

MATLAB Programming: A Quick Start


Files that contain MATLAB language code are called *M-files*. M-files can be *functions* that accept arguments and produce output, or they can be *scripts* that execute a series of MATLAB statements. For MATLAB to recognize a file as an M-file, its name must end in . m.

You create M-files using a text editor, then use them as you would any other MATLAB function or command. The process looks like this:

- 1 Create an M-file using a text editor.

```
function c = myfile(a, b)
c = sqrt((a.^2)+(b.^2))
```

- 2 Call the M-file from the command line, or from within another M-file.



```
a = 7.5
b = 3.342
c = myfile(a, b)

c =

    8.2109
```

Kinds of M-Files

There are two kinds of M-files.

Script M-Files	Function M-Files
<ul style="list-style-type: none"> • Do not accept input arguments or return output arguments 	<ul style="list-style-type: none"> • Can accept input arguments and return output arguments
<ul style="list-style-type: none"> • Operate on data in the workspace 	<ul style="list-style-type: none"> • Internal variables are local to the function by default
<ul style="list-style-type: none"> • Useful for automating a series of steps you need to perform many times 	<ul style="list-style-type: none"> • Useful for extending the MATLAB language for your application

What's in an M-File?

This section shows you the basic parts of a function M-file, so you can familiarize yourself with MATLAB programming and get started with some examples.

```

Function definition line → function f = fact(n)
H1 (help 1) line → % FACT Factorial.
Help text → [% FACT(N) returns the factorial of N, usually denoted by N!.
             % Put simply, FACT(N) is PROD(1:N).
Function body → f = prod(1:n);

```

This function has some elements that are common to all MATLAB functions:

- A *function definition line*. This line defines the function name, and the number and order of input and output arguments.
- A *H1 line*. H1 stands for “help 1” line. MATLAB displays the H1 line for a function when you use `lookfor` or request help on an entire directory.
- *Help text*. MATLAB displays the help text entry together with the H1 line when you request help on a specific function.
- The *function body*. This part of the function contains code that performs the actual computations and assigns values to any output arguments.

The “Functions” section coming up provides more detail on each of these parts of a MATLAB function.

Creating M-Files: Accessing Text Editors

M-files are ordinary text files that you create using a text editor. MATLAB provides a built in editor, although you can use any text editor you like.

Note To open the editor on the PC, from the **File** menu choose **New** and then **M-File**.



Another way to edit an M-file is from the MATLAB command line using the `edit` command. For example,

```
edit poof
```

opens the editor on the file `poof.m`. Omitting a filename opens the editor on an untitled file.

You can create the `fact` function shown on the previous page by opening your text editor, entering the lines shown, and saving the text in a file called `fact.m` in your current directory.

Once you’ve created this file, here are some things you can do:

- List the names of the files in your current directory

```
what
```

- List the contents of M-file `fact.m`

```
type fact
```

- Call the `fact` function

```
fact(5)  
ans =
```

120

Scripts

Scripts are the simplest kind of M-file – they have no input or output arguments. They're useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line. Scripts operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations.

Simple Script Example

These statements calculate rho for several trigonometric functions of theta, then create a series of polar plots.

Comment line	→	% An M-file script to produce "flower petal" plots
Computations	→	theta = -pi : 0.01 : pi ;
		rho(1, :) = 2*sin(5*theta) . ^2;
		rho(2, :) = cos(10*theta) . ^3;
		rho(3, :) = sin(theta) . ^2;
Graphical output commands	→	rho(4, :) = 5*cos(3.5*theta) . ^3;
		for i = 1:4
		polar(theta, rho(i, :))
		pause
		end

Try entering these commands in an M-file called `petal s. m`. This file is now a MATLAB script. Typing `petal s` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Return** to move to the next plot. There are no input or output arguments; `petal s` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

Functions

Functions are M-files that accept input arguments and return output arguments. They operate on variables within their own workspace. This is separate from the workspace you access at the MATLAB command prompt.

Simple Function Example

The average function is a simple M-file that calculates the average of the elements in a vector.

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Non-vector input results in an error.
[m, n] = size(x);
if ~( (m == 1) | (n == 1) | (m == 1 & n == 1) )
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

If you would like, try entering these commands in an M-file called `average.m`. The average function accepts a single input argument and returns a single output argument. To call the average function, enter:

```
z = 1:99;

average(z)

ans =
    50
```

Basic Parts of a Function M-File

A function M-file consists of:

- A function definition line
- A H1 line
- Help text
- The function body
- Comments

Function Definition Line

The function definition line informs MATLAB that the M-file contains a function, and specifies the argument calling sequence of the function. The function definition line for the average function is:

```
function y = average(x)
```

input argument
function name
output argument
keyword

All MATLAB functions have a function definition line that follows this pattern.

