MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

MATLAB® Programming Tips

Version 6

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

MATLAB Programming Tips

(Excerpted from the MATLAB documentation for Programming and Data Types)

Printing History: July 2002 Online only First printing for MATLAB 6.5 (Release 13)

MATLAB Programming Tips

This chapter is a categorized compilation of tips for the MATLAB® programmer. Each item is relatively brief to help you to browse through them and find information that is useful. Many of the tips include a link to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

Command and Function Syntax

This section covers the following topics:

- "Syntax Help"
- "Command and Function Syntaxes"
- "Command Line Continuation"
- "Completing Commands Using the Tab Key"
- "Recalling Commands"
- "Clearing Commands"
- "Suppressing Output to the Screen"

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 "Calling Functions" in the MATLAB "Programming and Data Types" documentation

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.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
          example_number, ...
          number_of_lines)
```

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 “Entering Long Lines” in the MATLAB “Development Environment” documentation

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 MATLAB “Development Environment” documentation

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 (**View -> Command History**) to see all previous commands. Double-click on the one you want to execute.

For more information: See “Recalling Previous Lines,” and “Command History” in the MATLAB “Development Environment” documentation

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

This section covers the following topics:

- "Using the Help Browser"
- "Help on Functions from the Help Browser"
- "Help on Functions from the Command Window"
- "Topical Help"
- "Paged Output"
- "Writing Your Own Help"
- "Help for Subfunctions and Private Functions"
- "Help for Methods and Overloaded Functions"

Using the Help Browser

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

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

Some of the features of the Help Browser are listed below.

Feature	Description
Product Filter	Establish which products to find help on
Contents	Look up topics in the Table of Contents
Index	Look up help using the documentation Index
Search	Search the documentation for one or more words
Demos	See what demos are available; run selected demos
Favorites	Save bookmarks for frequently used Help pages

For more information: See “Finding Information with the Help Browser” in the MATLAB “Development Environment” documentation

Help on Functions from the Help Browser

To find help on any function from the Help Browser, do either of the following:

- Select the **Contents** tab of the Help Browser, open the **Contents** entry labeled `MATLAB`, and find the two subentries shown below. Use one of these to look up the function you want help on.
  - Functions — By Category
  - Functions — Alphabetical List
- 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
`arith`	Arithmetic operators
`relop`	Relational and logical operators
`punct`	Special character operators
`slash`	Arithmetic division operators
`paren`	Parentheses, braces, and bracket operators
`precedence`	Operator precedence
`lists`	Comma separated lists
`strings`	Character strings
`function_handle`	Function handles and the `@` operator
`debug`	Debugging functions
`java`	Using Java from within MATLAB
`fileformats`	A list of readable file formats
`change_notification`	Windows directory 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” in the MATLAB “Development Environment” documentation

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 with M-files. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subdirectory `@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

This section covers the following topics:

- "Workspace Browser"
- "Using the Find and Replace Utility"
- "Commenting Out a Block of Code"
- "Creating M-Files from Command History"
- "Editing M-Files in EMACS"

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 **View -> Workspace** to open the browser.

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

For more information: See "MATLAB Workspace" in the MATLAB "Development Environment" documentation

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace utility. Click on **View -> Current Directory**, and then click on the binoculars icon at the top of the **Current Directory** window.

When entering search text, you don't 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 a String" in the MATLAB "Development Environment" documentation

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 “Commenting” in the MATLAB “Development Environment” documentation

Creating M-Files from Command History

If there is part of your current MATLAB session that you would like to put into an M-file, this is easily done using the Command History window:

1. Open this window by selecting **View -> 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 M-File** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing M-Files in EMACS

If you use Emacs, you can download editing modes for editing M-files 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” in the MATLAB “Development Environment” documentation

M-File Functions

This section covers the following topics:

- "M-File Structure"
- "Using Lowercase for Function Names"
- "Getting a Function's Name and Path"
- "What M-Files Does a Function Use?"
- "Dependent Functions, Built-Ins, Classes"

M-File Structure

An M-File 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 M-file" in the MATLAB "Programming and Data Types" documentation

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.

Specifically, MATLAB requires that you use lowercase when calling any built-in function. For M-file 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 your M-files more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the name of an M-file that is currently being executed, use the following function in your M-file code.

