

MATLAB®

Programming Tips

R2012a

MATLAB®

How to Contact MathWorks



www.mathworks.com
comp.soft-sys.matlab
www.mathworks.com/contact_TS.html

Web
Newsgroup
Technical Support



suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
service@mathworks.com
info@mathworks.com

Product enhancement suggestions
Bug reports
Documentation error reports
Order status, license renewals, passcodes
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Programming Tips

© COPYRIGHT 2002–2012 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

July 2002	Online only	New for MATLAB® 6.5 (Release 13)
June 2004	Online only	Revised for MATLAB® 7.0 (Release 14)
March 2005	Online only	Revised for MATLAB® 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB® 7.1 (Release 14SP3)
September 2007	Online only	Rereleased for Version 7.5 (Release 2007b)
March 2008	Online only	Rereleased for Version 7.6 (Release 2008a)
October 2008	Online only	Rereleased for Version 7.7 (Release 2008b)
March 2009	Online only	Rereleased for Version 7.8 (Release 2009a)
September 2009	Online only	Rereleased for Version 7.9 (Release 2009b)
March 2010	Online only	Rereleased for Version 7.10 (Release 2010a)
September 2010	Online only	Rereleased for Version 7.11 (Release 2010b)
April 2011	Online only	Rereleased for Version 7.12 (Release 2011a)
September 2011	Online only	Rereleased for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)

Programming Tips

1

Introduction	1-2
Command and Function Syntax	1-3
Syntax Help	1-3
Command and Function Syntaxes	1-3
Command Line Continuation	1-3
Completing Commands Using the Tab Key	1-4
Recalling Commands	1-4
Clearing Commands	1-5
Suppressing Output to the Screen	1-5
Help	1-6
Using the Help Browser	1-6
Help on Functions from the Help Browser	1-6
Help on Functions from the Command Window	1-7
Topical Help	1-7
Paged Output	1-8
Writing Your Own Help	1-8
Help for Subfunctions and Private Functions	1-8
Help for Methods and Overloaded Functions	1-9
Development Environment	1-10
Workspace Browser	1-10
Using the Find Utility	1-10
Commenting Out a Block of Code	1-11
Creating Functions from Command History	1-11
Editing Functions in EMACS	1-11
Functions	1-12
Function Structure	1-12
Using Lowercase for Function Names	1-12
Getting a Function's Name and Path	1-13
What Files Does a Function Use?	1-13
Dependent Functions, Built-Ins, Classes	1-14

Function Arguments	1-15
Getting the Input and Output Arguments	1-15
Variable Numbers of Arguments	1-15
String or Numeric Arguments	1-16
Passing Arguments in a Structure	1-16
Passing Arguments in a Cell Array	1-16
Program Development	1-18
Planning the Program	1-18
Using Pseudo-Code	1-18
Selecting the Right Data Structures	1-18
General Coding Practices	1-19
Naming a Function Uniquely	1-19
The Importance of Comments	1-19
Coding in Steps	1-20
Making Modifications in Steps	1-20
Functions with One Calling Function	1-20
Testing the Final Program	1-20
Debugging	1-21
The MATLAB Debug Functions	1-21
More Debug Functions	1-21
The MATLAB Graphical Debugger	1-22
A Quick Way to Examine Variables	1-22
Setting Breakpoints from the Command Line	1-22
Finding Line Numbers to Set Breakpoints	1-23
Stopping Execution on an Error or Warning	1-23
Locating an Error from the Error Message	1-23
Using Warnings to Help Debug	1-23
Making Code Execution Visible	1-24
Debugging Scripts	1-24
Variables	1-25
Rules for Variable Names	1-25
Making Sure Variable Names Are Valid	1-25
Do Not Use Function Names for Variables	1-26
Checking for Reserved Keywords	1-26
Avoid Using i and j for Variables	1-26
Avoid Overwriting Variables in Scripts	1-27
Persistent Variables	1-27
Protecting Persistent Variables	1-27
Global Variables	1-27

