MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

Using MATLAB

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.

Printing History: December 1996   First printing   First Printing for MATLAB 5
June 1997   Second printing   Revised for MATLAB 5.1
January 1998   Third printing   Revised for MATLAB 5.2
January 1999   Fourth printing   Revised for MATLAB 5.3 (Release 11)
November 2000   Fifth printing   Revised for MATLAB 6 (Release 12)
June 2001   Online only   Revised for MATLAB 6.1 (Release 12.1)
July 2002   Online only   Revised for MATLAB 6.5 (Release 13)

Contents

Development Environment

Running MATLAB Functions

3

Getting Help

4

Workspace, Search Path, and File Operations

5

Importing and Exporting Data

6

7

Editing and Debugging M-Files

8
Interfacing with Source Control Systems

9
Using Notebook

Mathematics

Polynomials and Interpolation

11

Function Functions

13

Differential Equations

14

Sparse Matrices

15

Programming and Data Types

Character Arrays (Strings)

17

Multidimensional Arrays

18

Structures and Cell Arrays

19

21

Maximizing MATLAB Performance

22

MATLAB Programming Tips

23

External Interfaces and
the MATLAB API

A

Development Environment

The MATLAB® development environment is a set of tools to help you use MATLAB functions and files. Many of these tools are graphical user interfaces.

"Starting and Quitting MATLAB" on page 1-1	Run MATLAB, including startup and shutdown options.
"Using the Desktop" on page 2-1	Use the MATLAB desktop to manage common tools. Configure the desktop, set preferences, and use other features.
"Running MATLAB Functions" on page 3-1	Work with functions in the Command Window and Command History window.
"Getting Help" on page 4-1	Get help using the Help browser, help functions, printed documentation, demos, and other methods.
"Workspace, Search Path, and File Operations" on page 5-1	Use the Workspace browser, Array Editor, search path tool, Current Directory browser, and equivalent functions.
"Importing and Exporting Data" on page 6-1	Bring data created by other applications into the MATLAB workspace using the Import Wizard. Package MATLAB workspace variables for use by other applications.
"Editing and Debugging M-Files" on page 7-1	Use the MATLAB graphical Editor/Debugger and debugging functions to create and change M-files.
"Interfacing with Source Control Systems" on page 8-1	Access your source control system from within MATLAB, Simulink®, and Stateflow®.
"Using Notebook" on page 9-1	Access the numeric computation and visualization software of MATLAB from within a word processing environment (Microsoft Word).

Starting and Quitting MATLAB

Starting MATLAB

Instructions for starting MATLAB depend on your platform. For a list of supported platforms, see the system requirements in the installation documentation for your platform, or the Products section of the MathWorks Web site, `http://www.mathworks.com`.

Starting MATLAB on Windows Platforms

To start MATLAB on a Microsoft Windows platform, double-click the MATLAB shortcut icon 🦗 on your Windows desktop (or single-click for Active Desktop). The shortcut was automatically created by the installer in the installation directory.

If you start MATLAB from a DOS window, type `matlab` at the DOS prompt.

After starting MATLAB, the MATLAB desktop opens—see Chapter 2, "Using the Desktop." All of the desktop components that were open when you last shut down MATLAB will be opened on startup.

If you use a virus scanner, your settings may slow down MATLAB startup. For example, if you use McAfee VirusScan and startup seems slow, you can try setting the McAfee options to scan program files only and see if MATLAB startup becomes faster.

Starting MATLAB on UNIX Platforms

To start MATLAB on a UNIX platform, type `matlab` at the operating system prompt.

After starting MATLAB, the MATLAB desktop opens—see Chapter 2, "Using the Desktop." On UNIX platforms, if the `DISPLAY` environment variable is not set or is invalid, the desktop will not display. On some UNIX platforms, the desktop is not supported – see the Release Notes for details.

Startup Directory for MATLAB

The startup directory is the current directory in MATLAB when it first starts, and depends on your platform and installation. You can specify a different startup directory.

Startup Directory on Windows Platforms

On Windows platforms, when you installed MATLAB, the default startup directory was set to $matlabroot/work, where $matlabroot is the directory where MATLAB files are installed.

Startup Directory on UNIX Platforms

On UNIX platforms, the initial current directory is the directory you are in on your UNIX file system when you start MATLAB.

Changing the Startup Directory

You can start MATLAB in a different directory from the default. The directory you specify will be the current working directory when MATLAB starts. To change the startup directory:

1 Create a startup.m file—see "Using the Startup File for MATLAB, startup.m" on page 1-4.

2 In the startup.m file, include the cd function to change to the new directory.

3 Put the startup.m file in the current startup directory.

For Windows Platforms Only. For Windows platforms, this is another way to change the startup directory:

1 Right-click the MATLAB shortcut icon and select **Properties** from the context menu.

The **Properties** dialog box for matlab.exe opens to the **Shortcut** page.

2 Enter the new startup directory in the **Start in** field and click **OK**.

The next time you start MATLAB using that shortcut icon, the current directory will be the one you specified in step 2.

You can make multiple shortcuts to start MATLAB, each with its own startup directory, and with each startup directory having different startup options.

Startup Options

You can define startup options for MATLAB that instruct MATLAB to perform certain operations when you launch it. There are two ways to specify startup options for MATLAB:

- "Using the Startup File for MATLAB, startup.m" on page 1-4
- "Adding Startup Options for Windows Platforms" on page 1-4 or "Adding Startup Options for UNIX Platforms" on page 1-6

Using the Startup File for MATLAB, startup.m

At startup, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. The file `matlabrc.m`, which is in the `local` directory, is reserved for use by The MathWorks, and by the system manager on multiuser systems.

The file `startup.m` is for you to specify startup options. For example, you can modify the default search path, predefine variables in your workspace, or define Handle Graphics defaults. Creating a `startup.m` file with the line

```
addpath /home/me/mytools
cd /home/me/mytools
```

adds `/home/me/mytools` to your default search path and makes `mytools` the current directory upon startup.

On Windows platforms, place the `startup.m` file in `$matlabroot/toolbox/local`, where `$matlabroot` is the directory in which MATLAB is installed.

On UNIX workstations, place the `startup.m` file in the directory named `matlab` off your home directory, for example, `~/matlab`.

Adding Startup Options for Windows Platforms

You can add selected startup options (also called command flags) to the target path for your Windows shortcut for MATLAB, or include the option if you start MATLAB from a DOS window. To do so:

1 Right-click the MATLAB shortcut icon ![icon] and select **Properties** from the context menu.

   The **Properties** dialog box for `matlab.exe` opens to the **Shortcut** panel.

2 In the **Target** field, after the target path for matlab.exe, add one or more of the startup options listed here. See the example following step 3.

Option	Description
/automation	Start MATLAB as an automation server, minimized and without the MATLAB splash screen. For more information, see "COM and DDE Support" in the External Interfaces documentation.
/c licensefile	Set LM_LICENSE_FILE to licensefile. It can have the form port@host.
/logfile logfilename	Automatically write output from MATLAB to the specified log file.
/minimize	Start MATLAB minimized and without the MATLAB splash screen.
/nosplash	Start MATLAB without displaying the MATLAB splash screen.
/r M_file	Automatically run the specified M-file immediately after MATLAB starts. This is also referred to as calling MATLAB in batch mode.
/regserver	Modify the Windows registry with the appropriate COM entries for MATLAB. For more information, see "COM and DDE Support" in the External Interfaces documentation.
/unregserver	Modify the Windows registry to remove the COM entries for MATLAB. Use this option to reset the registry. For more information, see "COM and DDE Support" in the External Interfaces documentation.

You can use a hyphen (-) instead of a slash (/), for example, -nosplash.

3 Click **OK**.

Example—Setting Startup Options to Automatically Run an M-File. To start MATLAB and automatically run the file results.m, set the **Target** field in your Windows shortcut to

```
D:/matlabr13/bin/win32/matlab.exe /r results
```

Startup Options If You Run MATLAB from a DOS Window. If you run MATLAB from a DOS window, include the startup options listed in the preceding table after the matlab startup function.

For example, to start MATLAB and automatically run the file results.m, type

```
matlab /r results
```

Adding Startup Options for UNIX Platforms

Include startup options (also called command flags) after the matlab startup command.

For example, to start MATLAB without the splash screen, type

```
matlab -nosplash
```

For more details, see the matlab reference page.

Option	Description
-arch	Run MATLAB assuming architecture arch.
-arch/ext	Run the version of MATLAB with the extension ext, if it exists, assuming architecture arch.
-c licensefile	Set LM_LICENSE_FILE to licensefile. It can have the form port@host.
-check_malloc	Set the MATLAB_MEM_MGR environment variable to 'debug'. This starts MATLAB memory integrity checking.
-**D**debugger [options]	Start MATLAB with the specified debugger.
-debug	Turn on MATLAB internal debugging.

Option	Description (Continued)
-display Xserver	Send X commands to Xserver.
-ext	Run the version of MATLAB with the extension ext, if it exists.
-h or -help	Display startup options (without starting MATLAB).
-logfile log	Make a copy of any output to the Command Window in the file log. This includes all crash reports.
-mwvisual visualid	Specify the default X visual to use for figure windows.
-n	Display final values of environment variables and arguments passed to MATLAB (without starting MATLAB).

Option	Description (Continued)
`-nodesktop`	Start MATLAB without bringing up the MATLAB desktop. Use this option to run without an X-window, for example, in VT100 mode, or in batch processing mode. Note that if you pipe to MATLAB using the > constructor, the nodesktop option is used automatically.
	With nodesktop, you can still use most development environment tools by starting them via a function. Specifically use
	• edit to open the Editor/Debugger
	• helpbrowser to open the Help browser
	• filebrowser to open the Current Directory browser
	• workspace to open the Workspace browser
	• openvar to open the Array Editor
	• profile viewer to open the Profiler
	You cannot use the LaunchPad and the Command History window in -nodesktop mode.
	Don't use nodesktop to provide a command line interface. If you prefer a command line interface, select **View -> Desktop Layout -> Command Window Only**.

Option	Description (Continued)
`-nojvm`	Start MATLAB without loading the Java VM. This minimizes memory usage and improves initial startup speed. With nojvm, you cannot use the desktop, or any of the tools that require Java. The restrictions are the same as those described under Platform Limitations in the Release Notes.
`-nosplash`	Start MATLAB without displaying the splash screen during startup.

Toolbox Path Caching

For performance reasons, MATLAB caches toolbox directory information across sessions. The caching features are mostly transparent to you. However, if MATLAB does not see the latest versions of your M-files or if you receive warnings about the toolbox path cache, you might need to update the cache.

Startup Using Cache File

Upon startup, MATLAB gets information from a cache file to build the toolbox directory cache. It displays the message

```
Using Toolbox Path Cache. Type "help toolbox_path_cache" for more
info.
```

Because of the cache file, startup is faster, especially if you run MATLAB from a network server or if you have many toolbox directories. When you end a session, MATLAB updates the cache file.

MATLAB does not use the cache file at startup if you clear the **Enable toolbox path cache** check box in General Preferences. Instead, it creates the cache by reading from the operating system directories, which is slower than using the cache file.

Updating the Cache

How the Toolbox Path Cache Works. MATLAB caches (essentially, stores in a known files list) the names and locations of files in $matlabroot/toolbox directories. These directories are for MathWorks supplied files that should not

change except for product installations and updates. Caching those directories provides better performance during a session because MATLAB does not actively monitor those directories.

We strongly recommend that you save any M-files you create and any MathWorks-supplied M-files that you edit in a directory that is *not* in the `$matlabroot/toolbox` directory tree. If you keep your files in `$matlabroot/toolbox` directories, they may be overwritten when you install a new version of MATLAB.

When to Update the Cache. When you add files to `$matlabroot/toolbox` directories, the cache and the cache file need to be updated. MATLAB updates the cache and cache file automatically when you install toolboxes or toolbox updates using the MATLAB installer. MATLAB also updates the cache and cache file automatically when you use MATLAB tools, such as when you save files from the MATLAB Editor to `$matlabroot/toolbox` directories.

When you add or remove files in `$matlabroot/toolbox` directories by some other means, MATLAB might not recognize those changes. For example, when you

- Save new files in `$matlabroot/toolbox` directories using an external editor
- Use operating system features and commands to add or remove files in `$matlabroot/toolbox` directories

MATLAB displays this message

```
Undefined function or variable
```
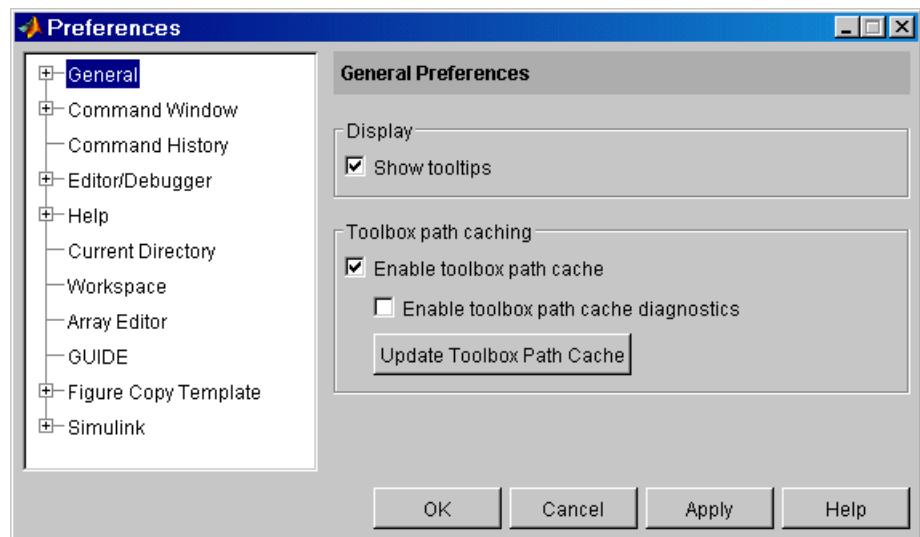
You need to update the cache so MATLAB will recognize the changes you made in `$matlabroot/toolbox` directories.

Steps to Update the Cache. To update the cache and the cache file

1 Select **File -> Preferences -> General**.

The **General Preferences** panel appears.

2 Click **Update Toolbox Path Cache** and click **OK**.

Function Alternative. To update the cache, use `rehash toolbox`. To also update the cache file, use `rehash toolboxcache`. For more information, see `rehash`.

Additional Diagnostics with Toolbox Path Caching

To display information about startup time when you start MATLAB, select the **Enable toolbox path cache diagnostics** in General Preferences.

Quitting MATLAB

To quit MATLAB at any time, do one of the following:

- Click the close box ⊠ in the MATLAB desktop.
- Select **Exit MATLAB** from the desktop **File** menu.
- Type quit at the Command Window prompt.

MATLAB closes immediately, without issuing a warning. If you want to see a warning, use the finishdlg.m script, as described in the next paragraph.

Running a Script When Quitting MATLAB

When MATLAB quits, it runs the script finish.m, if finish.m exists in the current directory or anywhere on the MATLAB search path. You create the file finish.m. It contains functions to run when MATLAB terminates, such as saving the workspace or displaying a confirmation dialog box. There are two sample files in $matlabroot/toolbox/local that you can use as the basis for your own finish.m file:

- finishsav.m—Includes a save function so the workspace is saved to a MAT-file when MATLAB quits.
- finishdlg.m—Displays a confirmation dialog box that allows you to cancel quitting.

For more information, see finish in the online Function Reference.

2

Using the Desktop

The desktop is the main interface for working with MATLAB.
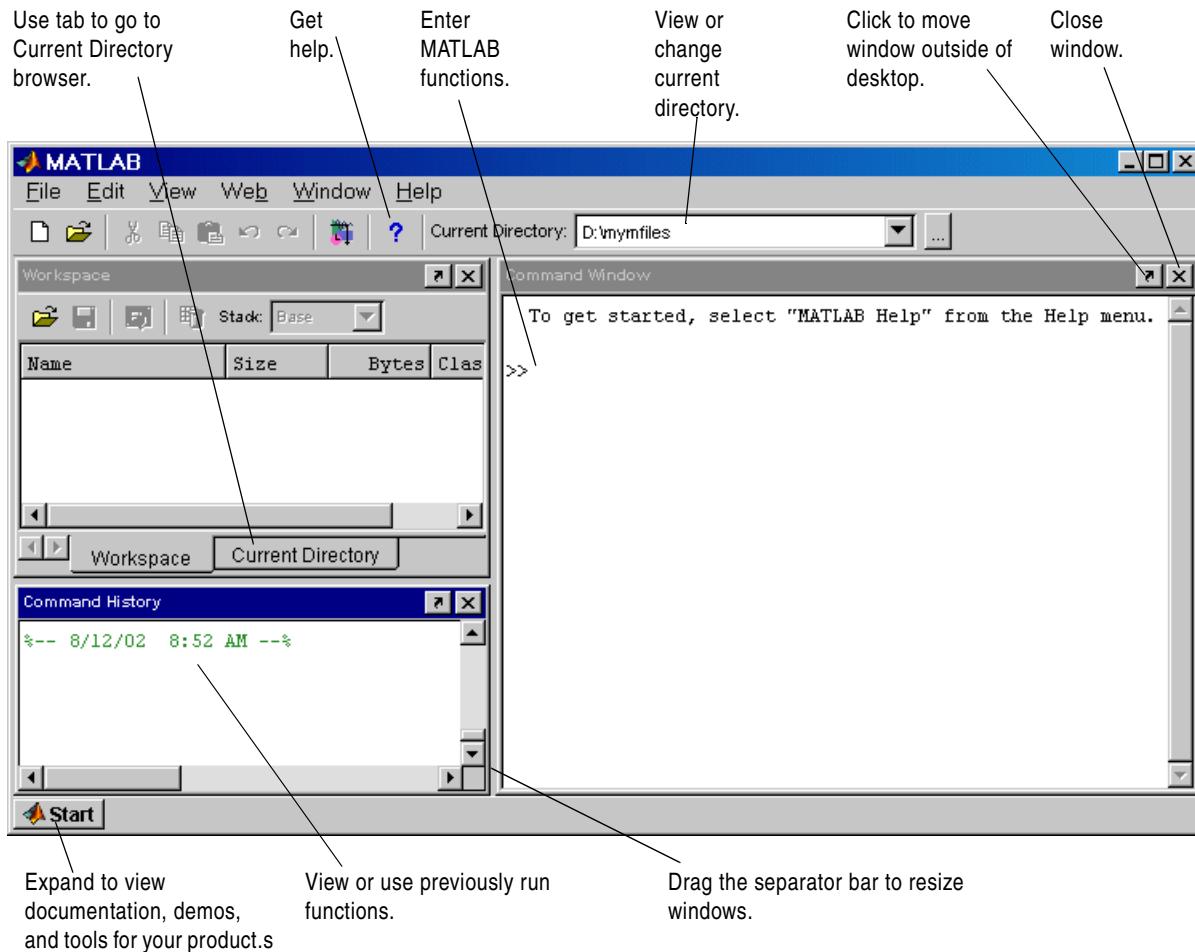
What the Desktop Is

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB. Think of the desktop as your instrument panel for MATLAB. The main things you need to know about the desktop are

- "Desktop Tools" on page 2-4—All the tools managed by the desktop
- "Configuring the Desktop" on page 2-10—Arranging the tools in the desktop
- "Common Desktop Features" on page 2-20—Features you can use in many of the tools, such as context menus
- "Setting Preferences" on page 2-30—Setting options for the look and performance of desktop and other tools

The first time MATLAB starts, the desktop appears as shown in the following illustration.

Use tab to go to Current Directory browser.

Get help.

Enter MATLAB functions.

View or change current directory.

Click to move window outside of desktop.

Close window.

Expand to view documentation, demos, and tools for your product.s

View or use previously run functions.

Drag the separator bar to resize windows.

Desktop Tools

The following tools are managed by the MATLAB desktop, although not all of them appear by default when you first start. If you prefer a command-line interface, you can use equivalent functions to perform most of the features found in the MATLAB desktop tools. You need to use these equivalent functions to perform the operations in M-files. Instructions for using these function equivalents are provided with the documentation for each tool.

- Start Button and Launch Pad—Run tools and access documentation for all of your MathWorks products.
- Command Window—Run MATLAB functions.
- Command History—View a log of the functions you entered in the Command Window, copy them, and execute them.
- Help Browser—View and search the documentation for the full family of MATLAB products.
- Current Directory Browser—View MATLAB files and related files, perform file operations such as open, and find content.
- Workspace Browser—View and make changes to the contents of the workspace.
- Array Editor—View array contents in a table format and edit the values.
- Editor/Debugger—Create, edit, and debug M-files (files containing MATLAB functions).
- Profiler—Assess the performance of your M-files using this graphical interface.

Other MATLAB tools and windows, such as figure windows, are not managed by the desktop.

Start Button and Launch Pad

The MATLAB Start button and the Launch Pad are similar tools that provide easy access to tools, demos, and documentation for all your MathWorks products. The MATLAB Start button has a menu interface that you access only when you need it. The Launch Pad presents information as a tree view in a window that is always visible if you have the Launch Pad open.
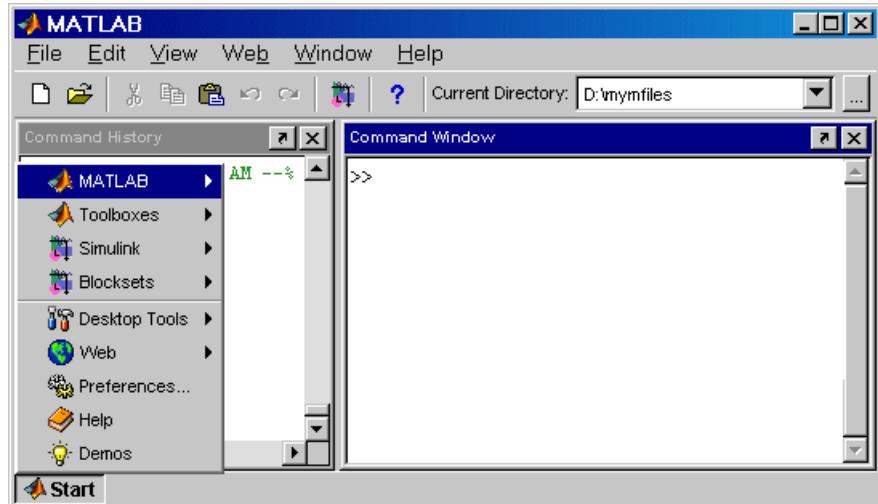
For details, see

- "Using the Start Button" on page 2-5
- "Using the Launch Pad" on page 2-6
- "Icons in the Start Button and Launch Pad" on page 2-8
- "Refreshing the Start Button and Launch Pad" on page 2-8
- "Adding Your Own Entries to the Start Button and Launch Pad" on page 2-8

Using the Start Button

The MATLAB Start button provides easy access to tools, demos, and documentation for all your MathWorks products.

1 Click the Start button to view product categories and desktop tools installed on your system.

2 Navigate through the menu on the Start button to the item you want to open, and then select that item. See also "Icons in the Start Button and Launch Pad" on page 2-8.

The item you selected opens.
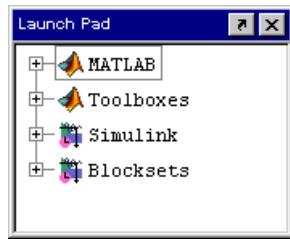
For example, select **Toolboxes -> Signal Processing -> Filter Design and Analysis Tool** to open that tool.

Using the Launch Pad

The Launch Pad provides easy access to tools, demos, and documentation for all your MathWorks products.

1 To open it, select **Launch Pad** from the **View** menu in the MATLAB desktop.
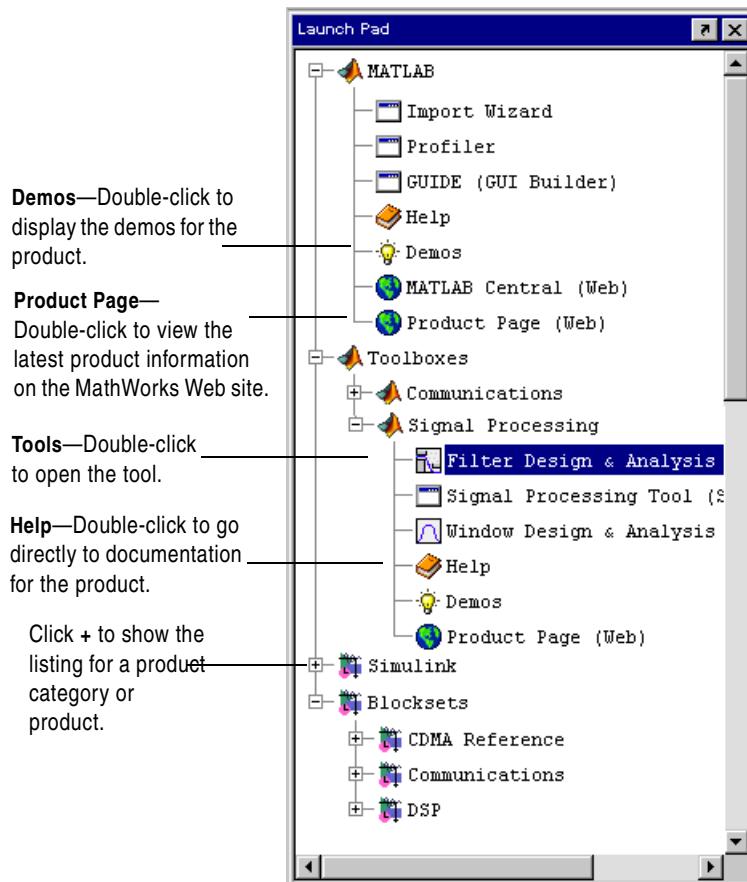
The product categories are listed.

2 To expand the listings for a product category or product, click the **+** to the left of the product. To collapse the listings, click the **-** to the left of the product. Navigate through the product categories and products to the tool you want to open.

3 Double-click a tool to open it. This example shows how to open the Filter Design and Analysis tool in the Signal Processing Toolbox.

Sample of listings in Launch Pad—you see listings for all products installed on your system.

Demos—Double-click to display the demos for the product.

Product Page— Double-click to view the latest product information on the MathWorks Web site.

Tools—Double-click to open the tool.

Help—Double-click to go directly to documentation for the product.

Click **+** to show the listing for a product category or product.

Launch Pad

- MATLAB
  - Import Wizard
  - Profiler
  - GUIDE (GUI Builder)
  - Help
  - Demos
  - MATLAB Central (Web)
  - Product Page (Web)
- Toolboxes
  - Communications
  - Signal Processing
    - Filter Design & Analysis
    - Signal Processing Tool (S
    - Window Design & Analysis
    - Help
    - Demos
    - Product Page (Web)
- Simulink
- Blocksets
  - CDMA Reference
  - Communications
  - DSP

Icons in the Start Button and Launch Pad

Icons help you quickly locate a particular type of product or tool. This legend describes the action performed when you double-click an entry with one of these associated icons in the Start button or Launch Pad.

Icon	Description of Action When Opened
📙	Documentation roadmap page for that product opens in the Help browser.
💡	Demo interface in Help browser opens, with the demos for that product selected.
⬜	Selected tool opens.
🌐	Product Page, which contains the latest product information on the MathWorks Web site, opens in your Web browser.

Refreshing the Start Button and Launch Pad

The Start button and Launch Pad include entries for all products found on the MATLAB search path when the MATLAB session was started. If you change the search path after the start of a session, such as by adding a toolbox directory, the Start button and Launch Pad are not automatically updated. Right-click in the Launch Pad and select **Refresh** from the context menu to update the Start button and Launch Pad so they reflect all products on the current search path.

Adding Your Own Entries to the Start Button and Launch Pad
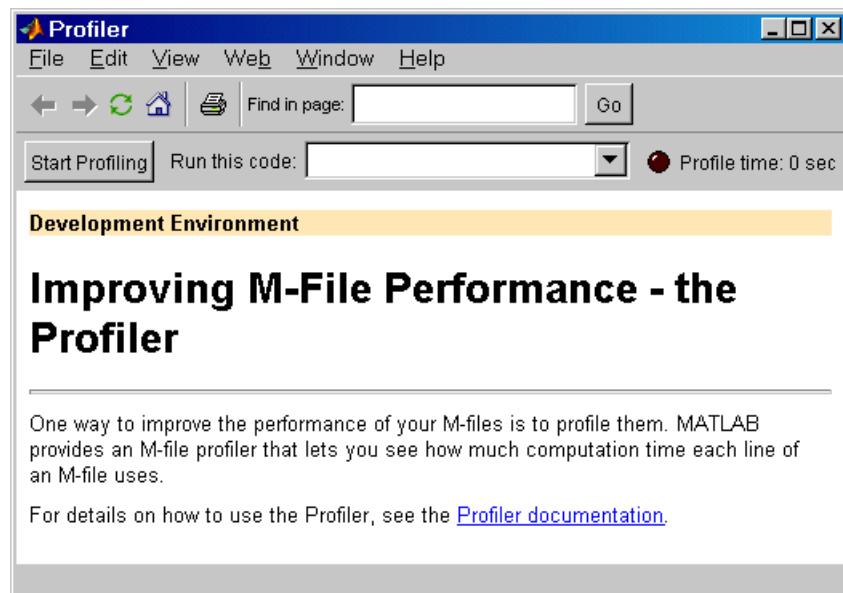
You can add your own entries to the Start button and Launch Pad by creating an info.xml file. To see an example, select one of the existing entries in the Launch Pad, right-click, and select **View Source** from the context menu. The info.xml file for that product appears. The line for the tool you selected appears highlighted.

Create a similar info.xml file for your own application and put it in a folder that is on the search path. Right-click in the Launch Pad and select **Refresh** from the context menu. The **Start** button and Launch Pad now include the entries you added.

Profiler

The Profiler is a tool that shows you where an M-file is spending its time. It is also useful to use when you are debugging or trying to understand how an M-file works.

Select **View -> Profiler** to open the tool. For details, see "Measuring Performance" in the MATLAB Programming and Data Types documentation.

Configuring the Desktop

You can modify the desktop configuration to best meet your needs. Configure the MATLAB desktop by

- "Opening and Closing Desktop Tools" on page 2-10
- "Resizing Windows" on page 2-12
- "Moving Windows" on page 2-13
- "Using Predefined Desktop Configurations" on page 2-19

When you end a session, MATLAB saves its desktop configuration. The next time you start MATLAB, the desktop is restored the way you left it.

Opening and Closing Desktop Tools

As part of configuring the MATLAB desktop so that it best meets your needs, you can use the following features:

- "Opening Desktop Tools" on page 2-10—Open only those tools you use.
- "Going to Documents in Desktop Tools" on page 2-11—Go directly to opened M-files, figures, and more.
- "Closing Desktop Tools" on page 2-12—Close those tools you do not use.

Opening Desktop Tools
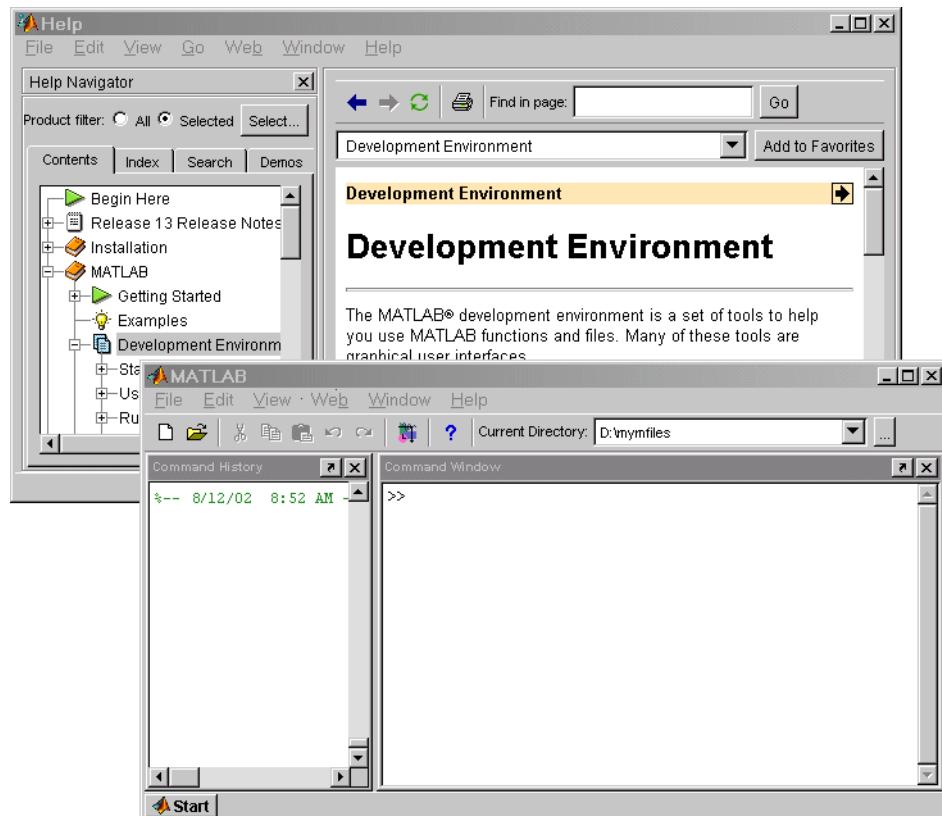
To open a tool from the desktop, select the tool from the **View** menu or the **Start** button. The tool opens in the location it occupied the last time it was open.

There are a few tools controlled by the desktop that are not on the **View** menu:

- Array Editor—Open it by double-clicking a variable in the Workspace Browser.
- Editor/Debugger—Open it by creating a new M-file or opening an existing M-file. For instructions, see "Starting the Editor/Debugger" on page 7-3.

Another way to open a tool is by using a function. For example, `helpbrowser` opens the Help browser. These functions are documented with each tool.
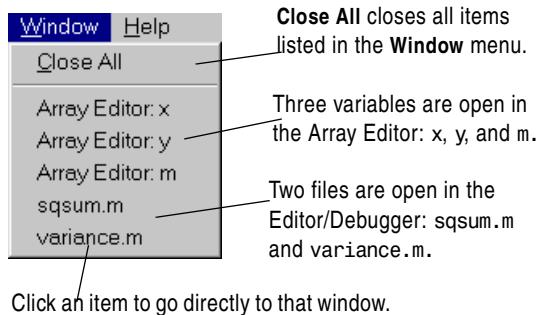
The following example shows how the MATLAB desktop might look with the Command Window, Command History, and Help browser open.

Going to Documents in Desktop Tools

The **Window** menu displays all open Editor/Debugger documents and variables in the Array Editor. On Windows platforms, it also shows figure windows and Simulink models. Select an entry in the **Window** menu to go directly to that window or tabbed document. Select **Close All** to close all items listed in the **Window** menu.

For example, the **Window** menu in the following illustration shows three documents open in the Array Editor and two documents open in the Editor/Debugger. Selecting variance.m, for example, makes the Editor/Debugger window with the file variance.m become the active window.

Close All closes all items listed in the **Window** menu.

Three variables are open in the Array Editor: x, y, and m.

Two files are open in the Editor/Debugger: sqsum.m and variance.m.

Click an item to go directly to that window.

Closing Desktop Tools

To close a desktop tool, do one of the following:

• Select the item in the **View** menu (the item becomes cleared).

• Click the close box ⊠ in the window's title bar.

• Select **Close** from the **File** menu to close the current window.

The window closes.

Resizing Windows
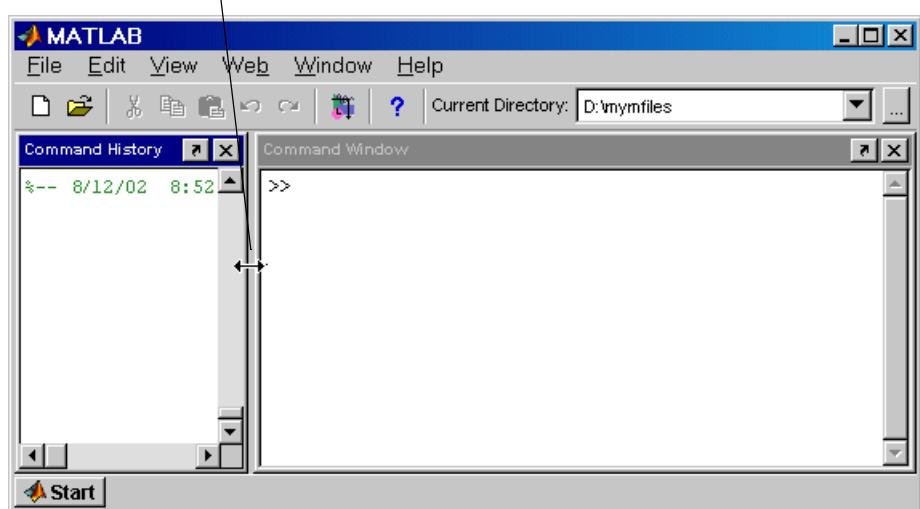
To resize windows in the MATLAB desktop, use the separator bar, which is the bar between two windows:

1 Move the cursor onto the separator bar.

The cursor assumes a different shape. On Windows platforms, it is a double-headed arrow ↔. On UNIX, it is an arrow with a bar.

2 Drag the separator bar to change the sizes of the windows.

Drag separator bar to resize windows in the desktop.

To resize the MATLAB desktop itself or any tool outside the desktop, use the convention for your platform. For example, on Windows, drag any edge or corner of the window.

Moving Windows

There are three basic ways to move MATLAB desktop windows:

- "Moving Windows Within the MATLAB Desktop" on page 2-14
- "Moving Windows Out of the MATLAB Desktop" on page 2-15 and "Moving Windows into the MATLAB Desktop" on page 2-16
- "Grouping (Tabbing) Windows Together" on page 2-17

Moving Windows Within the MATLAB Desktop

To move a window to another location in the MATLAB desktop:
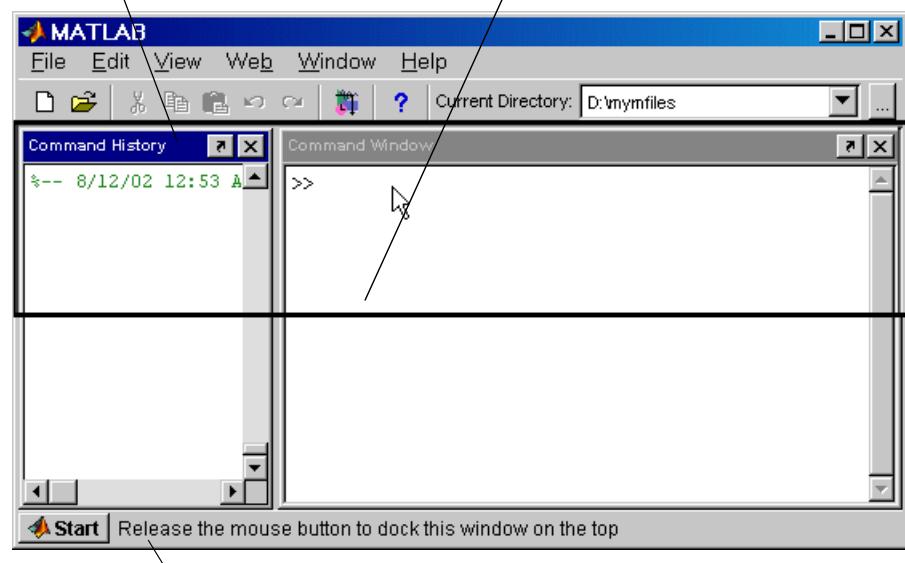
1 Drag the title bar of the window toward where you want the window to be located.

As you drag the window, an outline of it appears. When the outline nears a position where you can dock (keep) it, the outline snaps to that location. The status bar displays instructions about moving the window while you drag the outline.

In the following example, the Command History window is originally to the left of the Command Window and is being dragged above the Command Window. When the top of the Command History window touches the bottom of the toolbar, the outline appears.

Drag the title bar of the Command
History window to move it.
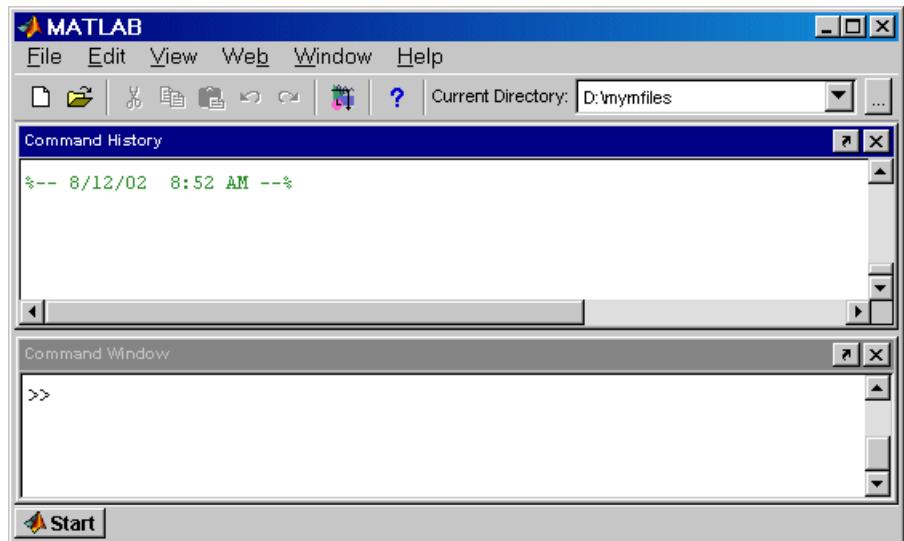
Release the mouse when the outline of the Command
History window appears where you want it to be.

The status bar displays instructions about moving the window.

2 Release the mouse to dock the window at the new location.

Other windows in the desktop resize to accommodate the new configuration. The following example shows how the desktop looks after you move the Command History window above the Command Window.
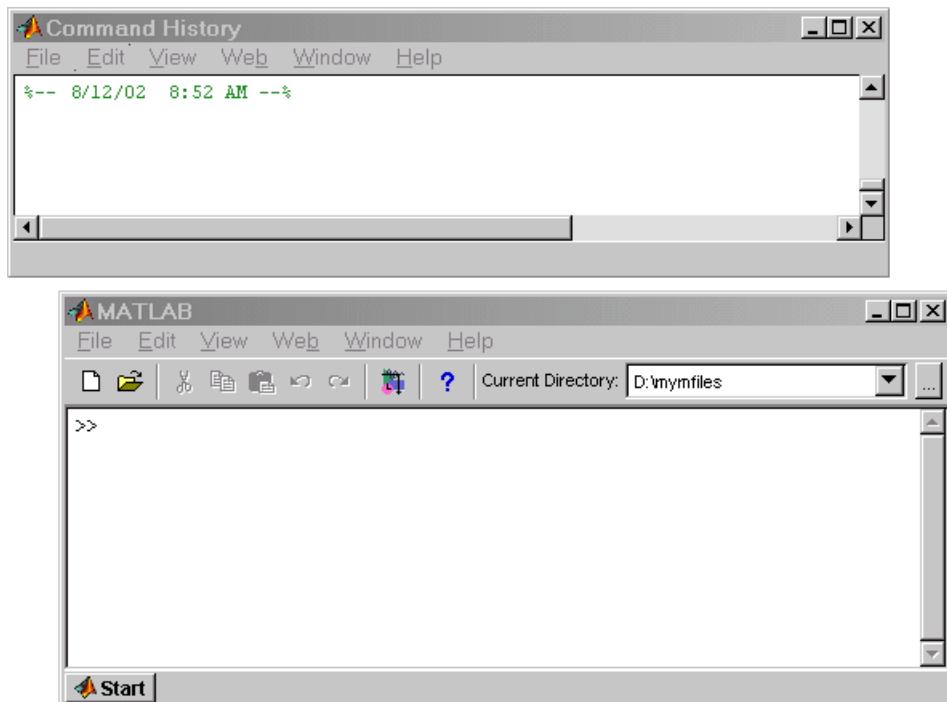
Moving Windows Out of the MATLAB Desktop

To move a window outside the MATLAB desktop, do one of the following:

- Click the arrow ▣ in the title bar of the window you want to move outside the desktop.
- Select **Undock** for that tool from the **View** menu; the window must be the currently active window.
- Drag the title bar of the window outside the desktop. As you drag, an outline of the window appears. When the cursor is outside the MATLAB desktop, release the mouse.

The window appears outside the MATLAB desktop.

In the following example, the Command History window has been moved outside the desktop.

Moving Windows into the MATLAB Desktop

To move a window that is outside the MATLAB desktop into the desktop, select **Dock** for that tool from its **View** menu.

Grouping (Tabbing) Windows Together
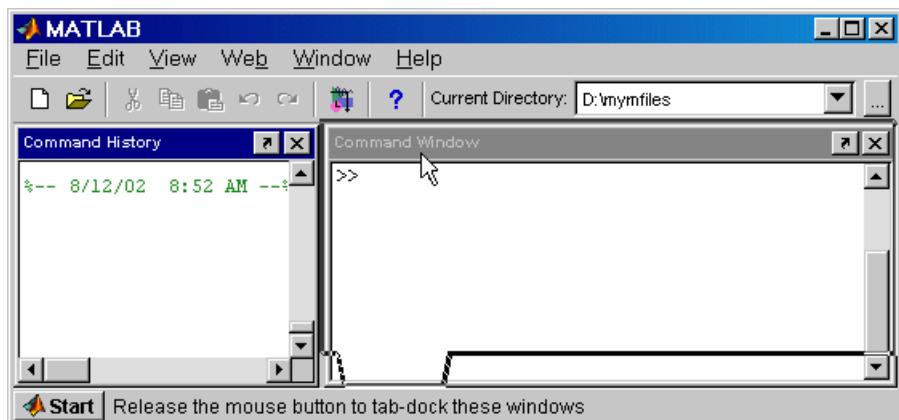
You can group windows so that they occupy the same space in the MATLAB desktop, with access to the individual windows via tabs. These are the main features in working with tabbed windows:

- "Grouping Windows" on page 2-17
- "Viewing Tabbed Windows" on page 2-18
- "Moving Tabbed Windows" on page 2-18
- "Closing Tabbed Windows" on page 2-18

Grouping Windows. To group (also called "to tab") windows together:
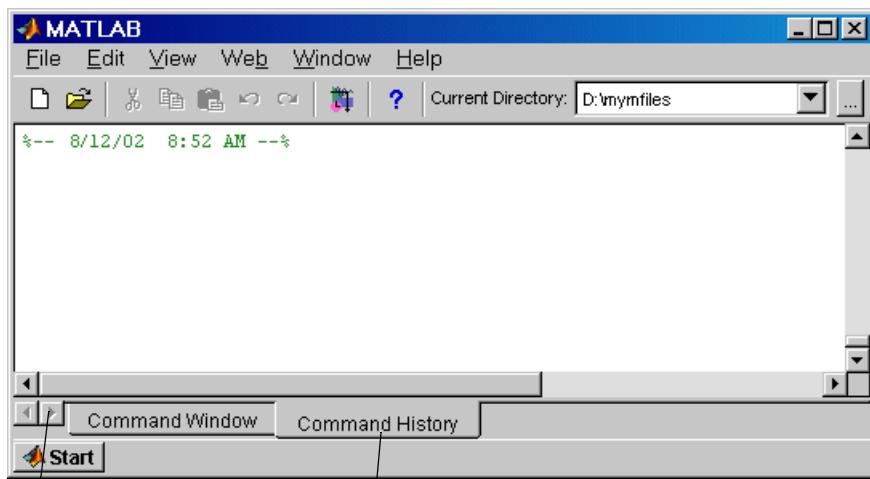
1 Drag the title bar of one window in the desktop on top of the title bar of another window in the desktop.

