time`.

## **Local Variables**

The names of the input variables given in the function definition line are local to that function.

This means that other variable names can be used when you call the function.

All variables inside a function are erased after the function finishes executing, except when the same variable names appear in the output variable list used in the function call.

## Global Variables

The `global` command declares certain variables global, and therefore their values are available to the basic workspace and to other functions that declare these variables global.

The syntax to declare the variables `a`, `x`, and `q` is

```
global a x q
```

Any assignment to those variables, in any function or in the base workspace, is available to all the other functions declaring them global.

## Application: Finding Zeros of a Function

You can use the `fzero` function to find the zero of a function of a single variable, which is denoted by  $x$ . One form of its syntax is

```
fzero('function', x0)
```

where `function` is a string containing the name of the function, and `x0` is a user-supplied guess for the zero.

The `fzero` function returns a value of  $x$  that is near `x0`. It identifies only points where the function crosses the  $x$ -axis, not points where the function just touches the axis.

For example, `fzero('cos', 2)` returns the value 1.5708.



## Using `fzero` with User-Defined Functions

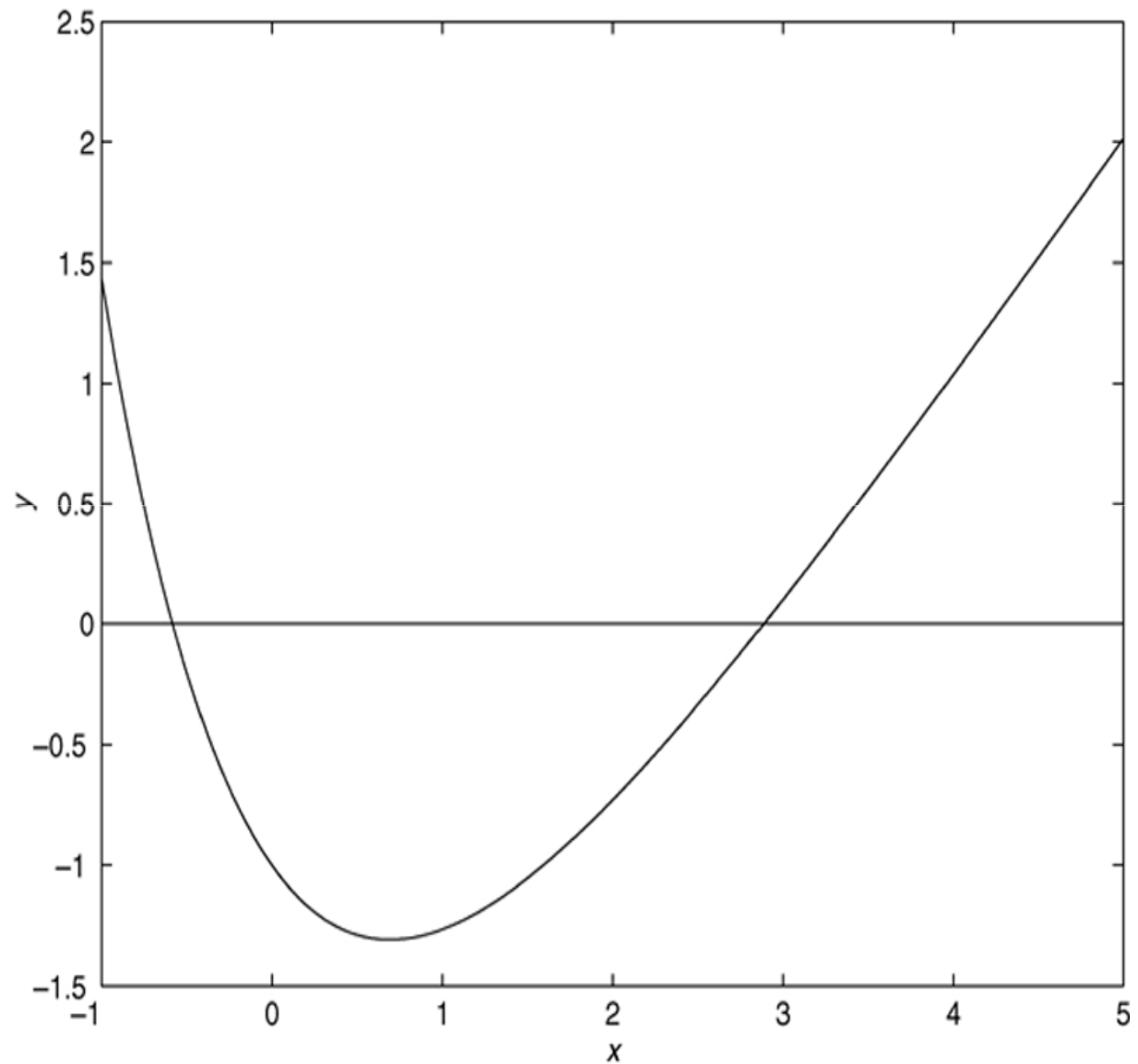
To use the `fzero` function to find the zeros of more complicated functions, it is more convenient to define the function in a function file.