Strings	1-29
Creating Strings with Concatenation	1-29
Comparing Methods of Concatenation	1-29
Store Arrays of Strings in a Cell Array	1-30
Converting Between Strings and Cell Arrays	1-30
Search and Replace Using Regular Expressions	1-30
Evaluating Expressions	1-32
Find Alternatives to Using eval	1-32
Assigning to a Series of Variables	1-32
Short-Circuit Logical Operators	1-32
Changing the Counter Variable within a for Loop	1-33
MATLAB Path	1-34
Precedence Rules	1-34
Adding a Folder to the Search Path	1-35
Handles to Functions Not on the Path	1-35
Making Toolbox File Changes Visible to MATLAB	1-36
Making Nontoolbox File Changes Visible to MATLAB	1-36
Change Notification on Windows	1-37
Program Control	1-38
Using break, continue, and return	1-38
Using switch Versus if	1-39
MATLAB case Evaluates Strings	1-39
Multiple Conditions in a case Statement	1-39
Implicit Break in switch-case	1-39
Variable Scope in a switch	1-40
Catching Errors with try-catch	1-40
Nested try-catch Blocks	1-41
Forcing an Early Return from a Function	1-41
Save and Load	1-42
Saving Data from the Workspace	1-42
Loading Data into the Workspace	1-42
Viewing Variables in a MAT-File	1-43
Appending to a MAT-File	1-43
Save and Load on Startup or Quit	1-44
Saving to an ASCII File	1-44
Files and Filenames	1-45
Naming Functions	1-45

Naming Other Files	1-45
Passing Filenames as Arguments	1-46
Passing Filenames to ASCII Files	1-46
Determining Filenames at Run-Time	1-46
Returning the Size of a File	1-46
Input/Output	1-48
Common I/O Functions	1-48
Loading Mixed Format Data	1-48
Reading Files with Different Formats	1-49
Interactive Input into Your Program	1-49
Starting MATLAB	1-50
Getting MATLAB to Start Up Faster	1-50
Operating System Compatibility	1-51
Executing O/S Commands from MATLAB	1-51
Searching Text with grep	1-51
Constructing Paths and Filenames	1-51
Finding the MATLAB Root Folder	1-52
Temporary Directories and Filenames	1-52
For More Information	1-53
Current CSSM	1-53
Archived CSSM	1-53
MATLAB Technical Support	1-53
MATLAB Central	1-53
MATLAB Newsletters (Digest, News & Notes)	1-53
MATLAB Documentation	1-53
MATLAB Index of Examples	1-54

Programming Tips

- “Introduction” on page 1-2
- “Command and Function Syntax” on page 1-3
- “Help” on page 1-6
- “Development Environment” on page 1-10
- “Functions” on page 1-12
- “Function Arguments” on page 1-15
- “Program Development” on page 1-18
- “Debugging” on page 1-21
- “Variables” on page 1-25
- “Strings” on page 1-29
- “Evaluating Expressions” on page 1-32
- “MATLAB Path” on page 1-34
- “Program Control” on page 1-38
- “Save and Load” on page 1-42
- “Files and Filenames” on page 1-45
- “Input/Output” on page 1-48
- “Starting MATLAB” on page 1-50
- “Operating System Compatibility” on page 1-51
- “For More Information” on page 1-53

Introduction

This section is a categorized compilation of tips for the MATLAB® programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic.

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see *Improving Performance and Memory Usage*.

Command and Function Syntax

In this section...

“Syntax Help” on page 1-3

“Command and Function Syntaxes” on page 1-3

“Command Line Continuation” on page 1-3

“Completing Commands Using the Tab Key” on page 1-4

“Recalling Commands” on page 1-4

“Clearing Commands” on page 1-5

“Suppressing Output to the Screen” on page 1-5

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3           % Command syntax
functionname('arg1','arg2','arg3')    % Function syntax
```

For more information: See “Command vs. Function Syntax”.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (...). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
fprintf ('Example %d shows a command coded on %d lines.\n', ...
        exampleNumber, ...
        numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...  
    to another line, resulting in an error.'
```

For more information: See “Continue Long Statements on Multiple Lines”.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;  
set(f, 'papTuT','cT)                % Type this line.  
set(f, 'paperunits','centimeters') % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT  
PaperOrientation  PaperPositionMode  PaperType      Parent  
PaperPosition    PaperSize          PaperUnits
```

For more information: See Tab Completion in the Command Window.

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.
- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.

- Open the Command History window (**Desktop > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Command History Window.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);    % Create matrix A, but do not display it.
```

Help

In this section...