The outline of the window you are dragging overlies the target window, and the bottom of the outline includes a tab. In the following example, the Command History window is originally to the left of the Command Window and its title bar is being dragged on top of the title bar of the Command Window.

2 Release the mouse.

Both windows occupy the same space and labeled tabs appear at the bottom of that space. In the following example, the Command History and Command Window are tabbed together, with the Command History tab currently selected.

Use arrows to show any tabs that are not in view. In this example, the arrows are grayed, indicating all tabs are in view.

There are labeled tabs for each tool tabbed together in the window. Click a tab to view that tool.

Viewing Tabbed Windows. To view a tabbed window, click the window's tab. The window moves to the foreground and becomes the currently active window. If there are more tabs in a window than are currently visible, use the arrows to the left of the tabs to see additional tabs.

Moving Tabbed Windows. To move a tabbed window to another location, drag the title bar or the tab to the new location. You can move it inside or outside the MATLAB desktop.

Closing Tabbed Windows. When you click the close box ⊠ for a window that is part of a group of windows tabbed together, that window closes. You cannot

close all the tabbed windows at one time; instead close each window individually.

Using Predefined Desktop Configurations

There are six predefined MATLAB desktop configurations, which you can select from the **View -> Desktop Layout** menu:

- **Default**—Contains the Command Window and the Command History with the Current Directory browser and the Workspace browser tabbed together.

- **Command Window Only**—Contains only the Command Window. This makes MATLAB appear similar to how it looked in older versions.

- **Simple**—Contains the Command History and Command Window, side by side.

- **Short History**—Contains the Current Directory browser and Workspace browser tabbed together above the Command Window and a small Command History.

- **Tall History**—Contains the Command History along the left, and the Current Directory browser and Workspace browser tabbed together above the Command Window.

- **Five Panel**—Contains the Launch Pad above the Command History along the left, the Workspace browser above the Current Directory browser in the center, and the Command Window on the right.

After selecting a predefined configuration, you can move, resize, and open and close windows.
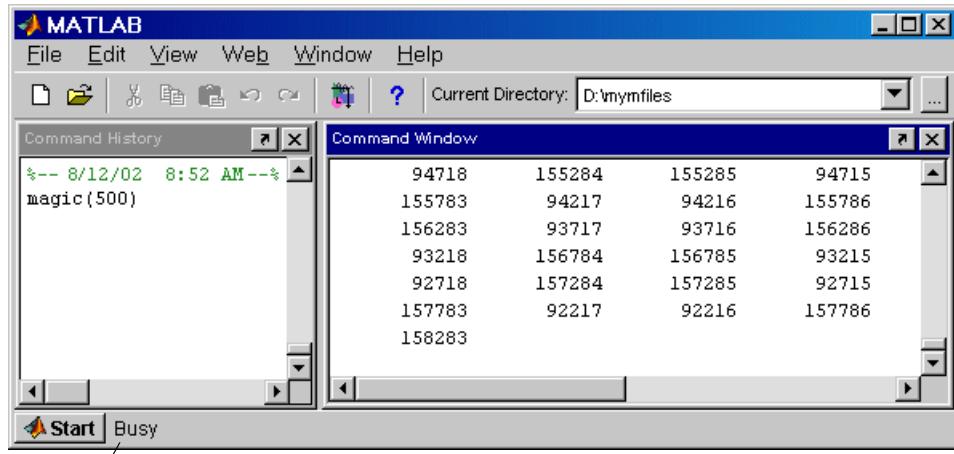
Common Desktop Features

These common features are available for the desktop tools:

- "Status Bar" on page 2-20
- "Desktop Toolbar" on page 2-21
- "Context Menus" on page 2-22
- "Keyboard Shortcuts" on page 2-22
- "Selecting Multiple Items" on page 2-24
- "Using the Clipboard" on page 2-24
- "Page Setup Options for Printing" on page 2-25
- "Accessing The MathWorks on the Web" on page 2-28

Status Bar

In the lower left corner of the desktop is the status bar. It displays messages, such as when MATLAB is busy executing statements or when the Profiler is on.
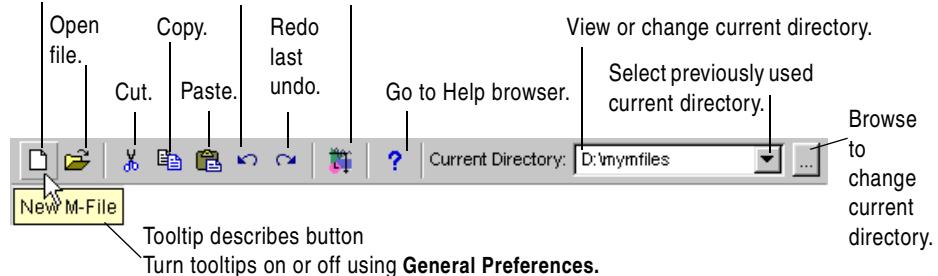
Status bar message indicates MATLAB is busy executing a statement.

Desktop Toolbar

The toolbar in the MATLAB desktop provides easy access to frequently used operations. Hold the cursor over a button and a tooltip appears describing the item. Note that some of the tools also have toolbars within their windows.

Create new M-file. Undo last edit. Open Simulink library browser.

Open
file.   Copy.   Redo   View or change current directory.
                          last
                          undo.   Select previously used
                Cut.   Paste.   Go to Help browser.   current directory.
                                                                                    Browse
                                                                                    to
                                                                                    change
                                                                                    current
                                                                                    directory.

Current Directory: D:\mymfiles

New M-File

Tooltip describes button
Turn tooltips on or off using **General Preferences.**
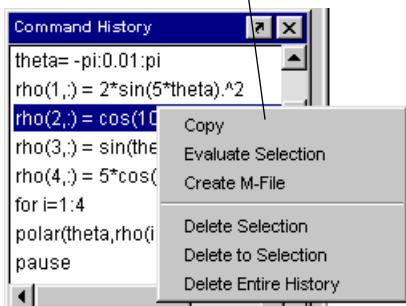
Current Directory Field

The **Current Directory** field in the toolbar shows the MATLAB current working directory. You can change the current directory using this field and perform other file operations using the Current Directory browse. For instructions, see "File Operations" on page 5-24.

Context Menus

Many of the features of the MATLAB desktop tools are available from context menus, also known as pop-up menus. To access a context menu, right-click a selection. The context menu for it appears, presenting the available actions. For example, following is the context menu for a selection in the Command History window. In general, this documentation does not specifically note where context menus can be used. The exception is where the context menu is the only means of accessing the feature.

Access context (pop-up) menus by right-clicking a selection.

Keyboard Shortcuts

You can access many of the menu items using keyboard shortcuts (sometimes called accelerators) for your platform, such as using **Ctrl+X** to perform a **Cut** on Windows platforms. Many of the menu items show the shortcuts. Additional standard shortcuts for your platform usually work but only one is listed with each menu item.

On Windows platforms, you can also use mnemonics, such as **Alt+F** to open the **File** menu. Mnemonics are listed with the menu item. For example, on the **File** menu the **F** in **File** is underlined, which indicates that **Alt+F** opens the menu. Note that more recent versions of Windows do not automatically show the mnemonics on the menu. For example, you might need to hold down the **Alt** key while the window has focus in order to see the shortcuts on the menus. Or in Windows 2000, go to the **Display Control Panel**, select **Effects**, and clear the item **Hide keyboard navigation indicators until I use the Alt key**. See your Windows documentation for details.

Following are some general shortcuts that are not listed on menu items.

Key	Result
Enter	The equivalent of double-clicking, it performs the default action for a selection. For example, pressing **Enter** while a statement in the Command History window is selected runs that statement in the Command Window.
Escape	Cancels the current action.
Ctrl+Tab or **Ctrl+F6**	Moves to the next open tool in the desktop, or, in a tool containing multiple tabs, like the Editor/Debugger, moves to the next tab.
Tab	Advances to the next button or field in a window or dialog box. In the Command Window, completes a statement if the tab completion preference is selected.
Space Bar	Activates the button that has focus, for example, the **OK** button in a dialog box.
+ or **-** or *****	Use these keys on the numeric keypad to expand and collapse items in tree views. The Help browser **Navigator** pane and the Launch Pad use tree views, for example. Use **+** to expand the selected item, use **-** to collapse the selected item, and use * to recursively expand it, meaning open all items contained in the selected item. Note that * is not valid in the Help browser.
Ctrl+Shift+Tab	Moves to the previous tab in the desktop, where the tab is for a tool, or for a file in the Editor/Debugger. When used in the Editor/Debugger in tabbed mode outside the desktop, moves to the previous open file.

Key	Result (Continued)
Ctrl+Page Up	Moves to the next tab within a group of items tabbed together.
Ctrl+Page Down	Moves to the previous tab within a window.
Alt+F4	Closes the desktop or the window outside the desktop.
Alt+Space	Displays the Windows system menu.

On the Alpha platform, even though shortcuts are not listed on the menu items, most standard Alpha shortcuts work.

For other shortcuts available in the Command Window, see "Tab Completion" on page 3-9.

Selecting Multiple Items

In many of the desktop tools, you can select multiple items and then select an action to perform on all the selected items. Select multiple items using the standard practices for your platform.

For example, if your platform is Windows, do the following to select multiple items:

1 Click the first item you want to select.

2 Hold the **Ctrl** key and then click the next item you want to select. Repeat this step until you have selected all the items you want.

 To select contiguous items, select the first item, hold the **Shift** key, and then select the last item.

Now you can perform an action, such as delete, on the selected items.

Using the Clipboard

You can cut and copy a selection from a desktop tool to the clipboard and then paste it from the clipboard into another tool or application. Use the **Edit** menu, toolbar, context menus, or standard keyboard shortcuts. For example, you can

copy a selection of statements from the Command History window and paste them into the desktop.

Use **Paste** to move items copied to the clipboard from other applications. The **Paste Special** item in the **Edit** menu opens the selection on the clipboard in the Import Wizard. You can use this to copy data from another application, such as Excel, into MATLAB. For details, see Chapter 6, "Importing and Exporting Data."

To undo the most recent cut, copy, or paste command, select **Undo** from the **Edit** menu. Use **Redo** to reverse the **Undo**.

You can also copy by dragging a selection. For example, make a selection in the Command History window and drag it to the Command Window, which pastes it there. Edit the lines in the Command Window, if needed, and then press the **Enter** key to run the lines from the Command Window.

Page Setup Options for Printing

You can specify setup options when you print from the Command Window, Command History, and Editor. The setup options are essentially the same for each tool, with minor variations. This section covers
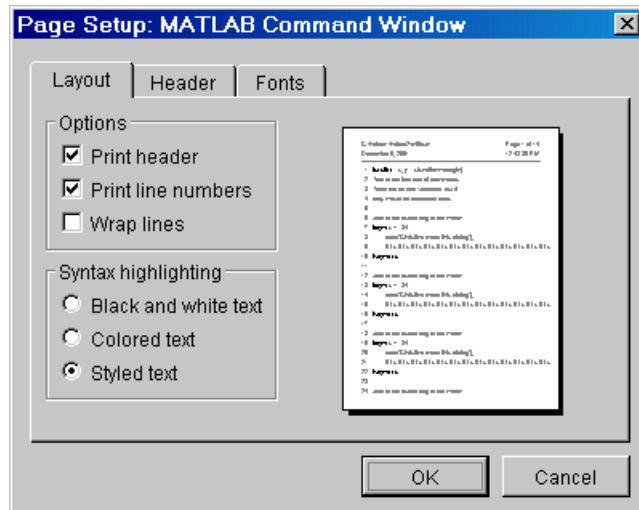
- "Specifying Page Setup Options" on page 2-26
- "Layout Options for Page Setup" on page 2-27
- "Header Options for Page Setup" on page 2-27
- "Fonts Options for Page Setup" on page 2-27

Specifying Page Setup Options

To specify page setup options:

1 In the tool you want to print from, for example, the Command Window, select **File -> Page Setup**.

The **Page Setup** dialog box opens for that tool.

2 Click the **Layout**, **Header**, or **Fonts** tab in the dialog box and set those options for that tool, as detailed in subsequent sections.

3 Click **OK**.

4 Select **File -> Print** in the tool you want to print from, for example, the Command Window.

The contents from the tool are printed, using the options you specified in **Page Setup**.

Layout Options for Page Setup

You can specify the following layout options. A preview window shows you the effects of your selections:

• **Print header**—Print the header specified in the **Header** tab.

• **Print line numbers**—Print line numbers.

• **Wrap lines**—Wrap any lines that are longer than the printed page width.

• **Syntax highlighting**—For keywords and comments that are highlighted in the Command Window, specify how they are to appear in print. Options are black and white text (that is, no highlighting), colored text, or styled text. Only keywords and comments that are input are highlighted in the Command Window and in print; output is not highlighted.

Header Options for Page Setup

If you elect to print a header by selecting **Print header** in the **Layout** tab, use **Header** options to specify how the elements of the header are to appear. A preview window shows you the effects of your selections:

• **Page number**—Format for the page number, for example # of n

• **Border**—Border style for the header, for example, Shaded box

• **Layout**—Layout style for the header, for example, Standard one line includes the date, time, and page number all on one line

Fonts Options for Page Setup

Specify the font to be used for the printed contents.

Use **Choose font** to select the element, either Body, Header, or Line numbers. Then, select the font to use for that element. Repeat for all applicable elements. If you did not select **Print header** or **Print line numbers** in the **Layout** tab, you do not need to specify the Header or Line numbers font.

For example, if you are setting **Page Setup** options for the Command Window, select **Use Command Window font** if you want the printed font for that element to be the same as the font specified for display in **Command Window Font & Colors** preferences.

If you want the font used for printing to be different from that specified for display, select **Use custom font** and specify the font characteristics for that element:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Accessing The MathWorks on the Web

You can access popular MathWorks Web pages from the MATLAB desktop. Select one of the following items from the **Web** menu. For most items, the selected Web page then opens in your default Web browser.

- **The MathWorks Web Site**—Links you to the home page of the MathWorks Web site (`http://www.mathworks.com`).
- **MATLAB Central**—Links you to the MATLAB Central Web site (`http://www.mathworks.com/matlabcentral`) for the MATLAB user community. It includes MATLAB contest entries and results, and a MATLAB screen saver as well as
  - **MATLAB File Exchange**—Links you to a code library of files contributed by MathWorks customers and employees available for download and use with MathWorks products.
  - **MATLAB Newsgroup Access**—Provides access to the Usenet newsgroup for MATLAB and related products, `comp.soft-sys.matlab`, which allows you to post and answer questions as well as view the archives of posts.
- **Products**—Links you to the MathWorks Products page (`http://www.mathworks.com/products/`), where you can get information about the full family of products.
- **Check for Updates**—In a dialog box, lists the version numbers for all MathWorks products installed on your system. Click **Check for Updates** in the dialog box to determine whether more recent versions of the products you have installed are available.
- **Membership**—Links you to the MATLAB Access page (`http://www.mathworks.com/mla/index.shtml`) for Access members. If you

are not a member, you can join online to help you keep up to date on the latest MATLAB developments.

- **Technical Support Knowledge Base**—Links you to the MathWorks Support page (`http://www.mathworks.com/support`), where you can look for solutions to problems you are having or report new problems.

Setting Preferences

Set preferences to modify the default behavior of some aspects of MATLAB, such as the font used in the Command Window. These topics are covered:

- "Using the Preferences Dialog Box" (p. 2-30)
- "Summary of Preferences" (p. 2-31)
- "General Preferences for MATLAB" (p. 2-32)
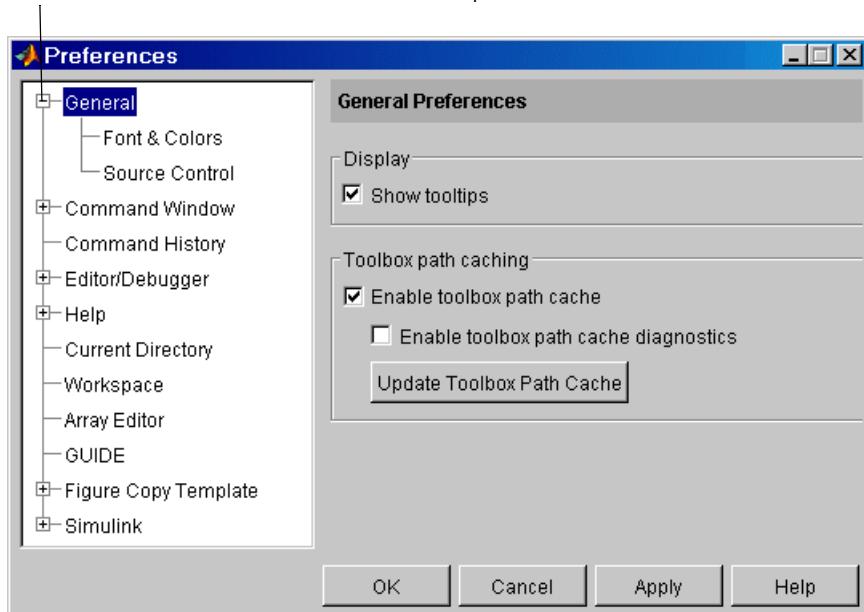
Using the Preferences Dialog Box

To set preferences:

1 Select **Preferences** from the **File** menu.

The **Preferences** dialog box opens. The page opened reflects the currently active window.

Select a tool and then click the + to list additional preferences for that tool.

2 In the left pane, select a tool. In the above example, **General** preferences are
selected, meaning the preferences apply to MATLAB in general but not to a
specific tool. Click the **+** to display more items for that tool, and then select
an item to set its preferences.

The right pane shows the preference you selected.

3 In the right pane, specify the preference values and click **Apply** or **OK** to set
the preferences.

The preferences take effect immediately.

Summary of Preferences

This table provides a summary; see the details for each tool.

Preference	What You Can Specify
General Preferences (p. 2-32)	Desktop display and toolbox path caching
Command Window (p. 3-24)	Numeric format and display, echo, fonts, colors, keyboard, indenting
Command History (p. 3-35)	Display, filtering, saving, drag and drop
Editor/Debugger (p. 7-46)	Editor type, startup options, fonts, colors, display, keyboard, indenting, autosaving
Help (p. 4-32)	Documentation location, products, PDF reader location (for UNIX), synchronization, fonts
Current Directory (p. 5-35)	Number of entries in history, file display options
Workspace (p. 5-8)	Fonts, confirm deletion of variables
Array Editor (p. 5-16)	Fonts, numeric format
GUIDE	Display options

Preference	What You Can Specify (Continued)
Figure Copy Template	Application, text, line, uicontrols, axis, format, background color, size
Simulink	Display, fonts, and simulation

The preferences file is `matlab.prf`. Type `prefdir` in the Command Window to see the location of the file. The `matlab.prf` file is loaded when MATLAB starts and is overwritten when you close MATLAB.

Preferences remain persistent across MATLAB sessions. Note that some tools allow you to control these aspects from within the tool without setting a preference—use that method if you only want the change to apply to the current session.

General Preferences for MATLAB

These preferences apply to all relevant tools in MATLAB.

Display

To show tooltips when you hold the cursor over a toolbar button in the MATLAB desktop, select the **Show tooltips** check box.

Toolbox path caching

See "Toolbox Path Caching" on page 1-9.

Font & Colors

Desktop font. Desktop font preferences specify the characteristics of the font used in tools under the control of the MATLAB desktop. The font characteristics are

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look. Lucida Console approximates the `fixedsys` font available in previous versions of MATLAB.
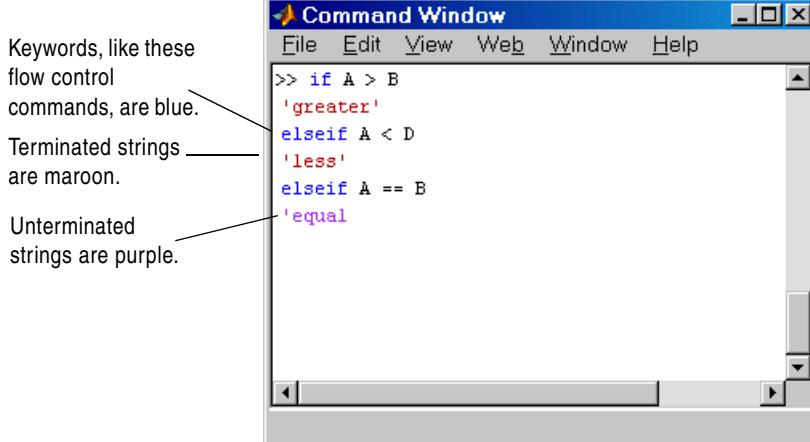
You can specify a different font for the Command Window, Editor/Debugger, Help browser, Workspace browser, and Array Editor using preferences for those tools.

Syntax highlighting colors. Select the colors to use to highlight syntax. For more information, see "Syntax Highlighting and Parentheses Matching" on page 3-6.

- **Keywords**—Flow control and other functions such as for and if are colored.
- **Comments**—All lines beginning with a % are colored.
- **Strings**—Single quotes and whatever is between them are colored.
- **Unterminated strings**—A single quote without a matching single quote and whatever follows the quote is colored.
- **System commands**—Commands such as the ! (shell escape) are colored.
- **Errors**—The error text is colored.

Click **Restore Default Colors** to return to the default settings. The following example uses the default values for color preferences.

Default colors

Keywords, like these flow control commands, are blue.

Terminated strings are maroon.

Unterminated strings are purple.

Source Control

Specify the source control system you want to interface MATLAB to. For more information, see "Interfacing with Source Control Systems" on page 8-1.

3

Running MATLAB Functions

The Command Window is the main way you communicate with MATLAB. It appears in the desktop when you first start MATLAB. Use the Command Window to run MATLAB functions (also referred to as commands) and perform MATLAB operations.
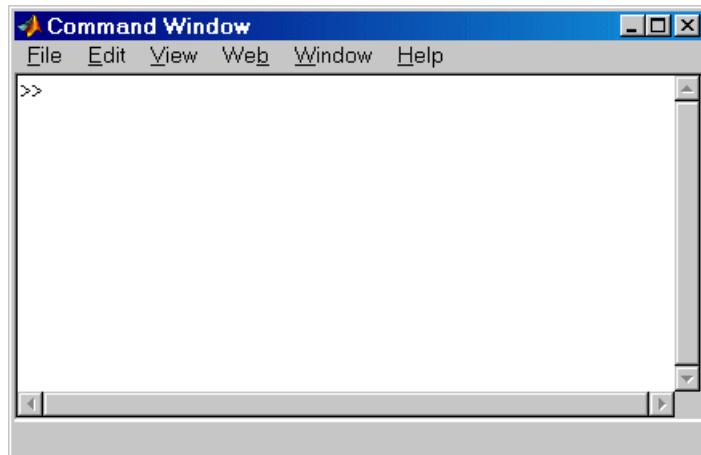
Opening the Command Window

To show the Command Window in the MATLAB Desktop, select **Command Window** from the **View** menu—see "Opening and Closing Desktop Tools" on page 2-10 for details.

If you prefer a simple command line interface without the other MATLAB desktop tools, select **View -> Desktop Layout -> Command Window Only**.

To undock the Command Window from the desktop, select **View -> Undock Command Window** while the Command Window has focus. This illustration shows the Command Window undocked from the MATLAB desktop.

Running Functions and Entering Variables

Entering Statements at the Command Line Prompt

The prompt (>>) in the Command Window indicates that MATLAB is ready to accept input from you. When you see the >> prompt, you can enter a variable or run a function. This prompt and your input are known as the command line.

For example, to create A, a 3-by-3 matrix, type

```
A = [1 2 3; 4 5 6; 7 8 10]
```

When you press the **Enter** or **Return** key after typing the line, MATLAB responds with

```
A =

    1    2    3
    4    5    6
    7    8   10
```

To run a function, type the function including all arguments and press **Return** or **Enter**. MATLAB displays the result. For example, type

```
magic(2)
```

and MATLAB returns

```
ans =
    1    3
    4    2
```

If you want to enter multiple lines before running, use **Shift+Enter** or **Shift+Return** after each line until the last. Then press **Enter** or **Return** to run all of the lines.

The K>> prompt in the Command Window indicates that MATLAB is in debug mode. For more information, see Chapter 7, "Editing and Debugging M-Files."

Evaluating a Selection

To run a selection in the Command Window, make the selection, and then right-click and select **Evaluate Selection** from the context menu.

Alternatively, after making a selection, press **Enter** or **Return**. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Opening a Selection

You can open a function, file, variable, or Simulink model from the Command Window. Select the name in the Command Window, and then right-click and select **Open Selection** from the context window. This runs the open function for the item you selected so that it opens in the appropriate tool:

- M-files and other text files open in the Editor
- Figure files (.fig) open in a figure window
- Variables open in the Array Editor
- Models open in Simulink

See open for details. If no file exists to work with the selected item, **Open selection** calls edit.

Hyperlinks to Run Functions

Use matlab: to create a hyperlink for specified text, which runs the specified function when clicked. For example, typing

```
disp('<a href= matlab:magic(4) >Generate magic square</a>')
```

displays

Generate magic square

When the user clicks the link "Generate magic square", MATLAB runs magic(4). You can use the disp or the fprintf functions with this feature.

Running One Process

In MATLAB, you can only run one process at a time. If MATLAB is busy running one function, any commands you issue will be buffered in a queue. The next command will run when the previous one finishes.

Controlling Input and Output

You can control and interpret input and output in the Command Window in these ways:

- "Case and Space Sensitivity" on page 3-5
- "Entering Multiple Functions in a Line" on page 3-5
- "Entering Long Lines" on page 3-6
- "Syntax Highlighting and Parentheses Matching" on page 3-6
- "Command Line Editing" on page 3-8
- "Clearing the Command Window" on page 3-11
- "Suppressing Output" on page 3-11
- "Paging of Output in the Command Window" on page 3-11
- "Formatting and Spacing Numeric Output" on page 3-12
- "Printing Command Window Contents" on page 3-13
- "Keeping a Session Log" on page 3-14

Case and Space Sensitivity

MATLAB is case sensitive. For example, you cannot run the function Plot but must instead use plot. Similarly, the variable a is not the same as the variable A. Note that if you use the help function, function names are shown in all uppercase, for example, PLOT, solely to distinguish them. Do *not* use uppercase when running the functions. Some functions for interfacing to Java actually used mixed case and the M-file help accurately reflects that.

Blank spaces around operators such as -, :, and ( ), are optional, but they improve readability.

Entering Multiple Functions in a Line

To enter multiple functions on a single line, separate the functions with a comma (,) or semicolon (;). Using the semicolon instead of the comma will suppress the output for the command preceding it. For example, put three functions on one line to build a table of logarithms by typing

```
format short; x = (1:10)'; logs = [x log10(x)]
```

and then press **Enter** or **Return**. The functions run in left-to-right order.

Entering Long Lines

If a statement does not fit on one line, enter a continuation ellipsis (...) at the end of the line to indicate it continues on the next line. Then press **Enter** or **Return**. Continue typing the statement on the next line. You can repeat the ellipsis to continue the statement across multiple lines. When you finish the statement, press **Enter** or **Return**.

For items in single quotes, such as strings, you must complete the string in the line on which it was started. For example, typing

```
headers = ['Author Last Name, Author First Name,   ...
'Author Middle Initial ]
```

results in

```
headers =
Author Last Name, Author First Name, Author Middle Initial
```

Syntax Highlighting and Parentheses Matching

Some entries appear in different colors to help you better find elements, such as matching if/else statements, which is known as syntax highlighting. Additional features allow you to easily see the matched pair for a parenthesis or other delimiter.

The colors and parentheses matching options listed here are the defaults. You can change them using preferences. See "Font & Colors Preferences for the Command Window" on page 3-26, and "Parentheses Matching Preferences" on page 3-28.
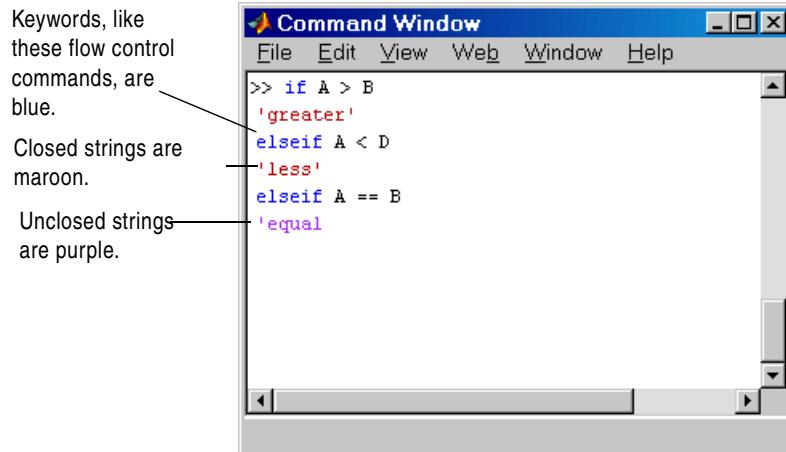
- Type a string and it is colored purple. When you close the string, it becomes maroon.
- Type a keyword, such as the flow control function for, or a continuation (ellipsis ...), and it is colored blue. Lines you enter between the opening and closing flow control functions are indented.
- Double-click an opening or closing delimiter, a parenthesis ( ), bracket [ ], or brace { }. This selects the characters between the delimiter and its mate.
- Type a closing (or opening) delimiter, and the matching opening (or closing) delimiter is highlighted briefly.

- Type a mismatched closing (or opening) delimiter and a strikethrough character appears on the delimiter. For example

    ```
    >> CC=C{:}
    ```

- Use an arrow key to move over an opening or closing delimiter. That delimiter and its matching closing or opening delimiter briefly appear underlined.
- Type a comment symbol, %, and what follows on the line appears in green. That information is treated by MATLAB as a comment.
- Type a system command, such as the ! (shell escape), and the line appears in gold.
- Errors appear in red.

Default colors are shown here – to change them, use **Preferences**.

Keywords, like these flow control commands, are blue.

Closed strings are maroon.

Unclosed strings are purple.

Note that output does not appear with syntax highlighting, except for errors.

3-7

Command Line Editing

These are time-saving features you can use in the Command Window:

- "Clipboard Features" on page 3-8
- "Recalling Previous Lines" on page 3-8
- "Tab Completion" on page 3-9
- "Arrow and Control Keys in the Command Window" on page 3-10

Clipboard Features

Use the **Cut**, **Copy**, **Paste**, **Undo**, and **Redo** features from the **Edit** menu when working in the Command Window. You can also access some of these features in the context menu for the Command Window.

Recalling Previous Lines

Use the arrow, tab, and control keys on your keyboard to recall, edit, and reuse functions you typed earlier. For example, suppose you mistakenly enter

```
rho = (1+ sqt(5))/2
```

MATLAB responds with

```
Undefined function or variable 'sqt'.
```

because you misspelled sqrt. Instead of retyping the entire line, press the up arrow ↑ key. The previously typed line is redisplayed. Use the left arrow key to move the cursor, add the missing r, and press **Enter** or **Return** to run the line. Repeated use of the up arrow key recalls earlier lines.

The functions you enter are stored in a buffer. You can use *smart recall* to recall a previous line whose first few characters you specify. For example, type the letters plo and then press the up arrow key. This recalls the last line that started with plo, as in the most recent plot function. Press **Enter** or **Return** to run the line. This feature is case sensitive. View the buffer of commands from the current and previous MATLAB sessions in the Command History.

Tab Completion

MATLAB completes the name of a function, variable, filename, structure, or handle graphics property if you type the first few letters and then press the **Tab** key. If there is a unique name, the name is automatically completed.

To use tab completion, you must have the tab completion preference selected.

For example, if you created a variable `costs_march`, type

```
costs
```

and press **Tab**. MATLAB completes the name, displaying

```
costs_march
```

Then press **Return** or **Enter** to run the statement. In this example, MATLAB displays the contents of `costs_march`.

If there is more than one name that starts with the letters you typed, press the **Tab** key again to see a list of the possibilities. For example, type

```
cos
```

and press **Tab**. MATLAB does not display anything, indicating there are multiple names beginning with `cos`. Press **Tab** again and MATLAB displays

```
cos        cosh        costfun
cos_tr     cosint      costs_march
```

The resulting list of possibilities includes the variable name you created, `costs_march`, but also includes functions that begin with `cos`. You can continue typing and press **Tab** again to further narrow the list of possibilities.

Tab Completion for Structures. For structures, type up to and including the period separator, and then press **Tab**. For example, type

```
mystruct.
```

and press **Tab** to display all fields of `mystruct`.

If you type a structure and include the start of a unique field after the period, pressing **Tab** completes that structure and field entry. For example, type

```
mystruct.n
```

and press **Tab**, which completes the entry `mystruct.name`, where `mystruct` contains no other fields that begin with `n`.

Note that the resulting lists from using tab completion might include files that are not valid commands, including private functions.

Arrow and Control Keys in the Command Window

Following is the list of arrow and control keys you can use in the Command Window. If the preference you select for "Command line key bindings" is **Emacs (MATLAB standard)**, you can also use the **Ctrl**+key combinations shown. See also general "Keyboard Shortcuts" on page 2-22.

Key	Control Key for Emacs (MATLAB standard) Preference	Operation
↑	**Ctrl+P**	Recall *p*revious line. See also "Command History" on page 3-30, which is a log of previously used functions, and "Keeping a Session Log" on page 3-14. Works only at command line.
↓	**Ctrl+N**	Recall *n*ext line. Works only at command line if you previously used the up arrow or **Ctrl+P**.
←	**Ctrl+B**	Move *b*ack one character.
→	**Ctrl+F**	Move *f*orward one character.
Ctrl+ →		Move *r*ight one word.
Ctrl+ ←		Move *l*eft one word.
Home	**Ctrl+A**	Move to beginning of command line.
End	**Ctrl+E**	Move to *e*nd of command line.
Ctrl+Home		Move to top of Command Window.
Ctrl+End		Move to end of Command Window.
Esc	**Ctrl+U**	Clear command line.

Key	Control Key for Emacs (MATLAB standard) Preference	Operation (Continued)
Delete	**Ctrl+D**	Delete character at cursor in command line.
Backspace	**Ctrl+H**	Delete character before cursor in command line.
	Ctrl+K	Cut contents (*k*ill) to end of command line.
Shift+Home		Highlight to beginning of command line.
Shift+End		Highlight to end of last line. Can start at any line in the Command Window.

Clearing the Command Window

Select **Clear Command Window** from the **Edit** menu to clear it. This does not clear the workspace, but only clears the view. Afterwards, you still can use the up arrow key to recall previous functions.

Function Alternative. Use clc to clear the Command Window. Similar to clc, the home function moves the prompt to the top of the Command Window.

Suppressing Output

If you end a line with a semicolon (;), MATLAB runs the statement but does not display any output when you press the **Enter** or **Return** key. This is particularly useful when you generate large matrices. For example, typing

```
A = magic(100);
```

and then pressing **Enter** or **Return** creates A but does not display the resulting matrix.

Paging of Output in the Command Window

If output in the Command Window is lengthy, it might not fit within the screen and will display too quickly for you to see it. Use the more function to control the paging of output in the Command Window. By default, more is off. When

you type more on, MATLAB displays only a page (a screen full) of output at a time. After the first screen displays, press one of the following keys.

Key	Action
Enter or **Return**	To advance to the next line
Space Bar	To advance to the next page
q	To stop displaying the output

You can scroll up in the Command Window to see input and output that no longer fit in the view.

Formatting and Spacing Numeric Output

By default, numeric output in the Command Windows is displayed as 5-digit scaled, fixed-point values. Use the text display preference to change the numeric format of output. The text display format affects only how numbers are shown, not how MATLAB computes or saves them.

Function Alternative

Use the format function to control the output format of the numeric values displayed in the Command Window. The format you specify applies only to the current session. More advanced alternatives are listed in the "See Also" section of the format reference page.

Examples of Formats

Here are a few examples of the various formats and the output produced from the following two-element vector x, with components of different magnitudes.

```
x = [4/3 1.2345e 6]

format short e
   1.3333e+000  1.2345e 006

format short
   1.3333     0.0000

format +
++
```

For a complete list and description of available formats, see the reference page for format. If you want more control over the output format, use the sprintf and fprintf functions.

Controlling Spacing

Use the text display preference or format function to control spacing in the output. Use

```
format compact
```

to suppress blank lines, allowing you to view more information in the Command Window. To include the blank lines, which can help make output more readable, use

```
format loose
```

Printing Command Window Contents

To print the complete contents of the Command Window, select **File -> Print**. To print only a selection, first make the selection in the Command Window and then select **File -> Print Selection**.

Specify printing options for the Command Window by selecting **File -> Page Setup**. For example, you can print with a header. For more information, see "Page Setup Options for Printing" on page 2-25.

Keeping a Session Log

The diary Function

The diary function creates a copy of your MATLAB session in a disk file, including keyboard input and system responses, but excluding graphics. You can view and edit the resulting text file using any word processor. To create a file on your disk called sept23.out that contains all the functions you enter, as well as MATLAB output, enter

```
diary('sept23.out')
```

To stop recording the session, use

```
diary('off')
```

Other Session Logs

There are two other means of seeing session information:

• The Command History, which contains a log of all functions executed in the current and previous sessions.

• The logfile startup option—see "Startup Options" on page 1-4.

Searching in the Command Window

You can search for a specified string that appears in the Command Window, where the string was either part of input you supplied, or output displayed by MATLAB. There are two search features for the Command Window:

- "Find Dialog" on page 3-15
- "Incremental Search" on page 3-18

After finding the text, you can copy and paste it to the prompt in the Command Window to run it, or into an M-file or other file.
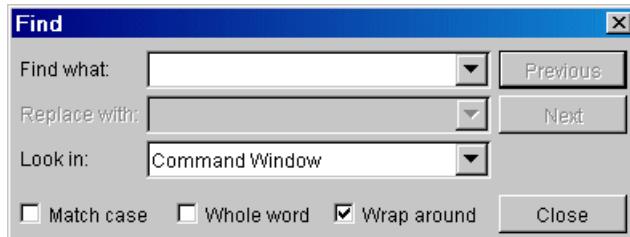
See also "Recalling Previous Lines" on page 3-8, "Tab Completion" on page 3-9, and "Arrow and Control Keys in the Command Window" on page 3-10 for techniques to reuse previous statements.

Find Dialog

To search for a string

1 Select **Find** from the **Edit** menu.

The **Find** dialog box appears. This is similar to the **Find** dialog box in the Editor/Debugger browser

2 In the **Find** dialog box, perform these steps to find all occurrences of the string you want to find in the Command Window:

a Type the string in the **Find what** field.

b Select Command Window from the **Look in** list box.

    c Limit the search using **Match case**, **Whole word**, or **Wrap around**. These settings are remembered for your next MATLAB session.

3 Click **Previous** or **Next** to find the previous occurrence or the next occurrence of the string in the Command Window, starting at the current cursor position.

The string is highlighted in the Command Window.
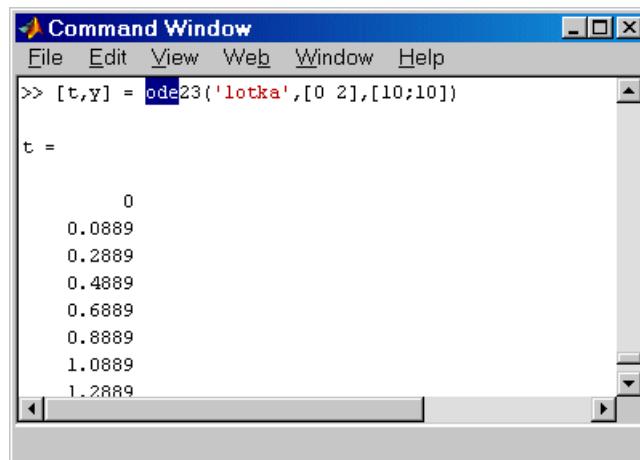
MATLAB beeps when a search for **Next** reaches the bottom of the Command Window, that is, the prompt, or when a search for **Previous** reaches the top of the Command Window. If you have **Wrap around** selected, it continues searching after beeping.

4 To find the next occurrence, click **Previous** or **Next** again.

Note that you can only search for strings currently displayed in the Command Window. To increase the amount of information maintained in the Command Window, increase the setting for "Command session scroll buffer size" in Command Window preferences, and do not clear the Command Window.

Example Using the Command Window Find Dialog

This example shows the **Find** dialog completed to find the string ode and shows the results, ode highlighted in the Command Window.
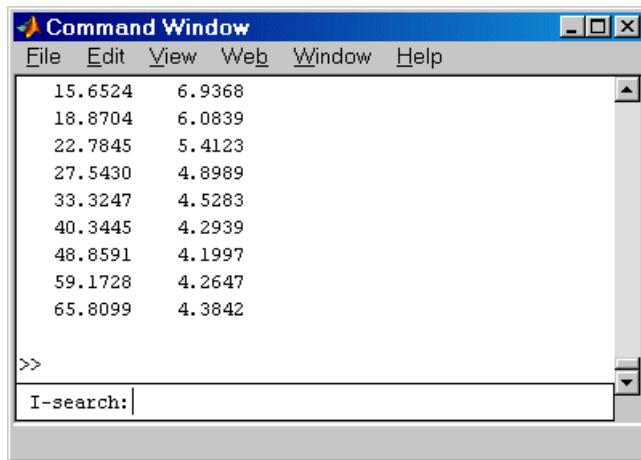
Incremental Search

With the incremental search feature, the cursor moves to the next or previous occurrence of the specified string in the Command Window. It is similar to the Emacs search feature.
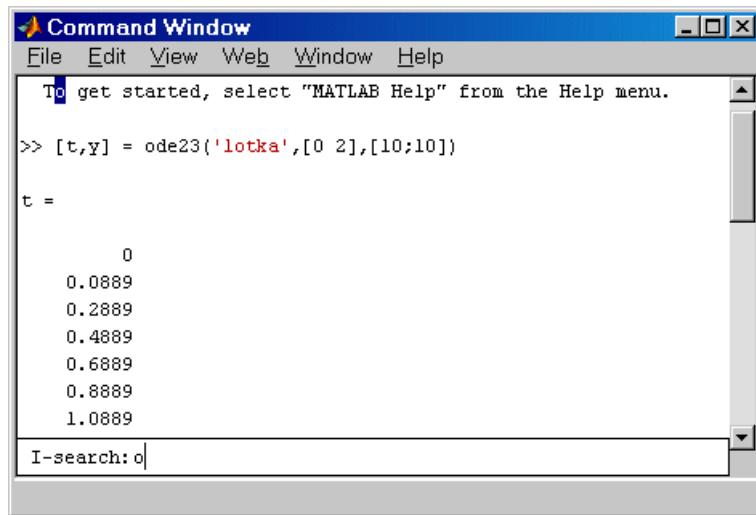
1 Position the cursor where you want the search to begin.

2 How you begin the incremental search depends on your setting for the "Command line key bindings" preference:

- Press **Ctrl+S** for **Emacs**, or
- Press **Ctrl+Shift+S** for **Windows**.

An incremental search field (**I-search:**) appears at the bottom of the Command Window.

3 In the **I-search** field, type the string you want to find. For example, type ode.

As you type the first letter, o, the first occurrence of that letter in the Command Window after the current cursor position is highlighted. In the example shown, the first occurrence of o is highlighted, the o in To in the startup message. Note that incremental search allows for case sensitivity— see "Case Sensitivity in Incremental Search" on page 3-20.

When you type the next letter, the first occurrence of the string becomes highlighted. In the example, when you add the letter d to the o so that the **I-search** field now has od, the od in ode becomes highlighted.

- If you mistype in the **I-search** field, use the **Back Space** key to remove the last letters and make corrections. Note that **Back Space** will first take you to any previous matches you just found using incremental search and will then remove the last letter in the **I-search** field.

- After finding the o, you can press **Ctrl+W** and the rest of the word that is found, in this case ode23, appears in the **I-search** field.

4 To find the next occurrence of ode in the Command Window, press **Ctrl+S**. To find the previous occurrence of the string, press **Ctrl+R**.

5 If you hear a beep, it either means that the string was not found, or it means you are at the end or beginning of the Command Window.

Press **Ctrl+S** or **Ctrl+R** again to wrap to the beginning or end of the file and continue the search. Either the next occurrence of the string will be highlighted, or you will hear another beep indicating the string is not in the Command Window.

6 To end the incremental search, press **Esc** or **Enter**, or any other non-character or number key.
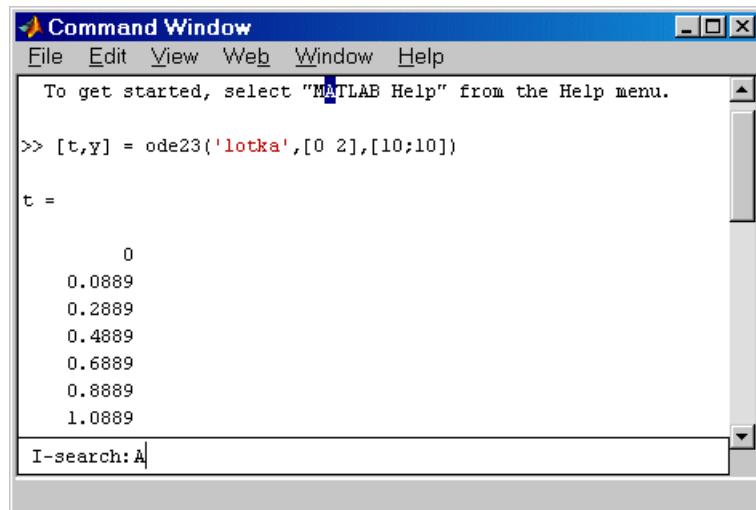
The **I-search:** field no longer appears in the window. The cursor is now located at the position where the string was last found, with the search string highlighted.

You can type **Ctrl+R** or **Ctrl+Shift+R** to display the **I-search** field to begin finding the previous occurrence rather than the next occurrence.