```
mfilename
```

To include the path along with the M-file name, use

```
mfilename('fullpath')
```

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

What M-Files Does a Function Use?

For a simple display of all M-files 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 M-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 M-files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

This section covers the following topics:

- "Getting the Input and Output Arguments"
- "Variable Numbers of Arguments"
- "String or Numeric Arguments"
- "Passing Arguments in a Structure"
- "Passing Arguments in a Cell Array"

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 `nargchk` 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)
disp(nargchk(2, 4, nargin))           % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))       % 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 "Passing Arguments" in the MATLAB "Programming and Data Types" documentation

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 don't have fieldnames 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

This section covers the following topics:

- “Planning the Program”
- “Using Pseudo-Code”
- “Selecting the Right Data Structures”
- “General Coding Practices”
- “Naming a Function Uniquely”
- “The Importance of Comments”
- “Coding in Steps”
- “Making Modifications in Steps”
- “Functions with One Calling Function”
- “Testing the Final Program”

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

For more information: See “Data Types” in the MATLAB “Programming and Data Types” documentation

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 an M-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.
- Don't 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" in the MATLAB "Programming and Data Types" documentation

Coding in Steps

Don't 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's 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, don't make widespread changes all at one time. It's 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've 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 M-file as the calling function, making it a subfunction.

For more information: See "Subfunctions" in the MATLAB "Programming and Data Types" documentation

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

This section covers the following topics:

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

The MATLAB Debug Functions

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

```
help debug
```

For more information: See "Debugging M-Files" in the MATLAB "Development Environment" documentation

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.
`whos`	List variables in the workspace.

Function	Description
`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.
`error`	Display specified error message.
`lasterr`	Return error message that was last issued.
`lasterror`	Return last error message and related information.
`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 M-Files,” and “Using Debugging Features” in the MATLAB “Development Environment” documentation

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

For more information: See “Setting Breakpoints” in the MATLAB “Development Environment” documentation

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 an M-file, also numbering each line. To display `copyfile.m`, use

```
dbtype copyfile
```

To display only lines 70 through 90, use

```
dbtype copyfile 70:90
```

Stopping Execution on an Error or Warning

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

For more information: See “Debug, Backtrace, and Verbose Modes” in the MATLAB “Programming and Data Types” documentation

Locating an Error from the Error Message

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

For more information: See “Types of Errors” in the MATLAB “Development Environment” documentation

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 and Data Types” 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.

For more information: See “Finding Errors” in the MATLAB “Development Environment” documentation

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

This section covers the following topics:

- “Rules for Variable Names”
- “Making Sure Variable Names Are Valid”
- “Don’t Use Function Names for Variables”
- “Checking for Reserved Keywords”
- “Avoid Using i and j for Variables”
- “Avoid Overwriting Variables in Scripts”
- “Persistent Variables”
- “Protecting Persistent Variables”
- “Global Variables”

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 “Naming Variables” in the MATLAB “Programming and Data Types” documentation

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 8th_column
ans =
     0
```

For more information: See "Naming Variables" in the MATLAB "Programming and Data Types" documentation

Don't 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 define a variable with a function name, you won't be able to call that function until you either `clear` the variable from memory, (unless you execute the function using `builtin`).