- “Using the Help Browser” on page 1-6
- “Help on Functions from the Help Browser” on page 1-6
- “Help on Functions from the Command Window” on page 1-7
- “Topical Help” on page 1-7
- “Paged Output” on page 1-8
- “Writing Your Own Help” on page 1-8
- “Help for Subfunctions and Private Functions” on page 1-8
- “Help for Methods and Overloaded Functions” on page 1-9

Using the Help Browser



Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

For more information: See “Ways to Get Function Help”.

Help on Functions from the Help Browser

You can find help on a MATLAB function in any of the following ways:

- Click the  **Functions** button in the left pane of the Help browser. This brings you to that part of the Function Reference documentation that is organized by category. To use an alphabetical list to get help on a specific function, click Alphabetical List at the top of that page.
- Click the  **MATLAB** button in the left pane of the Help browser. Look in the upper left corner of the page for links to either Functions: By

Category, or Functions: Alphabetical List and click there for the type of documentation access you prefer.

- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

```
help
```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help category`. For example,

```
help datafun
```

- To get help on a particular function, type `help functionname`. For example,

```
help sortrows
```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
<code>arith</code>	Arithmetic operators
<code>relop</code>	Relational and logical operators
<code>punct</code>	Special character operators
<code>slash</code>	Arithmetic division operators
<code>paren</code>	Parentheses, braces, and bracket operators
<code>precedence</code>	Operator precedence
<code>datatypes</code>	MATLAB classes, their associated functions, and operators that you can overload
<code>lists</code>	Comma separated lists

Topic Name	Description
strings	Character strings
function_handle	Function handles and the @ operator
debug	Debugging functions
java	Using Sun™ Java™ from within the MATLAB software.
changeNotification	Microsoft® Windows® change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type


```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented as MATLAB functions. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subfolder `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...
“Workspace Browser” on page 1-10
“Using the Find Utility” on page 1-10
“Commenting Out a Block of Code” on page 1-11
“Creating Functions from Command History” on page 1-11
“Editing Functions in EMACS” on page 1-11

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **Desktop > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace.

Using the Find Utility

Find any word or phrase in a group of files using the Find utility. Click

Desktop > Current Folder, click the  icon at the top of the **Current Folder** window, and then select **Find Files** from the menu that appears.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like win for windows, will not be found if enclosed in quotes.

For more information: See Finding and Replacing Text in the Current File.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

- 1 Highlight the block of text you would like to comment out.
- 2 Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Adding Comments.

Creating Functions from Command History

If there is part of your current MATLAB session that you would like to add to a function, this is easily done using the Command History window:

- 1 Open this window by selecting **Desktop > Command History**.
- 2 Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
- 3 Right-click once, and select **Create Script** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing Functions in EMACS

If you use Emacs, you can download editing modes for editing MATLAB functions with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

<http://www.mathworks.com/matlabcentral/>

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger.

Functions

In this section...

“Function Structure” on page 1-12

“Using Lowercase for Function Names” on page 1-12

“Getting a Function’s Name and Path” on page 1-13

“What Files Does a Function Use?” on page 1-13

“Dependent Functions, Built-Ins, Classes” on page 1-14

Function Structure

An MATLAB function consists of the components shown here:

```
function [x, y] = myfun(a, b, c)    % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See Basic Parts of a Function.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

Case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes them more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the file name for the function currently being executed, use the following function in your code.

```
mfilename
```

To include the path along with the file name, use:

```
x = mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What Files Does a Function Use?

For a simple display of all functions referenced by a particular function, follow the steps below:

- 1 Type `clear functions` to clear all functions from memory (see Note below).
- 2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.
- 3 Type `inmem` to display all MATLAB function files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

```
[mfiles, mexfiles] = inmem
```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to MATLAB function files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

“Getting the Input and Output Arguments” on page 1-15

“Variable Numbers of Arguments” on page 1-15

“String or Numeric Arguments” on page 1-16

“Passing Arguments in a Structure” on page 1-16

“Passing Arguments in a Cell Array” on page 1-16

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `narginchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
narginchk(2, 4)           % Allow 2 to 4 inputs
nargoutchk(0, 2)         % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
    y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
    1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number 75 as the string, '75'.

```
isnumeric(75)                isnumeric '75'
ans =                          ans =
    1                            0
```

For more information: See Command vs. Function Syntax.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The

advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...
“Planning the Program” on page 1-18
“Using Pseudo-Code” on page 1-18
“Selecting the Right Data Structures” on page 1-18
“General Coding Practices” on page 1-19
“Naming a Function Uniquely” on page 1-19
“The Importance of Comments” on page 1-19
“Coding in Steps” on page 1-20
“Making Modifications in Steps” on page 1-20
“Functions with One Calling Function” on page 1-20
“Testing the Final Program” on page 1-20

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what classes and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in a file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics® property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%-----  
% This function computes the ... <and so on>  
%-----
```

For more information: See `Comments` .