If you enter **Ctrl+S** or **Ctrl+R** after displaying the blank **I-search** field, the search term from your previous incremental search appears in the field. When you then use the **Back Space** key, you delete the entire previous search term, rather than just the last letter.

Case Sensitivity in Incremental Search

If you enter lowercase letters in the **I-search** field, for example, a, incremental search looks for both lowercase and uppercase instances of the letters, for example a and A. However, if you enter uppercase letters, for example, A, incremental search only looks for instances that match the cases you entered, for example, A.

If you enter mA in the I-search field, incremental search does not find MATLAB because the m in the **I-searc**h field is lowercase.

Running Programs

Running M-Files

Run M-files, files that contain code in the MATLAB language, the same way that you would run any other MATLAB function. Type the name of the M-file in the Command Window and press **Enter** or **Return**. The M-file must be in the MATLAB current directory or on the MATLAB search path—for details, see "Search Path" on page 5-18.

To display each function in the M-file as it executes, use the **Display** preference and check **Echo on**, or use the echo function set to on.

Interrupting a Running Program

You can interrupt a running program by pressing **Ctrl+C** or **Ctrl+Break** at any time.

On Windows platforms, you may have to wait until an executing built-in function or MEX-file has finished its operation. On UNIX systems, program execution will terminate immediately.

Running External Programs

The exclamation point character, !, is a *shell escape* and indicates that the rest of the input line is a command to the operating system. Use it to invoke utilities or run other programs without quitting from MATLAB. On UNIX, for example,

```
!vi yearlystats.m
```

invokes the vi editor for a file named yearlystats.m. After the program completes or you quit the program, the operating system returns control to MATLAB. See the functions unix and dos to run external programs that return results and status.

Opening M-Files

To open an M-file, select the file or function name in the Command Window, and then right-click and select **Open Selection** from the context window. The M-file opens in the Editor/Debugger.

Function Alternative

Use `open` or `edit` to open a file in the Editor. Use `type` to display the M-file in the Command Window.
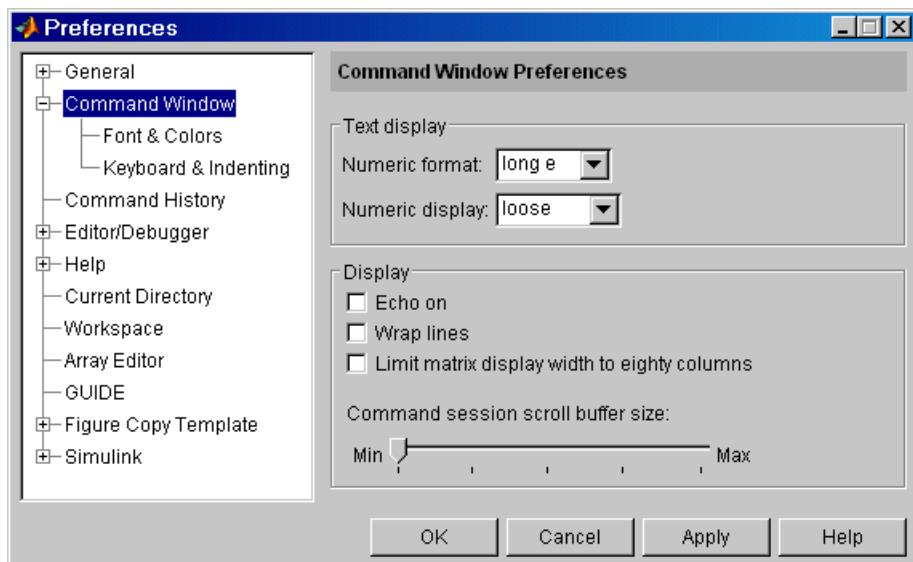
Examining Errors

If an error message appears when running an M-file, click the underlined portion of the error message, or position the cursor within the message and press **Ctrl+Enter**. The offending M-file opens in the Editor, scrolled to the line containing the error.

Preferences for the Command Window

Using preferences, you can, for example, specify the format for how numeric values are displayed, set echoing on automatically for each session, select font characteristics and the colors used for syntax highlighting, and choose among options for matching parentheses in syntax.

To set preferences for the Command Window, select **Preferences** from the **File** menu in the Command Window. The **Preferences** dialog box opens showing **Command Window Preferences**.

Set Command Window preferences for

- "Text Display and Display Preferences for the Command Window" on page 3-25
- "Font & Colors Preferences for the Command Window" on page 3-26
- "Keyboard and Indenting Preferences for the Command Window" on page 3-27

Text Display and Display Preferences for the Command Window

Text display

Specify how output appears in the Command Window:

- **Numeric format**—Output format of numeric values displayed in the Command Window. This affects only how numbers are displayed, not how MATLAB computes or saves them. The format reference page includes the list of available formats.
- **Numeric display**—Spacing of output in the Command Window. To suppress blank lines, use compact. To display blank lines, use loose. For more information, see the reference page for format.

Display

Set these options:

- **Echo on**—Select if you want commands running in M-files to display in the Command Window during the M-file execution. For more information, see the reference page for echo.
- **Wrap lines**—Select if you want a single line of input or output in the Command Window to instead break into multiple lines in order to fit within the current width of the Command Window. With this option selected, an entire line is visible without scrolling, and the horizontal scroll bar does not appear because it is not needed.
- **Limit matrix display width to eighty columns**—Select if you want MATLAB to display only 80 columns of matrix output, regardless of the width of the Command Window. Clear the check box if you make the Command Window wider than 80 columns and want matrix output to fill the width of the Command Window. See also the display reference page.

Command session scroll buffer size

Set how many lines are kept in the Command Window buffer. You can see these lines when you scroll up. A larger buffer means you can access more previous lines, but it requires more memory.

Font & Colors Preferences for the Command Window

Font

Command Window font preferences specify the characteristics of the font used in the Command Window. Select **Use desktop font** if you want the font in the Command Window to be the same as that specified for **General Font & Colors** preferences.

If you want the Command Window font to be different, select **Use custom font** and specify the font characteristics for the Command Window:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Colors

Specify the colors used in the Command Window:

- **Text color**—The color of nonspecial text. Special text uses colors specified for **Syntax highlighting**.
- **Background color**—The color of the background in the window
- **Syntax highlighting**—The colors to use to highlight syntax. If checked, click **Set Colors** to specify them. For a description of syntax highlighting, see "Syntax Highlighting and Parentheses Matching" on page 3-6.

Keyboard and Indenting Preferences for the Command Window

Command line key bindings

Specify the shortcuts to be used at the command line.

- **Emacs (MATLAB standard)**—Use the control keys listed in "Tab Completion" on page 3-9, which should be familiar to existing MATLAB users and Emacs users. For example, **Ctrl+A** moves the cursor to the beginning of the line.

- **Windows**—Use Windows standard control keys. For example, **Ctrl+A** selects entire contents of the Command Window.

Tab key

- **Tab size**—Number of spaces assigned to a tab stop when displaying output.

- **Enable up to n tab completions**—Check the box if you want to use tab completion when typing functions in the Command Window. Then enter a limit in the edit box. For example, if you enter 10, when you use the tab completion feature, MATLAB displays the list of possible completions if there are 10 or less. If there are more than 10, MATLAB displays a message stating there are more than 10 completions.

  Uncheck the box if you do not want to use the tab completion feature. MATLAB moves the cursor to the next tab stop when you press the **Tab** key, rather than completing a function.

Parentheses Matching Preferences

MATLAB alerts you to matches and mismatches in pairs of delimiters, that is, in parentheses ( ), brackets [ ], and braces { }, based upon the MATLAB language syntax rules.

Match parentheses while typing. Check the box if you want to be alerted to matches and mismatches in pairs of delimiters as you type them. Then choose how you want MATLAB to alert you to matches using **Show match with**. When you type a closing delimiter, MATLAB alerts you based on the option you choose:

- **Balance**—The cursor briefly moves to and highlights the matching delimiter.
- **Underline**—The delimiter you typed and its match are underlined briefly.
- **Highlight**—The delimiter you typed and its match are highlighted briefly.
- **Bold**—The delimiter you typed and its match are both bolded briefly.

Also choose how you want MATLAB to alert you to mismatches using **Show mismatch with**. When you type a closing delimiter that does not have an opening match, MATLAB alerts you based on the option you choose:

- **Beep**—MATLAB beeps.
- **Strikethrough**—The delimiter you typed is briefly crossed out.
- **Gray**—The delimiter you typed is briefly grayed out.
- **None**—There is no action.

Match parentheses on arrow key movement. Check the box if you want to be alerted to matches and mismatches in pairs of delimiters when you move the cursor over a delimiter using forward and back arrow keys. Then choose how you want MATLAB to alert you to matches using **Show match with**. When you move the cursor over a closing (or opening) delimiter, MATLAB alerts you based on the option you choose:

- **Underline**—Both delimiters in the pair are underlined briefly.
- **Highlight**—Both delimiters in the pair are highlighted briefly.
- **Bold**—Both delimiters in the pair are bolded briefly.

Also choose how you want MATLAB to alert you to mismatches using **Show mismatch with**. When you move the cursor over a delimiter that does not have a match, MATLAB alerts you based on the option you choose:
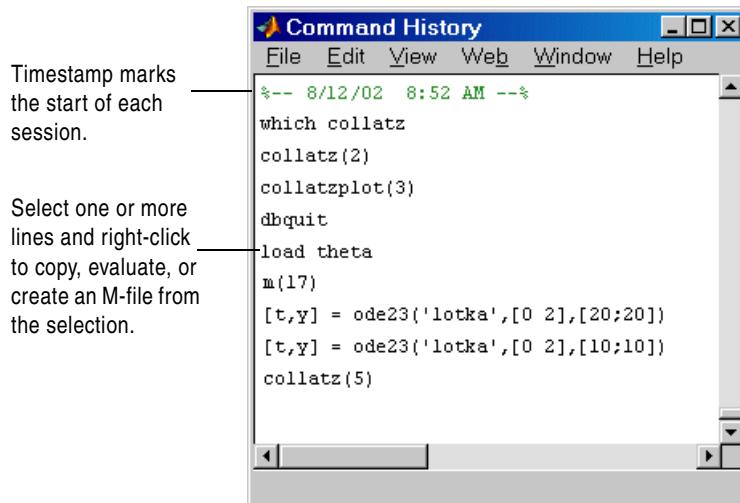
- **Beep**—MATLAB beeps.
- **Strikethrough**—The delimiter is briefly crossed out.
- **Gray**—The delimiter briefly appears in gray.
- **None**—There is no alert.

Command History

The Command History window appears the first time you start MATLAB. The Command History window displays a log of the statements most recently run in the Command Window. To show or hide the Command History window, use the **View** menu—see "Opening and Closing Desktop Tools" on page 2-10 for details.

Use the Command History window for

- "Viewing Statements in the Command History Window" on page 3-31
- "Running Statements from the Command History Window" on page 3-32
- "Copying Statements from the Command History Window" on page 3-32
- "Searching in the Command History" on page 3-33
- "Printing the Command History" on page 3-35
- "Preferences for Command History" on page 3-35

Timestamp marks the start of each session.

Select one or more lines and right-click to copy, evaluate, or create an M-file from the selection.

```
%-- 8/12/02  8:52 AM --%
which collatz
collatz(2)
collatzplot(3)
dbquit
load theta
m(17)
[t,y] = ode23('lotka',[0 2],[20;20])
[t,y] = ode23('lotka',[0 2],[10;10])
collatz(5)
```

Viewing Statements in the Command History Window

The log in the Command History window includes statements from the current session, as well as from previous sessions. The time and date for each session appear at the top of the history of functions for that session. Use the scroll bar or the up and down arrow keys to move through the Command History window.

The Command History file is history.m. Type prefdir in the Command Window to see the location of the file. The history.m file is loaded when MATLAB starts.

The Command History file is automatically saved throughout the MATLAB session according to the **Saving** preference you specified. You can choose to automatically exclude certain statements from being written to the Command History with the **Settings** preference. For details, see "Preferences for Command History" on page 3-35.

Deleting Entries in the Command History Window

Delete entries in the Command History window when you feel there are too many and it is inconvenient finding the ones you want. All entries remain until you delete them.

To delete entries in the Command History window, select an entry, or **Shift**+click or **Ctrl**+click to select multiple entries, or use **Ctrl+A** to select all entries. Then right-click and select one of the delete options from the context menu:

- **Delete**—Deletes the selection
- **Clear Command History**—Deletes all entries in the Command History window

You can also delete a selection by pressing the **Delete** key.

Another way to clear the entire history is by selecting **Clear Command History** from the **Edit** menu.

After deleting entries from the Command History, you will not be able to access those statements in the Command Window using features for "Recalling Previous Lines".

Running Statements from the Command History Window

Double-click any entry (entries) in the Command History window to execute that statement(s). For example, double-click `edit myfile` to open `myfile.m` in the Editor. You can also run an statement by right-clicking on it and selecting **Evaluate Selection** from the context menu, by selecting a statement and pressing **Enter** or **Return**, or by copying the entry to the Command Window, as described in the next section.

Copying Statements from the Command History Window

Select an entry, or **Shift**+click or **Ctrl**+click to select multiple entries, or use **Ctrl+A** to select all entries. Then you can do any of the following.

Action	How to Perform the Action
Run the statements in the Command Window	Copy the selection to the clipboard by right-clicking and selecting **Copy** from the context menu. Paste the selection into the Command Window. In the Command Window, edit the statement(s) if desired, and press **Enter** or **Return** to execute the statement(s). Alternatively, drag the selection to the Command Window, edit the statements if desired, and press **Enter** or **Return** to execute the statements. For this method, the Command History preference **Allow Drag and Drop editing** must be selected and both tools must be in the desktop.

Action	How to Perform the Action (Continued)
Copy the statement(s) to another window	Copy the selection to the clipboard by right-clicking and selecting **Copy** from the context menu. Paste the selection into an open M-file in the Editor or any application.
Create an M-file from the statement(s)	Right-click the selection and select **Create M-File** from the context menu. The Editor opens a new M-file that contains the statement(s) you selected from the Command History window. Also, you can drag selected statements from the Command History to an open file in the Editor if both tools are in the desktop.

Searching in the Command History
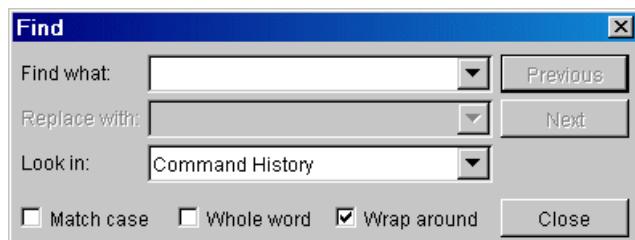
You can search for a specified string that appears in the Command History.

After finding the string, you can copy and paste it into an M-file or other file, or you can right-click and select **Evaluate Selectio**n to run the statement containing that string.

To search for a string

1 Select **Find** from the **Edit** menu.

The **Find** dialog box appears. This is similar to the **Find** dialog box in the Command Window.

2 Complete these steps in the **Find** dialog box to find all occurrences of the string you are looking for:

   a Type the string in the **Find what** field.

   b Select Command History from the **Look in** list box.

   c Limit the search as needed, using **Match case**, **Whole word**, or **Wrap around**. These settings are remembered for your next MATLAB session.

3 Click **Previous** or **Next**. to find the previous occurrence or the next occurrence of the string in the Command History, starting at the current cursor position.
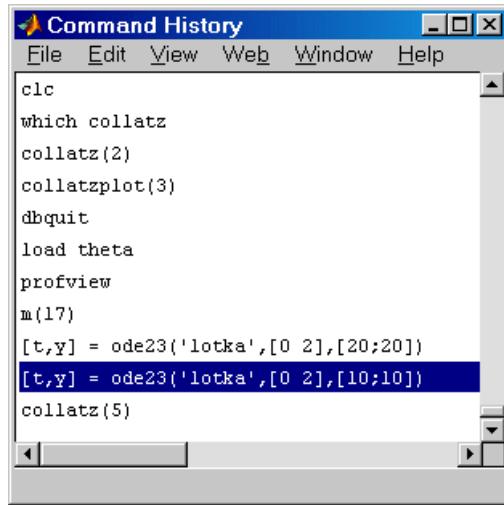
The entry containing the string is highlighted in the Command History.

When the search reaches the bottom or top of the Command History, it beeps, but if you have **Wrap around** selected, it continues searching.

4 To find the next occurrence, click **Previous** or **Next** again.

Example Using the Command History Find Dialog

This example shows the results of using find for ode.

Printing the Command History

To print the contents of the Command History window, select **File -> Print**. Specify options for printing by selecting **File -> Page Setup**. For example, you can print the history with a header. For more information, see "Page Setup Options for Printing" on page 2-25.

Preferences for Command History

Using Command History preferences, you can choose to exclude statements from the Command History and specify how often to save the Command History.

To set preferences for the Command History, select **Preferences** from the **File** menu in the Command History window. The **Preferences** dialog box opens showing **Command History Preferences**.

You can set preferences for

- "Settings" on page 3-36
- "Saving" on page 3-37

Settings

Set the following options. Note that if you choose to exclude statements from the Command History, you cannot access them using Command Window features such as "Recalling Previous Lines" on page 3-8.

Save exit/quit commands. Select the check box if you want `exit` and `quit` commands to be saved to the Command History.

Save consecutive duplicate commands. Select the check box if you want consecutive executions of the same statement to be saved to the Command History.

For example, with this option selected, if you run `magic(5)`, and the next statement you run is also `magic(5)`, the Command History saves two consecutive entries for `magic(5)`. With this option cleared, for the same example, the Command History saves one entry for `magic(5)`. If you then run `magic(10)`, the Command History saves two entries, `magic(5)` followed by `magic(10)`.

Save commands entered at input prompt. Select the check box if you want input supplied at the prompt following the `input` function to be included in the Command History. For example, with this option selected, if you type

```
input('Enter maximum: ')
```

MATLAB displays

```
Enter maximum:
```

If you then enter `500`, the Command History saves two statements

```
input('Enter maximum: ')
500
```

With the check box cleared, the Command History saves only one statement, `input('Enter maximum: ')`.

Allow Drag and Drop editing. If you select this check box, you can drag selected statements from the Command History to the Command Window or other

windows docked in the desktop. If you clear this check box, when you click on a selected entry, it becomes cleared.

Saving

Use **Saving** preferences to specify how often to automatically save the Command History during a MATLAB session.

Save history file on quit. Select this option to save the Command History when you end the MATLAB session. If the session does not end via a normal termination, that is, exit or quit function, **File -> Exit MATLAB**, or the MATLAB desktop close box, the history file will not be saved for that session.

Save after n commands. Select this option to save the Command History after n statements are added to the file. For example, if you select the option and set n to 10, after every 10 statements are added to the history file, the file will be automatically saved. Use this option if you don't want to risk losing entries to the saved history due to an abnormal termination.

Don't save history file. Select this option if you don't want to save the history file. This feature is useful when multiple users share the same machine and don't want other users to view the statements they have run.

4

Getting Help

The MathWorks provides online help and demonstrations for all products through the Help browser, help functions, and other methods.

Types of Documentation

The Help browser and help functions provide access to the following types of information. The icons shown here appear in the Help browser contents listing to help you quickly identify documentation by type.

Icon	Type of Documentation	Description and When to Use
▶	Getting Started	Primarily aimed at users new to a product, it contains instructions for a product's main features. Review Getting Started documentation before you begin using a product or feature for the first time. Then, to learn more, go to the user guides or reference pages.
📋	Release Notes	An overview of new products and features in a release, it also includes upgrade information and any known problems and limitations. Review the Release Notes for all your products when you first start using a new release.
📙 or 📘	Product	MATLAB and toolboxes use orange book icons 📙, while Simulink and Blocksets and related products use blue book icons 📘.
💡	Index of Examples	Accessible via the Help browser contents listing, this is an index of the major examples included in the user guide documentation.
📄	User Guides	This user guide material is comprehensive, containing overviews as well as detailed instructions. Consult it after reviewing Getting Started material.
📄	Reference Pages	Each function has a reference page that provides the syntax, description, examples, and other information for that function. It includes links to related functions and additional information. Reference pages are also provided for blocks and properties. Use a function reference page to learn details about the function.

Icon	Type of Documentation	Description and When to Use (Continued)
	Printable Documentation	Most products provide the online documentation in a printable format, PDF. Access PDF files from the Help browser and print them from your PDF reader, such as Adobe Acrobat.
	Product Pages	Available on the MathWorks Web site, a product page contains the latest product information, such as system requirements.
	Online Knowledge Base	This is the MathWorks Technical Support searchable knowledge base, maintained on the MathWorks Web site. It provides solutions to questions posed by users.
	Demos	MathWorks products come with examples that demonstrate features of the product. Many of the demos actually run code. Use the Help browser **Demos** tab to access demos for the products you have installed.
n.a.	M-File Help	Get M-file help in the Command Window to quickly access basic information for a function. It provides a brief description of a function and its syntax. It is called M-file help because the text of the help is a series of comments at the top of the M-file for a function.

Using the Help Browser
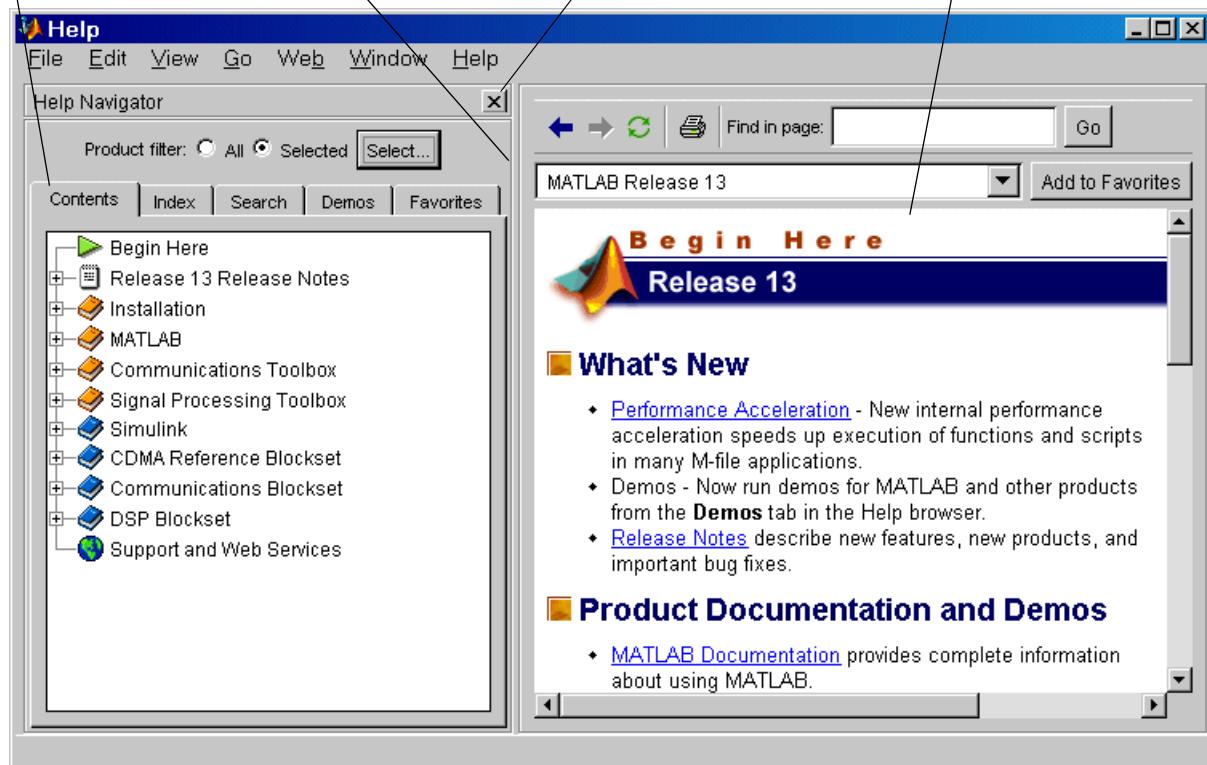
Use the Help browser to search and view documentation and demonstrations for MATLAB and your other MathWorks products. The Help browser is a Web browser integrated into the MATLAB desktop that displays HTML documents.

To open the Help browser, click the help button ? in the toolbar, or type helpbrowser in the Command Window. You can also access the Help browser by selecting **Help** from the **View** menu or by using the **Help** menu in any tool. The Help browser opens.

Tabs in the **Help Navigator** pane provide different ways to find information.   View documentation in the display pane.

Drag the separator bar to adjust the width of the panes.   Use the close box to hide the pane.

The Help browser consists of two panes:

- The Help Navigator, on the left, which you use to find information. It includes a **Product Filter** and **Contents**, **Index**, **Search**, **Demos**, and **Favorites** tabs. For more information, see "Finding Information with the Help Browser" on page 4-7.

- The display pane, on the right, which is for viewing documentation and demonstrations.
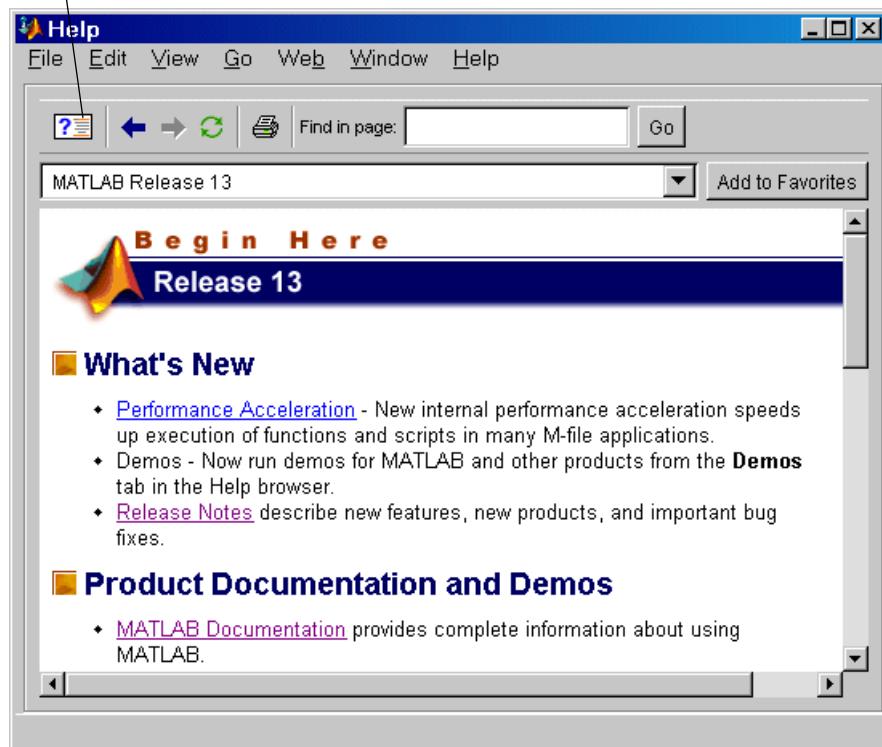
Resizing the Help Browser

To adjust the relative width of the two panes, drag the separator bar between them. You can also change the font in either of the panes—see "Help Fonts Preferences—Specifying Font Name, Style, and Size" on page 4-36.

Once you find the documentation you want, you can close the **Help Navigator** pane so there is more screen space to view the documentation itself. This is shown in the following figure. To close the **Help Navigator** pane, click the close box ☒ in the pane's upper right corner or select **View -> Help View Options -> Show Help Navigator** from the menu, which clears the checkmark it. To open the **Help Navigator** pane from the display pane, click the help navigator button ？☰ in the upper left corner of the Help browser, or select **View -> Help View Options -> Show Help Navigator**.

To show only the display pane, as in this illustration, click the close box at the top right of the **Help Navigator** pane.

Click this button to show the **Help Navigator** pane.

Finding Information with the Help Browser

Use the Help Navigator, the left pane in the Help browser, to find information. These sections describe the main features:

- "Using the Product Filter" on page 4-7—Show documentation only for specified products.
- "Viewing the Contents Listing in the Help Browser" on page 4-10—View an expandable table of contents for documentation.
- "Finding Documentation Using the Index" on page 4-13—Use keywords to find information.
- "Searching Documentation" on page 4-15—Find documentation using full-text and other forms of search.
- "Running Demonstrations" on page 4-23—View the demonstrations available and run them.
- "Favorites" on page 4-27—Designate favorite pages for later use.

Using the Product Filter

Use the **Product filter** in the **Help Navigator** to show documentation only for the products you specify. It is especially useful when you want to limit the search results to the most relevant ones.

To show documentation for all MathWorks products installed on your system, select **All**.

To show only a subset of the documentation for MathWorks products installed on your system, set the **Product filter** to **Selected**, which results in the following:

- The **Contents** listing shows only the subset of products you specify.
- The **Index** shows only index terms for the subset of products you specify.
- The **Search** feature only looks through the subset of products you specify.

To specify the subset of products, click the **Select** button. The **Help Product Filter** dialog box opens.

A checkmark appears for all products whose documentation is currently displayed in the Help Navigator. Make changes to the selected products and click **OK**. Then, with the **Product filter** set to **Selected**, the Help Navigator only shows documentation for those products you specified.

The product filter settings are remembered for your next MATLAB session.

Note that the Product filter does not apply to Demos and to the Online Knowledge Base search type. If you clear the selection for Release Notes, it applies to the Release Notes document that provides the overview for a release, not to the Release Notes for an individual product.

Example Using the Product Filter
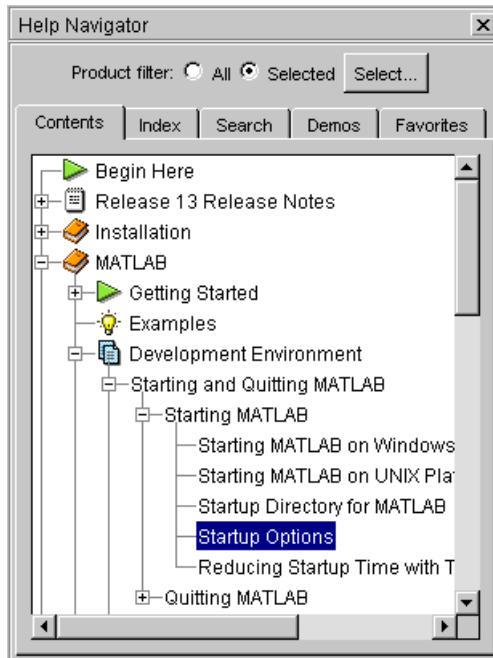
If you do a search and know the information you are seeking is in MATLAB or the Communications Toolbox:

1 In the **Help Product Filter**, click **Clear All** and then select `MATLAB` and `Communications Toolbox`.

2 In the Help Navigator, set the **Product filter** to **Selected**.

The **Contents** only shows MATLAB and the Communications Toolbox documentation, the **Index** only shows entries for MATLAB and the Communications Toolbox, and the **Search** feature only looks in and shows results for MATLAB and the Communications Toolbox.

Viewing the Contents Listing in the Help Browser

To list the titles and tables of contents for all product documentation, click the **Contents** tab in the **Help Navigator** pane.

Features of the **Contents** listing are

- "Navigating in the Contents Listing" on page 4-11
- "Icons in the Contents Listing" on page 4-11
- "Product Roadmap" on page 4-11
- "Product Pages" on page 4-12
- "Synchronize Contents Listing with Display Pane" on page 4-12

Navigating in the Contents Listing

In the **Contents** listing, you can

- Click the **+** to the left of an item to show the first page of that document or section in the display pane and expand the listing for that item in the Navigator pane.
- Click the **-** to the left of an item, or double-click the item, or press the left arrow key to collapse the listings for that item.
- Select an item to show the first page of that document or section in the display pane.
- Double-click an item or press the right arrow key to expand the listing for that item and show the first page of that document or section in the display pane.
- Use the down and up arrow keys to move through the list of items.

The **Contents** listing shows documentation for all products installed on your system, or only shows documentation for specified products if you have the **Product filter** set to **Selected**.

Icons in the Contents Listing

Icons for entries in the top levels of the Contents listing represent what type of documentation it is. This lets you quickly find the kind of information you need for a product. See the legend for icons in "Types of Documentation" on page 4-2.

Product Roadmap

When you select a product in the **Contents** pane (any entry with a book icon), such as MATLAB or the Communications Toolbox, a *roadmap* of the documentation for that product appears in the display pane. Some examples of what the roadmap might contain are links to

- The key documentation
- An index of major documentation examples
- The PDF version of the documentation, which is suitable for printing
- Related products

Product Pages

After expanding the listing for a product in the **Contents** pane, the last entry is **Product Page (Web)**. This allows you to link to the MathWorks Web site for the latest information about that product.

Synchronize Contents Listing with Display Pane
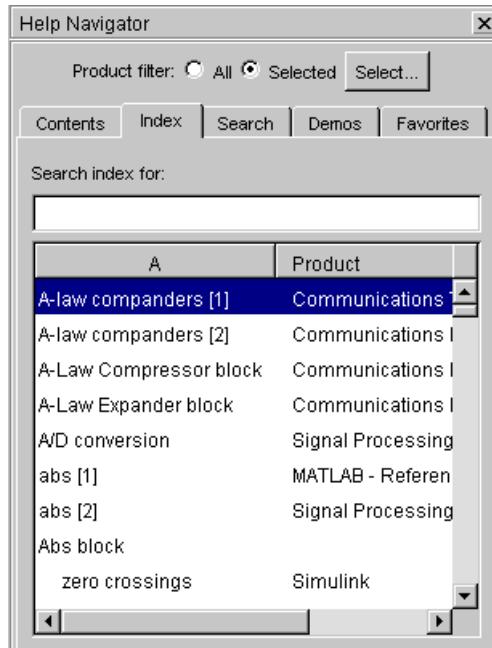
By default, the topic highlighted in the **Contents** pane matches the title of the page appearing in the display pane. The **Contents** listing is said to be synchronized with the displayed document. This feature is useful if you access documentation with a method other than the **Contents** pane, for example, using a link in a page in the display pane. With synchronization, you know what book and section the displayed page is part of. Note that synchronization only applies to the major headings in a document. For pages that begin with lower level headings, the **Contents** listing does not synchronize.

You can turn off synchronization. To do so, use preferences—see "General—Synchronizing the Contents Pane with the Displayed Page" on page 4-35.

Note that synchronization only applies to the **Contents** pane. The page shown in the display pane does not necessarily correspond to the selection in the **Search**, **Index**, **Demos**, or **Favorites** tabs. However, if you use the **Search**, **Index**, or **Favorites** page to display a page and then return to the **Contents** pane, the **Contents** pane synchronizes with the displayed page.

Finding Documentation Using the Index

To find specific index entries (selected keywords) in the MathWorks documentation for your products, use the **Index** in the **Help Navigator** pane.

1 Set the **Product filter** to **All** or **Selected**.

2 Click the **Index** tab.

3 Type a word or words in the **Search index for** field. As you type, the index displays matching entries and their subentries (indented). It might take a few moments for the display to appear. The index is not case sensitive.

The product and title of the document that includes the matching index entry are listed next to the index entry, which is useful when there are multiple matching index entries. You might have to make the **Help Navigator** pane wider to see the product and document.

4 Select an index entry from the list to display that page.

The page appears in the display pane, scrolled to the location where the index entry appears.

5 To see more matching entries, scroll through the results.

Tips for Using the Index
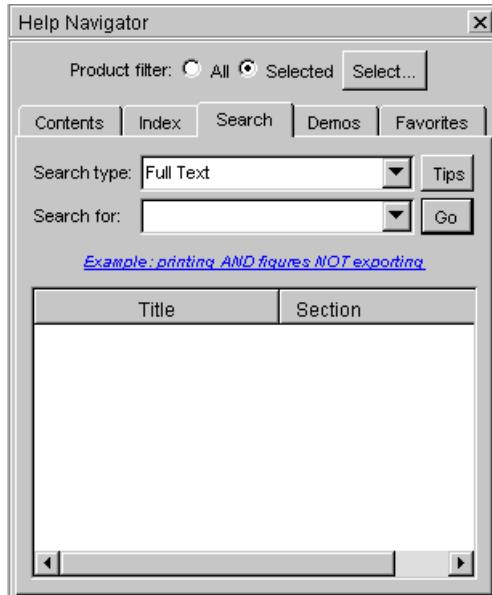
- Set the **Product filter** to **All** to see entries for all products.
- Set the **Product filter** to **Selected** and click **Select** to see entries only for selected products.
- Type a different term or reverse the order of the words you type. For example, if you are looking for `creating M-files`, type `M-files` and look for the subentry `creating`.
- After selecting an entry, search for the term in the displayed page using the **Find in page** field, located at the top of the display pane.
- See the product and section of documentation associated with the entry using the second column in the **Help Navigator** pane. You might need to make the pane wider to see it.
- Try the Search tab—for instructions, see "Searching Documentation" on page 4-15.

Searching Documentation

- "Using the Search Pane" on page 4-15
- "Examples of Search" on page 4-18
- "More About Search" on page 4-20
- "Tips for Using Search" on page 4-21

Using the Search Pane

To look for a specific word or phrase in the documentation, use the **Search** tab in the **Help Navigator** pane.

1 To limit or expand the products whose documentation is searched, set the **Product filter**.

2 Click the **Search** tab.

3 Select a **Search type**:

`Full Text`	Searches through all the text in the documentation.
	This can result in a large number of results. If so, try the techniques described in "Want Fewer Results" on page 4-21.
`Document Titles`	Searches through the headings in the documentation.
	This is the best way to search for overview information or for a broad topic.
`Function Name`	Searches for the reference page for the specified function, block, or property.
	This search type is the equivalent of running the `doc` function.
`Online Knowledge Base`	Connects to the MATLAB Web site and searches through the Technical Support information. It does not use the **Product filter**.

4 Type the word or words you want to look for in the **Search for** field, and click **Go** (or press **Enter** or **Return**). You can use Boolean operators between the words. For the `Function Name` search type, you can only enter a single word for a function.

The documents containing the exact search words are listed, grouped by product. For each result, the **Title** and **Section** of the document are shown so you can see the context for the search words. If you cannot see the **Section**, make the Navigator pane wider.

For the `Online Knowledge Base` search type, the results appear in the display pane.

5 Select an entry from the list of results.
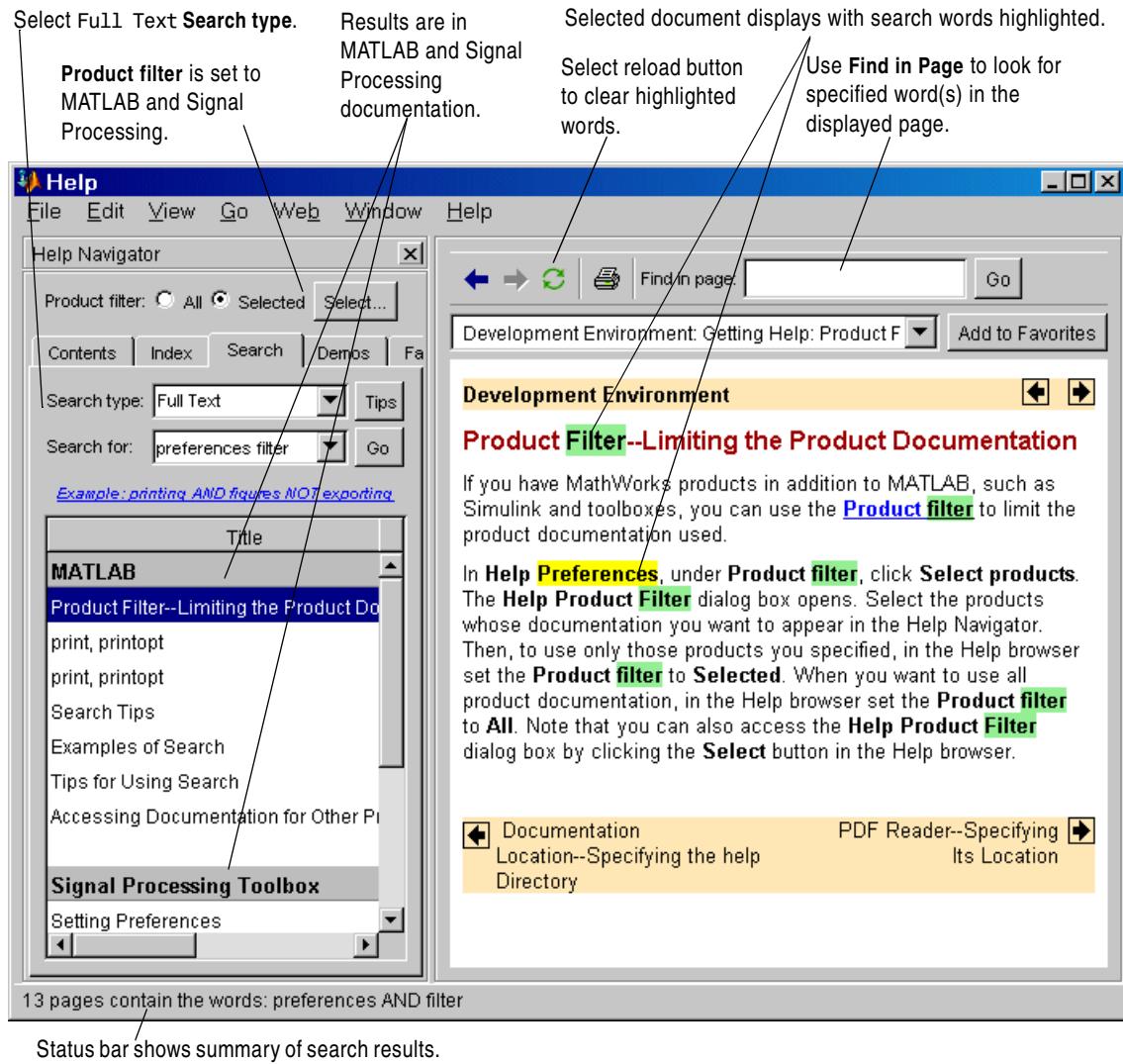
The selected page appears in the display pane with all occurrences of the search words highlighted (only for Full Text and Document Titles search types), with a different color for each word from the search term.

Search words remain highlighted until you view another page or until you click the page reload button ⟳ in the toolbar.

6 To see variants of a word in the page, for example `print` instead of `printing`, use the **Find in page** field.

Examples of Search

Full Text Search. This search looks through MATLAB documents for the words "preferences" and "filter".

Select `Full Text` **Search type**.

Product filter is set to MATLAB and Signal Processing.

Results are in MATLAB and Signal Processing documentation.

Selected document displays with search words highlighted.

Select reload button to clear highlighted words.

Use **Find in Page** to look for specified word(s) in the displayed page.

Status bar shows summary of search results.
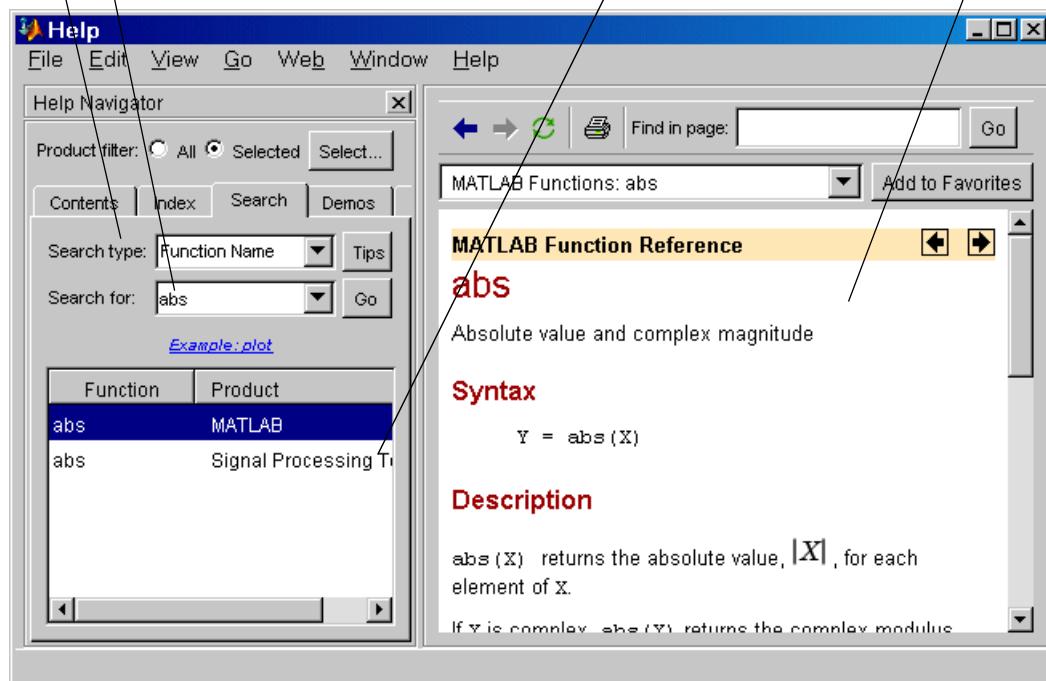
Function or Block Name Search. This shows how to find the reference page for the abs function.

Set the **Search type** to Function Name.

Enter the function or block name, for example, abs.

The reference page for the function or block displays.

If the function exists in multiple products, you can select the reference page for each product.

More About Search

Here are some guidelines that search uses.

- Insignificant words (a, an, the, of) are ignored.
- Search is not case sensitive.
- You cannot enter quotation marks around words to find exact phrases and you cannot use wildcards.
- Search does not find operators and special characters, such as +, so instead use the **Index**.
- If you are searching for information about an option, try including the hyphen (-) before the option, for example, `save -append`.
- The full text search type does not search the Online Knowledge Base.
- You cannot search for words in demos.
- If you search for a function but there are no results, try `help function` at the Command Window prompt. There are a few functions for which there is only M-file help.
- If you search for a function name that is used in multiple products (called an overloaded function), all the reference pages are listed. Use the **Product** column to identify the reference page for the product of interest.

Boolean Operators in Search. The search automatically performs a Boolean AND for multiple words. In the example `font preferences`, it finds all pages that have both the word `font` and the word `preferences`, although the page might not necessarily have the exact phrase "`font preferences`".

You can refine the search by including the Boolean operators AND, OR, and NOT between words. The operators must be in all capital letters and there must be a space before and after each operator. The Boolean operators are evaluated in left to right order.

Example Using Boolean Operators in Search. Type

```
print OR printing AND figure NOT exporting
```

to find all pages that contain the words `print` and `figure`, or `printing` and `figure`, but only if the page does not contain the word `exporting`. At the top of the results list are any pages that contain all the ANDed and ORed words in the page title.

Tips for Using Search

- "Want Fewer Results" on page 4-21
- "Want More Results" on page 4-22
- "Redoing a Search" on page 4-22
- "For More Information" on page 4-23

Want Fewer Results. If you see too many results for the search to be useful, try the following.