If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses. Use commas to separate multiple input or output arguments. Here's a more complicated example.

```
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets

```
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as those in the function definition line.

H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, "%." For the average function, the H1 line is:

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types `help function_name` at the MATLAB prompt. Further, the `lookfor` command searches on and displays only the H1 line. Because this line provides important summary

information about the M-file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your M-files by entering text on one or more comment lines, beginning with the line immediately following the H1 line. The help text for the average function is:

```
% AVERAGE(X), where X is a vector, is the mean of vector elements.  
% Nonvector input results in an error.
```

When you type `help function_name`, MATLAB displays the comment lines that appear between the function definition line and the first non-comment (executable or blank) line. The help system ignores any comment lines that appear after this help block.

For example, typing `help sin` results in

```
SIN      Sine.  
         SIN(X) is the sine of the elements of X.
```

Function Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the average function contains a number of simple programming statements.

```
Flow control   → [m, n] = size(x);  
Error message display → if ~(m == 1 | (n == 1) | (m == 1 & n == 1))  
               → error('Input must be a vector')  
               end  
Computation and assignment → y = sum(x)/length(x);
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.  
y = sum(x)           % Use the sum function.
```

The first comment line immediately following the function definition line is considered the H1 line for the function. The H1 line and any comment lines immediately following it constitute the online help entry for the file.

In addition to comment lines, you can insert blank lines anywhere in an M-file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for an M-file.

Help for Directories

You can make help entries for an entire directory by creating a file with the special name `Contents.m` that resides in the directory. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
hel p di rectory_name
```

If a directory does not contain a `Contents.m` file, typing `hel p di rectory_name` displays the first help line (the H1 line) for each M-file in the directory.

Function Names

MATLAB function names have the same constraints as variable names. MATLAB uses the first 31 characters of names. Function names must begin with a letter; the remaining characters can be any combination of letters, numbers, and underscores. Some operating systems may restrict function names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension `.m` appended. For example,

```
average.m
```

If the filename and the function definition line name are different, the internal name is ignored.

Thus, while the function name specified on the function definition line does not have to be the same as the filename, we strongly recommend that you use the same name for both.

How Functions Work

You can call function M-files from either the MATLAB command line or from within other M-files. Be sure to include all necessary arguments, enclosing input arguments in parentheses and output arguments in square brackets.

Function Name Resolution

When MATLAB comes upon a new name, it resolves it into a specific function by following these steps:

- 1 Checks to see if the name is a variable.
- 2 Checks to see if the name is a *subfunction*, a MATLAB function that resides in the same M-file as the calling function. Subfunctions are discussed on page 10-38.
- 3 Checks to see if the name is a *private function*, a MATLAB function that resides in a *private directory*, a directory accessible only to M-files in the directory immediately above it. Private directories are discussed on page 10-39.
- 4 Checks to see if the name is a function on the MATLAB search path. MATLAB uses the first file it encounters with the specified name.

If you duplicate function names, MATLAB executes the one found first using the above rules. It is also possible to overload function names. This uses additional dispatching rules and is discussed in Chapter 14, “Classes and Objects.”

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear

it using the `clear` command, or until you quit MATLAB. Variants of the `clear` command that you can use to clear functions from memory include:

`clear function_name` Remove specified function from workspace.

`clear functions` Remove all compiled M-functions.

`clear all` Remove all variables and functions

Creating P-Code Files

You can save a parsed version of a function or script, called P-code files, for later MATLAB sessions using the `pcode` command. For example,

```
pcode average
```

parses `average.m` and saves the resulting pseudocode to the file named `average.p`. This saves MATLAB from reparsing `average.m` the first time you call it in each session.

MATLAB is very fast at parsing so the `pcode` command rarely makes much of a speed difference.

One situation where `pcode` does provide a speed benefit is for large GUI applications. In this case, many M-files must be parsed before the application becomes visible.

Another situation for `pcode` is when, for proprietary reasons, you want to hide algorithms you've created in your M-file.

How MATLAB Passes Function Arguments

From the programmer's perspective, MATLAB appears to pass all function arguments by value. Actually, however, MATLAB passes by value only those arguments that a function modifies. If a function does not alter an argument but simply uses it in a computation, MATLAB passes the argument by reference to optimize memory use.

Function Workspaces

Each M-file function has an area of memory, separate from MATLAB's base workspace, in which it operates. This area is called the function workspace, with each function having its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can however, define variables as global variables explicitly, allowing more than one workspace context to access them.

Checking the Number of Function Arguments

The `nargin` and `nargout` functions let you determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a, b)
if (nargin == 1)
    c = a.^2;
elseif (nargin == 2)
    c = a + b;
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here's a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by whitespace or some other character. Given one input, the function assumes a default delimiter of whitespace; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists.