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It is much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It is better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you have changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same file as the calling function, making it a subfunction.

For more information: See “String Comparisons”.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

- “The MATLAB Debug Functions” on page 1-21
- “More Debug Functions” on page 1-21
- “The MATLAB Graphical Debugger” on page 1-22
- “A Quick Way to Examine Variables” on page 1-22
- “Setting Breakpoints from the Command Line” on page 1-22
- “Finding Line Numbers to Set Breakpoints” on page 1-23
- “Stopping Execution on an Error or Warning” on page 1-23
- “Locating an Error from the Error Message” on page 1-23
- “Using Warnings to Help Debug” on page 1-23
- “Making Code Execution Visible” on page 1-24
- “Debugging Scripts” on page 1-24

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging Process and Features.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
echo	Display function or script code as it executes.
disp	Display specified values or messages.
sprintf, fprintf	Display formatted data of different types.

Function	Description
whos	List variables in the workspace.
size	Show array dimensions.
keyboard	Interrupt program execution and allow input from keyboard.
return	Resume execution following a keyboard interruption.
warning	Display specified warning message.
MException	Access information on the cause of an error.
lastwarn	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the open function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging Process and Features.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific file line number.

- Break at the beginning of a specific subfunction.
- Break at the first executable line in a file.
- Break when a warning, or error, is generated.
- Break if any infinite or NaN values are encountered.

For more information: See Setting Breakpoints.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of the file, also numbering each line. To display `delaunay.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `dbstop if warning` to stop execution on any warning and enter debug mode.

For more information: See “Debugging Process and Features”.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the file being executed in its editor and places the cursor at the point of error.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on Warning Control in the MATLAB Programming Fundamentals documentation to find out how to selectively enable warnings.

For more information: See the warning function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...

“Rules for Variable Names” on page 1-25

“Making Sure Variable Names Are Valid” on page 1-25

“Do Not Use Function Names for Variables” on page 1-26

“Checking for Reserved Keywords” on page 1-26

“Avoid Using i and j for Variables” on page 1-26

“Avoid Overwriting Variables in Scripts” on page 1-27

“Persistent Variables” on page 1-27

“Protecting Persistent Variables” on page 1-27

“Global Variables” on page 1-27

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first *N* characters of the name, (where *N* is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first *N* characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

N = `namelengthmax`

N =

63

For more information: See “Variable Names”.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn  
ans =  
    0
```

For more information: See “Variable Names”.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it is a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See “Conflicts with Function Names”.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".  
Error: "End of Input" expected, "case" found.  
Error: Missing operator, comma, or semicolon.  
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See `Scripts`.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See `Persistent Variables`.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a

variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See Global Variables.

Strings

In this section...

- “Creating Strings with Concatenation” on page 1-29
- “Comparing Methods of Concatenation” on page 1-29
- “Store Arrays of Strings in a Cell Array” on page 1-30
- “Converting Between Strings and Cell Arrays” on page 1-30
- “Search and Replace Using Regular Expressions” on page 1-30

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;  
s = ['There are ' int2str(numChars) ' characters here']  
s = sprintf('There are %d characters here', numChars)
```

For more information: See “Creating Character Arrays” and Converting from Numeric to String.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function. However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See Cell Arrays of Strings.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...  
             'Phoenix      '];  
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};  
strcmp(charRecord, cellRecord2)  
ans =  
     0  
     1  
     0
```

For more information: See Converting to a Cell Array of Strings and String Comparisons.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.

For more information: See “Regular Expressions”.

Evaluating Expressions

In this section...

“Find Alternatives to Using `eval`” on page 1-32

“Assigning to a Series of Variables” on page 1-32

“Short-Circuit Logical Operators” on page 1-32

“Changing the Counter Variable within a for Loop” on page 1-33

Find Alternatives to Using `eval`

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that an `eval` statement that contains one or more commands will hide any dependencies on those commands from the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See “Alternatives to the `eval` Function”.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
    phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical AND and OR operators (`&&` and `||`) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators

are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function `myfun` unless the file that defines `myfun` exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See “Short-Circuit Operators”.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable `k` in the example below) in the body of a `for` loop. For example, this loop executes just 10 times, even though `k` is set back to 1 on each iteration.