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

```
which -all <name>
```

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" in the MATLAB "Programming and Data Types" documentation

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" in the MATLAB "Programming and Data Types" documentation

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 M-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" in the MATLAB "Programming and Data Types" documentation

Strings

This section covers the following topics:

- "Creating Strings with Concatenation"
- "Comparing Methods of Concatenation"
- "Store Arrays of Strings in a Cell Array"
- "Search and Replace Using Regular Expressions"
- "Converting Between Strings and Cell Arrays"

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:

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

For more information: See "Creating Character Arrays," and "Numeric/String Conversion" in the MATLAB "Programming and Data Types" documentation

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:

```
charRecord  = ['Allison Jones'; 'Development  '; 'Phoenix      '];
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See "Cell Arrays of Strings" in the MATLAB "Programming and Data Types" documentation

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" in the MATLAB "Programming and Data Types" documentation

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” in the MATLAB Programming and Data Types documentation

Evaluating Expressions

This section covers the following topics:

- “Find Alternatives to Using eval”
- “Assigning to a Series of Variables”
- “Short-Circuit Logical Operators”
- “Changing the Counter Variable within a for Loop”

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 `eval` statements cannot always be translated into C or C++ code by 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 MATLAB Technical Note 1103, “What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?” at URL `http://www.mathworks.com/support/tech-notes/1100/1103.shtml`

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 its M-file exists on the current path.

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

For more information: See "Short-Circuit Operators" in the MATLAB "Programming and Data Types" documentation

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
   disp(sprintf('Pass %d', 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

This section covers the following topics:

- “Precedence Rules”
- “File Precedence”
- “Adding a Directory to the Search Path”
- “Handles to Functions Not on the Path”
- “Making Toolbox File Changes Visible to MATLAB”
- “Making Nontoolbox File Changes Visible to MATLAB”
- “Change Notification on Windows”

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. Built-in function
3. Subfunction
4. Private function
5. Class constructor
6. Overloaded method
7. M-file in the current directory
8. M-file on the path

If the name is found to be an M-File on the path (No. 8 in the list), and there is more than one M-File on the path with the same name, MATLAB uses the one in the directory that is closest to the beginning of the path string.

For more information: See “Function Precedence Order” in the MATLAB “Programming and Data Types” documentation

File Precedence

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 directory, MATLAB selects the file to use according to the following precedence:

1. MEX-file
2. MDL-file (Simulink model)
3. P-Code file
4. M-file

For more information: See “Selecting Methods from Multiple Implementation Types” in the MATLAB “Programming and Data Types” documentation

Adding a Directory to the Search Path

To add a directory 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 directory and all of its subdirectories 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 subdirectories to the MATLAB path:

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

For more information: See “Search Path” in the MATLAB “Development Environment” documentation

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 directory 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 directory that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/create_fhandles.m
   fhset = @set_items
   fhsort = @sort_items
   fhdel = @delete_item
```

2 Run the script from your current directory to create the function handles:

```
run E:/testdir/create_fhandles
```

3 You can now execute one of the functions through its handle using `feval`.

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

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied directories, M-files (and MEX-files) in the `$MATLAB/toolbox` directories 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 `$MATLAB/toolbox` directories. If you add (or remove) files from these directories, 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 M-files outside of the toolbox directories, 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 change_notification
help change_notification_advanced
```

Program Control

This section covers the following topics:

- "Using break, continue, and return"
- "Using switch Versus if"
- "MATLAB case Evaluates Strings"
- "Multiple Conditions in a case Statement"
- "Implicit Break in switch/case"
- "Variable Scope in a switch"
- "Catching Errors with try/catch"
- "Nested try/catch Blocks"
- "Forcing an Early Return from a Function"

Using break, continue, and return

It's 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 don't 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 `case`s of that `switch` statement. The same holds true for `if/ifelse` statements.

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

```
      SWITCH/CASE                          IF/ELSEIF
switch choice
   case 1                          if choice == 1
      x = -pi:0.01:pi;                x = -pi:0.01:pi;
   case 2                          elseif choice == 2
      plot(x, sin(x));                plot(x, sin(x));
end                                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 by lasterr) and respond appropriately.