Problem	Try This
Too many products included in results	1. Set the **Product filter** to **Selected**. Product filter: ○ All ◉ Selected Select... 2. Click **Select** to specify the products whose documentation you want to search. For more information, see "Using the Product Filter" on page 4-7.
Search word is just mentioned in the results	Try the **Index** tab to see more important entries, or Change the **Search type** to • Document Titles—Find section titles that include the search term. • Function Name—Go to the reference page for a function, block, or property.

Problem	Try This
Too many irrelevant results	Type more than one word in the **Search for** field.
	Use Boolean operators (in all capitals), for example, `printing AND figures NOT exporting`.
Page is not relevant	Look at the **Section**, the second column in the search results list. If you cannot see it, make the pane wider.
	It tells you which section the resulting page is in, providing context for the result.

Want More Results. If you do not get many results, try the following.

Problem	Try This
No results for the product	If the **Product Filter** is set to **Selected**, be sure the product of interest is selected. Click **Select** to specify the products.
	For more information, see "Using the Product Filter" on page 4-7.
No results but you know the word is there	Try variations of the search words with an `OR` between the words.
	For example, search for `preference OR preferences` to find all pages that contain either the word `preference` or the word `preferences`.

Redoing a Search. If you run a search that is not what you want, you can stop it before it completes. Just enter the new search terms in the **Search for** field, make changes to the product filter and search type, and press **Go**. This stops the existing search and runs a search for the new terms.

For More Information. See also

- "More About Search" on page 4-20
- "Examples of Search" on page 4-18
- "Searching Documentation" on page 4-15

Running Demonstrations

Demos are installed even if you do not install the online help files. There are many different types of demonstrations. Some play a movie file showing how a feature works. Others run an M-file or tool, and can be interactive, requiring your input. The M-file code for most demonstrations is available for you to view and copy for use in your own applications.

See also **Examples** in the **Contents** pane for each product. These examples are similar to demos but are integrated in the documentation.

Topics for demos are

- "Viewing Demos" on page 4-23
- "Using Demos" on page 4-25
- "Notes" on page 4-26
- "See Also" on page 4-26

Viewing Demos

To see the available demos for the products you have installed:

1 Click the **Demos** tab in the **Help Navigator**.
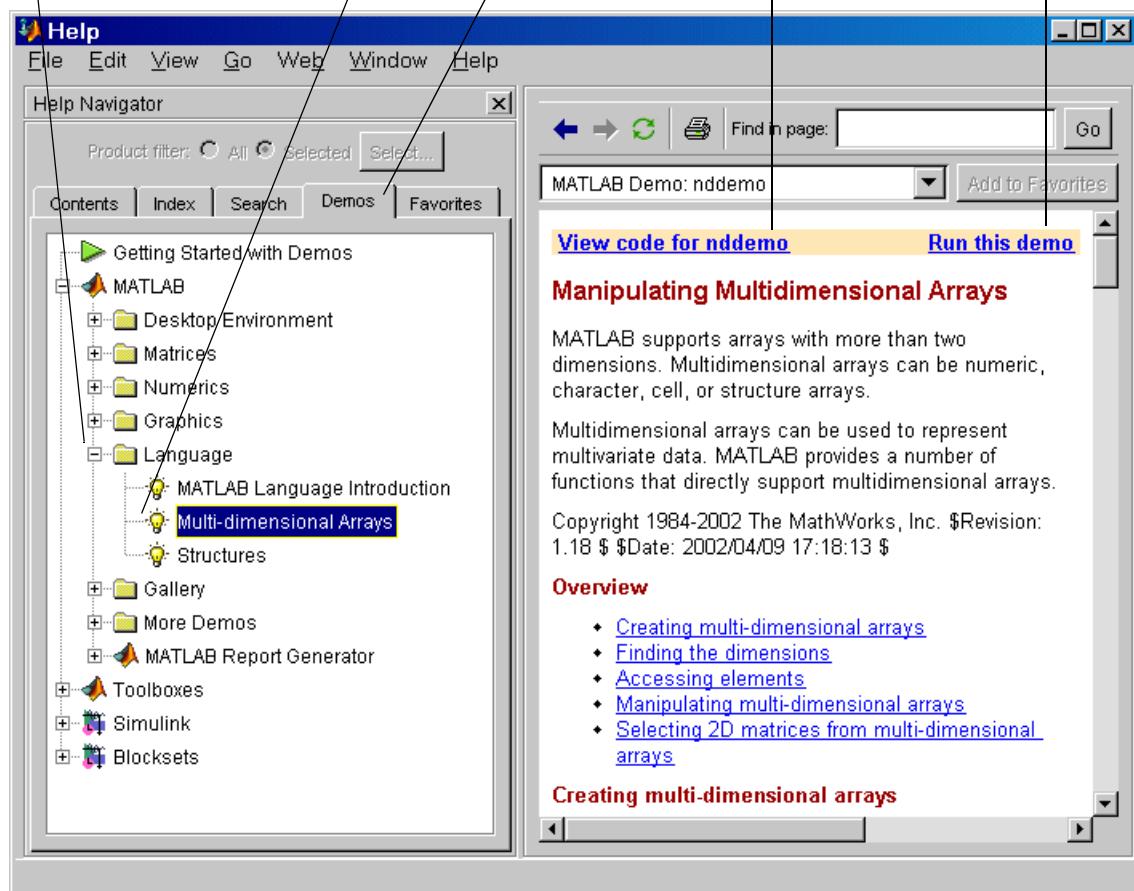
  You can also access demos from the **Start** button, and from the **Help** menu for some tools.

2 Click the **+** for a product area to list the products or categories that have demos.

3 Click **+** for a product or product category to list its demos.

4 Select a specific demo to use it. Information about the demo, including instructions for running it, appears in the display pane.

Access demos for all installed products using the **Demos** tab.

Click this link to run the demo.

Expand the listing for a product
and category to see its demos.

Select a demo.

The code for the demo is in the specified file. Click
this link to view the M-file code in your editor.
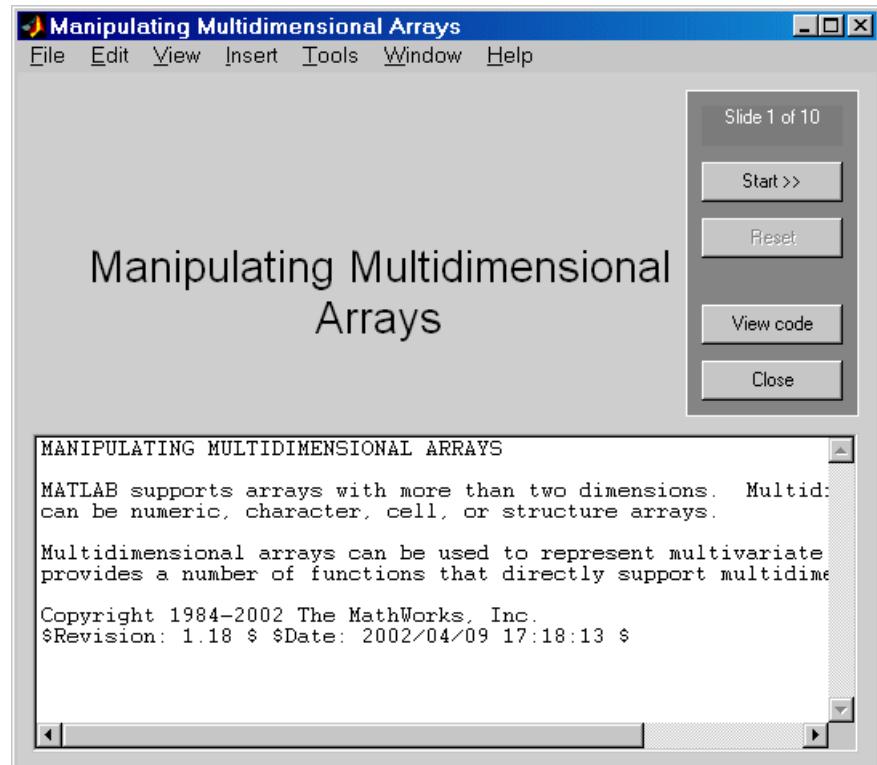
Some demos are not available from the Help browser. See for instructions to
access and run them.

Using Demos

After you select a demo, information about it appears in the display pane. For the example shown, **MATLAB -> Language -> Multi-dimensional Arrays**, you can

- View the source code (M-file) for the demo—click the **View code** link on the top left. For the example shown, it is nddemo.m. The M-file opens in the MATLAB Editor.
- Run the demo—click the link at the top right, **Run this Demo**. You can instead double-click the demo in the Navigator pane to run it.

When you run nddemo, the following appears. Follow the instructions shown to continue running the demo. In this example, click **Start>>**.

Notes

Some Help browser features do not apply to demos:

• You cannot use the **Search** tab to look for words or code contained in the demos. You can use the Current Directory browser or `lookfor` function to search for code in M-file demo files.

• You cannot add most demos to the **Favorites** tab.

• The **Product Filter** does not apply to demos.

• You cannot select **View -> Help View Options -> Page Source** for some demos.

Function Alternative

To view the **Demos** in the Help browser, type `demo` in the Command Window. You can go directly to the demos for a specific product. For example

```
demo toolbox signal
```

opens the **Demos** listing for the Signal Processing Toolbox.

To run one of the demos, you can type the demo name at the command line. For example, type

```
vibes
```

to run a visualization demonstration showing an animated L-shaped membrane.

Running Playshow Demos. To run playshow demos from the command line, type `playshow` followed by the demo name.

For example, if you type `quake`, the demo does not run. View the H1 line for `quake.m`, that is, the first comment line. It begins with two comment symbols (`%%`), indicating that `quake` is a playshow demo.

```
%% Loma Prieta Earthquake
```

Therefore, type `playshow quake` to run the demo.

See Also

See the Examples listing for each product in the **Contents** pane of the Help browser. It lists the major examples included in the documentation, some of which include sample code you can view, run, or copy.

Favorites

Adding Favorites

To designate a document page as a favorite (that is, to bookmark it), do one of the following:

- In the **Contents** listing, right-click an item and select **Add to Favorites** from the context menu.
- In the Help browser index or search results list, right-click an entry and select **Add to Favorites** from the context menu.
- While a page is open in the display pane, click the **Add to Favorites** button in the display pane toolbar.

The favorites file is matlab_help.hst. Type prefdir in the Command Window to see the location of the file. The matlab_help.hst file is loaded when MATLAB starts, and is overwritten with any changes you make during a session when you close MATLAB.

Going to Favorites

Click the **Favorites** tab in the **Help Navigator** to view a list of pages you previously designated as favorites (also known as bookmarks). It is the rightmost tab—to see it, you might need to use the separator bar to make the Navigator pane wider. From the list of favorites you can

- Select an entry—That page appears in the display pane.
- Remove an entry—Right-click the item in the favorites list and select **Remove** from the context menu, or press the **Delete** key.
- Rename an entry—Right-click the item in the favorites list and select **Rename** from the context menu. Type over the existing name to replace it with a new name.

Viewing Documentation in the Display Pane

After finding documentation with the Help Navigator, view the documentation in the display pane. The features available to you while viewing the documentation are

- "Browsing to Other Pages" on page 4-29
- "Following Links" on page 4-30
- "Revisiting Pages" on page 4-30
- "Finding Terms in Displayed Pages" on page 4-30
- "Copying Information" on page 4-31
- "Evaluating a Selection" on page 4-31
- "Viewing the Page Source (HTML)" on page 4-31
- "Viewing Web Pages and Other Documents" on page 4-31

Browsing to Other Pages
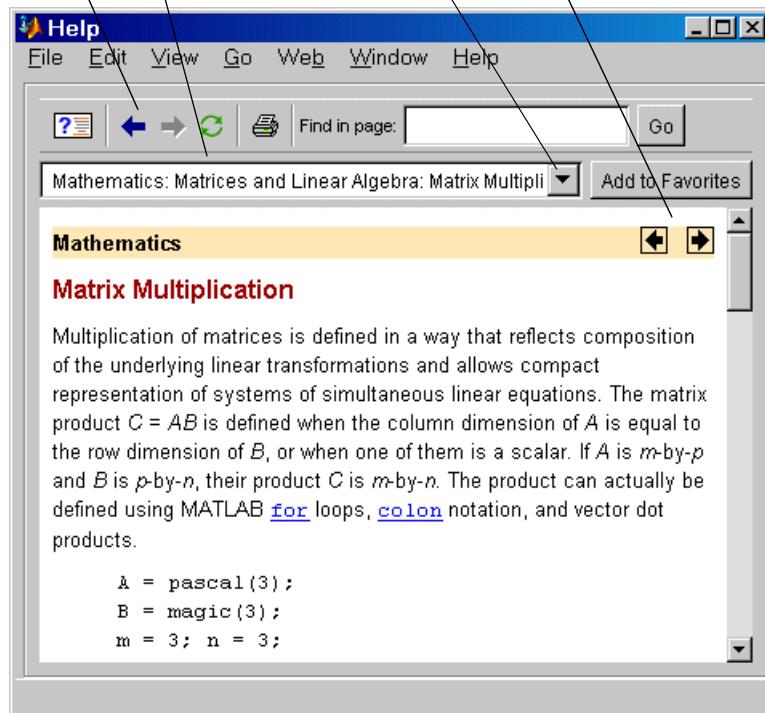
Use the arrow buttons in the page and in the toolbar to go to other pages.

Back button shows previous page you viewed in the Help browser.

Page title.

Return to pages previously viewed in the current MATLAB session.

Use left and right arrows in the page to go to the previous and next pages in the document.

View the next page in a document by clicking the right arrow ➡ at the top or bottom of the page. View the previous page in a document by clicking the left arrow ⬅ at the top or bottom of the page. These arrows allow you to move forward or backward within a single document. The arrows at the bottom of the page are labeled with the title of the page they go to.

View the page previously shown by clicking the back button ⬅ in the display pane toolbar. After using the back button, view the next page shown by clicking

the forward button ➡ in the display pane toolbar. These buttons work like the forward and back buttons of popular Web browsers. You can also go back or forward by right-clicking a page and selecting **Back** or **Forward** from the context menu.

Following Links

Click links in the displayed page to go to another place in the same page or a different page to get more information on the subject. Links appear underlined and in blue. Visited links appear in purple.

To open a linked page in a new Help browser window, press **Alt** and right-click the link or click the link using the middle mouse button.

Revisiting Pages

To display a page that you previously viewed in the current MATLAB session, select the page title from the drop-down list in the display pane toolbar (left of the **Add to Favorites** button).

Finding Terms in Displayed Pages

To find a phrase in the currently displayed page, follow these steps:

1 Type the phrase you are looking for in the **Find in page** field in the display pane toolbar. Then press **Enter** or **Return**, or click **Go**. You can type a partial word, for example, `preference` to find all occurrences of `preference` and `preferences`.

The page scrolls to the first occurrence of the phrase in the page and highlights it.

2 Press **Enter** or **Return** again to find the next occurrence in that page.

See "Searching Documentation" on page 4-15 for instructions on looking through all the documentation instead of just one page.

Copying Information

To copy information from the display pane, first select the information. Then right-click and select **Copy** from the context menu. You can then paste the information into another tool, such as the Command Window, or into another application, such as a word processing application.

Evaluating a Selection

To run code examples that appear in the documentation, select the code in the display pane. Then right-click and select **Evaluate Selection** from the context menu. The functions execute in the Command Window.

Viewing the Page Source (HTML)

To view the HTML source for the currently displayed page, select **View -> Help View Options -> Page Source**. The HTML version of the page appears in the Editor/Debugger. You can modify or copy the HTML source. To view a modified page, use the reload button in the display pane toolbar.

Viewing Web Pages and Other Documents
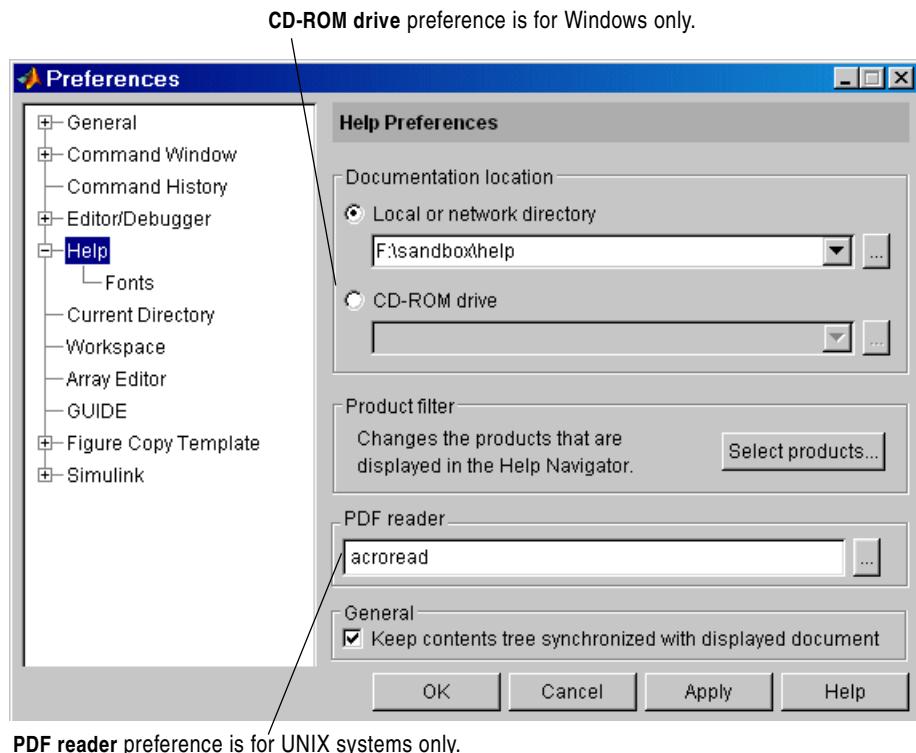
You can use the Help browser to view any Web page, although the Help browser might not support all the features of your usual Web browser. In the display pane page title field (to the left of **Add to Favorites**), type the URL and press the **Enter** key. For example, type `www.mathworks.com`. The MathWorks Web page appears in the Help browser.

You can also select **File -> Open** to open a file on your system in the Help browser. The Help browser supports `.html`, `.txt`, `.c`, and `.h` files.

Preferences for the Help Browser

Using preferences, you can specify the location of your help files, fonts used in the Help browser, and the products whose documentation you want to include in your Help browser.

To set preferences for the Help browser, select **Preferences** from the **File** menu. The **Preferences** dialog box opens. Select and expand **Help**.

CD-ROM drive preference is for Windows only.

PDF reader preference is for UNIX systems only.
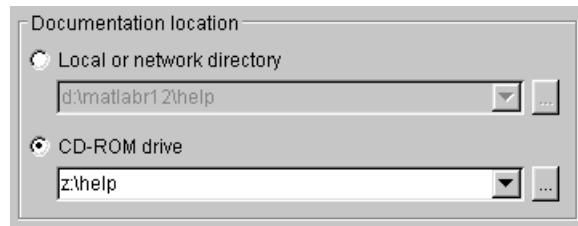
Set the preferences and then click **OK**. Help browser preferences consist of

- "Documentation Location—Specifying the help Directory" on page 4-33
- "Product Filter—Limiting the Product Documentation" on page 4-34
- "PDF Reader—Specifying Its Location" on page 4-35
- "General—Synchronizing the Contents Pane with the Displayed Page" on page 4-35
- "Help Fonts Preferences—Specifying Font Name, Style, and Size" on page 4-36

Documentation Location—Specifying the help Directory

Use the **Documentation location** preference to specify where the MATLAB help directory resides for your system.

The help directory contains the online help files used by the Help browser:

- If you elected to install the help files when you installed MATLAB, the documentation location should already be set to point to the help files. If the help files' location changes or you want to access different help files, change the **Documentation location** for **Local or network directory**. On Windows, use the **...** button to browse your file system to select the new location.
- On Windows platforms, if you did not install the help files during MATLAB installation, the Help browser attempts to find the files on the documentation CDs. You need to specify the **Documentation location** to be **CD-ROM drive**. Use the **...** button to browse your file system to select the drive's location and select the help directory, as shown in the following example. For PDF files, insert the PDF Documentation CD.

4-33

Function Alternative

Use the `docroot` function to set the documentation location.

For UNIX platforms that do not support Java GUIs, use the `docopt` function to specify the location of your `help` directory. The documentation displays in your default system Web browser and demos in a non-Java interface.

Installing Help Files

If you did not install the help files but want to access them from your system rather than from your CD, you can install the documentation using either of two methods. The first method is to use the MATLAB Installer program and only install documentation. See the Installation documentation for your platform for instructions. Note that you cannot install most PDF files this way.

The other method is to copy the following directories and their contents from the CDs to your system's hard drive under `$matlabroot`.

`help`—All files and subdirectories listed below.
`help/base/relnotes` and `help/base/utils`—All files.
`help/mapfiles`—All files.
`help/pdf_doc`—Each subdirectory and its files for PDF versions of toolbox documentation.
`help/support`—All files.
`help/techdoc`—All files and all subdirectories for MATLAB documentation.
`help/toolbox/...`—Each subdirectory and its files for toolbox documentation.

For Japanese documentation, use the `jhelp` directory and its contents instead of the `help` directory.

Product Filter—Limiting the Product Documentation

If you have MathWorks products in addition to MATLAB, such as Simulink and toolboxes, you can use the **Product filter** to limit the product documentation used.

1 In **Help Preferences**, under **Product filter**, click **Select products**.

   The **Help Product Filter** dialog box opens.

2 Select the products whose documentation you want to appear in the Help Navigator.

3 Then, to use only those products you specified, in the Help browser set the **Product filter** to **Selected**. When you want to use all product documentation, in the Help browser set the **Product filter** to **All**.

Note that you can also access the **Help Product Filter** dialog box by clicking the **Select** button in the Help browser.

PDF Reader—Specifying Its Location

If you want to access the PDF versions of the documentation, the Help system needs to know the location of your PDF reader (Adobe Acrobat).

For Windows systems, MATLAB reads the location from the registry, so you cannot specify its location using preferences.
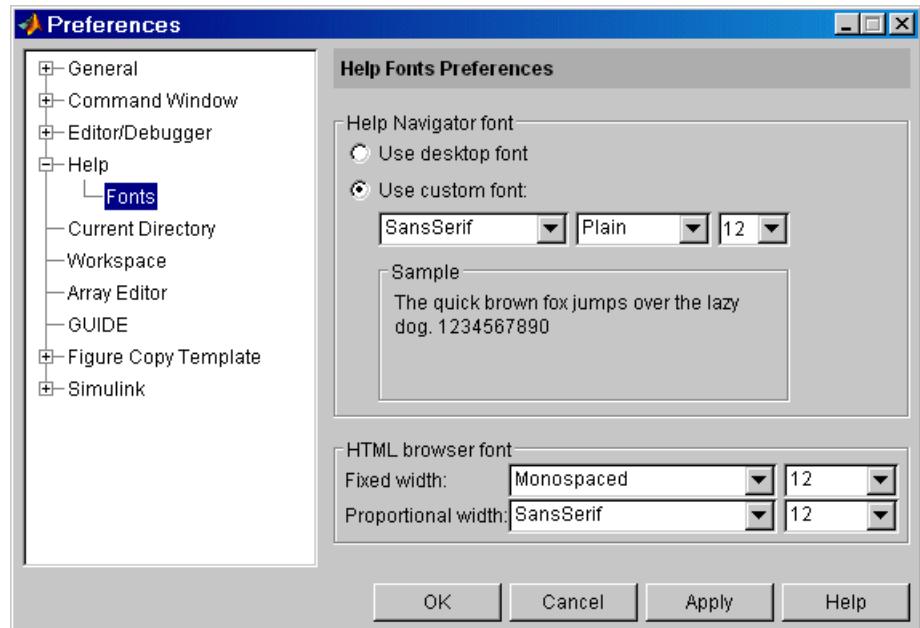
For UNIX systems, when you installed MATLAB, it looked for your system's PDF reader. If found, MATLAB automatically supplied the PDF reader location in the preferences **PDF reader** field. If MATLAB could not locate your PDF reader or if you moved your PDF reader since installation, change the location in the **PDF reader** field. Use the **...** button to browse your file system to select the location.

General—Synchronizing the Contents Pane with the Displayed Page

To turn synchronization off, clear the check box for **Keep contents tree synchronized with displayed document**, which is in **Help Preferences**, **General**. Select the check box to turn synchronization on. For more information, see "Synchronize Contents Listing with Display Pane" on page 4-12.

Help Fonts Preferences—Specifying Font Name, Style, and Size

You can specify the font type, style, and size used in the Help Navigator, and the font type and size used in the display pane. Expand the **Help** listing in the left pane of the **Preferences** dialog box and select **Fonts**. The **Help Fonts Preferences** panel appears.

Help Navigator Font

Use **Help Navigator font** preferences to specify the characteristics of the font in the Help Navigator. For example, specify a smaller font size for the Help Navigator to see more information without scrolling.

Select **Use desktop font** if you want the font in the Help Navigator to be the same as that specified under **General - Font & Colors**. If you want the Help Navigator font to be different, select **Use custom font** and specify the font characteristics for the Help Navigator:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

HTML Browser Font

Specify the font type and size used in the display pane for **Fixed width** and **Proportional width** fonts. In MathWorks documentation, most of the text uses proportional-width fonts. A fixed-width font is used for code examples, function names, and system input and output, as shown in this example.

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
```

To easily distinguish code, function names, and system input and output from surrounding text in the documentation, specify a different font for fixed width than for proportional width.

Printing Documentation

Printed documentation is provided with the major releases of some products and tools. The online documentation sometimes has information not included with the printed material and may be more current than the printed material. If you want to purchase printed documentation, see the online store at the MathWorks Web site at `http://www.mathworks.com`.

You can print the current page displayed in the Help browser, or print a page or an entire book from the PDF version of the documentation.

Printing a Page from the Help Browser

To print the page currently shown in the Help browser, click the print button in the display pane toolbar. The **Print** dialog box appears.

The **Pages** field in the **Print** dialog box shows the total number of pages to be printed and lets you specify the range of pages you want to print. If there is more than one page in the range, it means that multiple physical pages are needed to print the single page displayed in the Help browser.

Complete the dialog box and press **OK** to print the page.

Printing the PDF Version of Documentation

If you need to print only a few pages and if the quality does not need to be equivalent to pages in a printed book, you can print directly from the MATLAB Help browser—see "Printing a Page from the Help Browser" on page 4-38.

If you need to print more than a few pages of documentation, or if you need the pages to appear as if they came from a printed book, print the PDF version of the documentation. PDF documentation is shown and printed using your PDF reader, usually Adobe Acrobat Reader. The PDF documentation reproduces the look and feel of the printed book, complete with fonts, graphics, formatting, and images. In the PDF document, use links from the table of contents, index, or within the document to go directly to the page of interest.

To print a PDF version of documentation, follow these steps:

1 For Windows systems only, insert the PDF Documentation CD provided with MATLAB into your CD-ROM drive. PDF files are on the CD and are not installed on your system. (For UNIX systems, the PDF files are installed). If

you have problems, check the Help preferences—see "Documentation Location—Specifying the help Directory" on page 4-33.

2 In the Help browser, click the **Contents** tab and select the title (first entry) for a product, for example, MATLAB.

The Roadmap page opens for that product, providing links to key documentation for that product.

3 On the bottom of the Roadmap page, listed under **Printing the Documentation**, is a link for printing. Click that link.

If there is only one manual for the product, your PDF reader opens, displaying the table of contents and first page of the manual.

If there is more than one item you can print for the product, a page listing the choices appears. Select the item you want to print. Your PDF reader opens, displaying the documentation.

If you have problems, check the Help preferences—see "PDF Reader—Specifying Its Location" on page 4-35.

4 To print the documentation, select **Print** from the **File** menu in your PDF reader.

Using Help Functions

There are several help functions that provide different forms of help from the Help browser, or provide alternative ways to access help.

Function	Description
dbtype	Displays the M-file with line numbers. If you want to see only the input and output arguments for a function, use `dbtype function 1`, which displays the first line of the M-file.
demo	Displays the **Demos** pane in the Help browser, from which you can access demonstrations for the products you have installed.
doc	Displays the reference page for the specified function, block, or property in the Help browser. The reference page provides syntax, a description, examples, and links to related functions.
docopt	For UNIX platforms that do not support Java GUIs, use `docopt` to specify the location of help files. Documentation appears in a Web browser and demos in a non-Java interface.
docroot	Get or set the root directory for MATLAB help files.
help	Displays M-file help (a description and syntax) in the Command Window for the specified function or block.
helpbrowser	Opens the Help browser, the MATLAB interface for accessing documentation.
helpdesk	Opens the Help browser. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser.
helpwin	Displays in the Help browser a list of all functions, providing access to M-file help for the functions.

lookfor	Displays in the Command Window a list and brief description for all functions whose brief description includes the specified keyword.
web	Opens the specified URL in the specified Web browser, with the default being the MATLAB Help browser. You can use the web function in your own M-files to display documentation.

Viewing Function Reference Pages—the doc Function

To view the reference page for a function, block, or property in the Help browser, use doc. This is like using the Help browser search feature, with **Search type** set to **Function name**. For example, type

```
doc format
```

to view the reference page for the format function.

Overloaded Functions

When a function name is used in multiple products, it is said to be an overloaded function. The doc function displays the reference page for the first function having that name on the MATLAB search path, and lists the overloaded functions in the Command Window. To get help for an overloaded function, specify the name of the directory containing the function you want the reference page for, followed by the function name. For example, to display the reference page for the set function in the Database Toolbox, type

```
doc database/set
```

Getting Help in the Command Window—the help Function

To quickly view a brief description and syntax for a function in the Command Window, use the `help` function. For example, typing

```
help bar
```

displays a description and syntax for the `bar` function in the Command Window. This is called the M-file help. For other arguments you can supply, see the reference page for `help`.

If you need more information than the `help` function provides, use the `doc` function, which displays the reference page in the Help browser. It can include color, images, links, and more extensive examples than the M-file help. For example, typing

```
doc bar
```

displays the reference page for the `bar` function in the Help browser.

Note M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to make them stand out from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Java do use mixed case; the M-file help accurately reflects that, and you should use mixed case when typing them.

Overloaded Functions

When a function name is used in multiple products, it is said to be an overloaded function. The `help` function displays M-file help for the first function with that name found on the path, and lists the overloaded functions at the end. To get help for an overloaded function, specify the name of the directory containing the function you want help for, followed by the function name. For example, to get help for the `set` function in the Database Toolbox, type

```
help database/set
```

Creating M-File Help for Your Own M-Files

You can create M-file help for your own M-files and access it using the `help` command. See the `help` reference page for details.

Other Forms of Help

In addition to using the Help browser and help functions, these are the other ways to get help for MathWorks products:

- "Product-Specific Help Features" on page 4-44
- "Downloading M-Files" on page 4-44
- "Participating in the Newsgroup for MathWorks Products" on page 4-45
- "Contacting Technical Support" on page 4-45
- "Providing Feedback" on page 4-45
- "Getting Version and License Information" on page 4-46
- "Accessing Documentation for Other Products" on page 4-46

Product-Specific Help Features

In addition to the Help browser and help functions, some products and tools allow other methods for getting help. You will encounter some methods in the course of using a product, such as entries in the **Help** menu, **Help** buttons in dialog boxes, and selecting **Help** from a context menu. These methods all display context-sensitive help in the Help browser. Other methods for getting help, such as pressing the **F1** key, are described in the documentation for the product or tool that uses the method.

Downloading M-Files

You can download M-files contributed by users and developers of MATLAB, Simulink, and related products. Before you write an M-file yourself, especially if it seems to be a more generic utility, check the list of contributed files to see if someone has already written it. These files are freely contributed and can be used without charge by anyone who downloads them. To view the files available to download, go to the MATLAB Central File Exchange page on The MathWorks Web site, `http://www.mathworks.com/matlabcentral/fileexchange/index.jsp`. You can access this via the **Web** menu in any desktop component.

If you write M-files that you think would be of use to others, consider submitting them to the MATLAB Central File Exchange via the Web page.

Participating in the Newsgroup for MathWorks Products

The Usenet newsgroup for MATLAB and related products, `comp.soft-sys.matlab`, is read by thousands of users worldwide. Access the newsgroup to ask for or provide help or advice, and to share code or examples. You can read and submit postings as well as view and search through a sizable archive of postings using the MATLAB Central Newsgroup Access Web page on The MathWorks Web site, `http://www.mathworks.com/matlabcentral`. You can access this via the **Web** menu in any desktop component. First-time users to the newsgroup might want to read the FAQ listed on that page.

Contacting Technical Support

If your computer is connected to the Internet, you can contact MathWorks Technical Support for help with product problems.

• Find specific Technical Support information using the Help browser **Search** feature, with the **Search type** set to **Online Knowledge Base**. The knowledge base is a database on the MathWorks Web site, providing the most up-to-date solutions for questions posed by users.

• Select **Technical Support Knowledge Base** from the **Web** menu to go to the Technical Support Web page (`http://www.mathworks.com/support`). The page displays in your system's default Web browser. You can find out about other types of information including third-party books, ask questions, make suggestions, and report possible bugs.

Providing Feedback

To report any problems or provide any comments or suggestions to The MathWorks about the documentation and help features, send e-mail to `doc@mathworks.com`.

Alternatively, you can fill out a form on the Web. To access the form, go to the **Contents** pane in the Help browser. Select Begin Here from the top of the Contents listing. Scroll to the bottom of the Begin Here page and click the link **Feedback on Help**.

To suggest enhancements or provide feedback about MathWorks products, send e-mail to `suggest@mathworks.com`. To report problems, send e-mail to

bugs@mathworks.com or contact Technical Support at
http://www.mathworks.com/support/.

Getting Version and License Information

If you need the product version or license information, select **About** from the
Help menu for that product. The version is displayed in an **About** dialog box.
Click **Show License** in the dialog box to view license information. Note that the
information displayed does not cover your specific license agreement. If the
product does not have a **Help** menu, use the ver function. To see the license
number for MATLAB, type license in the Command Window. See also the ver,
version, and license functions.

Accessing Documentation for Other Products

The Help browser provides access to documentation for all products installed
on your system. If you want to look through documentation for MathWorks
products you do not have, you can

- View any product's online documentation at the MathWorks Web site,
  http://www.mathworks.com. Click **support** in the top menu bar. On the
  support Web page, under **Learn to use the products**, click **Documentation**.
  Select the product whose documentation you want to view. The
  documentation is displayed in your Web browser.

- You can access documentation for all products from the documentation CDs
  provided with MATLAB.There are PDF files for nearly all products. Use
  preferences for the Help browser to set the documentation location to the
  CD-ROM drive and use the product filter to specify those products whose
  documentation you want to see. For instructions, see "Preferences for the
  Help Browser" on page 4-32.

5

Workspace, Search Path, and File Operations

When you work with MATLAB, you'll need to understand the following important aspects of the development environment:

MATLAB Workspace (p. 5-2)

The workspace is the set of variables maintained in memory during a MATLAB session.

Use the Workspace browser or equivalent functions to view the workspace.

Viewing and Editing Workspace Variables with the Array Editor (p. 5-10)

View and make changes to variables using the Array Editor.

Search Path (p. 5-18)

MATLAB uses a search path to find M-files and other MATLAB related files.

View and change the path using the **Set Path** dialog box or equivalent functions.

File Operations (p. 5-24)

Search for, view, open, and make changes to MATLAB related directories and files, using the Current Directory browser or equivalent functions.

MATLAB Workspace

The MATLAB workspace consists of the set of variables (named arrays) built up during a MATLAB session and stored in memory. You add variables to the workspace by using functions, running M-files, and loading saved workspaces. For example, if you type

```
t = 0:pi/4:2*pi;
y = sin(t);
```

the workspace includes two variables, y and t, each having nine values.
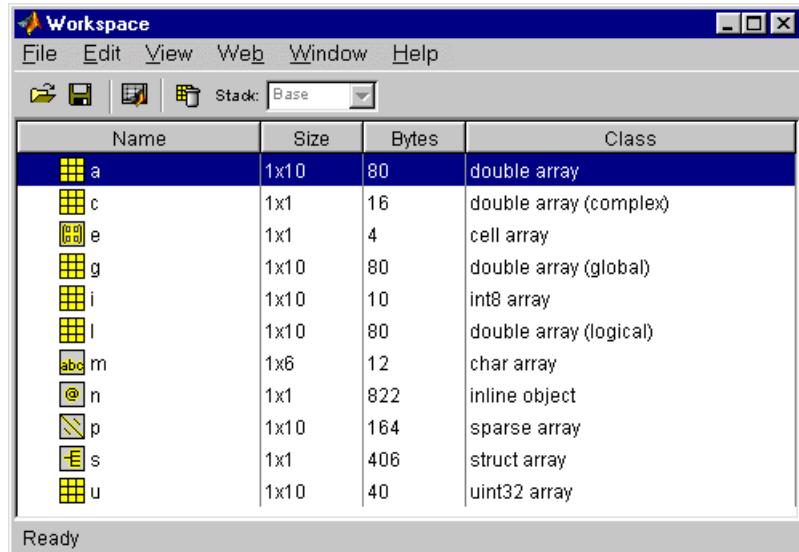
You can perform workspace operations and related features using the Workspace browser. Equivalent functions are available and are documented with each feature of the Workspace browser.

- "Opening the Workspace Browser" on page 5-2
- "Viewing the Current Workspace" on page 5-3
- "Saving the Current Workspace" on page 5-4
- "Loading a Saved Workspace and Importing Data" on page 5-6
- "Changing and Copying Variable Names" on page 5-7
- "Clearing Workspace Variables" on page 5-7
- "Viewing Base and Function Workspaces Using the Stack" on page 5-8
- "Creating Graphics from the Workspace Browser" on page 5-8
- "Opening Variables and Objects for Viewing and Editing" on page 5-8
- "Preferences for the Workspace Browser" on page 5-8.

Opening the Workspace Browser

To open the Workspace browser, select **Workspace** from the **View** menu in the MATLAB desktop, or type workspace at the Command Window prompt.

The Workspace browser opens.

Viewing the Current Workspace

The Workspace browser shows the name of each variable, its array size, its size in bytes, and the class. The icon for each variable denotes its class.

To resize the columns of information, drag the column header borders. To show or hide any of the columns, or to specify the sort order, select **Workspace View Options** from the **View** menu.

You can select the column on which to sort as well as reverse the sort order of any column. Click a column heading to sort on that column. Click the column heading again to reverse the sort order in that column. For example, to sort on **Size**, click the column heading once. To change from ascending to descending, click the heading again.

Function Alternative

Use who to list the current workspace variables. Use whos to list the variables and information about their size and class. For example:

```
who
   Your variables are:
   A        M        S        v

whos
   Name      Size        Bytes  Class
   A         4x4           128  double array
   M         8x1          2368  cell array
   S         1x1           398  struct array
   v         5x9            90  char array
   Grand total is 286 elements using 2984 bytes
```

Use exist to see if the specified variable is in the workspace.

Saving the Current Workspace

The workspace is not maintained across MATLAB sessions. When you quit MATLAB, the workspace is cleared. You can save any or all of the variables in the current workspace to a MAT-file, which is a MATLAB specific binary file. You can then load the MAT-file at a later time during the current or another session to reuse the workspace variables. MAT-files use a .mat extension.

Note The .mat extension is also used by Microsoft Access.

Saving All Variables

To save all of the workspace variables using the Workspace browser:

1 From the **File** menu, select **Save Workspace As**, or click the save button in the Workspace browser toolbar.

The **Save** dialog box opens.

2 Specify the location and **File name**. MATLAB automatically supplies the `.mat` extension.

3 Click **Save**.

The workspace variables are saved under the MAT-file name you specified.

You can also save the workspace variables from the desktop by selecting **Save Workspace As** from the **File** menu.

Saving Selected Variables

To save some but not all of the current workspace variables:

1 Select the variable in the Workspace browser. To select multiple variables, **Shift**+click or **Ctrl**+click.

2 Right-click and from the context menu, select **Save Selection As**.

The **Save to MAT-File** dialog box opens.

3 Specify the location and **File name**. MATLAB automatically supplies the `.mat` extension.

4 Click **Save**.

The workspace variables are saved under the MAT-file name you specified.

Function Alternative

To save workspace variables, use the `save` function followed by the filename you want to save to. For example,

```
save('june10')
```

saves all current workspace variables to the file `june10.mat`.

If you don't specify a filename, the workspace is saved to `matlab.mat` in the current working directory. You can specify which variables to save, as well as control the format in which the data is stored, such as ASCII. For these and other forms of the function, see the reference page for `save`. MATLAB provides additional functions for saving information—see Chapter 6, "Importing and Exporting Data."

Loading a Saved Workspace and Importing Data

To load saved variables into the workspace:

1 Click the load data button 📂 on the toolbar in the Workspace browser, or right-click in the Workspace browser and select **Import Data** from the context menu.

The **Open** dialog box opens.

2 Select the MAT-file you want to load and click **Open**.

The variables and their values, as stored in the MAT-file, are loaded into the current workspace. If any variables being loaded have the same names as variables in the current workspace, the values from the MAT-file replace the values in the current workspace. Any variables in the MAT-file that are not in the workspace are added to the workspace.

Function Alternative

Use load to open a saved workspace. For example,

```
load('june10')
```

loads all workspace variables from the file june10.mat.

Importing Data

MATLAB provides other methods and functions for loading information. You can use one of these methods, the Import Wizard, from the Workspace browser—select **Edit -> Paste Special** or use **Ctrl+V** to import data to MATLAB using the Import Wizard. For more information on the Import Wizard and other methods for loading information, see Chapter 6, "Importing and Exporting Data".

Changing and Copying Variable Names

To rename a variable in the workspace, right-click the variable the Workspace browser and select **Rename** from the context menu. Type the new variable name over the existing name and press **Enter** or **Return**.

To copy variable names to the clipboard, select the workspace variables and select **Edit -> Copy**. You can then paste the names, for example, into the Command Window. Multiple variables are comma separated.

Clearing Workspace Variables

You can clear a variable, which removes it from the workspace.

To clear a variable using the Workspace browser:

1 In the Workspace browser, select the variable, or **Shift**+click or **Ctrl**+click to select multiple variables. To select all variables, choose **Select All** from the **Edit** or context menus.

2 Press the **Delete** key or click the delete button on the toolbar.

3 A confirmation dialog box may appear. If it does, click **Yes** to clear the variables.

The confirmation dialog box appears if you specify it as a preference. See "Preferences for the Workspace Browser" on page 5-8 to change the preference.

To delete all variables at once, select **Clear Workspace** from the **Edit** menu in the Workspace browser.

Function Alternative

Use the clear function. For example,

```
clear A M
```

clears the variables A and M from the workspace.

Viewing Base and Function Workspaces Using the Stack

When you run M-files, MATLAB assigns each function its own workspace, called the function workspace, which is separate from the MATLAB base workspace. To access the base and function workspaces when debugging M-files, use the **Stack** field in the Workspace browser. The **Stack** field is only available in debug mode and otherwise is grayed out. The **Stack** field is also accessible from the Editor/Debugger. See "Debugging M-Files" on page 7-25 for more information.

Creating Graphics from the Workspace Browser

From the Workspace browser, you can generate a graph of a variable. Right-click the variable you want to graph. From the context menu, select **Graph Selection** and then choose the type of graph you want to create. The graph appears in a figure window. For more information about creating graphs in MATLAB, see the MATLAB Graphics documentation.

Opening Variables and Objects for Viewing and Editing

In the Workspace browser, double-click a variable and it opens in the Array Editor, where you can view and edit the contents of the variable. See "Viewing and Editing Workspace Variables with the Array Editor" on page 5-10 for more information about opening arrays.

Some toolboxes allow you to double-click an object in the Workspace browser to open a viewer or other tool appropriate for that object. For details, see the toolbox documentation for that object type.

Preferences for the Workspace Browser

You can specify as a preference the fonts to use in the Workspace browser and whether or not you want a confirmation dialog box to appear when you clear variables using the Workspace browser.

From the Workspace browser **File** menu, select **Preferences**. The **Preferences** dialog box opens to the **Workspace Preferences** panel.

Font

Workspace browser font preferences specify the characteristics of the font used in the Workspace browser. Select **Use desktop font** if you want the font in the Workspace browser to be the same as that specified for **General Font & Colors** preferences.

If you want the Workspace browser font to be different, select **Use custom font** and specify the font characteristics for the Workspace browser:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Confirm Deletion of Variables

Select **Confirm deletion of variables** if you want a confirmation dialog box to appear when you delete a variable.

Viewing and Editing Workspace Variables with the Array Editor

Use the Array Editor to view and edit a visual representation of one or two-dimensional numeric arrays, strings, and cell arrays of strings. The features of the Array Editor are
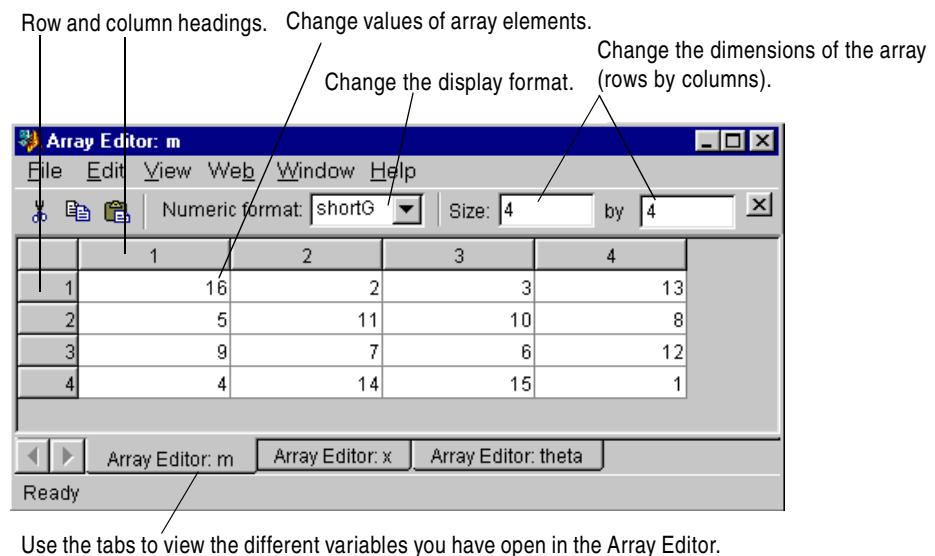
- "Opening the Array Editor" on page 5-10
- "Controlling the Display of Values in the Array Editor" on page 5-12
- "Navigating in the Array Editor" on page 5-12
- "Changing Array Size and Content of Elements in the Array Editor" on page 5-13
- "Cut, Copy, Paste, and Delete in the Array Editor" on page 5-13
- "Preferences for the Array Editor" on page 5-16

Opening the Array Editor

You can open the Array Editor from the Workspace browser:

1 In the Workspace browser, select the variable you want to open. **Shift**+click or **Ctrl**+click to select multiple variables, or use **Ctrl+A** to select all variables to open.

2 Click the open selection button 🖼 on the toolbar. For one variable, you can instead double-click it to open it.

The Array Editor opens, displaying the values for the selected variable.

Row and column headings. Change values of array elements.

Change the display format.

Change the dimensions of the array (rows by columns).

Use the tabs to view the different variables you have open in the Array Editor.

Repeat the steps to open additional variables in the Array Editor. Access each variable via its tab at the bottom of the window, or use the **Window** menu.

Function Alternatives

To open a variable in the Array Editor, use `openvar` with the name of the variable you want to open as the argument. For example, type

```
openvar('m')
```

MATLAB opens m in the Array Editor.

To see the contents of a variable in the workspace, just type the variable name at the Command Window prompt. For example, type

```
m
```

and MATLAB returns

```
m =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

5-11

Controlling the Display of Values in the Array Editor

In the Array Editor, select an entry in the **Numeric format** list box to control how numeric values are displayed. For descriptions of the formats, see the reference page for `format`. The format applies only to the Array Editor display for that variable while the Array Editor is open. It does not affect how MATLAB computes or saves the numeric value, nor does it affect the format used for display in the Command Window.