```
for k = 1:10
    fprintf('Pass %d\n', k)
    k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...
“Precedence Rules” on page 1-34
“Adding a Folder to the Search Path” on page 1-35
“Handles to Functions Not on the Path” on page 1-35
“Making Toolbox File Changes Visible to MATLAB” on page 1-36
“Making Nontoolbox File Changes Visible to MATLAB” on page 1-36
“Change Notification on Windows” on page 1-37

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

- 1 Variable
- 2 Nested function within the current function
- 3 Local function within the current file
- 4 Private function
- 5 Class constructor
- 6 Overloaded method
- 7 Function in the current folder
- 8 Function elsewhere on the path, in order of appearance

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the folder, MATLAB selects the file to use according to the following precedence:

- 1 Built-in function

- 2 MEX-function
- 3 Simulink® model, with file types in this order:
 - a SLX file
 - b MDL file
- 4 P-file (that is, an encoded program file with a .p extension)
- 5 Program file with a .m extension

For more information: See “Function Precedence Order”.

Adding a Folder to the Search Path

To add a folder to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a folder and all of its subfolders in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subfolders to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path folder as the functions. If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

- 1 Create a script in this off-path folder that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
    fhset = @setItem
    fhsort = @sortItems
    fhdel = @deleteItem
```

- 2 Run the script from your Current Folder to create the function handles:

```
run E:/testdir/createFhandles
```

- 3 You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied folders, MATLAB function files (and MEX-files) in the *matlabroot/toolbox* folders are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear functionname`. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot/toolbox* folders. If you add (or remove) files from these folders, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For functions outside of the toolbox folders, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear functionname`. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...
“Using break, continue, and return” on page 1-38
“Using switch Versus if” on page 1-39
“MATLAB case Evaluates Strings” on page 1-39
“Multiple Conditions in a case Statement” on page 1-39
“Implicit Break in switch-case” on page 1-39
“Variable Scope in a switch” on page 1-40
“Catching Errors with try-catch” on page 1-40
“Nested try-catch Blocks” on page 1-41
“Forcing an Early Return from a Function” on page 1-41

Using break, continue, and return

It is easy to confuse the break, continue, and return functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
break	for or while loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
continue	for or while loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
return	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement switch-case statements using if-elseif instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need strcmp to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between switch-case statements in MATLAB and C is that you can specify string values in MATLAB case statements, which you cannot do in C.

```
switch(method)
    case 'linear'
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with switch. The first case below tests for either a linear or bilinear method by using a cell array in the case statement.

```
switch(method)
    case {'linear', 'bilinear'}
        disp('Method is linear or bilinear')
    case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each case with a break statement, code execution falls through to the following case. In MATLAB, case statements do not fall

through; only one case may execute. Using `break` within a case statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is 52, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
    case 52
        disp('result is 52')
    case {52, 78}
        disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one case of any `switch` statement, variables defined within one case are not known in the other cases of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals 2, because `x` is undefined.

```
-- SWITCH-CASE --
switch choice
    case 1
        x = -pi:0.01:pi;
    case 2
        plot(x, sin(x));
end

-- IF-ELSEIF --
if choice == 1
    x = -pi:0.01:pi;
elseif choice == 2
    plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned in `MException` object, `err`) and respond appropriately:


```
try
    X = A * B
catch err
    errmsg = err.message;
    if(strfind(errmsg, 'Inner matrix dimensions'))
        disp('** Wrong dimensions for matrix multiply')
    end
end
```

For more information: See “The try-catch Statement”.