For example, if  $y = x + 2e^{-x} - 3$ , define the following function file:

```
function y = f1(x)
y = x + 2*exp(-x) - 3;
```

## Plot of the function $y = x + 2e^{-x} - 3$ . Figure 3.2-1

There is a  
zero near  $x =$   
 $-0.5$  and one  
near  $x = 3$ .



## Example (continued)

To find a more precise value of the zero near  $x = -0.5$ , type

```
>>x = fzero('f1',-0.5)
```

The answer is  $x = -0.5881$ .

## Finding the Minimum of a Function

The `fminbnd` function finds the minimum of a function of a single variable, which is denoted by  $x$ . One form of its syntax is

**`fminbnd('function', x1, x2)`**

where `function` is a string containing the name of the function. The `fminbnd` function returns a value of  $x$  that minimizes the function in the interval  $x1 \leq x \leq x2$ .

For example, `fminbnd('cos', 0, 4)` returns the value 3.1416.

When using `fminbnd` it is more convenient to define the function in a function file. For example, if  $y = 1 - xe^{-x}$ , define the following function file:

```
function y = f2(x)
y = 1-x.*exp(-x);
```

To find the value of  $x$  that gives a minimum of  $y$  for  $0 \leq x \leq 5$ , type

```
>>x = fminbnd('f2',0,5)
```

The answer is  $x = 1$ . To find the minimum value of  $y$ , type `y = f2(x)`. The result is  $y = 0.6321$ .

A function can have one or more *local* minima and a *global* minimum.