To specify a format for all variables in the Array Editor and keep it persistent across sessions, specify the format for the Array Editor using preferences—see "Preferences for the Array Editor" on page 5-16.

Navigating in the Array Editor

Use the following keys to move among elements in the Array Editor. Navigating in the Array Editor is much like navigating in Microsoft Excel.

Key	Result
Enter	Commit any changes to the element and move to next element, where next element is specified using "Preferences for the Array Editor" (default is down)
Tab	Move right Within a selection, also moves from the last column to the first column in the next row
Shift+Enter or **Shift+Tab**	Move in opposite direction of **Enter** or **Tab**
Page Up	Move up m rows, where m is the number of visible rows
Page Down	Move down m rows, where m is the number of visible rows
Home	Move to column 1
Ctrl+Home	Move to row 1, column 1
Shift+Home	Select to column 1
End	Move to last row in current column

Changing Array Size and Content of Elements in the Array Editor

To change the dimensions of an array, type the new values for the rows and columns in the **Size** fields. If you increase the size, the new rows and columns are added to the end and are filled with zeros. If you decrease the size, you will lose data—MATLAB removes the rows and columns from the end. Some data types do not allow you to change the dimension; for these variables, the **Size** field is not editable.

To change the value of an element in the Array Editor, click in that element and type a new value. Press **Enter** or **Return**, or click in another element to make the change take effect. You can specify where the cursor moves to after you press **Enter**—see "Preferences for the Array Editor" on page 5-16.

If you opened an existing MAT-file and made changes to it using the Array Editor, save that MAT-file if you want the changes to be saved. For instructions, see "Saving the Current Workspace" on page 5-4.

Cut, Copy, Paste, and Delete in the Array Editor

You can cut or copy selected elements, rows, and columns in an array and paste them to another position in that or another open array. To select a column or row, click in the row or column heading (the element that shows the row or column number). **Shift**+click to choose contiguous elements, rows, or columns in the array, or **Ctrl+A** to select all elements. For the cut, copy, and paste operations, use the **Edit** menu, the context menu, or the toolbar buttons.

When you cut elements, the value of each element you cut becomes 0. After cutting, select the elements whose value you want to replace with the cut elements and then choose **Paste**. If the shape of the elements you cut differs from the shape of the elements into which you are pasting, the Array Editor pastes all the elements, either by expanding the selection to be pasted into, or by expanding the array size to allow all the elements to be pasted. Pasting copied elements is the same as pasting cut elements, but the elements copied maintain their value rather than becoming 0.

Example Copying and Pasting Array Elements

In this example, two elements are copied but the selected area for pasting is only one element, so the Array Editor expands the selected area for pasting.

Two elements are selected and copied.

One element is selected as the paste area.

The Array Editor pastes all of the copied elements.

Example Cutting and Pasting Array Elements

In this example, the area selected for pasting requires the Array Editor to expand the array size in order for all cut elements to be pasted.

Two elements are selected and cut. →

The values of the cut elements become 0.

One element is selected as the paste area.

The Array Editor adds a column so that both cut elements can be pasted. The values for other new elements in the column added are 0.

Deleting Elements, Rows and Columns

You can clear elements, rows, or columns in the array by selecting them and then selecting **Delete** from the **Edit** menu or context menu. When you delete cells, a dialog box appears asking how you want the remaining cells to shift.

Exchanging Data with the Command Window

You can copy data from the Array Editor and paste it into the Command Window. You can also copy a value from the Command Window and paste it into an element in the Array Editor. Be sure the data types are compatible. For example, you cannot paste text from the Command Window into a numeric array in the Array Editor.

Exchanging Data with Excel

You can cut or copy cells from Microsoft Excel and paste them into the Array Editor. You can also cut or copy elements from the Array Editor and paste them into Excel.

Be sure the data types are compatible. For example, you cannot paste text from Excel into a numeric array in the Array Editor.

Preferences for the Array Editor

Using preferences for the Array Editor, you can specify

- "Font"—type, style, and size
- "Default Format" —how numeric values are displayed
- "Edit"—how the **Enter** key should behave

To set preferences for the Array Editor, select **Preferences** from the **File** menu. The **Preferences** dialog box opens showing **Array Editor Preferences**.

Font

Array Editor font preferences specify the characteristics of the font used in the Array Editor. Select **Use desktop font** if you want the font in the Array Editor to be the same as that specified for **General Font & Colors** preferences. If you want the Array Editor font to be different, select **Use custom font** and specify the font characteristics for the Array Editor:

- Type, for example, SansSerif
- Style, for example, bold
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look.

Default Format

Specify the output format of numeric values displayed in the Array Editor. This affects only how numbers are displayed, not how MATLAB computes or saves them. For more information, see "Controlling the Display of Values in the Array Editor" on page 5-12 or see the reference page for format.

Edit

You can specify where the cursor moves to after you type in an element and press **Enter**:

- If you want the cursor to remain at the element where you just typed, clear **Move selection after Enter**.
- If you want the cursor to move to another element, check **Move selection after Enter**, and then use **Direction** to specify how you want to cursor to move. For example, if you want the cursor to move right one element after you press **Enter**, select **Right**.

Search Path

This section covers the following topics:

Purpose of the Search Path

MATLAB uses a *search path* to find M-files and other MATLAB related files, which are organized in directories on your file system. These files and directories are provided with MATLAB and associated toolboxes. Any file you want to run in MATLAB must reside in a directory that is on the search path or in the current directory. By default, the files supplied with MATLAB and MathWorks toolboxes are included in the search path.

If you create any MATLAB related files, add the directories containing the files to the MATLAB search path. For instructions to view and modify the search path, see "Viewing and Setting the Search Path" on page 5-20.

Note Save any M-files you create and any MathWorks-supplied M-files that you edit in a directory that is not in the $matlabroot/toolbox directory tree. If you keep your files in $matlabroot/toolbox directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the $matlabroot/toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to $matlabroot/toolbox directories using an external editor or add or remove in from these directories using file system operations, run rehash toolbox before you use the files in the current session. If you make changes to existing files in $matlabroot/toolbox directories using an external editor, run clear functionname before you use the files in the current session. For more information, see rehash or "Toolbox Path Caching" on page 1-9.

How the Search Path Works

The search path is also referred to as the *MATLAB path*. Files included are considered to be *on the path*. When you include a directory on the search path, you *add it to the path*. Subdirectories must be explicitly added to the path; they are not on the path just because their parent directories are. The search path is stored in the file pathdef.m.

The order of directories on the path is relevant. MATLAB looks for a named element, for example, foo, as described here. If you enter foo at the MATLAB prompt, MATLAB performs the following actions:

1 Looks for foo as a variable.

2 Checks for foo as a built-in function.

3 Looks in the current directory for a file named foo.m.

4 Searches the directories on the MATLAB search path, in order, for foo.m.

Although the actual search rules are more complicated because of the restricted scope of private functions, subfunctions, and object-oriented functions, this simplified perspective is accurate for the ordinary M-files you usually work with.

The order of the directories on the search path is important if there is more than one function with the same name. When MATLAB looks for that function, only the first one in the search path order is found. Other functions with the same name are considered to be *shadowed* and cannot be executed. For more information, see "How MATLAB Determines Which Method to Call" in Programming and Data Types.

To see the pathname used, use which for a specified function. For more information, see the reference page for which.
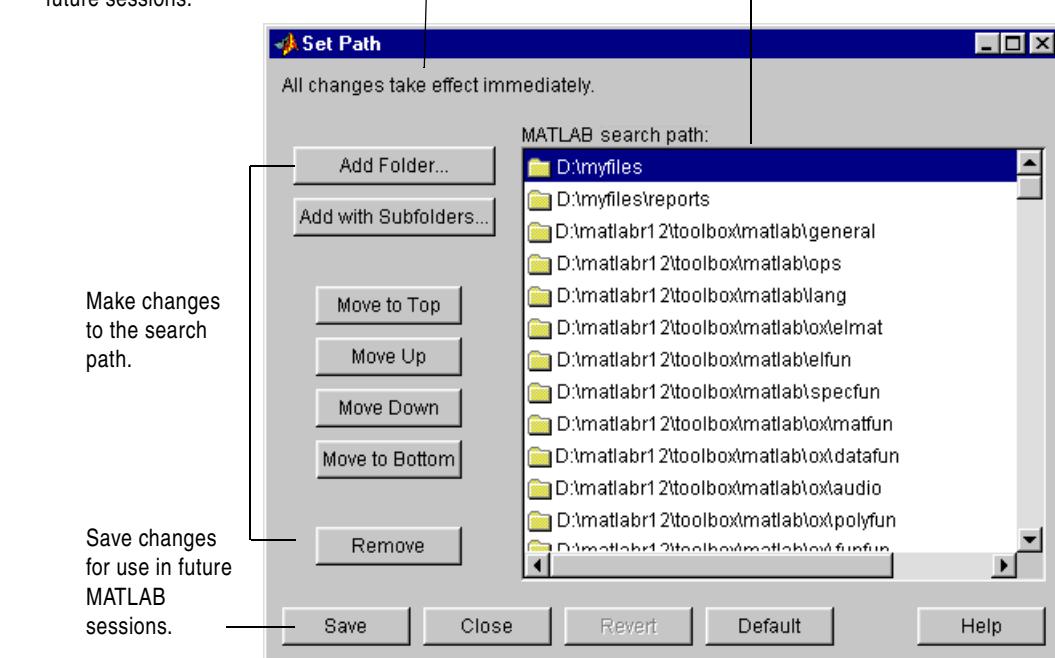
You can use a different pathdef.m if you store it in your startup directory—see "Startup Directory for MATLAB" on page 1-2.

Viewing and Setting the Search Path

Use the **Set Path** dialog box to view and modify the MATLAB search path and see files in directories that are on the path. Equivalent functions are documented for each feature of the **Set Path** dialog box.

Select **Set Path** from the **File** menu, or type pathtool at the Command Window prompt. The **Set Path** dialog box opens.

When you click one of these buttons, the change is made to the current search path. However, the search path is not automatically saved for future sessions.

Directories on the current MATLAB search path.

Make changes to the search path.

Save changes for use in future MATLAB sessions.

Set Path
All changes take effect immediately.

MATLAB search path:

- Add Folder...
- Add with Subfolders...
- Move to Top
- Move Up
- Move Down
- Move to Bottom
- Remove

D:\myfiles
D:\myfiles\reports
D:\matlabr12\toolbox\matlab\general
D:\matlabr12\toolbox\matlab\ops
D:\matlabr12\toolbox\matlab\lang
D:\matlabr12\toolbox\matlab\ox\elmat
D:\matlabr12\toolbox\matlab\elfun
D:\matlabr12\toolbox\matlab\specfun
D:\matlabr12\toolbox\matlab\ox\matfun
D:\matlabr12\toolbox\matlab\ox\datafun
D:\matlabr12\toolbox\matlab\ox\audio
D:\matlabr12\toolbox\matlab\ox\polyfun
D:\matlabr12\toolbox\matlab\ox\funfun

Save Close Revert Default Help

Use the **Set Path** dialog box for the following:

- "Viewing the Search Path" on page 5-21
- "Adding Directories to the Search Path" on page 5-21
- "Moving Directories Within the Search Path" on page 5-22
- "Removing Directories from the Search Path" on page 5-22
- "Restoring the Default Search Path" on page 5-23
- "Reverting to the Previous Path" on page 5-23
- "Saving Settings to the Path" on page 5-23
- "Editing pathdef.m" on page 5-23

Viewing the Search Path

The **MATLAB search path** field in the **Set Path** dialog box lists all of the directories on the search path.

Function Alternative. Use the `path` function to view the search path.

Adding Directories to the Search Path

To add directories to the MATLAB search path using the **Set Path** dialog box:

1 Click the **Add Folder** or the **Add with Subfolders** button.

- If you want to add only the selected directory but do not want to add all of its subdirectories, click **Add Folder**.
- If you want to add the selected directory and all of its subdirectories, click **Add with Subfolders**.

The **Browse for Folder** dialog box opens.

2 In the **Browse for Folder** dialog box, use the view of your file system to select the directory to add, and then click **OK**.

The selected directory, and subdirectories if specified, are added to the top of the search path. They remain on the search path until you end the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path—see "Saving Settings to the Path" on page 5-23.

You cannot add method directories (directories that start with @) or private directories directly to the search path. Instead, add their parent directories.

Adding Directories to the Path from the Current Directory Browser. In the Current Directory browser, select the directory, right-click, and select **Add to Path** from the context menu.

Function Alternative. To add directories to the search path, use addpath. The addpath function offers an option to get the path as a string and to concatenate multiple strings to form a new path.

You can include addpath in your startup M-file to automatically modify the path when MATLAB starts.

Moving Directories Within the Search Path

The order of files on the search path is relevant—for more information, see "How the Search Path Works" on page 5-19.

To modify the order of directories within the search path, first select the directory or directories you want to move. Then click one of the **Move** buttons, such as **Move to Top**.

The order changes and the new order of files on the search path remains in effect until you end the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path—see "Saving Settings to the Path" on page 5-23.

Function Alternative. While there is not a specific function to move directories, you can edit the pathdef.m file to make changes to the order of files.

Removing Directories from the Search Path

To remove directories from the MATLAB search path using the **Set Path** dialog box:

1 Select the directories to remove.

2 Click **Remove**.

The directories are removed from the search path for the remainder of the current MATLAB session. To use the newly modified search path in subsequent sessions, you need to save the path—see "Saving Settings to the Path" on page 5-23.

Function Alternative. To remove directories from the search path, use `rmpath`.

You can include `rmpath` functions in your `startup` M-file to automatically modify the path when MATLAB starts.

Restoring the Default Search Path

To restore the default search path, click **Default** in the **Set Path** dialog box. This changes the search path so that it uses the factory settings.

Reverting to the Previous Path

To restore the previous path, click **Revert** in the **Set Path** dialog box. This cancels any unsaved changes you've made in the **Set Path** dialog box.

Saving Settings to the Path

When you make changes to the search path, they remain in effect during the current MATLAB session. To keep the changes in effect for subsequent sessions, save the changes. To save changes using the **Set Path** dialog box, click **Save**.

The search path is stored in the `pathdef.m` file. By default, `pathdef.m` is stored in `$matlabroot/toolbox/local`.

Function Alternative. Use `path2rc` to save the current path to `pathdef.m`.

Editing pathdef.m

You can directly edit `pathdef.m` with a text editor to change the path.

If you do not have file system permission to edit `pathdef.m`, put `path` and `addpath` functions in your `startup.m` file to change your path defaults.

File Operations

MATLAB file operations use the current directory as a reference point. Any file you want to run must either be in the current directory or on the search path. Also, when you open a file in MATLAB, the starting point for the file **Open** dialog box is the current directory. The key tools for performing file operations are

- Current directory field
- Current directory browser

Current Directory Field

A quick way to view or change the current directory is by using the **Current Directory** field in the desktop toolbar.
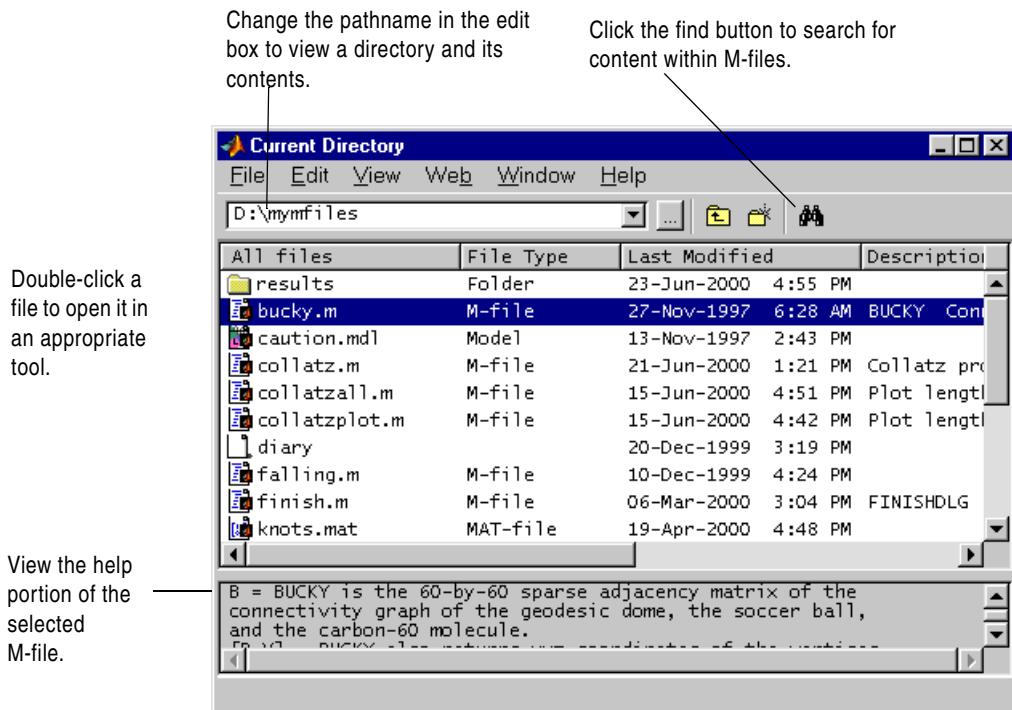
To change the current directory from this field, do one of the following:

- In the field, type the path for the new current directory.
- Click the down arrow to view a list of previous working directories, and select an item from the list to make that directory become the MATLAB current working directory. The directories are listed in order, with the most recently used at the top of the list. You can clear the list and set the number of directories saved in the list—see "Preferences for the Current Directory Browser" on page 5-35.
- Click the browse button (...) to set a new current directory.

Current Directory Browser

To search for, view, open, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser. Equivalent functions are documented for each feature of the Current Directory browser.

To open the Current Directory browser, select **Current Directory** from the **View** menu in the MATLAB desktop, or type filebrowser at the Command Window prompt. The Current Directory browser opens.

Change the pathname in the edit box to view a directory and its contents.

Click the find button to search for content within M-files.

Double-click a file to open it in an appropriate tool.

View the help portion of the selected M-file.

The main features of the Current Directory browser are

- "Viewing and Making Changes to Directories" on page 5-26
- "Creating, Renaming, Copying, and Removing Directories and Files" on page 5-28
- "Opening, Running, and Viewing the Content of Files" on page 5-30
- "Finding and Replacing Content Within Files" on page 5-33
- "Accessing Source Control Features" on page 5-35
- Setting "Preferences for the Current Directory Browser" on page 5-35

Note You generally cannot perform operations on files and directories for which you do not have proper permission. For example, you cannot copy a file to a read-only directory using the Current Directory browser. You can do so using `movefile`.

Viewing and Making Changes to Directories

The ways to view and make changes to directories are

- "Changing the Current Working Directory and Viewing Its Contents" on page 5-26
- "Adding Directories to the MATLAB Search Path" on page 5-27
- "Changing the Display" on page 5-27

Changing the Current Working Directory and Viewing Its Contents

To change the current directory, type the directory name in the pathname edit box in the Current Directory browser, and press the **Enter** or **Return** key. That directory becomes the current working directory and the files and subdirectories in it are listed.

To view a directory that has recently been displayed, click the down arrow at the right side of the pathname edit box in the Current Directory browser. The previously displayed directories are listed, sorted by most recent to least recent. Select an entry to view the contents of that directory. You can clear the list and set the number of directories saved in the list—see "Preferences for the Current Directory Browser" on page 5-35.

To view the contents of a subdirectory within the directory being displayed, double-click the subdirectory in the Current Directory browser, or select the subdirectory and press the **Enter** or **Return** key.

To move up one level in the directory structure, click the up button 🔼 in the Current Directory browser toolbar, or press the **Back Space** key.

Function Alternative. Use dir to view the contents of the current working directory or another specified directory.

Use what to see only the MATLAB related files in a directory. With no arguments, what displays the MATLAB related files in the current working directory. Use which to display the pathname for the specified function. Use exist to see if a directory or file exists. Use fileattrib to see or set file attributes, much like attrib in DOS or chmod in UNIX.

Adding Directories to the MATLAB Search Path

From the Current Directory browser, you can add directories to the MATLAB search path. Right-click and from the context menu, select **Add to Path**. Then select one of the options:

- **Current Directory**—Adds the current directory to the path.
- **Selected Folders**—Adds the directory selected in the Current Directory browser to the path.
- **Selected Folder and Subfolders**—Adds the directory selected in the Current Directory browser to the path, and adds all of its subdirectories to the path.

Changing the Display

To specify the types of files shown in the Current Directory browser, use **View -> Current Directory Filter**. For example, you can show only M-files. If **All Files** is selected and you want to see specific file types, first clear the selection for **All Files** and then select the specific file types.

You can sort the information shown in the Current Directory browser by column. Click the title of column on which you want to sort. The display is sorted, with the information in the that column shown in ascending order. Click a second time on the column title to sort the information in descending order.

Creating, Renaming, Copying, and Removing Directories and Files

If you have write permission, you can create, copy, remove, and rename MATLAB related files and directories for the directory shown in the Current Directory browser. If you do not have write permission, you can still copy files and directories to another directory, or you can use equivalent functions, such as movefile.

Creating New Files

To create a new file in the current directory:

1 Select **New** from the context menu or **File** menu and then select the type of file to create.

An icon for that file type, for example, an M-file icon , with the default name Untitled appears at the end of the list of files shown in the Current Directory browser.

2 Type over Untitled with the name you want to give to the new file.

3 Press the **Enter** or **Return** key.

The file is added.

4 To enter the contents of the new M-file, open the file—see "Opening, Running, and Viewing the Content of Files" on page 5-30. If you created the file using the context menu, the new file opens in the Editor with a template for writing an M-file function.

Function Alternative. Use the edit function to create a new M-file.

Creating New Directories

To create a new directory in the current directory:

1 Click the new folder button in the Current Directory browser toolbar, or select **New -> Folder** from context menu.

An icon, with the default name NewFolder appears at the end of the list of files shown in the Current Directory browser.

2 Type over NewFolder with the name you want to give to the new directory.

3 Press the **Enter** or **Return** key.

The directory is added.

Function Alternative. To create a directory, use the mkdir function. For example,

```
mkdir newdir
```

creates the directory newdir within the current directory.

Renaming Files and Directories

To rename a file or directory, select the item, right-click, and select **Rename** from the context menu. Type over the existing name with the new name for the file or directory, and press the **Enter** or **Return** key. The file or directory is renamed.

Function Alternative. You can use movefile to rename a file or directory. For example,

```
movefile('myfile.m','projectresults.m')
```

renames myfile.m to projectresults.m.

Cutting or Deleting Files and Directories

To cut or delete files and directories:

1 Select the files and directories to remove. Use **Shift**+click or **Ctrl**+click to select multiple items.

2 Right-click and select **Cut** or **Delete** from the context menu.

The files and directories are removed.

On Windows platforms, files you delete from the Current Directory browser go to the Recycle bin. If you do not want the selected items to go to the Recycle bin, press **Shift+Delete**. A confirmation dialog box displays before the items are deleted.

Function Alternative. To delete a file, use the delete function. For example,

```
delete('d:/mymfiles/testfun.m')
```

deletes the file testfun.m.

To delete a directory and optionally its contents, use rmdir. For example,

```
rmdir('myfiles')
```

removes the directory myfiles from the current directory.

Copying and Pasting Files

Using the Current Directory browser, you can copy and paste files, but not directories. To copy and paste files:

1 Select the files. Use **Shift**+click or **Ctrl**+click to select multiple items.

2 Right-click and select **Copy** from the context menu.

3 Move to the directory where you want to paste the files you just copied or cut.

4 Paste the files by right-clicking and selecting **Paste** from the context menu.

Function Alternative. Use movefile or copyfile to cut and paste or to copy and paste files or directories. For example, to make a copy of the file myfun.m in the current directory, assigning it the name myfun2.m, type

```
copyfile('myfun.m','myfun2.m')
```

Opening, Running, and Viewing the Content of Files

Opening Files

You can open a file using the open feature of the Current Directory browser. The file opens in the tool associated with that file type.

To open a file, select one or more files and perform one of the following actions:

- Press the **Enter** or **Return** key.
- Right-click and select **Open** from the context menu.
- Double-click the file(s).

The files open in the appropriate tools. For example, the Editor/Debugger opens for M-files, and Simulink opens for model (mdl) files.

To open any file in the Editor, no matter what type it is, select **Open as Text** from the context menu. One exception is P-files (.p), which you cannot open.

You can also import data from a file. Select the file, right-click, and select **Import Data** from the context menu. The Import Wizard opens. See Chapter 6, "Importing and Exporting Data" for instructions to import the data.

Function Alternative. Use the open function to open a file into the tool appropriate for the file, given its file extension. Default behavior is provided for standard MATLAB file types. You can extend the interface to include other file types and to override the default behavior for the standard files. For name.ext, open performs the following actions.

File Type	Extension	Action
Figure file	fig	Open figure name.fig in a figure window.
HTML file	html	Open HTML file name.html in the Help browser.
M-file	m	Open M-file name.m in the Editor.
MAT-file	mat	Open MAT-file name.mat in the Import Wizard.
Model	mdl	Open model name.mdl in Simulink.
P-file	p	Cannot open P-files.
PDF file	pdf	Open the PDF file name.pdf in the installed PDF reader, for example, Adobe Acrobat.
Variable	not applicable	Open the numeric or string array name in the Array Editor; open calls openvar.
Other	custom	Open name.custom by calling the helper function opencustom, where opencustom is a user-defined function.

To view the content of an ASCII file, such as an M-file, use the `type` function. For example

```
type('startup')
```

displays the contents of the file `startup.m` in the Command Window.

Running M-Files

To run an M-file from the Current Directory browser, select it, right-click, and select **Run** from the context menu. The results appear in the Command Window.

Viewing Help for an M-File

You can view help for the M-file selected in the Current Directory browser. From the context menu, select **View Help**. The reference page for that function appears in the Help browser, or if a reference page does not exist, the M-file help appears.

You can view the M-file help in the Current Directory browser—for instructions, see "Preferences for the Current Directory Browser" on page 5-35.

Finding and Replacing Content Within Files
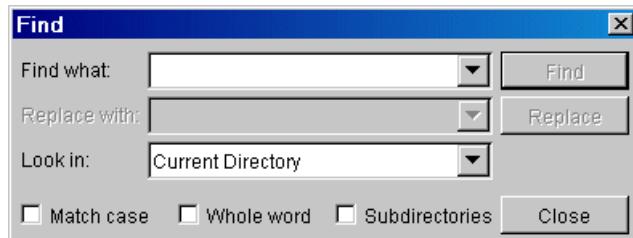
From the Current Directory browser, you can search for a specified string within files. If the file is open in the Editor, you can replace the specified string in a file.

Finding a Specified String Within a File

To search for a specified string in files:

1 Click the find button 🏶 in the Current Directory browser toolbar.
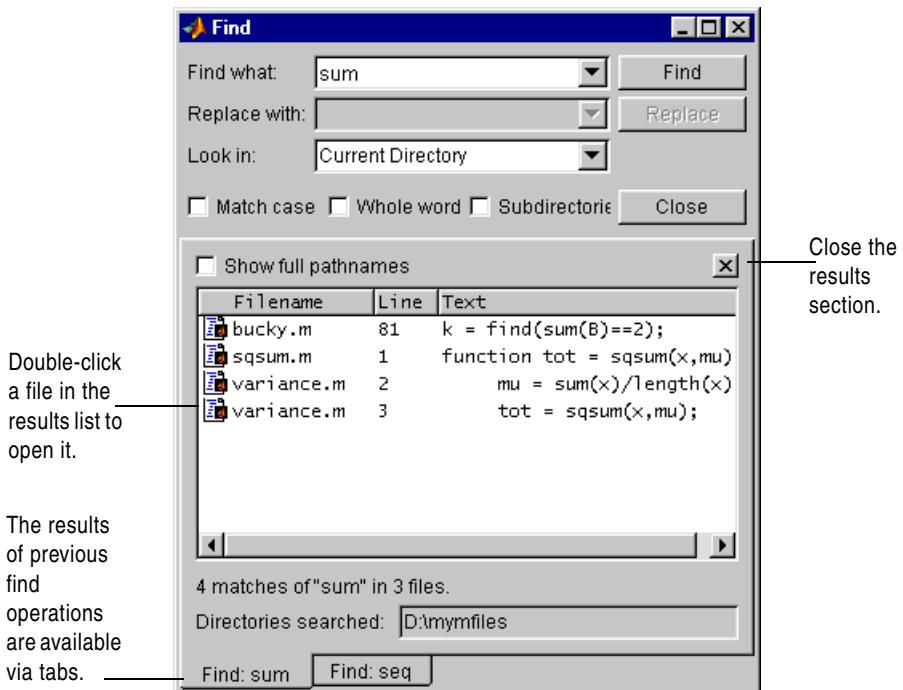
The **Find** dialog box appears. This is similar to the **Find & Replace** dialog box in the Editor.

2 Complete the **Find** dialog box to find all occurrences of the string you specify.

- Type the string in the **Find what** field.
- Select the directories or files to search through from the **Look in** listbox, or type a directory name directly in this field. If documents are open in the Editor, you can select those files and related files.
- Constrain the search by checking **Match case** or **Whole word**.
- Select **Subdirectories** if you want the search to also look through the subdirectories. Also select this if you want to find files in private directories or method directories (those that start with @).

3 Click **Find**.

Results appear in the lower part of the **Find** dialog box and include the filename, M-file line number, and content of that line.

Close the results section.

Double-click a file in the results list to open it.

The results of previous find operations are available via tabs.

4 Open any M-file(s) in the results list by doing one of the following:

- Double-clicking the file(s)
- Selecting the file(s) and pressing the **Enter** or **Return** key
- Right-clicking the file(s) and selecting **Open** from the context menu

The M-file(s) opens in the Editor, scrolled to the line number shown in the results section of the **Find** dialog box.

5 If you perform another search, the results of each search are accessible via tabs just below the current results list. Click a tab to see that results list as well as the search criteria.

Function Alternative. Use lookfor to search for the specified string in the help in all M-files on the search path.

Replacing a Specified String Within Files

After searching for a string within a file, you can replace the string:

1 Open the file in MATLAB Editor. You can open the file from the Current Directory browser **Find** dialog box results list—see step 4 in "Finding a Specified String Within a File" on page 5-33. Be sure that the file in which you want to replace the string is the current file in the Editor.

2 In the Editor, click the find button 🔍.

The Current Directory browser **Find** dialog box is replaced by the Editor **Find & Replace** dialog box, which looks very similar.

3 In the **Look in** field in the **Find & Replace** dialog box, select the name of the file in which you want to replace the string.

The **Replace** button in the **Find & Replace** dialog box becomes selectable.

4 In the **Replace with** field, type the text that is to replace the specified string.

5 Click **Replace** to replace the string in the selected line, or click **Replace All** to replace all instances in the currently open file.

The text is replaced.

6 To save the changes, select **Save** from the **File** menu in the Editor.
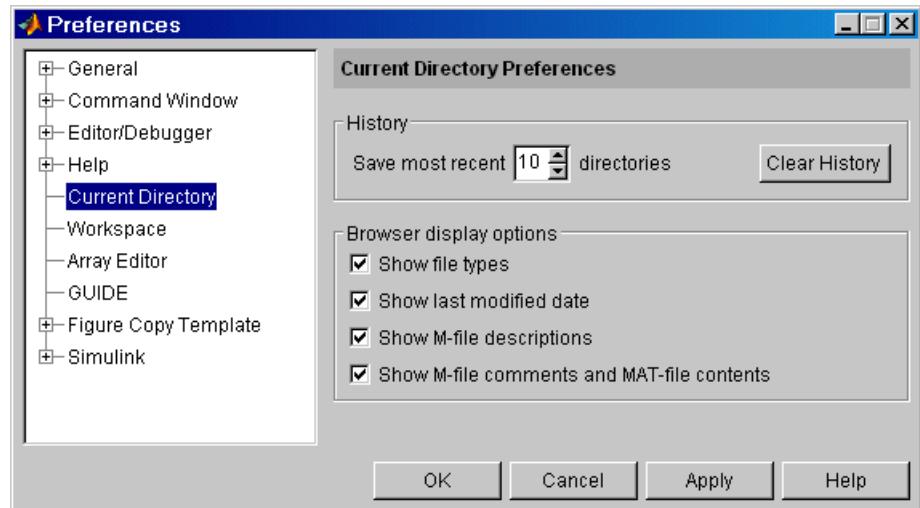
Accessing Source Control Features

Select a file or files in the Current Directory browser and right-click to view the context menu. From there you can access features for **Source Control**. For details on these features, see Chapter 8, "Interfacing with Source Control Systems".

Preferences for the Current Directory Browser

Using preferences, you can specify the number of recently used current directories to maintain in the history list as well as the type of information to display in the Current Directory browser.

From the Current Directory browser **File** menu, select **Preferences**. The **Current Directory Preferences** panel appears in the **Preferences** dialog box.

History

The dropdown list in the Current Directory browser toolbar, as well as in the MATLAB desktop **Current Directory** field, show the history of current directories, that is, the most recently used current directories.

Removing Directories. To remove the entries in the list, click **Clear History**. The list is cleared immediately.

Saving Directories. When the MATLAB session ends, the list of directories will be maintained. Use the **Save most recent directories** field to specify how many directories will appear on the list at the start of the next MATLAB session.

Browser Display Options

In the Current Directory browser, you can view the file type, last modified date, M-file descriptions, and M-file comments and MAT-file contents by selecting the appropriate **Browser display options**.

6

Importing and Exporting Data

Overview

MATLAB provides many ways to load data from disk files or the clipboard into the workspace, a process called *importing* data, and to save workspace variables to a disk file, a process called *exporting* data. Your choice of which mechanism to use depends on two factors:

- The operation you are performing, that is, whether you are importing or exporting data
- The format of the data: text, binary, or a standard format such as HDF

Note The easiest way to import data into MATLAB is to use the Import Wizard. When you use the Import Wizard, you do not need to know the format of the data. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically. For more information, see "Using the Import Wizard with Text Data" on page 6-5 and "Using the Import Wizard with Binary Data Files" on page 6-20. You can also use the Import Wizard to import HDF data. See "Using the HDF Import Tool" on page 6-31 for more information.

Text Data

In text format, the data values are American Standard Code for Information Interchange (ASCII) codes that represent alphabetic and numeric characters. ASCII text data can be viewed in a text editor. For more information about working with text data, see

- "Importing Text Data" on page 6-4
- "Exporting ASCII Data" on page 6-16

Binary Data

In binary format, the values are not ASCII codes and cannot be viewed in a text editor. Binary files contain data that represents images, sounds, and other information. For more information about working with binary data, see

- "Importing Binary Data" on page 6-20
- "Exporting Binary Data" on page 6-26

Other Formats

MATLAB also supports the importing of scientific data that uses the Hierarchical Data Format (HDF). For more information about working with HDF data, see

- "Importing HDF Data" on page 6-31
- "Exporting MATLAB Data in an HDF File" on page 6-58

Low-Level File I/O

MATLAB also supports C-style, low-level I/O functions that you can use with any data format. For more information, see "Using Low-Level File I/O Functions" on page 6-69.

Importing Text Data

The section describes various ways to import text data into MATLAB. It covers these topics:

"Using the Import Wizard with Text Data"	Describes how to use the MATLAB Import Wizard to import text data. This is the easiest way to import text data into MATLAB.
"Using Import Functions with Text Data" on page 6-9	Provides an overview of all the MATLAB text import functions.
"Importing Numeric Text Data" on page 6-11	Describes how to use the `load` command to import numeric data.
"Importing Delimited ASCII Data Files" on page 6-12	Describes how to use the `dlmread` command to import numeric data that is delimited by some character, such as a tab, comma, or space.
"Importing Numeric Data with Text Headers" on page 6-13	Describes how to use the `textread` command to import a table of delimited numeric data that includes text headers.
"Importing Mixed Alphabetic and Numeric Data" on page 6-14	Describes how to use the `textread` command to import data that mixes both numeric and text data.

Caution When you import data into the MATLAB workspace, you overwrite any existing variable in the workspace with the same name.

Using the Import Wizard with Text Data

To import text data using the Import Wizard, perform these steps:

1 Start the Import Wizard, by selecting the **Import Data** option on the MATLAB **File** menu. MATLAB displays a file selection dialog box. You can also use the uiimport function to start the Import Wizard.

To use the Import Wizard to import data from the clipboard, select the **Paste Special** option on the MATLAB **Edit** menu. You can also right-click in the MATLAB Command Window and choose **Paste Special** from the context menu. Skip to step 3 to continue importing from the clipboard.

2 Specify the file you want to import in the file selection dialog box and click **Open**. The Import Wizard opens the file and attempts to process its contents.

3 Specify the character used to separate the individual data items. This character is called the delimiter or column separator. The Import Wizard can determine the delimiter used in many cases. However, you might need to specify the character used in your text file. See "Specifying the Delimiter" on page 6-5 for more information. Once the Import Wizard has correctly processed the data, click **Next**.

4 Select the variables that you want to import. By default, the Import Wizard puts all the numeric data in one variable and all the text data in other variables, but you can choose other options. See "Selecting the Variables to Import" on page 6-7 for more information.

5 Click **Finish** to import the data into the workspace.
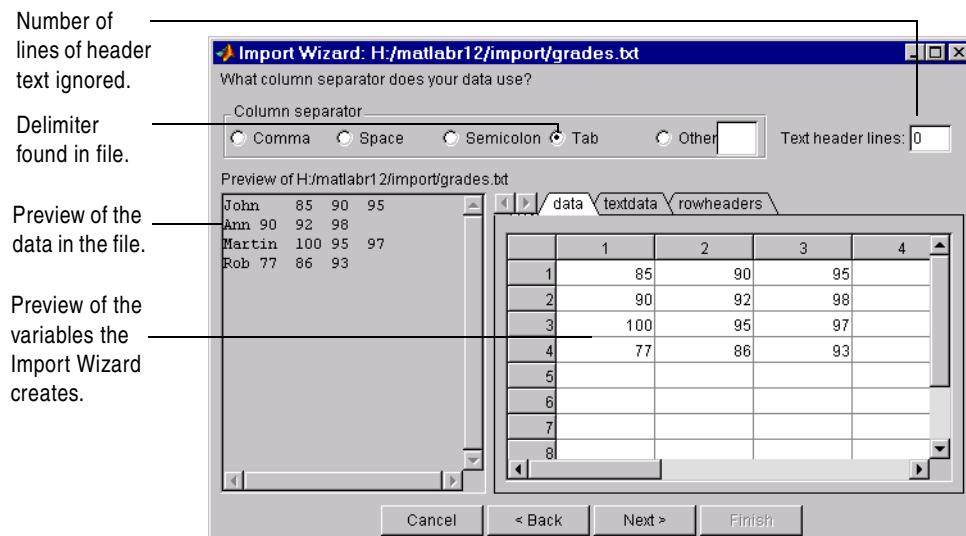
Specifying the Delimiter

When the Import Wizard opens a text file, or copies data from the clipboard, it displays a portion of the raw data in the preview pane of the dialog box. You can use this display to verify that the file contains the data you expected.

The Import Wizard also attempts to process the data, identifying the delimiter used in the data. The Import Wizard displays the variables it has created based on its interpretation of the delimiter, using tabbed panels to display multiple variables.

For example, in the following figure, the Import Wizard has opened this sample file, grades.txt.

```
John      85       90       95
Ann       90       92       98
Martin    100      95       97
Rob       77       86       93
```

Number of
lines of header
text ignored.

Delimiter
found in file.

Preview of the
data in the file.

Preview of the
variables the
Import Wizard
creates.

In the figure, note how the Import Wizard has correctly identified the tab character as the delimiter used in the file and has created three variables from the data:

- data contains all the numeric data in the file.
- textdata contains all the text found in the file.
- rowheaders contains the names in the left-most column of data.
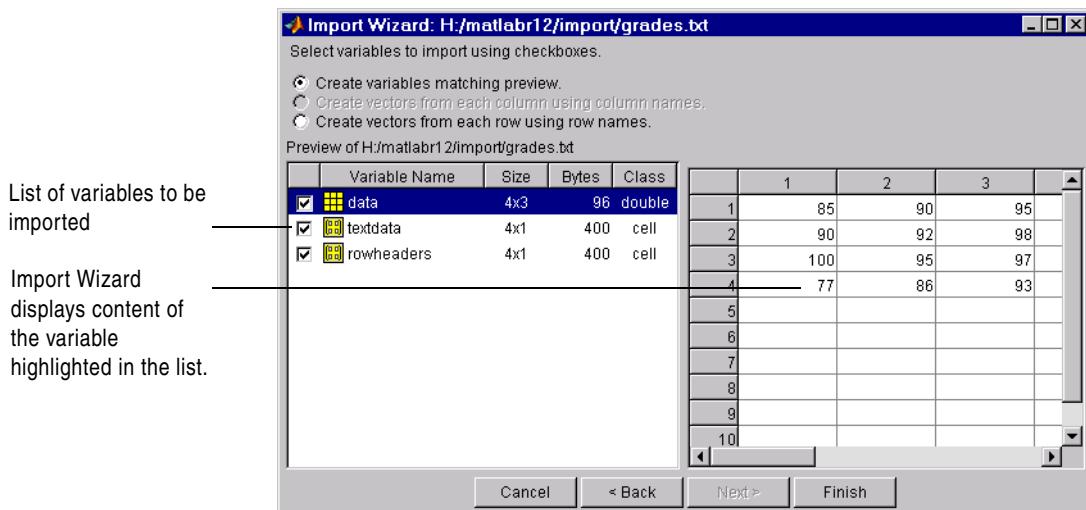
Handling Alphabetic Data. The Import Wizard recognizes data files that use row or column headers and extracts these headers into separate variables. It can also ignore any text header lines that might precede the data in a file.

Specifying Other Delimiters. If the Import Wizard cannot determine the delimiter used in the data, it displays a preview of the raw data, as before, but the variables it displays are not correct. If your data uses a character other than a comma, space, tab, or semicolon as a delimiter, you must specify it by clicking the **Other** button and entering the character in the text box. The Import Wizard immediately reprocesses the data, displaying the new variables it creates.

Selecting the Variables to Import

The Import Wizard displays a list of the variables it has created from your data. To select a variable to import, click in the check box next to its name. By default, all variables are selected.
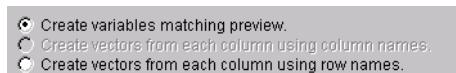
The Import Wizard displays the contents of the variable that is highlighted in the list in the right pane of the dialog box. To view the contents of one of the other variables, click it. Choose the variables you want to import and click **Next**.

List of variables to be imported

Import Wizard displays content of the variable highlighted in the list.

Changing the Variable Selection. By default, the Import Wizard puts all the numeric data in the file into one variable. If the file contains text data, the Import Wizard puts it in a separate variable. If the file contains row or column

headers, the Import Wizard puts them in separate variables, called `rowheaders` or `colheaders`, respectively.

In some cases, it might be more convenient to create a variable from each row or column of data and use the row header or column header text as the name of each variable. To do this, click the appropriate button from the list of buttons at the top of the dialog box.
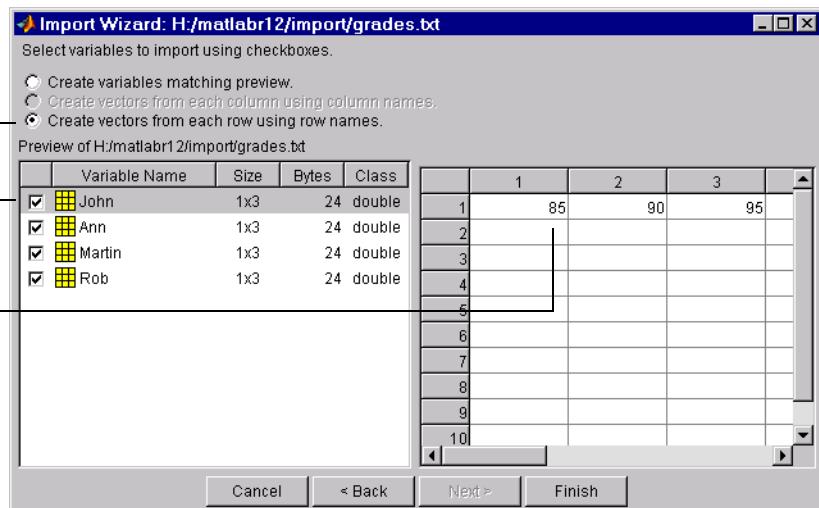
For example, it eases calculation of the student averages if you create a separate variable for each student, which contains that student's grades. To create these variables, click the **Create variables from each column using row names** button. When you click this option, the Import Wizard reprocesses the file, creating these new variables.

Select this option to create variables from row header.

List contains variables by row header.

The variable is a vector made from row of data file.

When you are satisfied with the list of variables to be imported, click **Next** to bring the data into the MATLAB workspace. This button also dismisses the Import Wizard. The Import Wizard displays a message in the MATLAB Command Window, reporting that it created variables in the workspace. In the

following example, note how the numeric text data in each variable is imported as an array of doubles.

```
Import Wizard created variables in the current workspace.
>> whos
  Name          Size          Bytes  Class

  Ann           1x3              24  double array
  John          1x3              24  double array
  Martin        1x3              24  double array
  Rob           1x3              24  double array

Grand total is 12 elements using 96 bytes
```

Using Import Functions with Text Data

To import text data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

The text data must be formatted in a uniform pattern of rows and columns, using a text character, called a *delimiter* or *column separator*, to separate each data item. The delimiter can be space, comma, semicolon, tab, or any other character. The individual data items can be alphabetic or numeric characters or a mix of both.