Nested try-catch Blocks

You can also nest try-catch blocks, as shown here. You can use this to attempt to recover from an error caught in the first try section:

```
try
    statement1                                % Try to execute statement1
catch
    try
        statement2                            % Attempt to recover from error
    catch
        disp 'Operation failed'              % Handle the error
    end
end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a return statement in the function at the point where you want to exit. For example,

```
if <done>
    return
end
```

Save and Load

In this section...
“Saving Data from the Workspace” on page 1-42
“Loading Data into the Workspace” on page 1-42
“Viewing Variables in a MAT-File” on page 1-43
“Appending to a MAT-File” on page 1-43
“Save and Load on Startup or Quit” on page 1-44
“Saving to an ASCII File” on page 1-44

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a diary file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, ...).

For more information: See Saving the Current Workspace and “Writing to Text Data Files with Low-Level I/O”.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.
- Read a binary or ASCII file using `load`.

- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, ...).

For more information: See Loading a Saved Workspace and Importing Data and “Importing Data”.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];    B = 12.5;    C = rand(4);  
save savefile;  
A = [6 7 8];  
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to save the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See “Writing to Delimited Data Files”.

Files and Filenames

In this section...

“Naming Functions” on page 1-45

“Naming Other Files” on page 1-45

“Passing Filenames as Arguments” on page 1-46

“Passing Filenames to ASCII Files” on page 1-46

“Determining Filenames at Run-Time” on page 1-46

“Returning the Size of a File” on page 1-46

Naming Functions

A valid name for a MATLAB function file is composed of a string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

`N = namelengthmax`

`N =`

`63`

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for a MATLAB function file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as the MATLAB function files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat           % Command syntax
load('mydata.mat')       % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii    % Command syntax
load('mydata.dat', '-ascii') % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

```
function myfun(datafile)
```

- Prompt for the filename using the `input` function

```
filename = input('Enter name of file: ', 's');
```

- Browse for the file using the `uigetfile` function

```
[filename, pathname] =
    uigetfile('*.mat', 'Select MAT-file');
```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
-- METHOD #1 --
s = dir('myfile.dat');
filesize = s.bytes

-- METHOD #2 --
fid = fopen('myfile.dat');
fseek(fid, 0, 'eof');
filesize = ftell(fid)
fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it is a folder (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...
“Common I/O Functions” on page 1-48
“Loading Mixed Format Data” on page 1-48
“Reading Files with Different Formats” on page 1-49
“Interactive Input into Your Program” on page 1-49

For more information and examples on importing and exporting data, see *MATLAB Data Import and Export*.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textscan`, `dlmread`, `dlmwrite`.

To select and import data from files interactively, select **File > Import Data**.

For more information: See “Supported File Formats”.

Loading Mixed Format Data

To load data that is in mixed formats, use `textscan` instead of `load`. The `textscan` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
fid = fopen('mydata.dat');  
c = textscan(fid, '%s %f %d', 1);  
fclose(fid);
```


returns

```
c =  
    {1x1 cell}    [12.3400]    [45]
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See <http://www.mathworks.com/support/solutions/data/1-17VEB.html> for a more detailed explanation.

For more information: See Toolbox Path Caching in MATLAB.

Operating System Compatibility

In this section...

“Executing O/S Commands from MATLAB” on page 1-51

“Searching Text with grep” on page 1-51

“Constructing Paths and Filenames” on page 1-51

“Finding the MATLAB Root Folder” on page 1-52

“Temporary Directories and Filenames” on page 1-52

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB ! operator.

On Windows, you can add an ampersand (&) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs and the system and dos function reference pages.

Searching Text with grep

grep is a powerful tool for performing text searches in files on UNIX® systems. To grep from within MATLAB, precede the command with an exclamation point (!grep).

For example, to search for the word warning in all MATLAB function files of the Current Folder, ignoring case, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the fullfile function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Folder

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the general toolbox folder:

```
fullfile(matlabroot, 'toolbox', 'matlab', 'general')
```

Temporary Directories and Filenames

If you need to locate the folder on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this folder.

To create a new file in this folder, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file folder, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

For More Information

In this section...
“Current CSSM” on page 1-53
“Archived CSSM” on page 1-53
“MATLAB Technical Support” on page 1-53
“MATLAB Central” on page 1-53
“MATLAB Newsletters (Digest, News & Notes)” on page 1-53
“MATLAB Documentation” on page 1-53
“MATLAB Index of Examples” on page 1-54

Current CSSM

<http://www.mathworks.com/matlabcentral/newsreader>

Archived CSSM

<http://mathforum.org/kb/forum.jspa?forumID=80>

MATLAB Technical Support

<http://www.mathworks.com/support/>

MATLAB Central

<http://www.mathworks.com/matlabcentral/>

MATLAB Newsletters (Digest, News & Notes)

<http://www.mathworks.com/company/newsletters/index.html>

MATLAB Documentation

<http://www.mathworks.com/help/>

MATLAB Index of Examples

http://www.mathworks.com/help/techdoc/demo_example.html