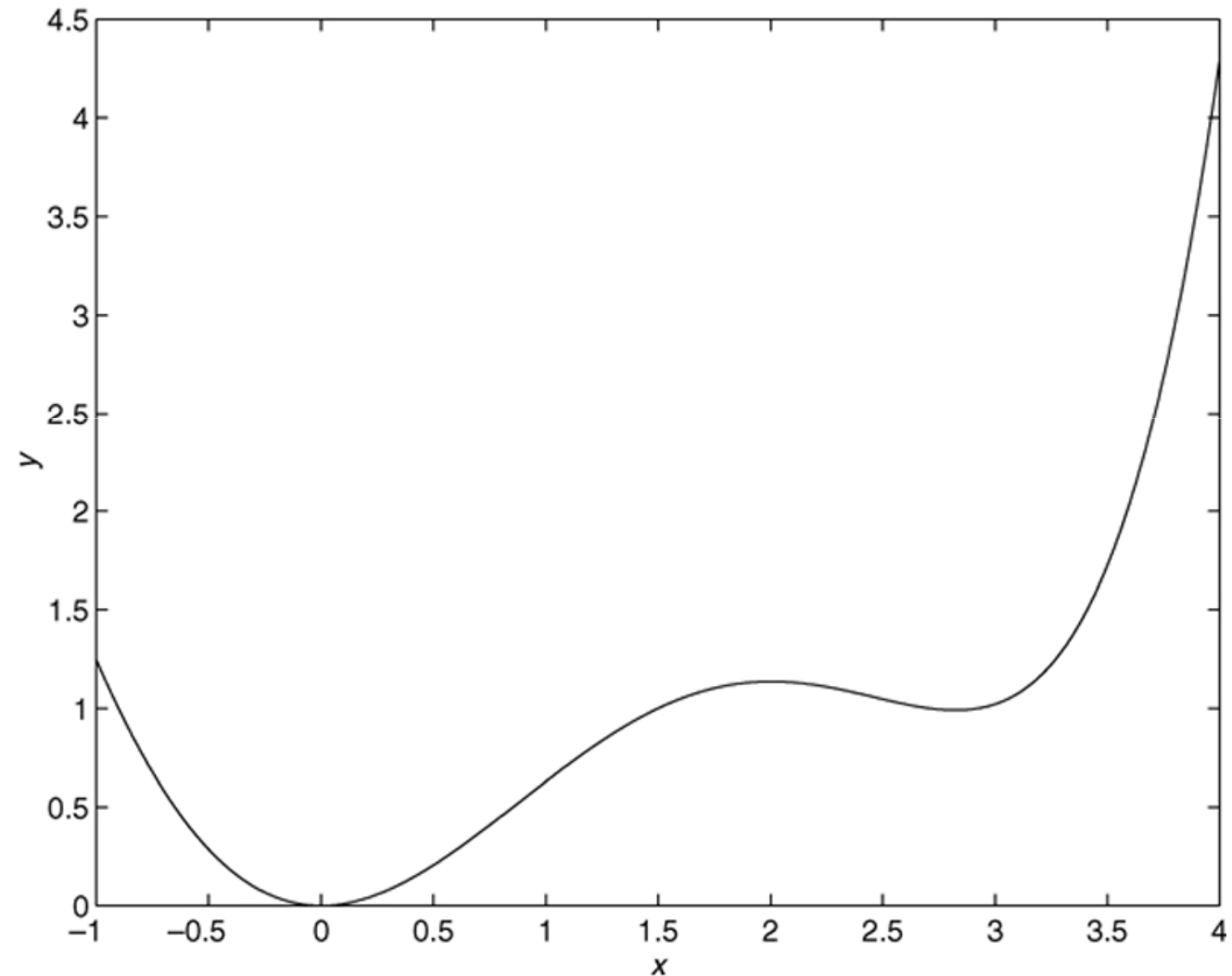
If the specified range of the independent variable does not enclose the global minimum, `fminbnd` will not find the global minimum.

Also, `fminbnd` might not find a minimum that occurs on a boundary.

**Plot of the function  $y = 0.025x^5 - 0.0625x^4 - 0.333x^3 + x^2$ .**

Figure 3.2–2

This function has one local and one global minimum. On the interval  $[1, 4]$  the minimum is at the boundary,  $x = 1$ .



To find the minimum of a function of more than one variable, use the `fminsearch` function. One form of its syntax is

**`fminsearch('function' , x0)`**

where `function` is a string containing the name of the function. The vector `x0` is a guess that must be supplied by the user.



To minimize the function  $f = xe^{-x^2 - y^2}$ , we first define it in an M-file, using the vector  $\mathbf{x}$  whose elements are  $\mathbf{x}(1) = x$  and  $\mathbf{x}(2) = y$ .

```
function f = f4(x)  
f = x(1) .* exp(-x(1).^2 - x(2).^2) ;
```

Suppose we guess that the minimum is near  $x = y = 0$ .  
The session is

```
>>fminsearch('f4',[0,0])  
ans =  
    -0.7071    0.000
```

Thus the minimum occurs at  $x = -0.7071$ ,  $y = 0$ .

**Function Handles: Now the preferred way to reference a function, instead of using single quotes.**

You can create a function handle to any function by using the *at* sign, @, before the function name. You may also name the handle if you wish, and use the handle to reference the function. For example, to create a handle named `th` to the MATLAB function `tanh`, you type

```
>>th = @tanh;
```

Or, you may simply use `@tanh` to reference the function.