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

For more information: See “Checking for Errors with try-catch” in the MATLAB “Programming and Data Types” documentation

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
```

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

This section covers the following topics:

- "Saving Data from the Workspace"
- "Loading Data into the Workspace"
- "Viewing Variables in a MAT-File"
- "Appending to a MAT-File"
- "Save and Load on Startup or Quit"
- "Saving to an ASCII File"

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 "Using the diary Command to Export Data," "Saving the Current Workspace," and "Using Low-Level File I/O Functions" in the MATLAB "Development Environment" documentation

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 “Using Low-Level File I/O Functions” in the MATLAB “Development Environment” documentation

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.

```
mydata_variables = 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 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 "Exporting ASCII Data" in the MATLAB "Development Environment" documentation

Files and Filenames

This section covers the following topics:

- "Naming M-files"
- "Naming Other Files"
- "Passing Filenames as Arguments"
- "Passing Filenames to ASCII Files"
- "Determining Filenames at Runtime"
- "Returning the Size of a File"

Naming M-files

M-file names must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed M-file name length (returned by the function `namelengthmax`).

```
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 an M-file.

```
isvarname <M-file name>
```

Naming Other Files

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

Depending on your operating system, you may be able to include certain non-alphanumeric 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.

```
saved_data = 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 Runtime

There are several ways that your function code can work on specific files without you having to hard-code 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 “Obtaining User Input” in the MATLAB “Programming and Data Types” documentation, and 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                               METHOD #2
s = dir('myfile.dat');               fid = fopen('myfile.dat');
filesize = s.bytes                   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's a directory (`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

This section covers the following topics:

- “File I/O Function Overview”
- “Common I/O Functions”
- “Readable File Formats”
- “Using the Import Wizard”
- “Loading Mixed Format Data”
- “Reading Files with Different Formats”
- “Reading ASCII Data into a Cell Array”
- “Interactive Input into Your Program”

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online Functions by Category reference. In the Help Browser **Contents**, click on **MATLAB -> Functions — By Category**, and then click on **File I/O**.

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 `textread`, `dlmread`, `dlmwrite`. Functions for I/O to text files with comma-separated values are `csvread`, `csvwrite`.

For more information: See “Text Files” in the MATLAB “Functions — By Category” reference documentation

Readable File Formats

Type `doc fileformats` to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel files) is to use the MATLAB Import Wizard. Open the Import Wizard by typing `uiimport <filename>` or by selecting **File -> Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Using the Import Wizard with Text Data," and "Using the Import Wizard with Binary Data Files" in the MATLAB Development Environment documentation

Loading Mixed Format Data

To load data that is in mixed formats, use `textread` instead of `load`. The `textread` 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:

```
 [names, x, y] = textread('mydata.dat', '%s %f %d', 1)
```

returns

```
names =
    'Sally'
x =
  12.34000000000000
y =
    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.

Reading ASCII Data into a Cell Array

A common technique used to read an ASCII data file into a cell array is