The text file can also contain one or more lines of text, called *header lines*, or can use text headers to label each column or row. The following example illustrates a tab-delimited text file with header text and row and column headers.

```
Text header line ─────────┐
                          │
                Class Grades for Spring Term
Column Headers ──────────  Grade1 Grade2 Grade3
                John      85      90      95
Row Headers ──── Ann       90      92      98
                Martin   100      95      97
                Rob       77      86      93
Tab-delimited ─────────────┘
data
```

To find out how your data is formatted, view it in a text editor. After you determine the format, scan the data format samples in Table 6-1 and look for the sample that most closely resembles the format of your data. Read the topic referred to in the table for more information.

Table 6-1: ASCII Data File Formats and MATLAB Import Commands

Data Format Sample	File Extension	Description
1 2 3 4 5 6 7 8 9 10	.txt .dat or other	See "Importing Numeric Text Data" on page 6-11 for more information. You can also use the Import Wizard for this data format. See "Using the Import Wizard with Text Data" on page 6-5 for more information.
1; 2; 3; 4; 5 6; 7; 8; 9; 10 or 1, 2, 3, 4, 5 6, 7, 8, 9, 10	.txt .dat .csv or other	See "Importing Delimited ASCII Data Files" on page 6-12 for more information. You can also use the Import Wizard for this data format. See "Using the Import Wizard with Text Data" on page 6-5 for more information.
Ann Type1 12.34 45 Yes Joe Type2 45.67 67 No	.txt .dat or other	See "Importing Numeric Data with Text Headers" on page 6-13 for more information.
Grade1 Grade2 Grade3 91.5 89.2 77.3 88.0 67.8 91.0 67.3 78.1 92.5	.txt .dat or other	See "Importing Numeric Data with Text Headers" on page 6-13 for more information. You can also use the Import Wizard for this data format. See "Using the Import Wizard with Text Data" on page 6-5 for more information.

If you are familiar with MATLAB import functions but are not sure when to use them, view Table 6-2, which compares the features of each function.

Table 6-2: ASCII Data Import Function Feature Comparison

Function	Data Type	Delimiters	Number of Return Values	Notes
csvread	Numeric data	Only commas	One	Primarily used with spreadsheet data. See also the binary format spreadsheet import functions.
dlmread	Numeric data	Any character	One	Flexible and easy to use.
fscanf	Alphabetic and numeric; however, both types returned in a single return variable	Any character	One	Part of low-level file I/O routines. Requires use of fopen to obtain file identifier and fclose after read.
load	Numeric data	Only spaces	One	Easy to use. Use the functional form of load to specify the name of the output variable.
textread	Alphabetic and numeric	Any character	Multiple return values.	Flexible, powerful, and easy to use. Use format string to specify conversions.

Importing Numeric Text Data

If your data file contains only numeric data, you can use many of the MATLAB import functions (listed in Table 6-2), depending on how the data is delimited. If the data is rectangular, that is, each row has the same number of elements, the simplest command to use is the load command. (The load command can also be used to import MAT-files, the MATLAB binary format for saving the workspace.)

For example, the file named `my_data.txt` contains two rows of numbers
delimited by space characters.

```
1 2 3 4 5
6 7 8 9 10
```

When you use `load` as a command, it imports the data and creates a variable
in the workspace with the same name as the filename, minus the file extension.

```
load my_data.txt;
whos
   Name          Size          Bytes   Class

   my_data       2x5            80     double array

my_data

my_data =
    1   2   3   4    5
    6   7   8   9   10
```

If you want to name the workspace variable something other than the file
name, use the functional form of `load`. In the following example, the data from
`my_data.txt` is loaded into the workspace variable `A`.

```
A = load('my_data.txt');
```

Importing Delimited ASCII Data Files

If your data file uses a character other than a space as a delimiter, you have a
choice of several import functions you can use. (See Table 6-2 for a complete
list.) The simplest to use is the `dlmread` function.

For example, consider a file named `ph.dat` whose contents are separated by
semicolons:

```
7.2;8.5;6.2;6.6
5.4;9.2;8.1;7.2
```

To read the entire contents of this file into an array named `A`, enter

```
A = dlmread('ph.dat', ';');
```

You specify the delimiter used in the data file as the second argument to dlmread. Note that, even though the last items in each row are not followed by a delimiter, dlmread can still process the file correctly. dlmread ignores space characters between data elements. So, the preceding dlmread command works even if the contents of ph.dat are

```
7.2;   8.5;       6.2;6.6
5.4;   9.2   ;8.1;7.2
```

Importing Numeric Data with Text Headers

To import an ASCII data file that contains text headers, use the textread function, specifying the headerlines parameter. textread accepts a set of predefined parameters that control various aspects of the conversion. (For a complete list of these parameters, see the textread reference page.) The headerlines parameter lets you specify the number of lines at the head of the file that textread should ignore.

For example, the file grades.dat contains formatted numeric data with a one-line text header.

```
Grade1 Grade2 Grade3
78.8    55.9   45.9
99.5    66.8   78.0
89.5    77.0   56.7
```

To import this data, use this command:

```
[grade1 grade2 grade3] = textread('grades.dat','%f %f %f',...
'headerlines',1)

grade1 =
    78.8000
    99.5000
    89.5000

grade2 =
    55.9000
    66.8000
    77.0000

grade3 =
```

```
45.9000
78.0000
56.7000
```

Importing Mixed Alphabetic and Numeric Data

If your data file contains a mix of alphabetic and numeric ASCII data, use the textread function to import the data. textread can return multiple output variables, and you can specify the data type of each variable.

For example, the file mydata.dat contains a mix of alphabetic and numeric data:

```
Sally     Type1 12.34 45 Yes
Larry     Type2 34.56 54 Yes
Tommy     Type1 67.89 23 No
```

Note To read an ASCII data file that contains numeric data with text column headers, see "Importing Numeric Data with Text Headers" on page 6-13.

To read the entire contents of the file mydata.dat into the workspace, specify the name of the data file and the format string as arguments to textread. In the format string, you include conversion specifiers that define how you want each data item to be interpreted. For example, specify %s for string data, %f for floating-point data, and so on. (For a complete list of format specifiers, see the textread reference page.)

For each conversion specifier in your format string, you must specify a separate output variable. textread processes each data item in the file as specified in the format string and puts the value in the output variable. The number of output variables must match the number of conversion specifiers in the format string.

In this example, textread reads the file mydata.dat, applying the format string to each line in the file until the end of the file.

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

```
        'Larry'
        'Tommy'

types =
    'Type1'
    'Type2'
    'Type1'

x =
   12.3400
   34.5600
   67.8900

y =
     45
     54
     23

answer =
    'Yes'
    'Yes'
    'No'
```

If your data uses a character other than a space as a delimiter, you must use the textread parameter 'delimiter' to specify the delimiter. For example, if the file mydata.dat used a semicolon as a delimiter, you would use this command:

```
[names,types,x,y,answer]=textread('mydata.dat','%s %s %f ...
    %d %s', 'delimiter',';')
```

See the textread reference page for more information about these optional parameters.

Exporting ASCII Data

This section describes how to use MATLAB functions to export data in several common ASCII formats. For example, you can use these functions to export a MATLAB matrix as a text file where the rows and columns are represented as space-separated, numeric values. The function you use depends on the amount of data you want to export and its format. Topics covered include

"Exporting Delimited ASCII Data Files" on page 6-17	Describes how to use the save command or the dlmwrite function
"Using the diary Command to Export Data" on page 6-18	Describes how to use the diary command to export data

If you are not sure which section describes your data, scan the data format samples in Table 6-3 and look for the sample that most nearly matches the data format you want to create. Then read the section referred to in the table.

If you are familiar with MATLAB export functions but are not sure when to use them, view Table 6-4, which compares the features of each function.

Note If C or Fortran routines for writing data files in the form needed by other applications exist, create a MEX file to write the data. See the External Interfaces/API documentation for more information.

Table 6-3: ASCII Data File Formats and MATLAB Export Commands

Data Format Sample	MATLAB Export Function
1 2 3 4 5 6 7 8 9 10	See "Exporting Delimited ASCII Data Files" on page 6-17 and "Using the diary Command to Export Data" on page 6-18 for information about these options.
1; 2; 3; 4; 5; 6; 7; 8; 9; 10;	See "Exporting Delimited ASCII Data Files" on page 6-17 for more information. The example shows a semicolon-delimited file, but you can specify another character as the delimiter.

Table 6-4: ASCII Data Export Function Feature Comparison

Function	Use With	Delimiter	Notes
csvwrite	Numeric data	Only comma	Primarily used with spreadsheet data. See also the binary format spreadsheet export functions.
diary	Numeric data or cell array	Only space	Can be used for small arrays. Requires editing of data file to remove extraneous text.
dlmwrite	Numeric data	Any character	Easy to use, flexible.
fprintf	Alphabetic and numeric data	Any character	Part of low-level file I/O routines. This is the most flexible command but also the most difficult to use. You must use fopen to obtain a file identifier before writing the data and fclose to close the file after writing the data.
save	Numeric data	Tab or space	Easy to use; output values are high precision.

Exporting Delimited ASCII Data Files

To export an array as a delimited ASCII data file, you can use either the save command, specifying the -ASCII qualifier, or the dlmwrite function. The save command is easy to use; however, the dlmwrite function provides more flexibility, allowing you to specify any character as a delimiter and to export subsets of an array by specifying a range of values.

Using the save Command

To export the array A,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

use the save command, as follows:

```
save my_data.out A -ASCII
```

If you view the created file in a text editor, it looks like this:

```
1.0000000e+000   2.0000000e+000   3.0000000e+000   4.0000000e+000
5.0000000e+000   6.0000000e+000   7.0000000e+000   8.0000000e+000
```

By default, save uses spaces as delimiters but you can use tabs instead of spaces by specifying the -tabs qualifier.

When you use save to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. If you write the character string 'hello' to a file, save writes the values

```
104 101 108 108 111
```

Using the dlmwrite Function

To export an array in ASCII format and specify the delimiter used in the file, use the dlmwrite function.

For example, to export the array A,

```
A = [ 1 2 3 4 ; 5 6 7 8 ];
```

as an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

If you view the created file in a text editor, it looks like this:

```
1;2;3;4 5;6;7;8
```

Note that dlmwrite does not insert delimiters at the end of rows.

By default, if you do not specify a delimiter, dlmwrite uses a comma as a delimiter. You can specify a space (' ') as a delimiter or, if you specify empty quotes (''), no delimiter.

Using the diary Command to Export Data

To export small numeric arrays or cell arrays, you can use the diary command. diary creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array A in your workspace,

```
A = [ 1 2 3 4; 5 6 7 8 ];
```

execute these commands at the MATLAB prompt to export this array using
`diary`:

1 Turn on the `diary` function. You can optionally name the output file `diary`
creates.

```
diary my_data.out
```

2 Display the contents of the array you want to export. This example displays
the array A. You could also display a cell array or other MATLAB data type.

```
A =
    1    2    3    4
    5    6    7    8
```

3 Turn off the `diary` function.

```
diary off
```

`diary` creates the file `my_data.out` and records all the commands executed
in the MATLAB session until it is turned off.

```
A =

    1    2    3    4
    5    6    7    8

diary off
```

4 Open the `diary` file `my_data.out` in a text editor and remove all the
extraneous text.

Importing Binary Data

This section describes how to import data in many common binary formats. Topics covered include:

"Using the Import Wizard with Binary Data Files" on page 6-20	Describes how to use the MATLAB Import Wizard to import binary data. This is the easiest way to import binary data into MATLAB.
"Using Import Functions with Binary Data" on page 6-22	Provides an overview of all the MATLAB functions for importing various types of binary data.

Caution When you import data into the MATLAB workspace, it overwrites any existing variable in the workspace with the same name.

Using the Import Wizard with Binary Data Files

To import binary data using the Import Wizard, perform these steps:

1 Start the Import Wizard, by selecting the **Import Data** option on the MATLAB **File** menu. MATLAB displays a file selection dialog box. You can also use the uiimport function to start the Import Wizard.

To use the Import Wizard to import data from the clipboard, select the **Paste Special** option on the MATLAB **Edit** menu. You can also right-click in the MATLAB command window and choose **Paste Special** from the context menu. Skip to step 3 to continue importing from the clipboard.

2 Specify the file you want to import in the file selection dialog box and click **Open**. The Import Wizard opens the file and attempts to process its contents. See "Viewing the Variables" on page 6-21 for more information.

3 Select the variables that you want to import. By default, the Import Wizard creates variables depending on the type of data in the file.

4 Click **Finish** to import the data into the workspace.

Viewing the Variables
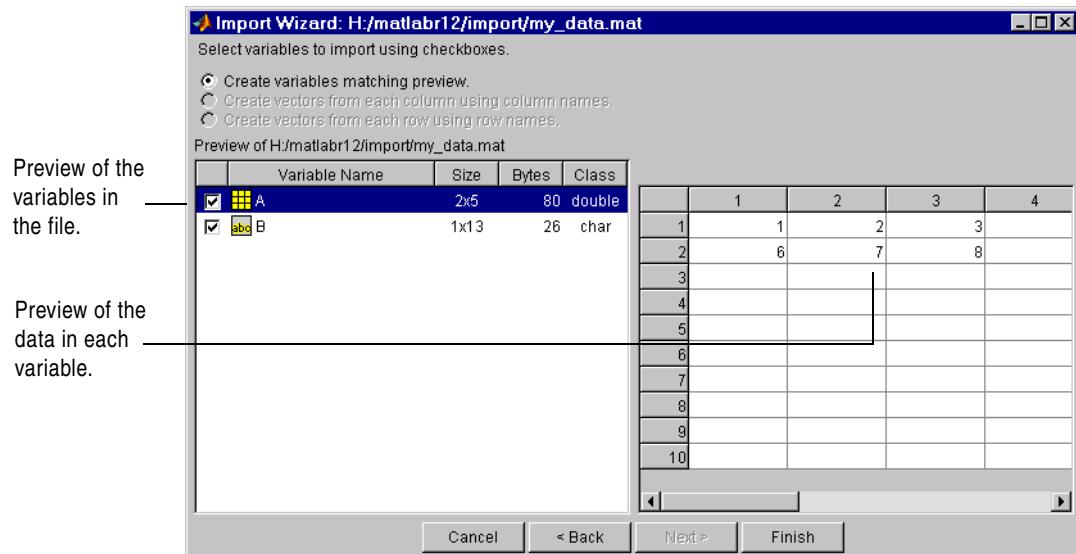
When the Import Wizard opens a binary data file, it attempts to process the data in the file, creating variables from the data it finds in the file.

For example, if you use the Import Wizard to import this sample MAT-file, my_data.mat,
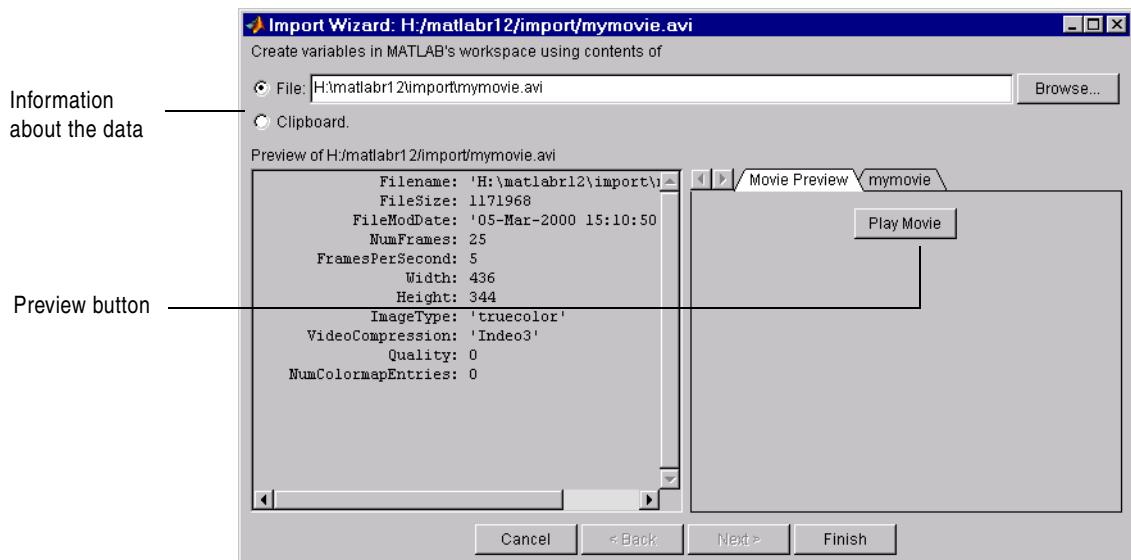
```
A =
    1 2 3 4 5
    6 7 8 9 10

B =
    a test string
```

it creates two variables, listed in the preview pane. To select a variable to import, click in the check box next to its name. All variables are preselected by default.

Preview of the variables in the file.

Preview of the data in each variable.

For other binary data types, such as images and sound files, the Import Wizard displays information about the data in the left pane and provides a preview button in the right pane of the dialog box. Click the preview button to view (or listen to) the data.

For example, when used to import a movie in Audio Video Interleaved (AVI) format, the Import Wizard displays this dialog box.

Information about the data

Preview button

Using Import Functions with Binary Data

To import binary data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

To find the function designed to work with a particular binary data format, scan the data formats listed in Table 6-5. The table lists the binary formats and the MATLAB high-level functions you use to import them, along with pointers to additional information. To view the alphabetical list of MATLAB binary data export functions, see Table 6-6.

Note If MATLAB does not support a high-level function that works with a particular binary data format, use the MATLAB low-level file I/O functions to import the data. You must know how the binary data is formatted in the file. See "Using Low-Level File I/O Functions" on page 6-69 for more information.

Table 6-5: Binary Data Formats and MATLAB Import Functions

Data Format	File Extension	Function
Audio files	`.au`	`auread` (on Sun systems)
	`.wav`	`wavread` (on Microsoft Windows systems)
Audio-video Interleaved (AVI)	`.avi`	`aviread`
Band-interleaved data	No standard file extension	`multibandread`
Common Data Format (HDF)	`.cdf`	`cdfread`
Hierarchical Data Format (HDF)	`.hdf`	`imread` Use `imread` only for HDF raster image files. For all other HDF files, see "Importing HDF Data" on page 6-31 for complete information.

Image files	`.bmp`
	`.cur`
	`.gif`
	`.hdf`
	`.ico`
	`.jpg (.jpeg)`
	`.pbm`
	`.pcx`
	`.pgm`
	`.png`
	`.pnm`
	`.ppm`
	`.ras`
	`.tif (.tiff)`
	`.xwd`
		`imread`

Data Format	File Extension	Function
MATLAB proprietary format (MAT-files)	`.mat`	`load` See "Loading a Saved Workspace and Importing Data" on page 5-6 for more information.
Spreadsheets (Excel or Lotus 123)	`.xls` `.wk1`	`xlsread`

Table 6-6: Binary Data Import Functions

Function	File Extension	Data Format
`auread`	`.au`	Import sound data in Sun Microsystems format.
`aviread`	`.avi`	Import audio-visual data in AVI format.
`cdfread`	`.cdf`	Import Common Data Format (CDF) data.
`hdf`	`.hdf`	Import data in Hierarchical Data Format (HDF). For HDF image file formats, use `imread`. For all other HDF files, see "Importing HDF Data" on page 6-31 for complete information.

`imread`	`.bmp`
	`.cur`
	`.gif`
	`.hdf`
	`.ico`
	`.jpg (.jpeg)`
	`.pbm`
	`.pcx`
	`.pgm`
	`.png`
	`.pnm`
	`.ppm`
	`.ras`
	`.tif (.tiff)`
	`.xwd`
		Import images in many formats.

Function	File Extension	Data Format
load	.mat	Import MATLAB workspace variables in MAT-files format.
multibandread	No standard file extension	Import three-dimensional, band-interleaved data.
wavread	.wav	Import sound data in Microsoft Windows format.
wk1read	.wk1	Import data in Lotus 123 spreadsheet format.
xlsread	.xls	Import data in Microsoft Excel spreadsheet format.

Exporting Binary Data

To export binary data in one of the standard binary formats, you can use the MATLAB high-level function designed to work with that format.

To find the function designed to work with a particular binary data format, scan the data formats listed in Table 6-7. The table lists the binary formats and the MATLAB high-level functions you use to import them, with pointers to sources of additional information.

Note If MATLAB does not support a high-level function that works with a data format, you can use the MATLAB low-level file I/O functions, if you know how the binary data is formatted in the file. See "Using Low-Level File I/O Functions" on page 6-69 for more information.

Table 6-7: Binary Data Formats and MATLAB Export Functions

Data Format	File Extension	Description
Audio files	`.au` `.wav`	Use the `auwrite` function to export audio files on Sun Microsystems platforms and the `wavwrite` function to export audio files in Microsoft Windows format.
Audio Video Interleaved (AVI)	`.avi`	You must use the `avifile`, `addframe`, and the `close` function overloaded for AVI data to export a sequence of MATLAB figures in AVI format. See "Exporting MATLAB Graphs in AVI Format" on page 6-28 for more information.
Band Interleaved	No standard file extension	Use the `multibandwrite` function.
Common Data Format (CDF)	`.cdf`	To export image data in CDF format, use `cdfwrite`.

Data Format	File Extension	Description
Hierarchical Data Format (HDF)	`.hdf`	To export image data in HDF format, use `imwrite`. For all other HDF formats, see "Exporting MATLAB Data in an HDF File" on page 6-58 for complete information.

Image files	`.bmp`
	`.gif`
	`.hdf`
	`.jpg (.jpeg)`
	`.pbm`
	`.pcx`
	`.pgm`
	`.png`
	`.pnm`
	`.ppm`
	`.ras`
	`.tif (.tiff)`
	`.xwd`
		Use the `imwrite` function to export image data in many different formats.

MATLAB proprietary format (MAT-files)	`.mat`	Use the `save` command to export data in MATLAB proprietary format. See "Saving the Current Workspace" on page 5-4 for more information.
Spreadsheets	`.wk1`	Use the `wk1write` function to export data in Lotus 123 format.

To view the alphabetical list of MATLAB binary data export functions, see Table 6-8.

Table 6-8: Binary Data Export Functions

Function	File Extension	Data Format
`auwrite`	`.au`	Export sound data in Sun Microsystems format.
`avifile`	`.avi`	Export audio-visual data in AVI format. Creates an AVI file object. See "Exporting MATLAB Graphs in AVI Format" on page 6-28 for more information.

Function	File Extension	Data Format
cdfwrite	.cdf	Export data as a Common Data Format file.
hdf	.hdf	Export data in Hierarchical Data Format (HDF). For HDF image file formats, use imwrite. For all other HDF files, see "Exporting MATLAB Data in an HDF File" on page 6-58 for complete information.
imwrite	.bmp .hdf .jpg (jpeg) .pbm .pcx .pgm .png .pnm .ppm .ras .tif (.tiff) .xwd	Export image files in many formats.
multibandwrite	No standard file extension	Export band interleaved data.
save	.mat	Export MATLAB variables in MAT-file format.
wavwrite	.wav	Export sound data on Microsoft Windows platforms.
wk1write	.wk1	Export data in Lotus 123 spreadsheet format.

Exporting MATLAB Graphs in AVI Format

In MATLAB, you can save a sequence of graphs as a *movie* that can then be played back using the movie function. You can export a MATLAB movie by saving it in MAT-file format, like any other MATLAB workspace variable. However, anyone who wants to view your movie must have MATLAB. (For more information about MATLAB movies, see the "Animation" section in the MATLAB Graphics documentation.)

To export a sequence of MATLAB graphs in a format that does not require MATLAB for viewing, save the figures in Audio Video Interleaved (AVI)

format. AVI is a file format that allows animation and video clips to be played on a PC running Windows or on UNIX systems.

Creating an AVI Format Movie

To export a sequence of MATLAB graphs as an AVI format movie, perform these steps:

1 Create an AVI file, using the avifile function.

2 Capture the sequence of graphs and put them into the AVI file, using the addframe function.

3 Close the AVI file, using the close function, overloaded for AVI files.

Note To convert an existing MATLAB movie into an AVI file, use the movie2avi function.

For example, this code example exports a sequence of MATLAB graphs as the AVI file mymovie.avi.

```
aviobj = avifile('mymovie.avi','fps',5);

for k=1:25
    h = plot(fft(eye(k+16)));
    set(h,'EraseMode','xor');
    axis equal;
    frame = getframe(gca);
    aviobj = addframe(aviobj,frame);
end

aviobj = close(aviobj);
```

Note the following items in this code example:

• The avifile function creates an AVI file and returns a handle to an AVI file object. AVI file objects support properties that let you control various characteristics of the AVI movie, such as colormap, compression, and quality. (See the avifile reference page for a complete list.) avifile uses default

values for all properties, unless you specify a value. In the example, the call to `avifile` explicitly sets the value of the frames per second (`fps`) property.

- The example uses a `for` loop to capture the series of graphs to be included in the movie. You typically use `addframe` to capture a sequence of graphs for AVI movies. However, because this particular MATLAB animation uses `XOR` graphics, you must call `getframe` to capture the graphs and then call `addframe` to add the captured frame to the movie. See the `addframe` reference page for more information.

- The example calls the `close` function to finish writing the frames to the file and to close the file.

Importing HDF Data

Hierarchical Data Format (HDF) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF-EOS is an extension of HDF developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS).

HDF and HDF-EOS files can contain multidimensional numeric arrays or text data, called *data sets,* in HDF terminology. MATLAB provides three ways to import HDF or HDF-EOS data sets into the MATLAB workspace:

Using the HDF Import Tool	Describes a graphical user interface you can use to navigate through the data sets and metadata in an HDF file and import selected data sets from the file.
Using the MATLAB high-level import function	Describes how to use `hdfread` to import data from an HDF file.
Using the MATLAB low-level HDF functions	Describes how to use the MATLAB interface to the HDF and HDF-EOS application programming interfaces (APIs). For example, to import an HDF scientific data set, use the `hdfsd` function.

Note MATLAB supports Version 4.1r3 of the NCSA multifile APIs to HDF data. The multifile APIs replace the original NCSA APIs. MATLAB does not support HDF version 5.0, which is a completely new format and is not compatible with Version 4.1r3. To find more information about HDF, see "Getting More Information About HDF" on page 6-67.

Using the HDF Import Tool

To import data using the HDF Import Tool, perform these steps:

1 Open the HDF file in the HDF Import Tool.

2 Select a data set to import.

3 Specify the subset of the data set that you want to import. This is an optional step.

4 Import the data.

The following sections provide more detail about each of these steps.

Opening a File in the HDF Import Tool

To open an HDF file in the HDF Import Tool, select the **Import Data** option from the MATLAB **File** menu. MATLAB displays a file selection dialog box. If you select an HDF file, the Import Wizard automatically starts the HDF Import Tool.
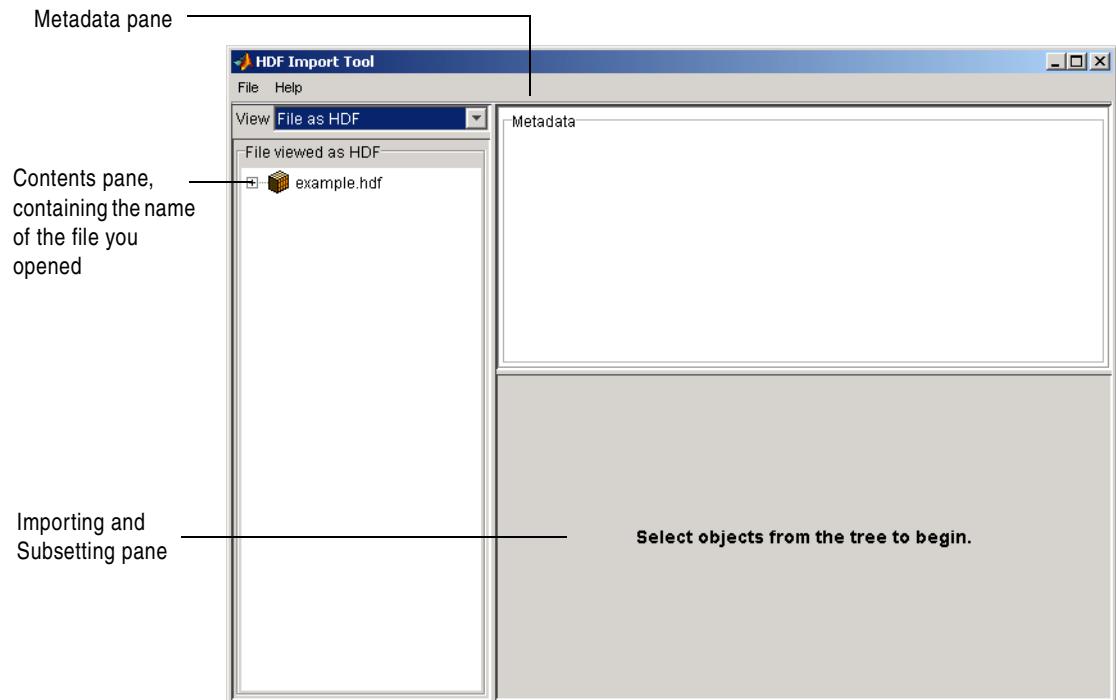
You can also open a file with the HDF Import Tool by entering the hdftool command at the MATLAB command line:

```
hdftool('example.hdf')
```

If you use the hdftool function without arguments, it starts the HDF Import Tool and automatically opens a file selection dialog box.

Note You can open multiple files in the HDF Import Tool at the same time.

The **HDF Import Tool** contains three panes: the **Contents** pane, **Metadata** pane, and the **Importing and Subsetting** pane. Initially, the **Contents** pane contains the name of the file you opened and the other panes are empty.

Metadata pane

Contents pane, containing the name of the file you opened

Importing and Subsetting pane

Selecting a Data Set to Import
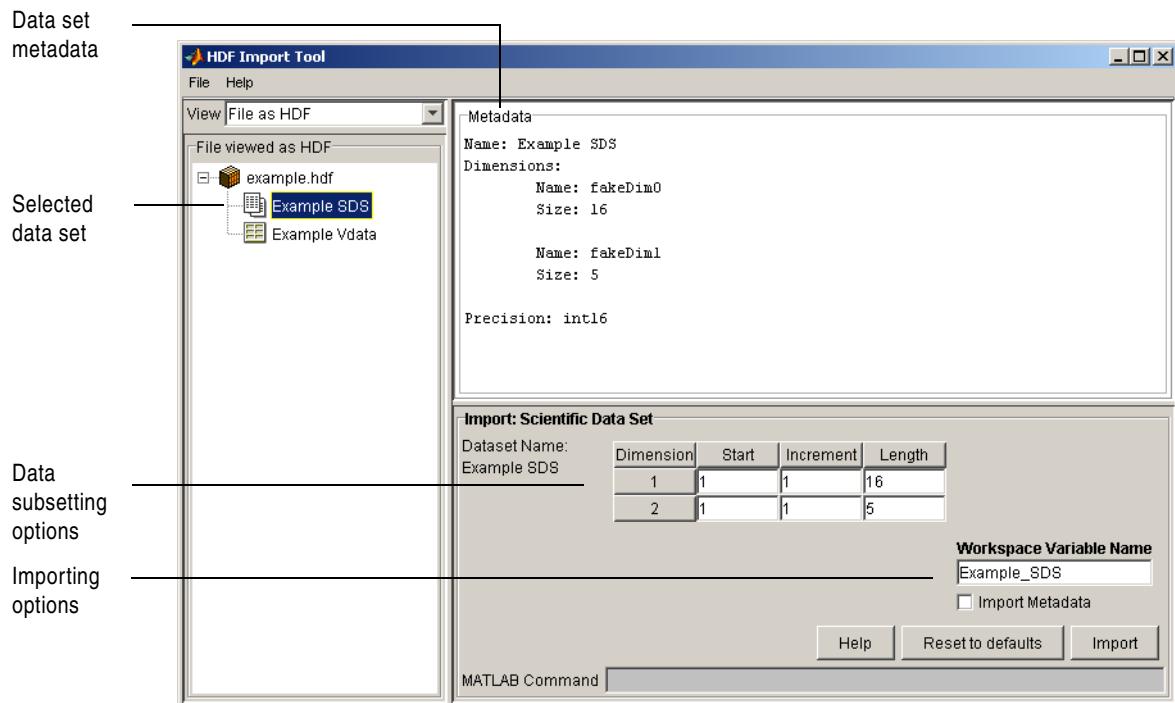
To select a data set to import, navigate through the contents of the file in the **Contents** pane. Click the plus sign at the left of the filename. This expands the hierarchical table of contents, listing the data sets in the file.

When you select a data set in the **Contents** pane, the HDF Import Tool displays the metadata associated with the data set in the **Metadata** pane, such as the size of each dimension.

The HDF Import Tool displays any subsetting options that exist for that type of data in the **Importing and Subsetting** pane.

Note The **Importing and Subsetting** pane might remain empty, even after you have selected items in the **Contents** pane, if the item is not a data set. For certain types of data, such as HDF-EOS Grid data, you have to navigate down several levels of the hierarchy to reach a data set.
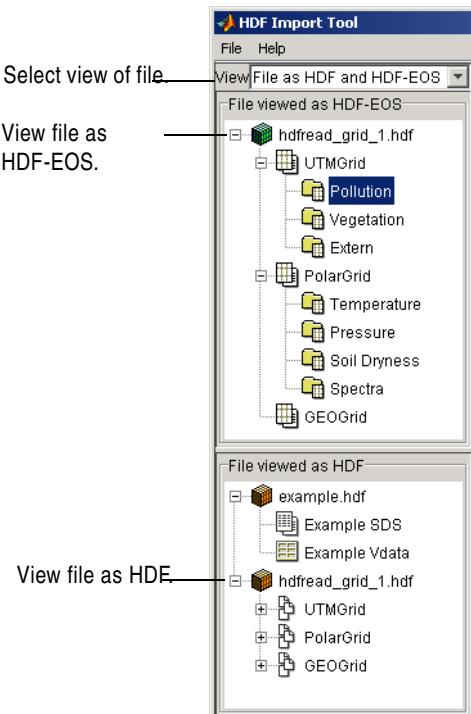
For example, this figure shows the expanded table of contents in the **Contents** pane, with the data set Example SDS selected. Note how the **Metadata** pane contains information about the data set. The **Importing and Subsetting** pane displays the subsetting options available and the workspace variable name assigned to the data set by default. See "Data Subsetting Options" on page 6-36 for more information about these data subsetting options.

Selecting the View.   For HDF-EOS files, the HDF Import Tool **Contents** pane can provide two different views of the file contents: as an HDF file or as an HDF-EOS file. HDF-EOS files can be viewed as HDF files because the EOS standard is built on the HDF standard. HDF files, however, cannot be viewed as EOS files.

You can switch the **Contents** pane to provide only an HDF view or only an HDF-EOS view of your files, or you can display both views of your files simultaneously, as illustrated in this figure.

Note Although HDF-EOS files can appear in both windows of the **Contents** pane, remember that you are getting two different views of the same file.

Data Subsetting Options

When you select a data set, the **Importing and Subsetting** pane displays the subsetting options available for that type of data set. For data sets that support multiple, mutually exclusive subsetting options, like HDF-EOS Grid data, the contents of the **Importing and Subsetting** pane change when you select one of the options. The following sections describe these subsetting options for all supported data set types. For general information about the tool, see "Using the HDF Import Tool" on page 6-31.

- "HDF Scientific Data (SD)"
- "HDF Vdata"
- "HDF-EOS Grid Data" on page 6-37
- "HDF-EOS Point Data" on page 6-41
- "HDF-EOS Swath Data" on page 6-43
- "HDF Raster Image Data" on page 6-46

Using these data subsetting options effectively requires an understanding of the HDF and HDF-EOS data formats. To find more information about these formats, see "Getting More Information About HDF" on page 6-67.

HDF Scientific Data (SD)

HDF Scientific Data (SD) data sets are multidimensional arrays. You can import a subset of an HDF SD data set by specifying the location, range, and values to be read from the data set.

Dimension	Start	Increment	Length
1	1	1	16
2	1	1	5

The HDF Import Tool displays the subsetting options available, where each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.

- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read. The **Length** parameter automatically updates when you change the value of **Start** or **Increment**.

HDF Vdata
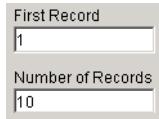
HDF Vdata data sets are tables. You can import a subset of an HDF Vdata data set in two ways:

- By field name
- By record

Fields. Select a specific field you want to import.

Records. Specify the range of records you want to import.

HDF-EOS Grid Data

In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

- Direct Index
- Geographic Box
- Interpolation
- Pixels
- Tile
- Time

• User-Defined

Direct Index. You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and values to be read along each dimension.

Dimension	Start	Increment	Length
1	1	1	16
2	1	1	5

Each row represents a dimension in the data set and each column represents these subsetting parameters:

• **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set.

• **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.

• **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box. You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in.

You define the rectangular area of interest by specifying two points that are two corners of the box:

• **Corner 1**— Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.

• **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

When specifying geographic box subsetting, you can optionally further define the subset of data you are interested in by using Time parameters (see "Time" on page 6-40) and by specifying other User-Defined subsetting parameters (see "User-Defined" on page 6-45).
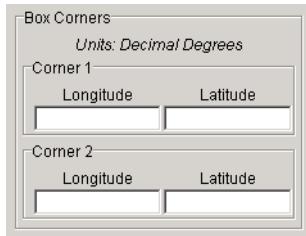
Interpolation. Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

You define the region used for bilinear interpolation by specifying two points that are two corners of the interpolation area:

- **Corner 1**— Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.
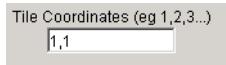
Pixels. You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid.

You define the box by specifying two points that define two corners of the box:

- **Corner 1**— Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

Tile. In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. HDF-EOS Grid data can be stored as tiles. If it is, you can import a subset of the Grid data set by specifying the coordinates of the tile you are interested in.

Tile coordinates are 1-based, with the upper left corner of a two-dimensional data set identified as 1,1. In a three-dimensional data set, this tile would be referenced as 1,1,1.

Time. You can import a subset of the Grid data set by specifying a time period.
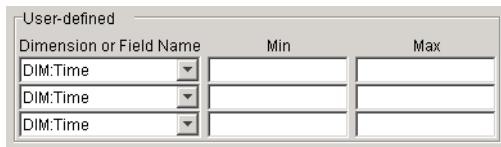
Specify these values:

- **Start**— Specifies the start time.
- **Stop** — Specifies the endpoint in the time span.

Note The units used (hours, minutes, seconds) to specify the time are defined by the data set.

Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters (see "User-Defined").

User-Defined. You can import a subset of the Grid data set by specifying user-defined parameters.

Specify these values:
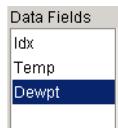
- **Dimension or Field Name** — Specifies the name of the dimension or field to be read from. Dimension names are prefixed with the characters DIM:.
- **Min** — Specifies the start of a range. For dimensions, **min** represents the start of a range of *elements* to extract. For fields, **min** represents the start of a range of *values* to extract.
- **Max** — Specifies the endpoint of a range. For dimensions, **max** represents the end of a range of *elements* to extract. For fields, **max** represents the end of a range of *values* to extract.

HDF-EOS Point Data
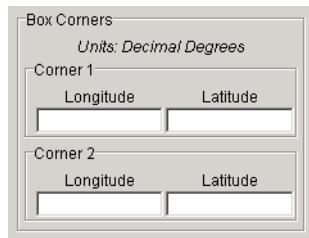
HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying any of these parameters:

- Field name
- Rectangular area of interest
- Record
- Time

Fields. Select a specific field you want to import.

Rectangular Area. You can import a subset of an HDF-EOS Point data set by specifying the rectangular area that you are interested in.

You define the rectangular area of interest by specifying two points that are two corners of the box:

- **Corner 1**— Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees.Typically, **Corner 2** is the lower-right corner of the box.

Records. Specify the range of records you want to import.

Time. You can import a subset of the HDF-EOS Point data set by specifying a time period.

Specify these values:

- **Start**— Specifies the start time.
- **Stop** — Specifies the endpoint in the time span.

Note The units used (hours, minutes, seconds) to specify the time are defined by the data set.

HDF-EOS Swath Data

HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

- Direct indexing
- Geographic region
- Time
- User-Defined

Direct Index. You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and values to be read along each dimension.

Dimension	Start	Increment	Length
1	1	1	16
2	1	1	5

Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box. You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in. When you use this subsetting method, you can also specify the Cross Track Inclusion Mode and the Geolocation Mode.

Define the area by specifying two points that specify two corners of the box:

- **Corner 1**— Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.
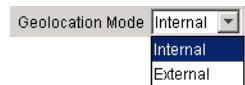
For Swath data, you must also specify the **Cross Track Inclusion Mode**. This determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

Select from these values:

- **AnyPoint**— Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath. This is the default.
- **Endpoint** — All of the area defined by the box overlaps with the swath.

For Swath data, you must also specify **Geolocation Mode.** This specifies whether geolocation fields and data must be in the same swath.

Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

Time. You can import a subset of the Swath data set by specifying a time period.

Specify these values:

- **Start**— Specifies the start time.
- **Stop** — Specifies the endpoint in the time span.

Note The units used (hours, minutes, seconds) to specify the time are defined by the data set.

When you use this subsetting method, you must also specify the Cross Track Inclusion Mode and the Geolocation Mode. You can optionally also specify user-defined subsetting options.

User-Defined. You can import a subset of the Swath data set by specifying user-defined parameters.

Specify these values:

- **Dimension or Field Name** — Specifies the name of the dimension or field to be read from. Dimension names are prefixed with the characters DIM:.

6-45

- **Min** — Specifies the start of a range. For dimensions, **min** represents the start of a range of *elements* to extract. For fields, **min** represents the start of a range of *values* to extract.
- **Max** — Specifies the endpoint of a range. For dimensions, **max** represents the end of a range of *elements* to extract. For fields, **max** represents the end of a range of *values* to extract.

HDF Raster Image Data

There are no subsetting options available for HDF raster image data.

Importing Data Using the HDF Import Tool
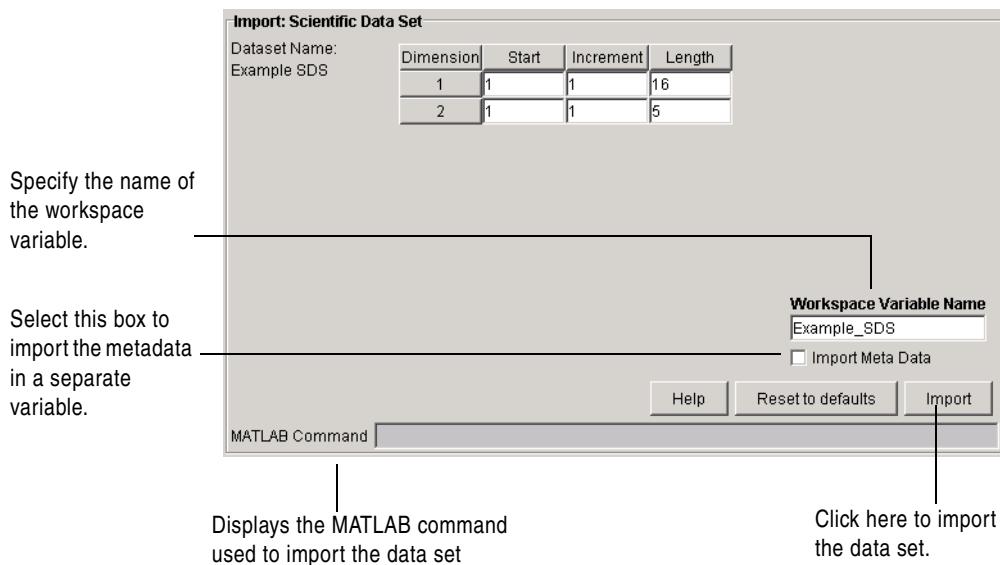
To import the data set you have selected, click the **Import** button in the bottom right corner of the **Importing and Subsetting** pane.

In this illustration, the HDF scientific data set, Example SDS, is selected.

By default, the HDF Import Tool uses the name of the data set as the name of the MATLAB workspace variable. In the figure, the variable name is Example_SDS. You can change the variable name by entering text in the **Workspace Variable Name** text box.

To import the metadata associated with the data set, select the **Import Metadata** check box. The HDF Import Tool creates a second variable in the workspace with the same name with "_info" appended to it. For example, the name of the metadata variable in the figure is Example_SDS_info.

The **MATLAB Command** text window contains the command the import tool used to import the data set. This text is not editable but you can select it to copy and paste it into the MATLAB command window or text editor. To reuse this command on other HDF or HDF-EOS files with the same organization, change the filename in the command string.

Specify the name of the workspace variable.

Select this box to import the metadata in a separate variable.

Displays the MATLAB command used to import the data set

Click here to import the data set.

Using the MATLAB High-Level Import Function

You can import data from an HDF or HDF-EOS file using the `hdfread` function. This function hides many of the low-level details required by the HDF protocol, described in "Using the HDF Command-Line Interface" on page 6-47.

For example, to read a data set in an HDF file using the low-level functions, you need to open the file, select the data set, read the data set, and close the file. The `hdfread` function performs these operations for you.

Using the HDF Command-Line Interface

To import HDF data into MATLAB, you can use the routines in the HDF API associated with the particular HDF data type. Each API has a particular programming model, that is, a prescribed way to use the routines to open the HDF file, access data sets in the file, and read data from the data sets.

To illustrate this concept, this section describes the programming model of one particular HDF API: the Scientific Data (SD) API. To view a complete list of the HDF APIs supported by MATLAB, see "HDF APIs Supported by MATLAB" on

page 6-68. For information about working with other HDF APIs, see the official NCSA documentation.

Note The following sections, when referring to specific routines in the HDF SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

The HDF SD Import Programming Model

To import data in HDF SD format, you must use API routines to perform these steps:

1 Open the file containing HDF SD data sets.

2 Select the data set in the file that you want to import.

3 Read the data from the data set.

4 Close access to the data set and HDF file.

There are several additional steps that you might also need to perform, such as retrieving information about the contents of the HDF file or the data sets in the file. The following sections provide more detail about the basic steps as well as optional steps.

Opening HDF Files

To import an HDF SD data set, you must first open the file containing the data set. In the HDF SD API, you use the SDstart routine to open the file and initialize the HDF interface. In MATLAB, you use the hdfsd function with start specified as the first argument.

SDstart accepts these arguments:

- A text string specifying the name of the file you want to open
- A text string specifying the mode in which you want to open it

For example, this code opens the file mydata.hdf for read access:

```
sd_id = hdfsd('start','mydata.hdf','read');
```

If SDstart can find and open the file specified, it returns an HDF SD file identifier, named sd_id in the example. Otherwise, it returns -1.