## Function Handles (continued)

A common use of a function handle is to pass the function as an argument to another function. For example, we can plot  $\tanh(x)$  over  $0 \leq x \leq 6$  as follows:

```
>>plot([0:0.01:6],th([0:0.01:6]))
```

In this simple example, use of a handle does not save many keystrokes. However, there are more effective ways to use handles, as we will see.

## Application of Handles: Finding Zeros of a Function

You can use a function handle with the `fzero` function to find the zero of a function of a single variable, which is denoted by `x`. One form of its syntax is

```
fzero(@function, x0)
```

where `@function` is the handle to the function being analyzed, and `x0` is a user-supplied guess for the zero.

For example, `fzero(@cos, 2)` returns the value 1.5708.

## Function Handles (continued)

Other examples of the use of function handles:

```
>>x = fminbnd(@f2,0,5)
```

where `@f2` is the handle to the function `f2` created earlier, and

```
>>fminsearch(@f4,[0,0])
```

where `@f4` is the handle to the function `f4` created earlier

## Methods for Calling Functions (continued)

1. As a character string identifying the appropriate function M-file, which is

```
function y = fun1(x)
y = x.^2-4;
```

The function may be called as follows, to compute the zero over the range  $0 \leq x \leq 3$ :

```
>>[x, value] = fzero('fun1',[0, 3])
```

## Methods for Calling Functions (continued)

2. As a function handle to an existing function M-file:

```
>>[x, value] = fzero(@fun1,[0, 3])
```

3. As an “inline” function object:

```
>>fun1 = 'x.^2-4';
```

```
>>fun_inline = inline(fun1);
```

```
>>[x, value] = fzero(fun_inline,[0, 3])
```

## Methods for Calling Functions (continued)

4. As a string expression:

```
>>fun1 = 'x.^2-4';  
>>[x, value] = fzero(fun1,[0, 3])
```

or as

```
>>[x, value] = fzero('x.^2-4',[0, 3])
```



## Methods for Calling Functions (continued)

The function handle method (method 2) is the fastest method, followed by method 1.

In addition to speed improvement, another advantage of using a function handle is that it provides access to *subfunctions*, which are normally not visible outside of their defining M-file.

## Types of User-Defined Functions

The following types of *user-defined* functions can be created in MATLAB.

- The *primary* function is the first function in an M-file and typically contains the main program. Following the primary function in the same file can be any number of subfunctions, which can serve as subroutines to the primary function.

## **Types of User-Defined Functions (continued)**

Usually the primary function is the only function in an M-file that you can call from the MATLAB command line or from another M-file function.

You invoke this function using the name of the M-file in which it is defined.

We normally use the same name for the function and its file, but if the function name differs from the file name, you must use the file name to invoke the function.

## Types of User-Defined Functions (continued)

- *Anonymous* functions enable you to create a simple function without needing to create a M-file for it. You can construct an anonymous function either at the MATLAB command line or from within another function or script. Thus, anonymous functions provide a quick way of making a function from any MATLAB expression without the need to create, name, and save a file.

## **Types of User-Defined Functions (continued)**

- *Subfunctions* are placed in the primary function and are called by the primary function. You can use multiple functions within a single primary function M-file.

## Types of User-Defined Functions (continued)

- *Nested* functions are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the M-file.

The difference between nested functions and subfunctions is that subfunctions normally cannot be accessed outside of their primary function file.

## Types of User-Defined Functions (continued)

- *Overloaded* functions are functions that respond differently to different types of input arguments. They are similar to overloaded functions in any object-oriented language.

For example, an overloaded function can be created to treat integer inputs differently than inputs of class double.

## **Types of User-Defined Functions (continued)**

- *Private* functions enable you to restrict access to a function. They can be called only from an M-file function in the parent directory.



The term *function function* is not a separate function type but refers to any function that accepts another function as an input argument, such as the function `fzero`.

You can pass a function to another function using a function handle.

## Anonymous Functions

Anonymous functions enable you to create a simple function without needing to create an M-file for it. You can construct an anonymous function either at the MATLAB command line or from within another function or script. The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

where `arglist` is a comma-separated list of input arguments to be passed to the function, and `expr` is any single, valid MATLAB expression.