```

function [token, remainder] = strtok(string, delimiters)
Function requires at least one input. → if nargin < 1, error('Not enough input arguments.');
```

```

    token = []; remainder = [];
    len = length(string);
    if len == 0
        return
    end
    if (nargin == 1)
        delimiters = [9:13 32]; % White space characters
    end
    i = 1;
    while (any(string(i) == delimiters))
        i = i + 1;
        if (i > len), return, end
    end
    start = i;
    while (~any(string(i) == delimiters))
        i = i + 1;
        if (i > len), break, end
    end
    finish = i - 1;
    token = string(start:finish);
    if (nargout == 2)
        remainder = string(finish + 1:end);
    end
end

```

If one input, use white space delimiter.

Determine where non-delimiter characters begin.

Find where token ends.

For two output arguments, count characters after first delimiter (remainder).

Note strtok is a MATLAB M-file in the strfun directory.

Note that the order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Variable Numbers of Arguments

The `varargin` and `varargout` functions let you pass any number of inputs or return any number of outputs to a function. MATLAB packs all of the specified input or output into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data – one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on.

Here's an example function that accepts any number of two-element vectors and draws a line to connect them.

Cell array indexing

```
function testvar(varargin)
    for i = 1:length(varargin)
        → x(i) = varargin{i}(1);
          y(i) = varargin{i}(2);
    end
    xmin = min(0, min(x));
    ymin = min(0, min(y));
    axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
    plot(x, y)
```

Coded this way, the `testvar` function works with various input lists; for example,

```
testvar([2 3], [1 5], [4 8], [6 5], [4 2], [2 3])
testvar([-1 0], [3 -5], [4 2], [1 1])
```

Unpacking varargin Contents

Because `varargin` contains all the input arguments in a cell array, it's necessary to use cell array indexing to extract the data. For example,

```
y(i) = varargin{i}(2);
```

Cell array indexing has two subscript components:

- The cell indexing expression, in curly braces
- The contents indexing expression(s), in parentheses

In the code above, the indexing expression `{i}` accesses the *i*'th cell of `varargin`. The expression `(2)` represents the second element of the cell contents.

Packing varargout Contents

When allowing any number of output arguments, you must pack all of the output into the varargout cell array. Use nargin to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of x coordinates and the second represents y coordinates. It breaks the array into separate [xi yi] vectors that you can pass into the testvar function on the previous page.

```

function [varargout] = testvar2(arrayin)
for i = 1:nargout
    varargout{i} = arrayin(i,:)
end

```

Cell array assignment →

The assignment statement inside the for loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see “Structures and Cell Arrays” in Chapter 13.

Here’s how to call testvar2.

```

a = {1 2; 3 4; 5 6; 7 8; 9 0};
[p1, p2, p3, p4, p5] = testvar2(a);

```

varargin and varargout in Argument Lists

varargin or varargout must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of varargin and varargout.

```

function [out1, out2] = example1(a, b, varargin)
function [i, j, varargout] = example2(x1, y1, x2, y2, flag)

```

Local and Global Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in M-files:

- You do not need to type or declare variables. Before assigning one variable to another, however, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable if needed, or overwrites its current value if it already exists.
- MATLAB variable names consist of a letter followed by any number of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so A and a are not the same variable.
- MATLAB uses only the first 31 characters of variable names.

Ordinarily, each MATLAB function, defined by an M-file, has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose you want to study the effect of the interaction coefficients, α and β , in the Lotka-Volterra predator-prey model

$$\begin{aligned}\dot{y}_1 &= y_1 - \alpha y_1 y_2 \\ \dot{y}_2 &= -y_2 + \beta y_1 y_2\end{aligned}$$

Create an M-file, `lotka.m`.