The HDF SD API supports several file access modes. You use a symbolic constant, defined by the HDF SD API, to specify each mode. In MATLAB, you specify these constants as text strings. You can specify the full HDF constant or one of the abbreviated forms listed in the table.

HDF File Creation Mode	HDF Symbolic Constant	MATLAB String
Create a new file	'DFACC_CREATE'	'create'
Read access	'DFACC_RDONLY'	'read' or 'rdonly'
Read and write access	'DFACC_RDWR'	'rdwr' or 'write'

Retrieving Information About an HDF File

After opening a file, you can get information about what the file contains using the SDfileinfo routine. In MATLAB, you use the hdfsd function with fileinfo specified as the first argument. This function returns the number of data sets in the file and whether the file includes any global attributes. (For more information about global attributes, see "Retrieving Attributes from an HDF File" on page 6-50.)

As an argument, SDfileinfo accepts the SD file identifier, sd_id, returned by SDstart. In this example, the HDF file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)

ndatasets =
     3

nglobal_atts =
     1

status =
     0
```

Retrieving Attributes from an HDF File

HDF files are self-documenting; that is, they can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an HDF file are called *global* attributes. (You can also associate attributes with data sets or dimensions. For more information about these attributes, see "Including Metadata in an HDF File" on page 6-64.)

In the HDF SD API, you use the SDreadattr routine to retrieve global attributes from an HDF file. In MATLAB, you use the hdfsd function, specifying readattr as the first argument. As other arguments, you specify

- The HDF SD file identifier (sd_id) returned by the SDstart routine.
- The index value specifying the attribute you want to view. HDF uses zero-based indexing, so the first global attribute has index value 0, the second has index value 1, and so on.

For example, this code returns the contents of the first global attribute, which is simply the character string my global attribute.

```
attr_idx = 0;
[attr, status] = hdfsd('readattr', sd_id, attr_idx)

attr =
    my global attribute

stat =
    0
```

MATLAB automatically sizes the return value, attr, to fit the data in the attribute.

Retrieving Attributes by Name. Attributes have names as well as values. If you know the name of an attribute, you can use the SDfindattr function to determine its index value so you can retrieve it. In MATLAB, you use the hdfsd function, specifying findattr as the first argument.

As other arguments, you specify

- The HDF SD file identifier, when searching for global attributes
- A text string specifying the name of the attribute

SDfindattr searches all the global attributes associated with the file. If it finds an attribute with this name, SDfindattr returns the index of the attribute. You can then use this index value to retrieve the attribute using SDreadattr.

This example uses SDfindattr to obtain the index for the attribute named my_att and then passes this index as an argument to SDreadattr:

```
attr_idx = hdfsd('findattr',sd_id,'my_att');
[attr, status] = hdfsd('readattr', sd_id, attr_idx);
```

Selecting Data Sets in HDF Files

After opening an HDF file, you must specify the data set in the file that you want to read. An HDF file can contain multiple data sets. In the HDF SD API, you use the SDselect routine to select a data set. In MATLAB, you use the hdfsd function, specifying select as the first argument.

As arguments, this function accepts

- The HDF SD file identifier (sd_id) returned by SDstart.
- The index value specifying the attribute you want to view. HDF uses zero-based indexing, so the first global attribute has index value 0, the second has index value 1, and so on.

For example, this code selects the third data set in the HDF file identified by sd_id. If SDselect finds the specified data set in the file, it returns an HDF SD data set identifier, called sds_id in the example. If it cannot find the data set, it returns -1.

Note Do not confuse HDF SD *file* identifiers, named sd_id in the examples, with HDF SD *data set* identifiers, named sds_id in the examples.

```
sds_idx = 2; % HDF uses zero-based indexing.
sds_id = hdfsd('select',sd_id,sds_idx)
```

Retrieving Data Sets by Name

Data sets in HDF files can be named. If you know the name of the data set you are interested in, but not its index value, you can determine its index by using the SDnametoindex routine. In MATLAB, use the hdfsd function, specifying nametoindex as the first argument.

Getting Information About a Data Set

After you select a data set in an HDF file, you can obtain information about the data set, such as the number and size of the array dimensions. You need this information to read the data set using the SDreaddata function. (See "Reading Data from an HDF File" on page 6-54 for more information.)

In the HDF SD API, you use the SDgetinfo routine to gather this information. In MATLAB, use the hdfsd function, specifying getinfo as the first argument. In addition, you must specify the HDF SD data set identifier returned by SDselect (sds_id).

This table lists the information returned by SDgetinfo.

Data Set Information Returned	MATLAB Data Type
Name	Character array
Number of dimensions	Scalar
Size of each dimension	Vector
Data type of the data stored in the array	Character array
Number of attributes associated with the data set	Scalar

For example, this code retrieves information about the data set identified by sds_id:

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] =
              hdfsd('getinfo',sds_id)
dsname =
     A

dsndims =
     2

dsdims =
     5     3

dstype =
```

```
        double

dsatts =
        0

stat =
        0
```

Retrieving Data Set Attributes

Like HDF files, HDF SD data sets are self-documenting; that is, they can optionally include information, called *attributes*, that describes the data in the data set. Attributes associated with a data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information about these attributes, see "Including Metadata in an HDF File" on page 6-64.)

In the HDF SD API, you use the SDreadattr routine to retrieve local attributes. In MATLAB, use the hdfsd function, specifying readattr as the first argument. As other arguments, specify

• The HDF SD data set identifier (sds_id) returned by SDselect.

• The index of the attribute you want to view. HDF uses zero-based indexing, so the first global attribute has index value 0, the second has index value 1, and so on.

This code example returns the contents of the first attribute associated with a data set identified by sds_id. In this case, the value of the attribute is the character string 'my local attribute'. MATLAB automatically sizes the return value, ds_attr, to fit the value of the attribute:

```
attr_idx = 0;
[ds_attr, status] = hdfsd('readattr', sds_id, attr_idx)

ds_attr =
    my local attribute

stat =
    0
```

Reading Data from an HDF File

After you open an HDF file and select a data set in the file, you can read the entire data set, or part of the data set. In the HDF SD API, you use the SDreaddata routine to read a data set. In MATLAB, use the hdfsd function, specifying readdata as the first argument. As other arguments, specify

- The HDF SD data set identifier (sds_id) returned by SDselect
- The location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector in HDF terminology
- The number of elements along each dimension to skip between each read operation, specified as a vector of scalar values, called the *stride* vector in HDF terminology
- The total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector in HDF terminology

For example, to read the entire contents of a data set containing this 3-by-5 matrix of numeric values,

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

you could use this code:

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] =
                          hdfsd('getinfo',sds_id);

ds_start = zeros(1,ds_ndims); % Creates the vector [0 0]
ds_stride = [];
ds_edges = ds_dims;

[ds_data, status] =
          hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
```

In this example, note the following:

- The return values of SDgetinfo are used to specify the dimensions of the return values and as arguments to SDreaddata.
- To read from the beginning of a data set, specify zero for each element of the start vector (ds_start). Note how the example uses SDgetinfo to determine the length of the start vector.
- To read every element of a data set, specify one for each element of the stride vector or specify an empty array ([]).
- To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set.

Note Use the dimensions vector returned by SDgetinfo, dsdims, to set the value of the edges vector, because SDgetinfo returns these values in row-major order, the ordering used by HDF. MATLAB stores data in column-major order. An array referred to as a 3-by-5 array in MATLAB is described as a 5-by-3 array in HDF.

Reading a Portion of a Data Set

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read.

For example, the following code fragment uses SDreaddata to read the entire second row of this sample data set:

```
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
```

Note that in the start, stride, and edges arguments, you must specify the dimensions in column-major order, that is, [columns,rows]. In addition, note that you must use zero-based indexing in these arguments:

```
ds_start = [0 1] % Start reading at the first column, second row
ds_stride = []; % Read each element
ds_edges = [5 1]; % Read a 1-by-5 vector of data

[ds_data, status] =
            hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);
```

```
disp(ds_data)
    6    7    8    9    10
```

For more information about specifying ranges in data sets, see "Writing MATLAB Data to an HDF File" on page 6-61.

Closing HDF Files and HDF Data Sets

After reading data from a data set in an HDF file, you must close access to the data set and the file. The HDF SD API includes functions to perform these tasks. See "Closing HDF Data Sets" on page 6-63 for more information.

MATLAB HDF Function Calling Conventions

Each HDF API includes many individual routines to read data from files, write data to files, and perform other related functions. For example, the HDF SD API includes separate C routines to open (SDopen), close (SDend), and read data (SDreaddata).

MATLAB, instead of supporting a corresponding function for each individual HDF API routine, supports a single function for each HDF API. You use this single function to access all the individual routines in the HDF API, specifying the name of the individual HDF routine as the first argument.

For example, to call the HDF SD API routine to terminate access to an HDF file in a C program, you use

```
status = SDend(sd_id);
```

To call this routine from MATLAB, use the MATLAB function associated with the API. By convention, the name of the MATLAB function associated with an HDF API includes the API acronym in the function name. For example, the MATLAB function used to access routines in the HDF SD API is called hdfsd.

As the first argument to this function, specify the name of the API routine, minus the acronym, and pass the remaining arguments expected by the routine in the order they are required. Thus, to call the SDend routine from MATLAB, use this syntax:

```
status = hdfsd('end',sd_id);
```

Note For some HDF API routines, particularly those that use output arguments to return data, the MATLAB calling sequence is different. (See "Handling HDF Routines with Output Arguments" on page 6-57 for more information.) Refer to the MATLAB online Function Reference to make sure you have the correct syntax.

Handling HDF Routines with Output Arguments

When calling HDF API routines that use output arguments to return data, you must specify all output arguments as return values. For example, in C syntax, the SDfileinfo routine returns data about an HDF file in two output arguments, ndatasets and nglobal_atts:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified last.

Handling HDF Library Symbolic Constants

The C versions of the HDF APIs use symbolic constants, defined in header files, to specify modes and data types. For example, the SDstart routine uses a symbolic constant to specify the mode in which to open an HDF file:

```
sd_id = SDstart("my_file.hdf",DFACC_RDONLY);
```

When calling this routine from MATLAB, specify these constants as text strings:

```
sd_id = hdfsd('start','my_file.hdf','DFACC_RDONLY')
```

In MATLAB, you can specify the entire constant or leave off the prefix. For example, in this call to SDstart, you can use any of these variations as the constant text string: 'DFACC_RDONLY', 'dfacc_rdonly', or 'rdonly'. Note that you can use any combination of uppercase and lowercase characters.

Exporting MATLAB Data in an HDF File

To export data from MATLAB in an HDF file, you must use the functions in the HDF API associated with the HDF data type. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. (In HDF terminology, the numeric arrays stored in HDF files are called data sets.)

To illustrate this concept, this section describes the programming model of one particular HDF API: the HDF Scientific Data (SD) API. To view a complete list of the HDF APIs supported by MATLAB, view "HDF APIs Supported by MATLAB" on page 6-68.

This section covers these topics:

"The HDF SD Export Programming Model" on page 6-59	Provides an overview of the HDF SD programming model, including how to create an HDF file, create an HDF data set in the file, write data to the data set, and close the file
"Including Metadata in an HDF File" on page 6-64	Describes how to write metadata to an HDF SD file
"Using the MATLAB HDF Utility API" on page 6-66	Describes the functions in the MATLAB HDF Utility API, such as the `MLlistinfo` function
"Getting More Information About HDF" on page 6-67	Lists some additional sources of information about HDF

Note This section does not attempt to describe all HDF features and routines. To use the MATLAB HDF functions effectively, you must refer to the official NCSA documentation. See "Getting More Information About HDF" on page 6-67.

The HDF SD Export Programming Model

The programming model for exporting HDF SD data involves these steps:

1 Create the HDF file, or open an existing one.

2 Create a data set in the file, or select an existing one.

3 Write data to the data set.

4 Close access to the data set and the HDF file.

You can optionally include information in the HDF file that describes your data. See "Including Metadata in an HDF File" on page 6-64 for more information.

Creating an HDF File

To export MATLAB data in HDF format, you must first create an HDF file, or open an existing one. In the HDF SD API, you use the SDstart routine. In MATLAB, use the hdfsd function, specifying start as the first argument. As other arguments, specify

- A text string specifying the name you want to assign to the HDF file (or the name of an existing HDF file)
- A text string specifying the HDF SD interface file access mode

For example, this code creates an HDF file named mydata.hdf:

```
sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');
```

When you specify the DFACC_CREATE access mode, SDstart creates the file and initializes the HDF SD multifile interface, returning an HDF SD file identifier, named sd_id in the example.

If you specify DFACC_CREATE mode and the file already exists, SDstart fails, returning -1. To open an existing HDF file, you must use HDF read or write modes. For information about using SDstart in these modes, see "Opening HDF Files" on page 6-48.

Creating an HDF Data Set

After creating the HDF file, or opening an existing one, you must create a data set in the file for each MATLAB array you want to export. In the HDF SD API,

you use the SDcreate routine to create data sets. In MATLAB, you use the hdfsd function, specifying create as the first argument. To write data to an existing data set, you must obtain the HDF SD data set identifier. See "Selecting Data Sets in HDF Files" on page 6-51 for more information.

This table lists the other arguments to SDcreate.

Argument	MATLAB Data Type
Valid HDF SD file identifier	Returned from SDstart
Name you want assigned to the data set	Text string
Data type of the data set	Text string. For information about specifying data types, see "Importing HDF Data" on page 6-31.
Number of dimensions in the data set. This is called the *rank* of the data set in HDF terminology.	Scalar numeric value
Size of each dimension	Vector

The values you assign to these arguments depend on the MATLAB array you want to export. For example, to export the following MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ];
```

you could set the values of these arguments as in this code fragment:

```
ds_name = 'A';
ds_type = 'double';
ds_rank = ndims(A);
ds_dims = fliplr(size(A));

sds_id = hdfsd('create',sd_id,ds_name,ds_type,ds_rank,ds_dims);
```

If SDcreate can successfully create the data set, it returns an HDF SD data set identifier, (sds_id). Otherwise, SDcreate returns -1.

Note In this example, note how the code fragment reverses the order of the values in the dimensions argument (ds_dims). This processing is necessary because the MATLAB size function returns the dimensions in column-major order and HDF expects to receive dimensions in row-major order.

Once you create a data set, you cannot change its characteristics. You can, however, modify the data it contains. To do this, initiate access to the data set, using SDselect, and write to the data set as described in "Writing MATLAB Data to an HDF File" on page 6-61.

Writing MATLAB Data to an HDF File

After creating an HDF file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF SD API, you use the SDwritedata routine. In MATLAB, use the hdfsd function, specifying writedata as the first argument.

This table lists the other arguments to SDwritedata.

Argument	MATLAB Data Type
Valid data set identifier (sds_id)	Returned by SDcreate
Location in the data set where you want to start writing data, called the *start* vector in HDF terminology	Vector of index values
Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF terminology	Vector of scalar values
Total number of elements to write along each dimension, called the *edges* vector in HDF terminology	Vector of scalar values
MATLAB array to be written	Array of doubles

> **Note** You must specify the values of the start, stride, and edges arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by the `size` function before assigning it as the value of the edges argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, the following code fragment writes this MATLAB 3-by-5 array of doubles,

```
A = [ 1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15 ];
```

into an HDF file:

```
ds_start = zeros(1:ndims(A)); % Start at the beginning
ds_stride = [];               % Write every element.
ds_edges = fliplr(size(A));   % Reverse the dimensions.

stat = hdfsd('writedata',sds_id,
                         ds_start, ds_stride, ds_edges, A)
```

If it can write the data to the data set, SDwritedata returns 0; otherwise, it returns -1.

> **Note** SDwritedata queues write operations. To ensure that these queued write operations are executed, you must close the file, using the SDend routine. See "Closing an HDF File" on page 6-63 for more information. As a convenience, MATLAB provides a function, MLcloseall, that you can use to close all open data sets and file identifiers with a single call. See "Using the MATLAB HDF Utility API" on page 6-66 for more information.

Writing Data to Portions of Data Sets

To write less than the entire data set, use the start, stride, and edges vectors to specify where you want to start writing data and how much data you want to write.

For example, the following code fragment uses SDwritedata to replace the values of the entire second row of the sample data set

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

with the vector B:

```
B = [ 9 9 9 9 9]
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as (2,1). The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1] % Start writing at the first column, second row.
ds_stride = []; % Write every element.
ds_edges = [5 1]; % Each row is a 1-by-5 vector.

stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Closing HDF Data Sets

After writing data to a data set in an HDF file, you must close access to the data set. In the HDF SD API, you use the SDendaccess routine to close a data set. In MATLAB, use the hdfsd function, specifying endaccess as the first argument. As the only other argument, specify a valid HDF SD data set identifier, sds_id in this example:

```
stat = hdfsd('endaccess',sds_id);
```

Closing an HDF File

After writing data to a data set and closing the data set, you must also close the HDF file. In the HDF SD API, you use the SDend routine. In MATLAB, use the hdfsd function, specifying end as the first argument. As the only other argument, specify a valid HDF SD file identifier, sd_id in this example:

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF file before closing it.

Note Closing an HDF file executes all the write operations that have been queued using SDwritedata. As a convenience, the MATLAB HDF Utility API provides a function, MLcloseall, that can close all open data set and file identifiers with a single call. See "Using the MATLAB HDF Utility API" on page 6-66 for more information.

Including Metadata in an HDF File

You can optionally include information in an HDF file that describes your data. HDF defines an separate annotation API; however, the HDF SD API includes an annotation capability. This section only describes the annotation capabilities of the HDF SD API. For information about the Annotation API, see the official NCSA documentation.

Types of Attributes

Using the HDF SD API, you can associate attributes with three types of HDF objects:

- An entire HDF file — File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF file — Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set — Dimension attributes provide information about one particular dimension of a data set.

Multiple Attributes

You can associate multiple attributes with a single HDF object. HDF maintains an attribute index for each object. The attribute index is zero-based. The first attribute has index value 0, the second has index value 1, and so on. You access an attribute by its index value.

Each attribute has the format name=value, where name (called label in HDF terminology) is a text string up to 256 characters in length and value contains one or more entries of the same data type. A single attribute can have multiple values.

Associating Attributes with HDF SD Objects

In the HDF SD API, you use the SDsetattr routine to associate an attribute with a file, data set, or dimension. In MATLAB, use the hdfsd function, specifying setattr as the first argument. As other arguments, specify

- A valid HDF SD identifier associated with the object. This value can be a file identifier (sd_id), a data set identifier (sds_id), or a dimension identifier (dim_id).
- A text string that defines the name of the attribute. The SD interface supports predefined attributes that have reserved names and, in some cases, data types. For information about these attributes, see "Creating Predefined Attributes" on page 6-65.
- The attribute value.

For example, this code creates a global attribute, named my_global_attr, and associates it with the HDF file identified by sd_id.

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the SDsetattr routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF function can determine the data type and size of the attribute from the value you specify.

Creating Predefined Attributes

Predefined attributes are identical to user-defined attributes except that the HDF SD API has already defined their names and data types. For example, the HDF SD API defines an attribute, named cordsys, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings 'cartesian', 'polar', and 'spherical'.

Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA documentation.

In the HDF SD API, you create predefined attributes the same way you create user-defined attributes, using the SDsetattr routine. In MATLAB, use the hdfsd function, specifying setattr as the first argument:

```
attr_name = 'cordsys';
attr_value = 'polar';

status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as SDsetdatastrs, are sometimes easier to use, especially when you are reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

Using the MATLAB HDF Utility API

In addition to the standard HDF APIs, listed in "HDF APIs Supported by MATLAB" on page 6-68, MATLAB supports utility functions that are designed to make using HDF in the MATLAB environment easier.

For example, the MATLAB utility API includes a function, MLlistinfo, that you can use to view all types of open HDF identifiers, such as HDF SD file identifiers. MATLAB updates these lists whenever HDF identifiers are created or closed.

This code obtains a list of all open HDF file and data set identifiers, using the MLlistinfo function. In this example, only two identifiers are open:

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
   262144
Open scientific data file identifiers:
   393216
No open Vdata identifiers
No open Vgroup identifiers
```

```
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Closing All Open HDF Identifiers

To close all open HDF identifiers in a single call, use the `MLcloseall` function. This call closes all open HDF identifiers:

```
hdfml('closeall')
```

Getting More Information About HDF

To use the MATLAB HDF interface functions effectively, you must use this documentation in conjunction with the official HDF documentation, available in many formats at the NCSA Web site (`hdf.ncsa.uiuc.edu`). In particular, consult the following documentation:

- The *HDF User's Guide*, which describes key HDF concepts and programming models and provides tutorial information about using the library routines
- The *HDF Reference Manual*, which provides detailed reference information about the hundreds of HDF routines, their arguments, and return values

The National Aeronautics and Space Administration (NASA) has created an extension of HDF as one of the data standards for the Earth Observing System (EOS). For information about this extension to HDF, consult the official HDF-EOS documentation at the EOS Web site (`hdfeos.gsfc.nasa.gov/hdfeos/workshop.html`).

Finally, for details about the syntax of the MATLAB HDF functions, consult the MATLAB online Function Reference. For most HDF routines, the MATLAB syntax is essentially the same as the HDF version; however, for certain routines, the MATLAB version has a different syntax.

HDF APIs Supported by MATLAB

The following table lists all the HDF APIs supported by MATLAB (listed alphabetically by acronym). The table includes the MATLAB utility functions API, named `ML`.

Application Programming Interface	Acronym	Description
Annotations	AN	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.
General Raster Images	DF24 DFR8	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes. Note: Use the MATLAB functions `imread` and `imwrite` with HDF raster image formats.
HDF Utilities	H, HD, and HE	Provides functions to open and close HDF files and handle errors.
MATLAB HDF Utilities	ML	Provides utility functions that help you work with HDF files in the MATLAB environment.
Scientific Data	SD	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.
V Groups	V	Creates and retrieves groups of other HDF data objects, such as raster images or V data.
V Data	VS VF VH	Stores, manages, and retrieves multivariate data stored as records in a table.

Using Low-Level File I/O Functions

MATLAB includes a set of low-level file I/O functions that are based on the I/O functions of the ANSI Standard C Library. If you know C, therefore, you are probably familiar with these routines.

For example, the MATLAB file I/O functions use the same programming model as the C language routines. To read or write data, you perform these steps:

1 Open the file, using `fopen`. `fopen` returns a file identifier that you use with all the other low-level file I/O routines.

2 Operate on the file.

   a Read binary data, using `fread`.

   b Write binary data, using `fwrite`.

   c Read text strings from a file line-by-line, using `fgets/fgetl`.

   d Read formatted ASCII data, using `fscanf`.

   e Write formatted ASCII data, using `fprintf`.

3 Close the file, using `fclose`.

This section also describes how these functions affect the current position in the file where read or write operations happen and how you can change the position in the file.

Note While the MATLAB file I/O commands are modeled on the C language I/O routines, in some ways their behavior is different. For example, the `fread` function is *vectorized*; that is, it continues reading until it encounters a text string or the end of file. These sections, and the MATLAB reference pages for these functions, highlight any differences in behavior.

Opening Files

Before reading or writing a text or binary file, you must open it with the `fopen` command.

```
fid = fopen('filename','permission')
```

Specifying the Permission String

The permission string specifies the kind of access to the file you require. Possible permission strings include

- r for reading only
- w for writing only
- a for appending only
- r+ for both reading and writing

Note Systems such as Microsoft Windows that distinguish between text and binary files might require additional characters in the permission string, such as 'rb' to open a binary file for reading.

Using the Returned File Identifier (fid)

If successful, fopen returns a a nonnegative integer, called a *file identifier* (fid). You pass this value as an argument to the other I/O functions to access the open file. For example, this fopen statement opens the data file named penny.dat for reading:

```
fid = fopen('penny.dat','r')
```

If fopen fails, for example if you try to open a file that does not exist, fopen

- Assigns -1 to the file identifier.
- Assigns an error message to an optional second output argument. Note that the error messages are system dependent and are not provided for all errors on all systems. The function ferror can also provide information about errors.

Test the file identifier each time you open a file in your code. For example, this code loops until a readable filename is entered.

```
fid=0;
while fid < 1
    filename=input('Open file: ', 's');
    [fid,message] = fopen(filename, 'r');
```

```
        if fid ==  1
          disp(message)
        end
    end
```

Now assume that `nofile.mat` does not exist but that `goodfile.mat` does exist. On one system, the results are

```
Open file: nofile.mat
Cannot open file. Existence? Permissions? Memory? . . .

Open file: goodfile.mat
```

Opening Temporary Files and Directories

The `tempdir` and `tempname` commands assist in locating temporary data on your system.

Function	Purpose
tempdir	Get temporary directory name.
tempname	Get temporary filename.

You can create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary can mean only that the file is not backed up.

A function named `tempdir` returns the name of the directory or folder that has been designated to hold temporary files on your system. For example, issuing `tempdir` on a UNIX system returns the `/tmp` directory.

MATLAB also provides a `tempname` function that returns a filename in the temporary directory. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first.

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

> **Note** The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

Reading Binary Data

The `fread` function reads all or part of a binary file (as specified by a file identifier) and stores it in a matrix. In its simplest form, it reads an entire file and interprets each byte of input as the next element of the matrix. For example, the following code reads the data from a file named `nickel.dat` into matrix A.

```
fid = fopen('nickel.dat','r');
A = fread(fid);
```

To echo the data to the screen after reading it, use `char` to display the contents of A as characters, transposing the data so it is displayed horizontally.

```
disp(char(A'))
```

The `char` function causes MATLAB to interpret the contents of A as characters instead of as numbers. Transposing A displays it in its more natural horizontal format.

Controlling the Number of Values Read

`fread` accepts an optional second argument that controls the number of values read (if unspecified, the default is the entire file). For example, this statement reads the first 100 data values of the file specified by `fid` into the column vector A.

```
A = fread(fid,100);
```

Replacing the number `100` with the matrix dimensions `[10 10]` reads the same 100 elements into a 10-by-10 array.

Controlling the Data Type of Each Value

An optional third argument to `fread` controls the data type of the input. The data type argument controls both the number of bits read for each value and the interpretation of those bits as character, integer, or floating-point values.

MATLAB supports a wide range of precisions, which you can specify with MATLAB specific strings or their C or Fortran equivalents.

Some common precisions include

- 'char' and 'uchar' for signed and unsigned characters (usually 8 bits)
- 'short' and 'long' for short and long integers (usually 16 and 32 bits, respectively)
- 'float' and 'double' for single- and double-precision floating-point values (usually 32 and 64 bits, respectively)

Note The meaning of a given precision can vary across different hardware platforms. For example, a 'uchar' is not always 8 bits. fread also provides a number of more specific precisions, such as 'int8' and 'float32'. If in doubt, use precisions that are not platform dependent. See fread for a complete list of precisions.

For example, if fid refers to an open file containing single-precision floating-point values, then the following command reads the next 10 floating-point values into a column vector A.

```
A = fread(fid,10,'float');
```

Writing Binary Data

The fwrite function writes the elements of a matrix to a file in a specified numeric precision, returning the number of values written. For instance, these lines create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, each stored as 4-byte integers.

```
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
```

In this case, fwrite sets the count variable to 25 unless an error occurs, in which case the value is less.

Controlling Position in a File

Once you open a file with `fopen`, MATLAB maintains a file position indicator that specifies a particular location within a file. MATLAB uses the file position indicator to determine where in the file the next read or write operation will begin. The following sections describe how to

- Determine whether the file position indicator is at the end of the file
- Move to a specific location in the file
- Retrieve the current location of the file position indicator
- Reset the file position indicator to the beginning of the file

Determining End-of-File

The `fseek` and `ftell` functions let you set and query the position in the file at which the next input or output operation takes place:

- The `fseek` function repositions the file position indicator, letting you skip over data or back up to an earlier part of the file.
- The `ftell` function gives the offset in bytes of the file position indicator for a specified file.

The syntax for `fseek` is

```
status = fseek(fid,offset,origin)
```

`fid` is the file identifier for the file. `offset` is a positive or negative offset value, specified in bytes. `origin` is one of the following strings that specify the location in the file from which to calculate the position.

`'bof'`	Beginning of file
`'cof'`	Current position in file
`'eof'`	End of file

Understanding File Position

To see how `fseek` and `ftell` work, consider this short M-file.

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
status = fclose(fid);
```

This code writes out the numbers 1 through 5 to a binary file named `five.bin`. The call to `fwrite` specifies that each numerical element be stored as a `short`. Consequently, each number uses two storage bytes.

Now reopen `five.bin` for reading.

```
fid = fopen('five.bin','r');
```

This call to `fseek` moves the file position indicator forward 6 bytes from the beginning of the file.

```
status = fseek(fid,6,'bof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	

File Position Indicator

This call to `fread` reads whatever is at file positions 7 and 8 and stores it in variable `four`.

```
four = fread(fid,1,'short');
```

The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`.

```
position = ftell(fid)

position =

    8
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	

File Position Indicator

This call to `fseek` moves the file position indicator back 4 bytes.

6-75

```
status = fseek(fid,-4,'cof');
```

File Position	bof	1	2	3	4	5	6	7	8	9	10	eof
File Contents		0	1	0	2	0	3	0	4	0	5	
File Position Indicator					↑							

Calling fread again reads in the next value (3).

```
three = fread(fid,1,'short');
```

Reading Strings Line by Line from Text Files

MATLAB provides two functions, fgetl and fgets, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that fgets copies the newline character to the string vector but fgetl does not.

The following M-file function demonstrates a possible use of fgetl. This function uses fgetl to read an entire file one line at a time. For each line, the function determines whether an input literal string (literal) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0
   tline = fgetl(fid);
   matches = findstr(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
end
fclose(fid);
```

For example, consider the following input data file called `badpoem`.

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string `'an'` appears in this file, use `litcount`.

```
litcount('badpoem','an')
2: Oranges and lemons,
1: Pineapples and tea.
3: Orangutans and monkeys,
```

Reading Formatted ASCII Data

The `fscanf` function is like the `fscanf` function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers for `fscanf` begin with a `%` character; common conversion specifiers include

Conversion Specifier	Description
%s	Match a string
%d	Match an integer in base 10 format
%g	Match a double-precision floating-point value

You can also specify that `fscanf` skip a value by specifying an asterisk in a conversion specifier. For example, `%*f` means skip the floating-point value in the input data; `%*d` means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf

Despite all the similarities between the MATLAB and C versions of `fscanf`, there are some significant differences. For example, consider a file named `moon.dat` for which the contents are as follows.

```
3.654234533
2.71343142314
5.34134135678
```

The following code reads all three elements of this file into a matrix named
MyData.

```
fid = fopen('moon.dat','r');
MyData = fscanf(fid,'%g');
status = fclose(fid);
```

Notice that this code does not use any loops. Instead, the fscanf function
continues to read in text as long as the input format is compatible with the
format specifier.

An optional size argument controls the number of matrix elements read. For
example, if fid refers to an open file containing strings of integers, then this
line reads 100 integer values into the column vector A.

```
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix A.

```
A = fscanf(fid,'%5d',[10 10]);
```

A related function, sscanf, takes its input from a string instead of a file. For
example, this line returns a column vector containing 2 and its square root.

```
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```

Writing Formatted Text Files

The fprintf function converts data to character strings and outputs them to the screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; fprintf copies text directly.

Common conversion specifiers include

Conversion Specifier	Description
%e	Exponential notation
%f	Fixed point notation
%g	Automatically select the shorter of %e and %f

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function:

```
x = 0:0.1:1;
y = [x; exp(x)];
```

The code below writes x and y into a newly created file named exptable.txt.

```
fid = fopen('exptable.txt','w');
fprintf(fid,'Exponential Function\n\n');
fprintf(fid,'%6.2f  %12.8f\n',y);
status = fclose(fid);
```

The first call to fprintf outputs a title, followed by two carriage returns. The second call to fprintf outputs the table of numbers. The format control string specifies the format for each line of the table:

• A fixed-point value of six characters with two decimal places

• Two spaces

• A fixed-point value of twelve characters with eight decimal places

fprintf converts the elements of array y in column order. The function uses the format string repeatedly until it converts all the array elements.

Now use fscanf to read the exponential data file.

```
fid = fopen('exptable.txt','r');
title = fgetl(fid);
[table,count] = fscanf(fid,'%f %f',[2 11]);
table = table';
status = fclose(fid);
```

The second line reads the file title. The third line reads the table of values, two floating-point values on each line, until it reaches end of file. count returns the number of values matched.

A function related to fprintf, sprintf, outputs its results to a string instead of a file or the screen. For example,

```
root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));
```

Closing a File

When you finish reading or writing, use fclose to close the file. For example, this line closes the file associated with file identifier fid.

```
status = fclose(fid);
```

This line closes all open files.

```
status = fclose('all');
```

Both forms return 0 if the file or files were successfully closed or -1 if the attempt was unsuccessful.

MATLAB automatically closes all open files when you exit from MATLAB. It is still good practice, however, to close a file explicitly with fclose when you are finished using it. Not doing so can unnecessarily drain system resources.

Note Closing a file does not clear the file identifier variable fid. However, subsequent attempts to access a file through this file identifier variable will not work.

Exchanging Files with the Internet

MATLAB provides a set of functions for exchanging files with the Internet. These include

- "Downloading URL Content"
- "ZIP Functions"
- "Sending E-Mail"

Downloading URL Content

From within MATLAB, you can read and save the content of a URL. The urlread function reads the content to a string variable in the MATLAB workspace. The urlwrite function saves the content to a file.

Example—Retrieving Content from a URL

This example downloads the contents of the Top Authors list at MATLAB Central File Exchange, assigning the results to the string s in the MATLAB workspace.

1 Retrieve the URL content.

```
s =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
TopFiles.jsp?type=category&id=&value=TopAuthors');
```

2 After retrieving the content, perform MATLAB functions on the variable s, such as

```
findstr(s,'My_name')
```

The next example downloads the files submitted for Signal Processing, Communications, and DSP from MATLAB Central File Exchange, saving the results to samples.html in the MATLAB current directory.

```
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange
/Category.jsp?type=category&id=1','samples.html');
```

After downloading, you can view the file in your Web browser.

ZIP Functions

You can compress and uncompress files and directories from MATLAB using the zip and unzip functions. For example:

```
zip('d:/mymfiles/viewlet.zip','$matlabroot/demos/guide.viewlet')
```

creates a ZIP file from guide.viewlet. For details, see the reference pages for zip and unzip.

Sending E-Mail

Use sendmail to send an electronic mail message and, optionally, attachments, to a list of addresses. This is an example of the syntax:

```
sendmail('user@otherdomain.com','Test subject','Test message',
{'directory/attach1.doc'});
```

If MATLAB cannot read the SMTP mail server from your system registry, you get an error. You need to identify the outgoing SMTP mail server for your electronic mail application, which is usually listed in preferences. Or, consult your e-mail system administrator. Then provide the information to MATLAB, using

```
setpref('Internet','SMTP_Server','myserver.myhost.com');
```

7

Editing and Debugging M-Files

Ways to Edit and Debug M-Files in MATLAB

There are several methods for creating, editing, and debugging M-files, which are files containing MATLAB code.

Task	Option	Instructions
Creating and Editing M-files	MATLAB Editor	"Starting the Editor/Debugger" on page 7-3
	MATLAB Editor in stand-alone mode (without running MATLAB)	"Opening the Editor Without Starting MATLAB" on page 7-7
	Any text editor, such as Emacs or vi	Specify the other editor as the default using preferences—see preferences for "Editor" on page 7-47
Debugging M-files	General debugging tips	"Types of Errors" and "Finding Errors" on page 7-26
	MATLAB Debugger	"Using Debugging Features" on page 7-30
	MATLAB debugging functions	"Using Debugging Features" on page 7-30

Use preferences for the Editor/Debugger to set up the editing and debugging environment to best meet your needs.

To learn more about writing M-files, see "Programming and Data Types".

Starting the Editor/Debugger
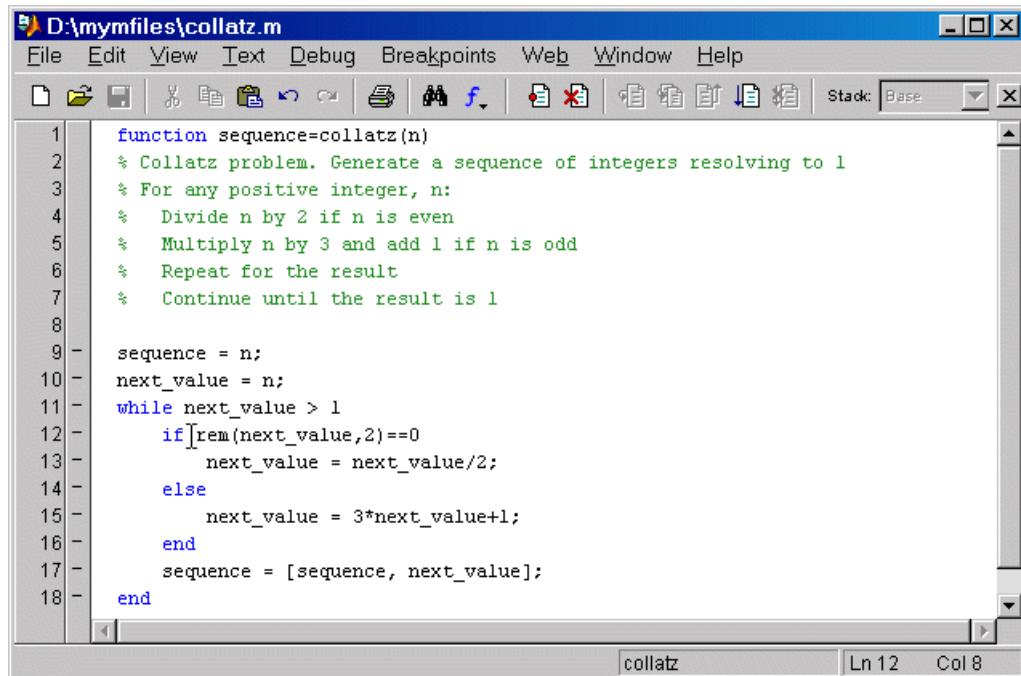
The MATLAB Editor/Debugger provides a graphical user interface for basic text editing features for any file type, as well as for M-file debugging. The Editor/Debugger is a single tool that you can use for editing, debugging, or both. There are various ways to start the Editor/Debugger—see these sections:

- "Creating a New M-File in the Editor/Debugger" on page 7-4
- "Opening Existing M-Files in the Editor/Debugger" on page 7-5
- "Opening the Editor Without Starting MATLAB" on page 7-7 (no Debugger)

After starting the Editor/Debugger, follow the instructions for

- "Creating and Editing M-Files with the Editor/Debugger" on page 7-9
- "Debugging M-Files" on page 7-25
- "Closing the Editor/Debugger" on page 7-7

This shows the Editor/Debugger opened to an existing M-file.

If the Editor/Debugger window is not wide enough, the toolbar buttons on the right are not shown and the menu wraps. All toolbar functions are available from equivalent menu items. To see all toolbar buttons, make the Editor/Debugger window wider.

To dock the Editor/Debugger inside the MATLAB desktop, select **Dock M-File** from the **View** menu.

To change the default appearance and behavior of the Editor/Debugger, follow the instructions in "Preferences for the Editor/Debugger" on page 7-46.

Creating a New M-File in the Editor/Debugger

To create a new M-file in the Editor/Debugger, either click the new file button □ on the MATLAB toolbar, or select **File -> New -> M-file** from the MATLAB desktop. You can also create a new M-file using the context menu in the

Current Directory browser—see "Creating New Files" on page 5-28. The Editor/Debugger opens, if it is not already open, with an untitled file in the MATLAB current directory from which you can create an M-file.

If the Editor/Debugger is open, create more new files by using the new file button □ on the toolbar, or select **File -> New -> M-file**.

Function Alternative

Type `edit` in the Command Window to create a new M-file in the Editor/Debugger. Type `edit filename.m` to open the M-file `filename.m` in the Editor/Debugger.

If `filename.m` does not exist, a prompt appears asking if you want to create a new file titled `filename.m`.

- If you click **Yes**, the Editor/Debugger creates a blank file titled `filename.m`. If you do not want the dialog to appear in this situation, select that check box in the dialog. Then, the next time you type `edit filename.m`, the file is created without first prompting you. If you later want that dialog to appear, specify it in preferences for "Prompt" (p. 7-52).

- If you click **No**, the Editor/Debugger does not create a new file. If you do not want the dialog to appear in this situation, select that check box in the dialog In that case, the next time you type `edit filename.m`, a "file not found" message appears. If you later want that dialog to appear, specify it in preferences for "Prompt" (p. 7-52).

Opening Existing M-Files in the Editor/Debugger

To open an existing M-file in the Editor/Debugger, click the open button 🖿 on the desktop or Editor/Debugger toolbar, or select **File -> Open**.

The **Open** dialog box opens, listing all MATLAB files. You can see different files by changing the selection for **Files of type** in the dialog box. Select the file and click **Open**. If you click the open icon from the desktop toolbar, the current directory files are shown, but if you open it from the Editor, the files in the directory for the current file are shown.

You can also open files from the Current Directory browser—see "Opening Files" on page 5-30. You can select a file to open from the most recently used files, which are listed at the bottom of the **File** menu in the Editor/Debugger

and all other desktop tools. You can change the number of files appearing on the list—see "Preferences for the Editor/Debugger" on page 7-46.

If the Editor/Debugger is not already open, it opens with the file displayed. If it is already open, the file appears either in its own window or as a tab in the current window as specified in the preference for "Opening files in editor" on page 7-50. To make a document in the Editor/Debugger become the current document, click it or use the **Window** menu or tabs.

You can set a preference that instructs MATLAB, upon startup, to automatically open the files that were open when the previous MATLAB session ended. For instructions, see the **On restart** preference in "General Preferences for the Editor/Debugger" on page 7-47.

Function Alternative

Use the edit or open function to open an existing M-file in the Editor/Debugger. For example, type

```
edit collatz.m
```

to open the file collatz.m in the Editor/Debugger, where collatz.m is on the search path or in the current directory. Use the relative or absolute pathname for the file you want to open if it is not on the search path or in the current directory.

Opening a Selection

Within a file in the Editor/Debugger, select a filename, right-click, and select **Open Selection** from the context menu to open that file. See "Opening a Selection in an M-File" on page 7-21 for details.

Getting Help for a Function

Within a file in the Editor/Debugger, select a function, right-click, and select **Help on Selection** from the context menu. The reference page for that function opens in the Help browser, or if the reference page does not exist, the M-file help is shown instead.

Accessing Your Source Control System

If you use a source control system for M-files, you can access it from within the Editor/Debugger using **File -> Source Control**. For more information, see Chapter 8, "Interfacing with Source Control Systems".

Opening the Editor Without Starting MATLAB

On Windows platforms, you can use the MATLAB Editor without starting
MATLAB. To do so, double-click an M-file in Windows Explorer. The M-file
opens in the MATLAB Editor. To open the Editor without a file, open
`$matlabroot/bin/win32/meditor.exe`. Regardless of the type of MATLAB
license you have, you can open multiple instances of `meditor` because it is not
considered an instance of MATLAB.

When you open the MATLAB Editor without starting MATLAB, the Editor is
a stand-alone application. You cannot debug M-files from it, evaluate a
selection, access source control features, dock the Editor in the MATLAB
desktop, nor access help from it. It remains a stand-alone application, even if
you subsequently open MATLAB. Other than these limitations, you can use the
editing features as described in "Creating and Editing M-Files with the
Editor/Debugger" on page 7-9.

For Windows platforms, when MATLAB is installed, the stand-alone Editor is
automatically associated with files having a `.m` extension. If you double-click
an M-file, the stand-alone Editor opens. You can change the association using
Windows Explorer so that files with a `.m` extension will open in the
Editor/Debugger in MATLAB.

Closing the Editor/Debugger

To close the Editor/Debugger, click the close box in the title bar of the
Editor/Debugger. This is different from the close box in the toolbar of the
Editor/Debugger, which closes the current file when multiple files are open in
a single window.

Note When you close the current file and it is the only file open, then the
Editor/Debugger closes as well.

Close box for Editor/Debugger.

Close box for current file. If no other files are open, closes the Editor/Debugger.

🔻 D:\mymfiles\collatz.m	_ □ ×
File Edit View Text Debug Breakpoints Web Window Help	

If multiple files are open, with each in a separate Editor/Debugger window, close each window separately. To close all files at once, select **Close All** from the **Window** menu.

When you close the Editor/Debugger and any of the open files have unsaved changes, you are prompted to save the files.

Creating and Editing M-Files with the Editor/Debugger

After opening an existing file or creating a new file, enter text in the Editor/Debugger. Follow the same rules you would use for entering text in the Command Window as described in these sections:

- "Case and Space Sensitivity" on page 3-5
- "Entering Multiple Functions in a Line" on page 3-5
- "Entering Long Lines" on page 3-6
- "Suppressing Output" on page 3-11
- "Formatting and Spacing Numeric Output" on page 3-12

In addition, use the editing features described in these sections:

- "Appearance of an M-File" on page 7-9, including syntax highlighting
- "Navigating in an M-File" on page 7-13, including go to and find features
- "Opening a Selection in an M-File" on page 7-21
- "Saving M-Files" on page 7-21
- "Running M-Files from the Editor/Debugger" on page 7-23
- "Printing an M-File" on page 7-23
- "Closing M-Files" on page 7-24

Appearance of an M-File

The following features make M-files more readable:

- "Syntax Highlighting" on page 7-10
- "Indenting" on page 7-10
- "Commenting" on page 7-10
- "Showing Balanced Delimiters" on page 7-12
- "Line and Column Numbers" on page 7-12
- "View Function or Subfunction" on page 7-13

You can specify the default behaviors for some of these—see "Preferences for the Editor/Debugger" on page 7-46.

Syntax Highlighting

Some entries appear in different colors to help you better find matching elements, such as if/else statements. This is called syntax highlighting and is used in the Command Window as well as in the Editor/Debugger. For more information, see the Command Window documentation for "Syntax Highlighting and Parentheses Matching" on page 3-6.

Indenting

Flow control entries are automatically indented to aid in reading loops, such as while/end statements.

To move the current or selected lines further to the left, select **Decrease Indent** from the **Text** menu. To move the current or selected lines further to the right, select **Increase Indent** from the **Text** menu. If after using these features you want to apply automatic indenting to selected lines, select **Smart Indent** from the **Text** menu, or right-click and select it from the context menu. For more information about smart indenting, see the preference for smart indent. See also "Keyboard and Indenting Preferences for the Editor/Debugger" on page 7-52.