## Anonymous Functions (continued)

To create a simple function called `sq` to calculate the square of a number, type

```
>>sq = @(x) x.^2;
```

To improve readability, you may enclose the expression in parentheses, as `sq = @(x) (x.^2);`. To execute the function, type the name of the function handle, followed by any input arguments enclosed in parentheses. For example,

```
>>sq([5,7])
```

```
ans =
```

```
25
```

```
49
```

## Anonymous Functions (continued)

You might think that this particular anonymous function will not save you any work because typing `sq([5, 7])` requires nine keystrokes, one more than is required to type `[5, 7].^2`.

Here, however, the anonymous function protects you from forgetting to type the period (.) required for array exponentiation.

Anonymous functions are useful, however, for more complicated functions involving numerous keystrokes.

## Anonymous Functions (continued)

You can pass the handle of an anonymous function to other functions. For example, to find the minimum of the polynomial  $4x^2 - 50x + 5$  over the interval  $[-10, 10]$ , you type

```
>>poly1 = @(x) 4*x.^2 - 50*x + 5;  
>>fminbnd(poly1, -10, 10)  
ans =  
    6.2500
```

If you are not going to use that polynomial again, you can omit the handle definition line and type instead

```
>>fminbnd(@(x) 4*x.^2 - 50*x + 5, -10, 10)
```

## Multiple Input Arguments

You can create anonymous functions having more than one input. For example, to define the function

$\sqrt{x^2 + y^2}$ , type

```
>>sqrtsum = @(x,y) sqrt(x.^2 + y.^2);
```

Then type

```
>>sqrtsum(3, 4)
```

```
ans =
```

```
5
```

As another example, consider the function defining a plane,  $z = Ax + By$ . The scalar variables  $A$  and  $B$  must be assigned values before you create the function handle. For example,

```
>>A = 6; B = 4;  
>>plane = @(x,y) A*x + B*y;  
>>z = plane(2,8)  
z =  
44
```

## Calling One Function within Another

One anonymous function can call another to implement function composition. Consider the function  $5 \sin(x^3)$ . It is composed of the functions  $g(y) = 5 \sin(y)$  and  $f(x) = x^3$ . In the following session the function whose handle is `h` calls the functions whose handles are `f` and `g`.

```
>>f = @(x) x.^3;  
>>g = @(x) 5*sin(x) ;  
>>h = @(x) g(f(x)) ;  
>>h(2)  
ans =  
    4.9468
```



## Variables and Anonymous Functions

Variables can appear in anonymous functions in two ways:

- As variables specified in the argument list, as for example `f = @(x) x.^3;`, and

## Variables and Anonymous Functions (continued)

- As variables specified in the body of the expression, as for example with the variables A and B in `plane = @(x,y) A*x + B*y`.

When the function is created MATLAB captures the values of these variables and retains those values for the lifetime of the function handle. If the values of A or B are changed after the handle is created, their values associated with the handle do not change.

This feature has both advantages and disadvantages, so you must keep it in mind.

## Subfunctions

A function M-file may contain more than one user-defined function. The first defined function in the file is called the *primary function*, whose name is the same as the M-file name. All other functions in the file are called *subfunctions*.

Subfunctions are normally “visible” only to the primary function and other subfunctions in the same file; that is, they normally cannot be called by programs or functions outside the file. However, this limitation can be removed with the use of function handles.

## **Subfunctions (continued)**

Create the primary function first with a function definition line and its defining code, and name the file with this function name as usual.

Then create each subfunction with its own function definition line and defining code.

The order of the subfunctions does not matter, but function names must be unique within the M-file.

## Precedence When Calling Functions

The order in which MATLAB checks for functions is very important. When a function is called from within an M-file, MATLAB first checks to see if the function is a built-in function such as `sin`.

If not, it checks to see if it is a *subfunction* in the file, then checks to see if it is a *private* function (which is a function M-file residing in the `private` subdirectory of the calling function).