```
function yp = lotka(t, y)
%LOTKA Lotka-Volterra predator-prey model.
global ALPHA BETA
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA
ALPHA = 0.01
BETA = 0.02
[t, y] = ode23('lotka', 0, 10, [1; 1]);
plot(t, y)
```

The two global statements make the values assigned to ALPHA and BETA at the command prompt available inside the function defined by `lotka.m`. They can be modified interactively and new solutions obtained without editing any files.

For your MATLAB application to work with global variables:

- Declare the variable as `global` in every function that requires access to it. To enable the workspace to access the global variable, also declare it as `global` from the command line.
- In each function, issue the `global` command before the first occurrence of the variable name. The top of the M-file is recommended.

MATLAB global variable names are typically longer and more descriptive than local variable names, and sometimes consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code and reduce the chance of accidentally redefining a global variable.

Persistent Variables

A variable may be defined as `persistent` so that it does not change value from one call to another. Persistent variables may be used within a function only. Persistent variables remain in memory until the M-file is cleared or changed.

`persistent` is exactly like `global`, except that the variable name is not in the global workspace, and the value is reset if the M-file is changed or cleared.

Three MATLAB functions support the use of persistent variables.

<code>ml lock</code>	Prevents an M-file from being cleared.
<code>munlock</code>	Unlocks an M-file that had previously been locked by <code>ml lock</code> .
<code>mi sl ocked</code>	Indicates whether an M-file can be cleared or not.

Special Values

Several functions return important special values that you can use in your M-files.

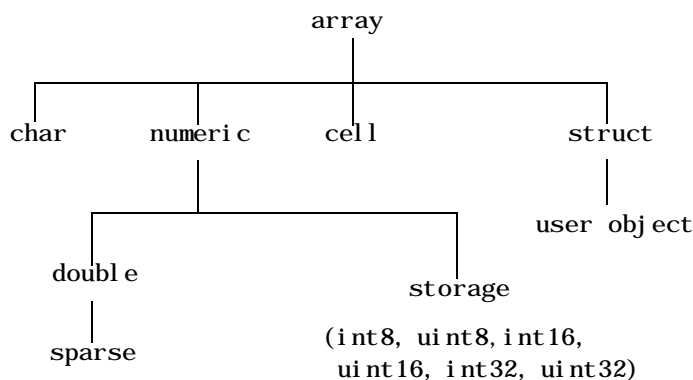
<code>ans</code>	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in <code>ans</code> .
<code>eps</code>	Floating-point relative accuracy. This is the tolerance MATLAB uses in its calculations.
<code>real max</code>	Largest floating-point number your computer can represent.
<code>real mi n</code>	Smallest floating-point number your computer can represent.
<code>pi</code>	3. 1415926535897. . .
<code>i , j</code>	Imaginary unit.
<code>i nf</code>	Infinity. Calculations like $n/0$, where n is any nonzero real value, result in <code>i nf</code> .
<code>NaN</code>	Not-a-Number, an invalid numeric value. Expressions like $0/0$ and $i nf/i nf$ result in a NaN, as do arithmetic operations involving a NaN. $n/0$, where n is complex, also returns NaN.
<code>computer</code>	Computer type.
<code>fl ops</code>	Count of floating-point operations.
<code>versi on</code>	MATLAB version string.

All of these special functions and constants reside in MATLAB's `el mat` directory, and provide online help. Here are several examples that use them in MATLAB expressions.

```
x = 2*pi ;  
A = [3+2i 7-8i ] ;  
tol = 3*eps;
```

Data Types

There are six fundamental data types (classes) in MATLAB, each one a multidimensional array. The six classes are `double`, `char`, `sparse`, `storage`, `cell`, and `struct`. The two-dimensional versions of these arrays are called *matrices* and are where MATLAB gets its name.



You will probably spend most of your time working with only two of these data types: the double precision matrix (`double`) and the character array (`char`) or string. This is because all computations are done in double-precision and most of the functions in MATLAB work with arrays of double-precision numbers or strings.

The other data types are for specialized situations like image processing (`uint8`), sparse matrices (`sparse`), and large scale programming (`cell` and `struct`).

You can't create variables with the types `numeric`, `array`, or `storage`. These *virtual* types serve only to group together types that share some common attributes.

The `storage` data types are for memory efficient storage only. You can apply basic operations such as subscripting and reshaping to these types of arrays but you can't perform any math with them. You must convert such arrays to `double` via the `double` function before doing any math operations.

You can define user classes and objects in MATLAB that are based on the `struct` data type. For more information about creating classes and objects, see "Classes and Objects: An Overview" in Chapter 14.

This table describes the data types in more detail.

Class	Example	Description
array		virtual data type
cell	{17 'hello' eye(2)}	Cell array. Elements of cell arrays contain other arrays. Cell arrays collect related data and information of a dissimilar size together.
char	'Hello'	Character array (each character is 16 bits long). Also referred to as a string.
double	[1 2; 3 4] 5+6i	Double precision numeric array (this is the most common MATLAB variable type).
numeric		virtual data type
sparse	speye(5)	Sparse double precision matrix (2-D only). The sparse matrix stores matrices with only a few nonzero elements in a fraction of the space required for an equivalent full matrix. Sparse matrices invoke special methods especially tailored to solve sparse problems.
storage		virtual data type
struct	a.day = 12; a.color = 'Red'; a.mat = magic(3);	Structure array. Structure arrays have field names. The fields contain other arrays. Like cell arrays, structures collect related data and information together.
uint8	uint8(magic(3))	Unsigned 8 bit integer array. The uint8 array stores integers in the range from 0 to 255 in 1/8 the memory required for a double precision array. No mathematical operations are defined for uint8 arrays.
user object	inline('sin(x)')	User-defined data type.