```
[a,b,c,d] = textread('data.txt', '%s %s %s %s');
mydata = cellstr([a b c d]);
```

For more information: See the `textread` and `cellstr` function reference pages

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.

For more information: See “Obtaining User Input” in the MATLAB “Programming and Data Types” documentation

Managing Memory

This section covers the following topics:

- “Useful Functions for Managing Memory”
- “Compressing Data in Memory”
- “Clearing Unused Variables from Memory”
- “Conserving Memory with Large Amounts of Data”
- “Matrix Manipulation with Sparse Matrices”
- “Structure of Arrays Rather Than Array of Structures”
- “Preallocating Is Better Than Growing an Array”
- “Use repmat When You Need to Grow Arrays”
- “Preallocating a Nondouble Matrix”
- “System-Specific Ways to Use Less Memory”
- “Out of Memory Errors on UNIX”
- “Reclaiming Memory on UNIX”
- “Out of Memory Errors and the JVM”
- “Memory Requirements for Cell Arrays”
- “Memory Required for Cell Arrays and Structures”
- “Preallocating Cell Arrays to Save Memory”

For more information: See “Making Efficient Use of Memory” in the MATLAB “Programming and Data Types” documentation

Useful Functions for Managing Memory

Several functions that are useful in managing memory are listed below.

Function	Description
`clear`	Remove variables from memory.
`pack`	Save existing variables to disk, and then reload them contiguously.
`save`	Selectively store variables to disk.

Function	Description
`load`	Reload a data file saved with the `save` function.
`quit`	Exit MATLAB and return all allocated memory to the system.

Compressing Data in Memory

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. If you get the `Out of Memory` message from MATLAB, the `pack` function may be able to compress some of your data in memory, thus freeing up larger contiguous blocks.

For more information: See "Ways to Conserve Memory" in the MATLAB "Programming and Data Types" documentation

Clearing Unused Variables from Memory

If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you no are longer using from memory. Use `clear` to do this.

Conserving Memory with Large Amounts of Data

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, `clear` the variable from memory and continue with the data generation.

Matrix Manipulation with Sparse Matrices

Matrices with values that are mostly zero are best stored in `sparse` format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse using the `sparse` function.

For more information: See "Converting Full Matrices into Sparse" in the MATLAB "Programming and Data Types" documentation, and the `sparse` function reference page

Structure of Arrays Rather Than Array of Structures

If your MATLAB application needs to store a large amount of data, and the definition of that data lends itself to being stored in either a structure of arrays or an array of structures, the former is preferable. A structure of arrays requires significantly less memory than an array of structures, and also has a corresponding speed benefit.

Preallocating Is Better Than Growing an Array

`for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect memory usage and performance. On each iteration, MATLAB has to allocate more memory for the growing matrix and also move data in memory whenever it cannot allocate a contiguous block. This can also result in fragmentation or in used memory not being returned to the operating system.

To avoid this, preallocate a block of memory large enough to hold the matrix at its final size. For example,

```
zeros(10000, 10000)        % Preallocate a 10000 x 10000 matrix
```

For more information: See “Preallocating Arrays” in the MATLAB “Programming and Data Types” documentation

Use repmat When You Need to Grow Arrays

If you do need to grow an array, see if you can do it using the `repmat` function. `repmat` gets you a contiguous block of memory for your expanding array.

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, it is more memory efficient and sometimes faster to use the `repmat` function for this.

The statement below uses `zeros` to preallocate a 100-by-100 matrix of `uint8`. It does this by first creating a full matrix of `doubles`, and then converting the matrix to `uint8`. This costs time and uses memory unnecessarily.

```
A = int8(zeros(100));
```

Using `repmat`, you create only one `double`, thus reducing your memory needs.

```
A = repmat(int8(0), 100, 100);
```

For more information: See “Preallocating Arrays” in the MATLAB “Programming and Data Types” documentation

System-Specific Ways to Use Less Memory

If you run out of memory often, you can allocate your larger matrices earlier in the MATLAB session and also use these system-specific tips:

- UNIX: Ask your system manager to increase your swap space.
- Windows: Increase virtual memory using the Windows Control Panel.

For UNIX systems, we recommend that your machine be configured with twice as much swap space as you have RAM. The UNIX command, `pstat -s`, lets you know how much swap space you have.

For more information: See “Platform-Specific Memory Topics” in the MATLAB “Programming and Data Types” documentation

Out of Memory Errors on UNIX

On UNIX systems, you may get `Out of Memory` errors when executing an operating system command from within MATLAB (using the shell escape (`!`) operator). This is because, when a process shells out to a subprocess, UNIX allocates as much memory for the subprocess as has been allocated for the parent process.

For more information: See “Running External Programs” in the MATLAB “Development Environment” documentation

Reclaiming Memory on UNIX

On UNIX systems, MATLAB does not return memory to the operating system even after variables have been cleared. This is due to the manner in which UNIX manages memory. UNIX does not accept memory back from a program until the program has terminated. So, the amount of memory used in a MATLAB session is not returned to the operating system until you exit MATLAB.

To free up the memory used in your MATLAB session, save your workspace variables, exit MATLAB, and then load your variables back in.

Out of Memory Errors and the JVM

You are more likely to encounter `Out of Memory` errors when using MATLAB 6.0 or greater, due to the use of the Java virtual machine within MATLAB. On UNIX systems, you can reduce the amount of memory MATLAB uses by starting MATLAB with the `-nojvm` option. When you use this option, you cannot use the desktop or any of the MATLAB tools that require Java.

Type the following at the UNIX prompt:

```
matlab -nojvm
```

For more information: See “Running MATLAB without the Java Virtual Machine” in the MATLAB “Programming and Data Types” documentation

Memory Requirements for Cell Arrays

The memory requirement for an M-by-N cell array containing the same type of data is

```
memreq = M*N*(100 + (num. of elements in cell)*bytes per element)
```

So a 5-by-6 cell array that contains a 10-by-10 real matrix in each cell takes up 27,000 bytes.

Memory Required for Cell Arrays and Structures

Contiguous memory is not required for an entire cell array or structure array. Since each of these is actually an array of pointers to other arrays, the memory for each array needs to be contiguous, but the entire memory collection does not need to be.

Preallocating Cell Arrays to Save Memory

Preallocation of cell arrays is recommended if you know the data type of your cells’ components. This makes it unnecessary for MATLAB to grow the cell array each time you assign values.

For example, to preallocate a 1000-by-200 cell array of empty arrays, use

```
A = cell(1000, 200);
```

For more information: See “Preallocating Arrays” in the MATLAB “Programming and Data Types” documentation

Optimizing for Speed

This section covers the following topics:

- "Finding Bottlenecks with the Profiler"
- "Measuring Execution Time with tic and toc"
- "Measuring Smaller Programs"
- "Speeding Up MATLAB Performance"
- "Vectorizing Your Code"
- "Functions Used in Vectorizing"
- "Coding Loops in a MEX-File for Speed"
- "Preallocate to Improve Performance"
- "Functions Are Faster Than Scripts"
- "Avoid Large Background Processes"
- "Load and Save Are Faster Than File I/O Functions"
- "Conserving Both Time and Memory"

For more information: See "Maximizing MATLAB Performance" in the MATLAB "Programming and Data Types" documentation

Finding Bottlenecks with the Profiler

A good first step to speeding up your programs is to use the MATLAB Profiler to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

To start the Profiler, type `profile viewer` or select **View -> Profiler** in the MATLAB desktop.

For more information: See "Measuring Performance" in the MATLAB "Programming and Data Types" documentation, and the `profile` function reference page

Measuring Execution Time with tic and toc

The functions `tic` and `toc` help you to measure the execution time of a piece of code. You may want to test different algorithms to see how they compare in execution time.

Use `tic` and `toc` as shown here.

```
tic
   - run the program section to be timed -
toc
```

For more information: See “Techniques for Improving Performance” in the MATLAB “Programming and Data Types” documentation, and the `tic`/`toc` function reference page

Measuring Smaller Programs

Programs can sometimes run too fast to get useful data from `tic` and `toc`. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run.