Then MATLAB checks for a standard M-file on your search path.

## **Precedence When Calling Functions (continued)**

Thus, because MATLAB checks for a subfunction before checking for private and standard M-file functions, you may use subfunctions with the same name as another existing M-file.

This feature allows you to name subfunctions without being concerned about whether another function exists with the same name, so you need not choose long function names to avoid conflict.

This feature also protects you from using another function unintentionally.

The following example shows how the MATLAB M-function `mean` can be superceded by our own definition of the mean, one which gives the root-mean square value.

The function `mean` is a subfunction.

The function `subfun_demo` is the primary function.

```
function y = subfun_demo(a)
y = a - mean(a);
%
function w = mean(x)
w = sqrt(sum(x.^2))/length(x);
```

## Example (continued)

A sample session follows.

```
>>y = subfn_demo([4, -4])  
y =  
    1.1716    -6.8284
```

If we had used the MATLAB M-function `mean`, we would have obtained a different answer; that is,

```
>>a=[4, -4];  
>>b = a - mean(a)  
b =  
     4     -4
```



Thus the use of subfunctions enables you to reduce the number of files that define your functions.

For example, if it were not for the subfunction `mean` in the previous example, we would have had to define a separate M-file for our `mean` function and give it a different name so as not to confuse it with the MATLAB function of the same name.

Subfunctions are normally visible only to the primary function and other subfunctions in the same file.

However, we can use a function handle to allow access to the subfunction from outside the M-file.

## Nested Functions

With MATLAB 7 you can now place the definitions of one or more functions within another function. Functions so defined are said to be *nested* within the main function. You can also nest functions within other nested functions.

## **Nested Functions (continued)**

Like any M-file function, a nested function contains the usual components of an M-file function.

You must, however, always terminate a nested function with an end statement.

In fact, if an M-file contains at least one nested function, you must terminate *all* functions, including subfunctions, in the file with an end statement, whether or not they contain nested functions.

## Example

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB function `fminbnd` to find the minimum point on a parabola. The `parabola` function constructs and returns a function handle `f` for the nested function `p`. This handle gets passed to `fminbnd`.

```
function f = parabola(a, b, c)
f = @p;
    function y = p(x)
        y = a*x^2 + b*x + c;
    end
end
```

## Example (continued)

In the Command window type

```
>>f = parabola(4, -50, 5);  
>>fminbnd(f, -10, 10)  
ans =  
    6.2500
```

Note that the function  $p(x)$  can see the variables  $a$ ,  $b$ , and  $c$  in the calling function's workspace.

Nested functions might seem to be the same as subfunctions, but they are not. Nested functions have two unique properties:

1. A nested function can access the workspaces of all functions inside of which it is nested. So for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the main function.

A variable assigned in a nested function can be read or overwritten by any of the functions containing that function.

**2.** If you construct a function handle for a nested function, the handle not only stores the information needed to access the nested function; it also stores the values of all variables shared between the nested function and those functions that contain it.

This means that these variables persist in memory between calls made by means of the function handle.

## Private Functions

Private functions reside in subdirectories with the special name `private`, and they are visible only to functions in the parent directory.

Assume the directory `rsmith` is on the MATLAB search path. A subdirectory of `rsmith` called `private` may contain functions that only the functions in `rsmith` can call. Because private functions are invisible outside the parent directory `rsmith`, they can use the same names as functions in other directories.



## Private Functions (continued)

Primary functions and subfunctions can be implemented as private functions.

Create a private directory by creating a subdirectory called `private` using the standard procedure for creating a directory or a folder on your computer, but do not place the private directory on your path.

## Section 3.4 Working with Data Files

**Importing Spreadsheet Files:** Some spreadsheet programs store data in the `.wk1` format. You can use the command `M = wk1read('filename')` to import this data into MATLAB and store it in the matrix `M`.

The command `A = xlsread('filename')` imports the Microsoft Excel workbook file `filename.xls` into the array `A`. The command `[A, B] = xlsread('filename')` imports all numeric data into the array `A` and all text data into the cell array `B`.