```
tic;
   for k=1:100
      - run the program -
   end;
toc
```

Speeding Up MATLAB Performance

MATLAB internally processes much of the code in M-file functions and scripts to run at an accelerated speed. The effects of performance acceleration can be particularly noticeable when you use `for` loops and, in some cases, the accelerated loops run as fast as vectorized code.

Implementing performance acceleration in MATLAB is a work in progress, and not all components of the MATLAB language can be accelerated at this time. Read “Performance Acceleration” in the MATLAB “Programming and Data Types” documentation to see how you can make the best use of this feature.

Vectorizing Your Code

It’s best to avoid the use of `for` loops in programs that cannot benefit from performance acceleration. Unlike compiled languages, MATLAB interprets each line in a `for` loop on each iteration of the loop. Most loops can be eliminated by performing an equivalent operation using MATLAB vectors instead. In many cases, this is fairly easy to do and is well worth the effort required to convert from using a loop.

For more information: See “Vectorizing Loops” in the MATLAB “Programming and Data Types” documentation

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are

`all`	`end`	`logical`	`repmat`	`squeeze`
`any`	`find`	`ndgrid`	`reshape`	`sub2ind`
`cumsum`	`ind2sub`	`permute`	`shiftdim`	`sum`
`diff`	`ipermute`	`prod`	`sort`	

Coding Loops in a MEX-File for Speed

If there are instances where you must use a `for` loop, consider coding the loop in a MEX-file. In this way, the loop executes much more quickly since the instructions in the loop do not have to be interpreted each time they execute.

For more information: See “Introducing MEX-Files” in the External Interfaces/API documentation

Preallocate to Improve Performance

MATLAB allows you to increase the size of an existing matrix incrementally, usually within a `for` or `while` loop. However, this can slow a program down considerably, as MATLAB must continually allocate more memory for the growing matrix and also move data in memory whenever a contiguous block cannot be allocated.

It is much faster to preallocate a block of memory large enough to hold the matrix at its final size. For example, to preallocate a 10000-by-10000 matrix, use

```
zeros(10000, 10000)        % Preallocate a 10000 x 10000 matrix
```

For more information: See “Preallocating Arrays” in the MATLAB “Programming and Data Types” documentation

Functions Are Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script. Every time a script is used in MATLAB, it is loaded into memory and evaluated one line at a time. Functions, on the other hand, are compiled into pseudo-code and loaded into memory once. Therefore, additional calls to the function are faster.

For more information: See "Techniques for Improving Performance" in the MATLAB "Programming and Data Types" documentation

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the MATLAB file I/O routines, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Conserving Both Time and Memory

The following tips have already been mentioned under "Managing Memory" on page 50, but apply to optimizing for speed as well:

- "Conserving Memory with Large Amounts of Data" on page 51
- "Matrix Manipulation with Sparse Matrices" on page 51
- "Structure of Arrays Rather Than Array of Structures" on page 52
- "Preallocating Is Better Than Growing an Array" on page 52
- "Preallocating a Nondouble Matrix" on page 52

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/25731.shtml` for a more detailed explanation.

For more information: See "Reduced Startup Time with Toolbox Path Caching" in the MATLAB "Development Environment" documentation

Operating System Compatibility

This section covers the following topics:

- "Executing O/S Commands from MATLAB"
- "Searching Text with grep"
- "Constructing Path and File Names"
- "Finding the MATLAB Root Directory"
- "Temporary Directories and Filenames"

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" in the MATLAB "Development Environment" documentation, 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`, ignoring case, in all M-Files of the current directory, you would use

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

Constructing Path and File Names

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 Directory

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox directories 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 directory:

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

Temporary Directories and Filenames

If you need to locate the directory 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 directory.

To create a new file in this directory, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file directory, 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');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop
- Stateflow

For more information: See “Running Demonstrations” in the MATLAB “Development Environment” documentation, and the `demo` function reference page

For More Information

Current CSSM

`news:comp.soft-sys.matlab`

Archived CSSM

`http://mathforum.org/epigone/comp.soft-sys.matlab/`

MATLAB Technical Support

`http://www.mathworks.com/support/`

Search Selected Online Resources

`http://www.mathworks.com/search/`

Tech Notes

`http://www.mathworks.com/support/tech-notes/1100/index.shtml`

MATLAB Central

`http://www.mathworks.com/matlabcentral/`

MATLAB Tips

`http://www.mathworks.com/products/gallery/tips/`

MATLAB Digest

`http://www.mathworks.com/company/digest/index.shtml`

MATLAB News & Notes

`http://www.mathworks.com/company/newsletter/index.shtml`

MATLAB Documentation

`http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml`

MATLAB Index of Examples

```
http://www.mathworks.com/access/helpdesk/help/techdoc/
   demo_example.shtml
```