Commenting
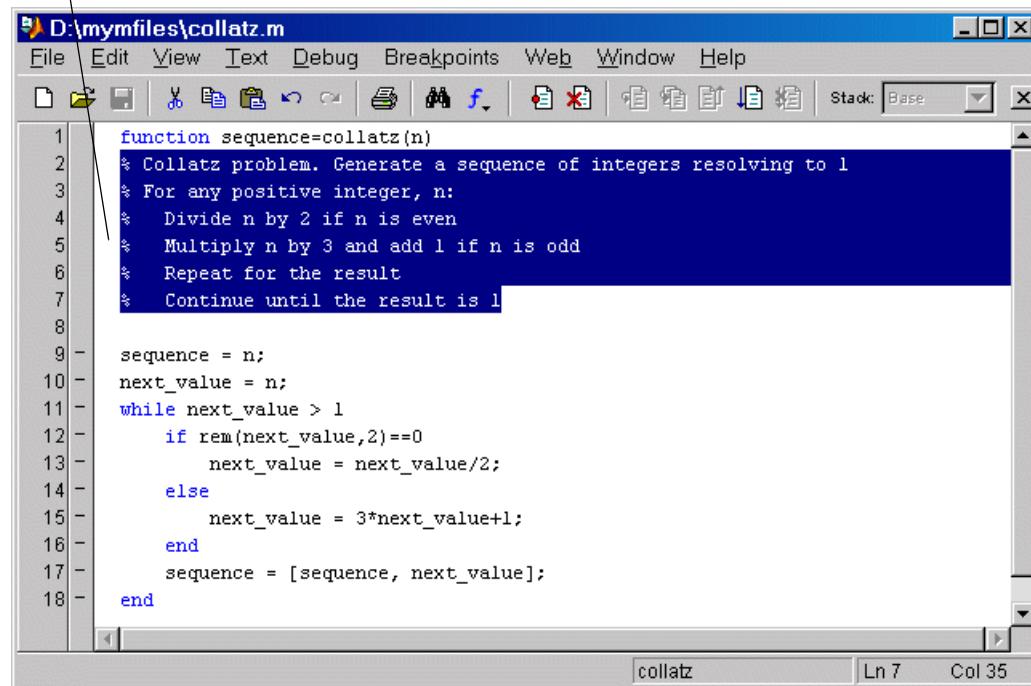
You can comment the current line or a selection of lines. To select a line, click just to the left of the line. The line becomes highlighted. Drag or **Shift**+click to select multiple lines. Then select **Comment** from the **Text** menu, or right-click and select it from the context menu. A comment symbol, %, is added at the start of the line, and the color of the text becomes green (or the color specified for comments in Editor/Debugger preferences).

You can make any line a comment by typing a % at the beginning of it. To put a comment within a line, type % followed by the comment text; MATLAB treats all the information after the % on that line as a comment.

You can also uncomment a selected group of lines—select **Uncomment** from the **Text** menu, or right-click and select it from the context menu.

Click in the area to the left of a line to select that line.
Drag or Shift+click to select multiple lines as shown here.

Select **Text -> Comment** to make all the selected lines comments.


```
1    function sequence=collatz(n)
2    % Collatz problem. Generate a sequence of integers resolving to 1
3    % For any positive integer, n:
4    %    Divide n by 2 if n is even
5    %    Multiply n by 3 and add 1 if n is odd
6    %    Repeat for the result
7    %    Continue until the result is 1
8
9    sequence = n;
10   next_value = n;
11   while next_value > 1
12       if rem(next_value,2)==0
13           next_value = next_value/2;
14       else
15           next_value = 3*next_value+1;
16       end
17       sequence = [sequence, next_value];
18   end
```

Formatting Comments. To make comment lines in M-files wrap when they reach a certain column:

1 Specify the maximum column number using preferences for the Editor. Under **M-file formatting**, set the **Max width**. See "M-file comment formatting" on page 7-48 for details.

2 Select contiguous comment lines that you want to limit to the specified maximum width.

3 Select **Text -> Format Selected Comments**.

The selected comment lines are reformatted so that no comment line in the selected area is longer than the maximum. Lines that were shorter than the specified maximum are merged to make longer lines if they are at the same level of indentation.

If you want comment lines to automatically be limited to the maximum width while you type, select the Editor preference to **Autowrap comments**—see "M-file comment formatting" on page 7-48.

Showing Balanced Delimiters

When you position the cursor inside a pair of delimiters, that is, inside ( ), [ ], or { }, and select **Balance Delimiters** from the **Text** menu, the string inside the pair of delimiters is highlighted. If there is not a pair of delimiters, instead MATLAB beeps. In this example, when the cursor is positioned before /norm, as indicated here

```
alfa = asin(T*v'./sqrt(diag(T*T'))/norm(v));
```

selecting **Balance Delimiters** highlights the selection as shown here.

```
alfa = asin(T*v'./sqrt(diag(T*T'))/norm(v));
```

For other appearance aids related to balancing delimiters, see "Syntax Highlighting" on page 7-10.

Line and Column Numbers

Line numbers are displayed along the left side of the Editor/Debugger window. To change the width of the line number column, drag the separator bar (to the right of the line numbers). Line numbers can be up to nine digits.

The line and column numbers for the current cursor position are shown in the far right side of the status bar in the Editor. You can elect not to show the line numbers using preferences—see "Show line numbers" on page 7-52.

View Function or Subfunction

View the function or subfunction the cursor is currently at in the right side of the status bar in the Editor.

Navigating in an M-File

There are several options for navigating in M-files:

Arrow and Control Keys to Navigate in the Editor

Following is the list of arrow and control keys you can use in the Editor. Control keys only work if the **Keyboard and Indenting** preference you select for "Key bindings" is **Emacs**.

Key	Control Key for Emacs Preference Only	Operation
↑	**Ctrl+P**	Move to *p*revious line.
↓	**Ctrl+N**	Move to *n*ext line.
←	**Ctrl+B**	Move *b*ack one character.
→	**Ctrl+F**	Move *f*orward one character.
Ctrl+ →		Move *r*ight one word.
Ctrl+ ←		Move *l*eft one word.
Home	**Ctrl+A**	Move to beginning of line.
End	**Ctrl+E**	Move to *e*nd of line.

Key	Control Key for Emacs Preference Only	Operation (Continued)
Delete	**Ctrl+D**	Delete character at cursor.
Backspace		Delete character before cursor.
	Ctrl+K	Delete (*k*ill) to end of line.
Shift+Home		Highlight to beginning of line.
Shift+End		Highlight to end of line.
Ctrl+Home		Move to top of file.
Ctrl+End		Move to end of file.

Going to a Line Number

Select **Go to Line** from the **Edit** menu. In the resulting dialog box, enter the **Line number** and click **OK**. The cursor moves to that line number in the current M-file.

Going to a Bookmark

You can set a bookmark at a line in a file in the Editor so you can quickly go to the bookmarked line. This is particularly useful in long files. For example, while working on a line, if you need to look at another part of the file, set a bookmark at the current line, go to the other part of the file, and then go back to the bookmark.

To set a bookmark, position the cursor anywhere in the line and select **Set Bookmark** from the **Edit** menu. A bookmark icon appears to the left of the line.

To go to a bookmark, select **Next Bookmark** or **Previous Bookmark** from the **Edit** menu.

To clear a bookmark, position the cursor anywhere in the line and select **Clear Bookmark** from the **Edit** menu.

Bookmarks are not saved when you close a file.

Going to a Function (Subfunction)

To go to a function in an M-file (referred to as a subfunction), click the function button f on the toolbar. Select the function you want to go to from the alphabetical listing of all functions in that M-file. The list does not include functions that are called from the M-file, but only lists lines in the current M-file that begin with a function statement.

The function or subfunction that the current line is part of is shown at the right side of the status bar.

Finding a Selection in the Current File

Within the current file, select a string. From the **Edit** menu, select **Find Selection**. The next occurrence of that string is highlighted. Select **Find Selection** again (or **Find Next**) to continue finding the next occurrences of the string.

To find the previous occurrence of a selected string (find backwards) in the current file, press **Ctrl+Shift+F3**, or select **Find Previous** from the **Edit** menu. The previous occurrence of that string is highlighted. Repeat to continue finding the previous occurrences of the string.
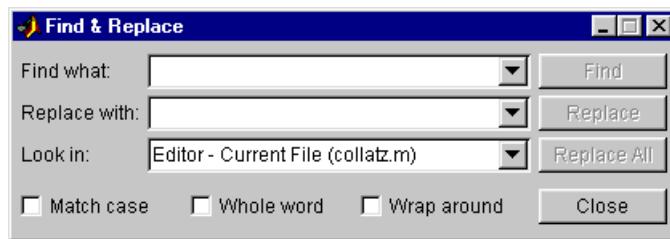
Finding and Replacing a String

You can search for a specified string within multiple files, and replace the string within a file.
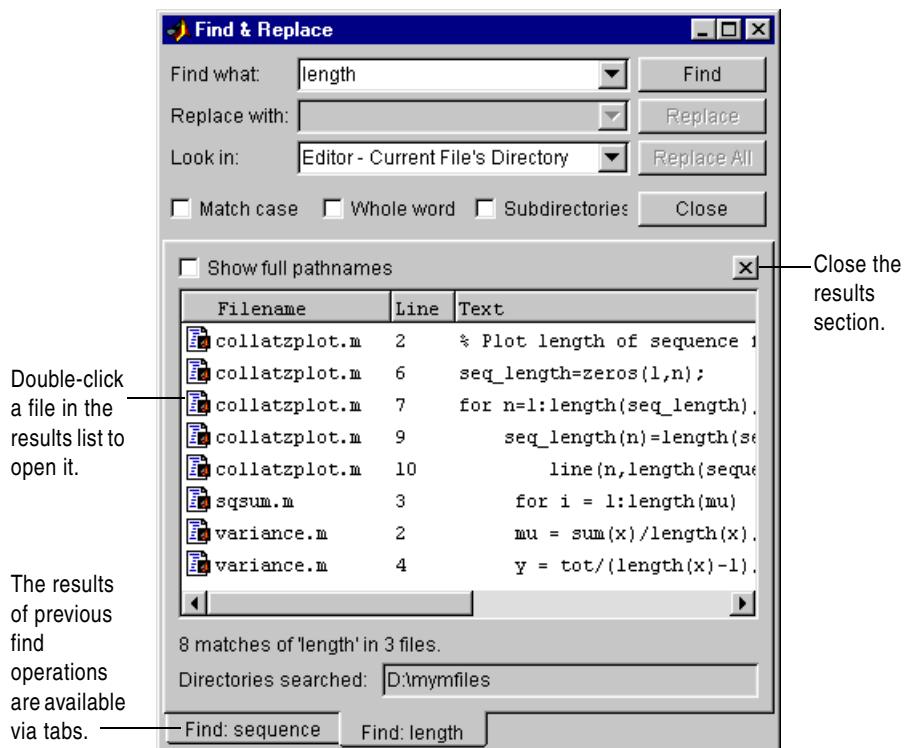
Finding a String. To search for a string in files:

1 Click the find button 🔍 in the Editor/Debugger toolbar, or select **Find and Replace** from the **Edit** menu.

The **Find & Replace** dialog box appears. This is similar to the **Find** dialog box in the Current Directory browser.

2 Complete the **Find & Replace** dialog box to find all occurrences of the string you specify.

   a Type the string in the **Find what** field.

   b Select the files or directories to search through from the **Look in** list box.

   c Limit the search using **Match case**, **Whole word**, or **Wrap around**. These settings are remembered for your next MATLAB session.

3 Click **Find**.

   - If you set **Look in** to Current File, the next occurrence of the string is highlighted in the file.
   - If you set **Look in** to search through multiple files, results appear in the lower part of the **Find & Replace** dialog box and include the filename, M-file line number, and content of that line.

Find & Replace

Find what: length

Replace with:

Look in: Editor - Current File's Directory

☐ Match case ☐ Whole word ☐ Subdirectories

Find | Replace | Replace All | Close

Close the results section.

☐ Show full pathnames

Filename	Line	Text
collatzplot.m	2	% Plot length of sequence 1
collatzplot.m	6	seq_length=zeros(1,n);
collatzplot.m	7	for n=1:length(seq_length),
collatzplot.m	9	seq_length(n)=length(se
collatzplot.m	10	line(n,length(seque
sqsum.m	3	for i = 1:length(mu)
variance.m	2	mu = sum(x)/length(x),
variance.m	4	y = tot/(length(x)-1),

Double-click a file in the results list to open it.

The results of previous find operations are available via tabs.

8 matches of 'length' in 3 files.

Directories searched: D:\mymfiles

Find: sequence | Find: length

4 Open any M-files in the results list by doing one of the following:

- Double-clicking a file
- Selecting files and pressing the **Enter** or **Return** key
- Right-clicking selected files and selecting **Open** from the context menu

The M-file opens, scrolled to the line number shown in the results section of the **Find & Replace** dialog box.

5 If you perform another search, the results of each search are accessible via tabs just below the results list. Click a tab to see that list of results as well as the search criteria.

Finding the Next or Previous Occurrence of the String. To find the next occurrence of the string you entered in the **Find & Replace** dialog box, click the **Find** button (or press the **Enter** key) if the dialog box is open (and has focus). If the **Find & Replace** dialog box is closed, select **Find Next** from the **Edit** menu.

To find the previous occurrence of that string (find backwards), select **Find Previous** from the **Edit** menu.

Function Alternative. Use lookfor to search for the specified string in the help in all M-files on the search path.

Replacing a String. After searching for a string within files, you can replace the specified content in the current file.

1 Open the file in the Editor if it is not already open. You can open it from the **Find & Replace** dialog box—see step 4 in "Finding a String" on page 7-16. Be sure that the file in which you want to replace the string is the current file in the Editor.

2 Be sure the **Look in** field in the **Find & Replace** dialog box shows the name of the file in which you want to replace the string.

3 In the **Replace with** field, type the text that is to replace the specified string.

4 Click **Replace** to replace the string in the selected line, or click **Replace All** to replace all instances in the currently open file.

   The text is replaced.

5 To save the changes, select **Save** from the **File** menu.

You can repeat this for multiple files.

Incremental Search

With the incremental search feature, the cursor moves to the next or previous occurrence of the specified string in the current file. It is similar to the Emacs search feature.

1 Position the cursor where you want the search to begin.

2 How you begin the incremental search depends on your setting for the Editor/Debugger "Key bindings" preference:

- Press **Ctrl+S** for **Emacs** or,

- Press **Ctrl+Shift+S** for **Windows**.

An incremental search field (**I-search:**) appears at the bottom of the current file window.

3 In the **I-search** field, type the string you want to find. For example, type `plot`.

As you type the first letter, `p`, the first occurrence of that letter in the file after the current cursor position is highlighted. In the example shown, the current line is 3 and the first occurrence of `p`, the `P` in `Prepare`, is highlighted.

If you enter a lowercase letter, for example, `p`, incremental search looks for both lowercase and uppercase instances, for example `p` and `P`. However, if

you enter an uppercase letter, for example, P, incremental search only looks for uppercase instances, for example, P.

```
3      % Prepare figure
```

When you type the next letter, the first occurrence of the string becomes highlighted. In the example, when you add the letter l to the p so that the **I-search** field now has pl, the pl in plot on line 8 is highlighted. When you add ot to the term in the **I-search** field, the whole word plot in line 8 is highlighted.

- If you mistype in the **I-search** field, use the **Back Space** key to remove the last letters and make corrections. Note that **Back Space** first takes you to any previous matches you just found using incremental search and then removes the last letter in the **I-search** field.

- After finding the p, you can press **Ctrl+W** and the rest of the word that is found, in this case plot, appears in the **I-search** field.

4 To find the next occurrence of plot in the file, press **Ctrl+S**. To find the previous occurrence of the string, press **Ctrl+R**.

5 If you hear a beep it either means that the string was not found, or it means you are at the end or beginning of the file.

Press **Ctrl+S** (or **Ctrl+R**) again to wrap to the beginning (or end) of the file and continue the search. Either the next occurrence of the string is highlighted, or you hear another beep indicating the string is not in the file.

6 To end the incremental search, press **Esc** or **Enter**, or any other noncharacter or number key except **Tab**.

The **I-search:** field no longer appears in the window. The cursor is now located at the position where the string was last found, with the search string highlighted.

You can type **Ctrl+R** (or **Ctrl+Shift+R** for Windows key bindings) to display the **I-search:** field to begin finding the previous occurrence rather than the next occurrence.

If you enter **Ctrl+S** or **Ctrl+R** after displaying the blank **I-search** field, the search term from your previous incremental search appears in the field. When you then use the **Back Space** key, you delete the entire previous search term, rather than just the last letter.

Opening a Selection in an M-File

You can open a subfunction, function, file, variable, or Simulink model from within a file in the Editor/Debugger. Select the name and then right-click and select **Open Selection** from the context menu. Based on what the selection is, the Editor performs a different action.

Selection	Action
Subfunction	Cursor moves to the subfunction within the current M-file. If no subfunction by that name is found in the current M-file, the Editor runs the open function on the selection, which opens the selection in the appropriate tool, as shown for the other selection types in this table.
M-file or other text file	Opens in the Editor.
Figure file (.fig)	Opens in a figure window.
Variable	Opens in the Array Editor.
Model	Opens in Simulink.
Other	If the selection is some other type, **Open selection** looks for a matching file in a private directory in the current directory and performs the appropriate action.

Saving M-Files

After making changes to an M-file, you see an asterisk (*) next to the filename in the title bar of the Editor/Debugger. This indicates there are unsaved changes to the file.

To save the changes, use one of the **Save** commands in the **File** menu:

- **Save**—Saves the file using its existing name. If the file is newly created, the **Save file as** dialog box opens, where you assign a name to the file and save it. Another way to save is by using the save button 💾 on the toolbar. If the file has not been changed, **Save** is grayed out, but you can instead use **Save As** to save to a different filename.

- **Save As**—The **Save file as** dialog box opens, where you assign a name to the file and save it. You do not need to type the `.m` extension because MATLAB automatically assigns the `.m` extension to the filename. If you do not want an extension, type a `.` (period) after the filename.

- **Save All**—Saves all named files to their existing filenames. For all newly created files, the **Save file as** dialog box opens, where you assign a name to each file and save it.

You cannot save a file while in debug mode. First, exit debug mode and then save the file.

Note Save any M-files you create and any MathWorks-supplied M-files that you edit in a directory that is not in the `$matlabroot/toolbox` directory tree. If you keep your files in `$matlabroot/toolbox` directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the `$matlabroot/toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to `$matlabroot/toolbox` directories using an external editor or add or remove in from these directories using file system operations, run `rehash toolbox` before you use the files in the current session. If you make changes to existing files in `$matlabroot/toolbox` directories using an external editor, run `clear functionname` before you use the files in the current session. For more information, see `rehash` or "Toolbox Path Caching" on page 1-9.

Autosave

As you make changes to a file in the Editor, every five minutes the Editor automatically saves a copy of the file to a file of the same name but with an `.asv` extension. The autosaved copy is useful if you have system problems and

lose changes made to your file. In that event, you can open the autosaved version, `filename.asv`, and then save it as `filename.m` to use the last good version of `filename`.

Use "Autosave Preferences for the Editor/Debugger" on page 7-56 to turn the autosave feature off or on, to specify the number of minutes between automatic saves, and to specify the file extension and location for autosaved files.

Autosaved files are not automatically deleted when you delete the source file. So if, for example, you rename a file, delete the `.asv` version of the original filename.

If the M-file you are editing is in a read-only directory, an autosave copy of the file is *not* made.

Running M-Files from the Editor/Debugger

You can run a script or a function that does not require an input argument directly from the Editor/Debugger. Click the run button on the toolbar, or select **Run** from the **Debug** menu.

If the file is not in a directory on the search path or in the current directory, a dialog box appears, presenting you with options that allow you to run the file. You can either change the current directory to the directory containing the file, or you can add the directory containing the file to the search path.

Note that if the file has unsaved changes, running it from the Editor/Debugger automatically saves the changes before running. In that event, the menu item is **Save and Run**.

See "Running an M-File with Breakpoints" on page 7-33 for additional information about running M-files while debugging.

Printing an M-File
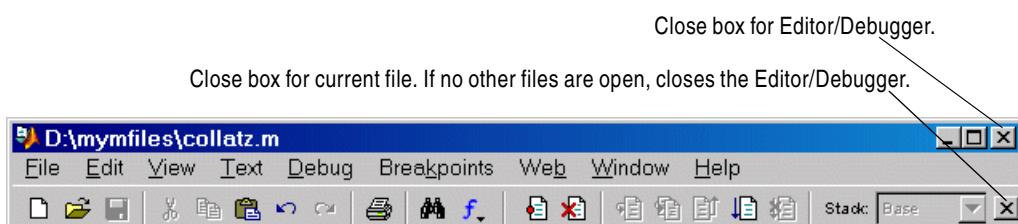
To print an entire M-file, select **File -> Print**, or click the print button on the toolbar. To print the current selection, select **File -> Print Selection**. Complete the standard print dialog box that appears.

Specify printing options for the Editor by selecting **File -> Page Setup**. For example, you can specify printing with a header. For more information, see "Page Setup Options for Printing" on page 2-25.

Closing M-Files

To close the current M-file, select **Close filename** from the **File** menu. If there are multiple files open in a single Editor/Debugger window, click the close box in the Editor toolbar to close the current M-file. This is different from the close box in the titlebar of the Editor/Debugger, which closes the Editor/Debugger, including all open files.

Note When you close the current file and it is the only file open, then the Editor/Debugger closes as well.

Close box for Editor/Debugger.

Close box for current file. If no other files are open, closes the Editor/Debugger.

If each file is open in a separate Editor/Debugger window, close all the files at once using the **Close All** item in the **Window** menu.

When you close a file that has unsaved changes, you are prompted to save the file.

Debugging M-Files

This section introduces general techniques for finding errors, and then illustrates MATLAB debugger features found in the Editor/Debugger and equivalent debugging functions using a simple example. It includes these topics:

- "Types of Errors" on page 7-25
- "Finding Errors" on page 7-26
- "Debugging Example—The Collatz Problem" on page 7-26
- "Trial Run for Example" on page 7-29
- "Using Debugging Features" on page 7-30

In addition to the Debugger and debugging functions, the Profiler included with MATLAB can be a useful tool to help you improve performance and detect problems in your M-files. For details, see Measuring Performance in the Programming and Data Types section of the MATLAB documentation.

Types of Errors

Debugging is the process by which you isolate and fix problems with your code. Debugging helps to correct two kinds of errors:

- Syntax errors—For example, misspelling a function name or omitting a parenthesis. "Syntax Highlighting" on page 7-10 helps you identify these problems, as does the process of setting breakpoints.

  When you run an M-file with a syntax error, MATLAB will most likely detect it and display an error message in the Command Window describing the error and showing its line number in the M-file. Click the underlined portion of the error message, or position the cursor within the message and press **Ctrl+Enter**. The offending M-file opens in the Editor, scrolled to the line containing the error. Use the pcode function to check for syntax errors in your M-file without running the M-file.

- Run-time errors—These errors are usually algorithmic in nature. For example, you might modify the wrong variable or perform a calculation incorrectly. Run-time errors are apparent when an M-file produces unexpected results.

Finding Errors

Usually, it is easy to find syntax errors based on MATLAB error messages. Run-time errors are more difficult to track down because the function's local workspace is lost when the error forces a return to the MATLAB base workspace. Use the following techniques to isolate the causes of run-time errors:

- Remove selected semicolons from the statements in your M-file. Semicolons suppress the display of intermediate calculations in the M-file. By removing the semicolons, you instruct MATLAB to display these results on your screen as the M-file executes.

- Add `keyboard` statements to the M-file. Keyboard statements stop M-file execution at the point where they appear and allow you to examine and change the function's local workspace. This mode is indicated by a special prompt:

    ```
    K>>
    ```

    Resume function execution by typing `return` and pressing the **Return** key.

- Comment out the leading function declaration and run the M-file as a script. This makes the intermediate results accessible in the base workspace.

- Use the `depfun` function to see the dependent functions.

- Use the MATLAB Editor/Debugger or debugging functions. They are useful for correcting run-time errors because you can access function workspaces and examine or change the values they contain. You can set and clear *breakpoints*, lines in an M-file at which execution halts. You can change workspace contexts, view the function call stack, and execute the lines in an M-file one by one.

The remainder of this section on debugging M-files describes the use of the Editor/Debugger and debugging functions using an example.

Debugging Example—The Collatz Problem

The example debugging session requires you to create two M-files, `collatz.m` and `collatzplot.m`, that produce data for the Collatz problem.

For any given positive integer, `n`, the Collatz function produces a sequence of numbers that always resolves to 1. If `n` is even, divide it by 2 to get the next integer in the sequence. If `n` is odd, multiply it by 3 and add 1 to get the next

integer in the sequence. Repeat the steps until the next integer is 1. The number of integers in the sequence varies, depending on the starting value, n.

The Collatz problem is to prove that the Collatz function will resolve to 1 for all positive integers. The M-files for this example are useful for studying the problem. The file collatz.m generates the sequence of integers for any given n. The file collatzplot.m calculates the number of integers in the sequence for any given integer and plots the results. The plot shows patterns that can be further studied.

Following are the results when n is 1, 2, or 3.

n	Sequence	Number of Integers in the Sequence
1	1	1
2	2 1	2
3	3 10 5 16 8 4 2 1	8

M-Files for the Collatz Problem

Following are the two M-files you use for the debugging example. To create these files on your system, open two new M-files. Select and copy the following code from the Help browser and paste it into the M-files. Save and name the files collatz.m and collatzplot.m. Save them to your current directory or add the directory where you save them to the search path. One of the files has an embedded error for purposes of illustrating the debugging features.

Code for `collatz.m`.

```
function sequence=collatz(n)
% Collatz problem. Generate a sequence of integers resolving to 1
% For any positive integer, n:
%   Divide n by 2 if n is even
%   Multiply n by 3 and add 1 if n is odd
%   Repeat for the result
%   Continue until the result is 1%

sequence = n;
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value];
end
```

Code for `collatzplot.m`.

```
function collatzplot(n)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and sequeence length
for m = 1:n
    plot_seq = collatz(m);
    seq_length(m) = length(plot_seq);
    line(m,plot_seq,'Marker','.','MarkerSize',9,'Color','blue')
    drawnow
end
```
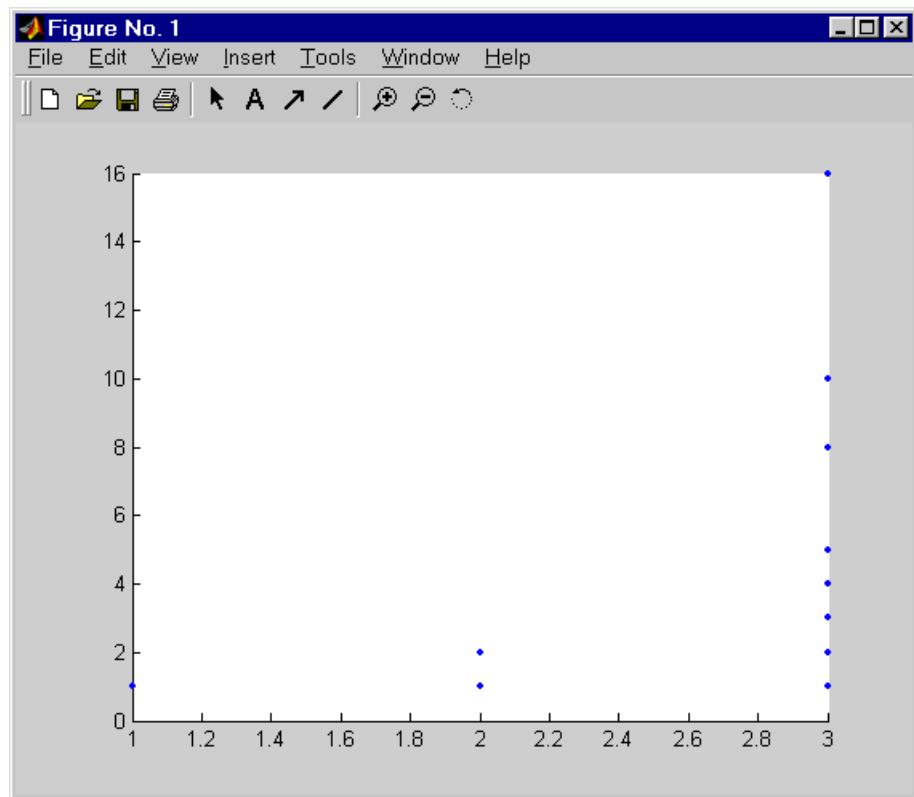
Trial Run for Example

Try out `collatzplot` to see if it works correctly. Use a simple input value, for example, 3, and compare the results to those shown in the preceding table. Typing

```
collatzplot(3)
```

produces the plot shown in the following figure.

The plot for 1 appears to be correct—when n = 1, the Collatz series is 1, and contains one integer. But for n = 2 and n = 3 it is wrong because there should be only one value plotted for each integer, the number of integers in the sequence, which the preceding table shows to be 2 (when n = 2) and 8 (when

n = 3). Instead, multiple values are plotted. Use MATLAB debugging features to isolate the problem.

Using Debugging Features

You can debug the M-files using the Editor/Debugger, which is a graphical user interface, and using debugging functions from the Command Window. You can use both methods interchangeably. The example describes both methods.

The debugging process consists of

- "Preparing for Debugging" on page 7-30
- "Setting Breakpoints" on page 7-30
- "Running an M-File with Breakpoints" on page 7-33
- "Stepping Through an M-File" on page 7-34
- "Examining Values" on page 7-36
- "Correcting Problems and Ending Debugging" on page 7-40

Preparing for Debugging

Do the following to prepare for debugging:

- Open the file—To use the Editor/Debugger for debugging, open it with the file you will run, in this example, collatzplot.m.
- Save changes—If you are editing the file, save the changes before you begin debugging. If you try to run a file with unsaved changes, the file is automatically saved before it runs.
- Add the files to a directory on the search path or put them in the current directory—Be sure the file you run and any files it calls are in directories that are on the search path. If all files to be used are in the same directory, you can instead make that directory be the current directory.

Setting Breakpoints
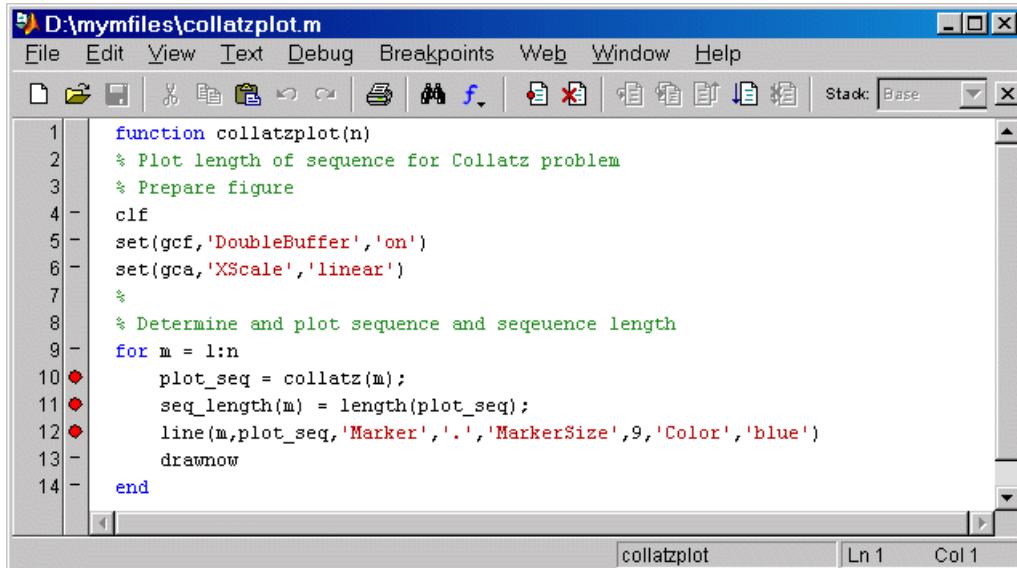
Set breakpoints to pause execution of the function so you can examine values where you think the problem might be. You can only set valid breakpoints at executable lines in saved files that are in the current directory or in directories on the search path.

When you create a new M-file, save it before setting breakpoints. You cannot set breakpoints while MATLAB is busy, for example, running an M-file, unless that M-file is paused at a breakpoint.

Breakpoints for the Example. It is unclear whether the problem in the example is in collatzplot or collatz. To start, set breakpoints in collatzplot.m at lines 10, 11, and 12. The breakpoint at line 10 allows you to step into collatz to see if the problem might be there. The breakpoints at lines 11 and 12 stop the program where you can examine the interim results.

Setting Breakpoints Using the Editor/Debugger. To set a breakpoint using the Editor/Debugger, click in the breakpoint alley at the line where you want to set the breakpoint. The breakpoint alley is the column to the right of the line number. Set breakpoints at lines that are preceded by a - (dash). Lines not preceded by a dash, such as comments or blank lines, are not executable—if you try to set a breakpoint there, it is actually set at the next executable line. Other ways to set a breakpoint are to position the cursor in the line and then click the set/clear breakpoint button 🔲 on the toolbar, or select **Set/Clear Breakpoint** from the **Breakpoints** menu or the context menu.
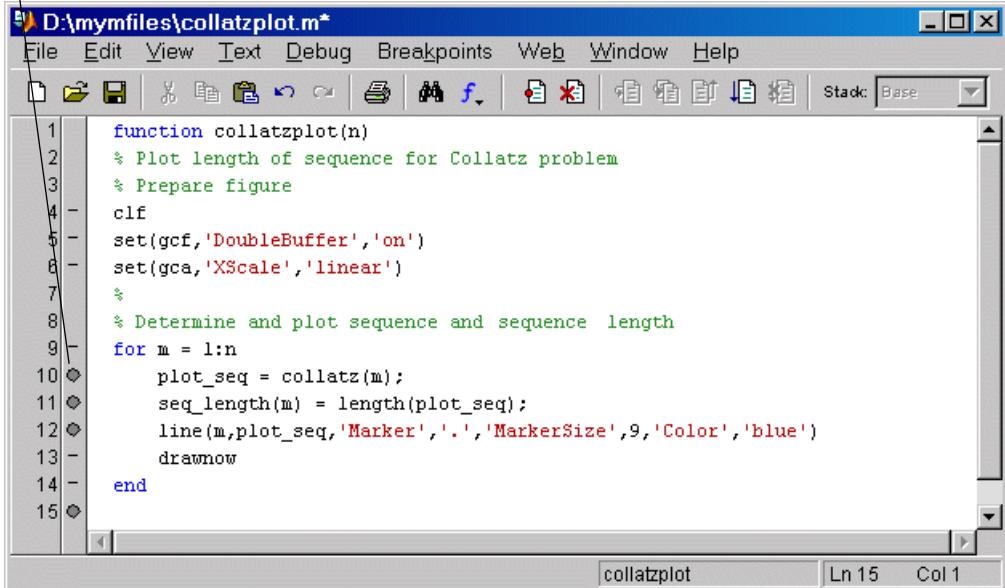
A breakpoint icon appears, as in the following illustration for the example.

Valid (Red) and Invalid (Gray) Breakpoints. Red breakpoints are valid breakpoints. If breakpoints are instead gray, they are not enabled and therefore not valid.

Breakpoints are gray, meaning they are not valid. In this example, it is because the file has not been saved since changes were made to it. Save the file to make the breakpoints valid (red).

Breakpoints are gray for either of these reasons:

- The file has not been saved since changes were made to it. Save the file to make breakpoints valid. The gray breakpoints become red, indicating they are now valid. Any gray breakpoints that were entered at invalid breakpoint lines automatically move to the next valid breakpoint line with the successful file save.

- There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. Fix the syntax error and save the file to make breakpoints valid.

Function Alternative. To set a breakpoint using the debugging functions, use `dbstop`. For the example, type

```
dbstop in collatzplot at 10
dbstop in collatzplot at 11
dbstop in collatzplot at 12
```

Some useful related functions are

- `dbtype`—Lists the M-file with line numbers in the Command Window.
- `dbstatus`—Lists breakpoints.

Setting Stops for Conditions. Use the items on the **Breakpoints** menu or the `dbstop` function equivalents to instruct the program to stop when it encounters a problem. For details, see `dbstop`.

- **Stop If Error**, or `dbstop if error`
- **Stop If Warning**, or `dbstop if warning`
- **Stop If NaN Or Inf** (for not-a-number or infinite value), or `dbstop if naninf` or `dbstop if infnan`
- **Stop If All Error**, or `dbstop if all error`

If the File Is Not on the Search Path or in the Current Directory. When you add or remove a breakpoint in a file that is not in a directory on the search path or in the current directory, a dialog box appears, presenting you with options that allow you to add or remove the breakpoint. You can either change the current directory to the directory containing the file, or you can add the directory containing the file to the search path.

Running an M-File with Breakpoints

After setting breakpoints, run the M-file from the Command Window or the Editor/Debugger.
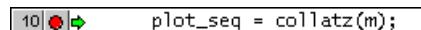
For the example, run `collatzplot` for the simple input value, 3, by typing in the Command Window

```
collatzplot(3)
```

The example, `collatzplot`, requires an input argument and therefore runs only from the Command Window and not from the Editor/Debugger.

Results of Running an M-File Containing Breakpoints. Running the M-file results in the following:

- The prompt in the Command Window changes to

  K>>

  indicating that MATLAB is in debug mode.
- The program is paused at the first breakpoint, which in the example is line 10 of collatzplot. This means that line 10 will be executed when you continue. The pause is indicated in the Editor/Debugger by the green arrow just to the right of the breakpoint as shown here.


```
10        plot_seq = collatz(m);
```

  If you use debugging functions from the Command Window and do not have the file you are debugging open in the Editor/Debugger, the line at which you are paused is displayed in the Command Window, if you do *not* have the Editor/Debugger preference for **Automatically open files while debugging** selected. For the example, it would show

  ```
  10   plot_seq = collatz(m);
  ```

- The function displayed in the **Stack** field on the toolbar changes to reflect the current function. The call stack includes subfunctions as well as called functions. If you use debugging functions, use dbstack to view the current call stack.
- If the file you are running is not in the current directory or a directory on the search path, you are prompted to either add the directory to the path or change the current directory.

While in debug mode, you can set breakpoints, step through programs, examine variables, and run other functions.

Stepping Through an M-File

While in debug mode, you can step through an M-file, pausing at points where you want to examine values.

Use the step buttons or the step items in the **Debug** menu of the Editor/Debugger, or use the equivalent functions.

Toolbar Button	Debug Menu Item	Description	Function Alternative
	Continue or **Run** or **Save and Run**	Continue execution of M-file until completion or until another breakpoint is encountered. The menu item says **Run** or **Save and Run** if a file is not already running.	dbcont
None	**Go Until Cursor**	Continue execution of M-file until the line where the cursor is positioned. Also available on the context menu.	None
	Step	Execute the current line of the M-file.	dbstep
	Step In	Execute the current line of the M-file and, if the line is a call to another function, step into that function.	dbstep in
	Step Out	After stepping in, run the rest of the called function or subfunction, leave the called function, and pause.	dbstep out

Stepping In. In the example, collatzplot is paused at line 10. Use the step-in button or type dbstep in in the Command Window to step into collatz and walk through that M-file. Stepping into line 10 of collatzplot takes you to line 9 of collatz. If collatz is not open in the Editor, it automatically opens if you have selected the Editor/Debugger preference to **Automatically open files while debugging**.

The pause indicator at line 10 of `collatzplot` changes to a hollow arrow ⇨, indicating that MATLAB control is now in a subfunction called from the main program. The call stack shows that the current function is now `collatz`.

In the called function, you can do the same things you can do in the main (calling) function—set breakpoints, run, step through, and examine values.

Stepping Out. In the example, the program is paused at step 9 in `collatz`. Because the problem results are correct for n = 1, continue running the program since there is no need to examine values until n = 2. One way to run through `collatz` is to step out, which runs the rest of `collatz` and returns to the next line in `collatzplot`, line 11. To step out, use the step-out button or type `dbstep out` in the Command Window.

Examining Values

While the program is paused, you can view the value of any variable currently in the workspace. Use the following methods to examine values:

- "Where to Examine Values" on page 7-36
- "Selecting the Workspace" on page 7-37
- "Viewing Values as Datatips in the Editor/Debugger" on page 7-37
- "Viewing Values in the Command Window" on page 7-39
- "Viewing Values in the Array Editor" on page 7-39
- "Evaluating a Selection" on page 7-39
- "Examining Values in the Example" on page 7-39

Many of these methods are used in "Examining Values in the Example" on page 7-39.

Where to Examine Values. When the program is paused, either at a breakpoint or at a line you have stepped to, you can examine values. Examine values when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as expected, then that line, or a previous line, contains an error.

In the example, because the results for n = 1 are correct, there is no need to examine values until n = 2. Click the continue button in `collatzplot` twice to move to line 10 of `collatzplot` again.

Selecting the Workspace. Variables assigned through the Command Window and created using scripts are considered to be in the base workspace. Variables created in each function have their own workspace. To examine a variable, you must first select its workspace. When you run a program, the current workspace is shown in the **Stack** field. In the example, the workspace is currently collatzplot. To examine values that are part of another function workspace currently running or the base workspace, first select that workspace from the list in the **Stack** field. If you use debugging functions, use dbup and dbdown to change to a different workspace.

Viewing Values as Datatips in the Editor/Debugger. In the Editor/Debugger, position the cursor to the left of a variable on that line. Its current value appears—this is called a datatip. It stays in view for a few seconds or until you move the cursor. If you have trouble getting the datatip to appear, click in the line and then move the cursor next to the variable.

In the example, step into collatz and then position the cursor over n in collatz—the datatip shows that n = 2, as expected. Note that the **Stack** shows collatz as the current function.

Hold the cursor over a variable to view its current value.

Viewing Values in the Command Window. You can do this while in debug mode, at the K>> prompt. To see the variables currently in the workspace, use who. Type a variable name in the Command Window and MATLAB displays its current value. For the example, to see the value of n , type

```
n
```

and MATLAB returns the expected result
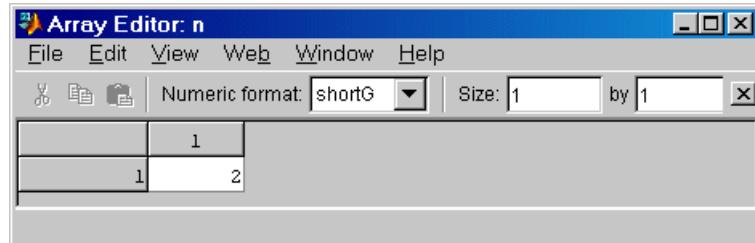
```
n =
   2
```

Viewing Values in the Array Editor. You can view the value of any variable in the Array Editor. To view the current variables, open the Workspace browser. In the Workspace browser, double-click a variable. The Array Editor opens, displaying the value for that variable. You can also open the Array Editor for a variable using openvar.

To see the value of n in the Array Editor for the example, type

```
openvar n
```

and the Array Editor opens, showing that n = 2 as expected.

Evaluating a Selection. Select a variable or equation in an M-file in the Editor/Debugger. Right-click and select **Evaluate Selection** from the context menu. MATLAB displays the value of the variable or equation in the Command Window. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Examining Values in the Example. In collatz, use the step button or the function dbstep. The program advances to line 10, where there is no need to examine values. Continue stepping until line 13. When you step again, the pause

indicator jumps to line 17, just after the if loop, as expected, given the code in line 13 for next_value = 2. When you step again to line 18, you can check the value of sequence in line 17 and see that it is 2 1 as expected for n = 2. Stepping again takes you from line 18 to line 11. At line 11, step again. Because next_value is 1, the while loop ends. The pause indicator is at line 11 and appears as a green down arrow ⬇. This indicates that processing in the called function is complete and program control will return to the calling program, in this case, collatzplot line 10. Step again from line 11 in collatz and execution is now paused at line 10 in collatzplot.

In collatzplot, step again to advance to line 11, then line 12. The variable seq_length in line 11 is a vector with the elements 1 2, which is correct.

Finally, step again to advance to line 13. Examining the values in line 12, m = 2 as expected, but the second variable, plot_seq, has two values, where only one value is expected. While plot_seq has the value expected, 2 1, it is the incorrect variable for plotting. Instead, seq_length(m) should be plotted.

Correcting Problems and Ending Debugging

These are some of the ways to correct problems and end the debugging session:

- "Changing Values and Checking Results" on page 7-40
- "Ending Debugging" on page 7-41
- "Clearing Breakpoints" on page 7-41
- "Correcting an M-File" on page 7-42

Many of these features are used in "Completing the Example" on page 7-42.

Changing Values and Checking Results. While debugging, you can change the value of a variable in the current workspace to see if the new value produces expected results. While the program is paused, assign a new value to the variable in the Command Window or in the Array Editor. Then continue running or stepping through the program. If the new value does not produce the expected results, the program has a different or another problem.

Ending Debugging. After identifying a problem, end the debugging session. You must end a debugging session if you want to change and save an M-file to correct a problem or if you want to run other functions in MATLAB.

Note It is best to quit debug mode before editing an M-file. If you edit an M-file while in debug mode, you can get unexpected results when you run the file. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See "Valid (Red) and Invalid (Gray) Breakpoints" on page 7-32 for details.

To end debugging, click the exit debug mode icon , or select **Exit Debug Mode** from the **Debug** menu.

You can instead use the function dbquit to end debugging.

After quitting debugging, the pause indicators in the Editor/Debugger display no longer appear, and the normal prompt >> appears in the Command Window instead of the debugging prompt, K>>. You can no longer access the call stack.

Clearing Breakpoints. Breakpoints remain in a file until you clear them or until they are automatically cleared.

Clear the breakpoints if you want the program to run uninterrupted, for example, after identifying and correcting a problem.

To clear a breakpoint in the Editor/Debugger, click the breakpoint icon for a line, or select **Set/Clear Breakpoint** from the **Breakpoints** or context menu. The breakpoint for that line is cleared.

To clear all breakpoints in all files, select **Clear All Breakpoints** from the **Breakpoints** menu, or click the equivalent button on the toolbar.

The function that clears breakpoints is dbclear. To clear all breakpoints, use dbclear all. For the example, clear all of the breakpoints in collatzplot by typing

```
dbclear all in collatzplot
```

Breakpoints are automatically cleared when you

- End the MATLAB session
- Clear the M-file using `clear name` or `clear all`

Correcting an M-File. To correct a problem in an M-file:

1 Quit debugging.

Do not make changes to an M-file while MATLAB is in debug mode. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See "Valid (Red) and Invalid (Gray) Breakpoints" on page 7-32 for details.

2 Make changes to the M-file.

3 Save the M-file.
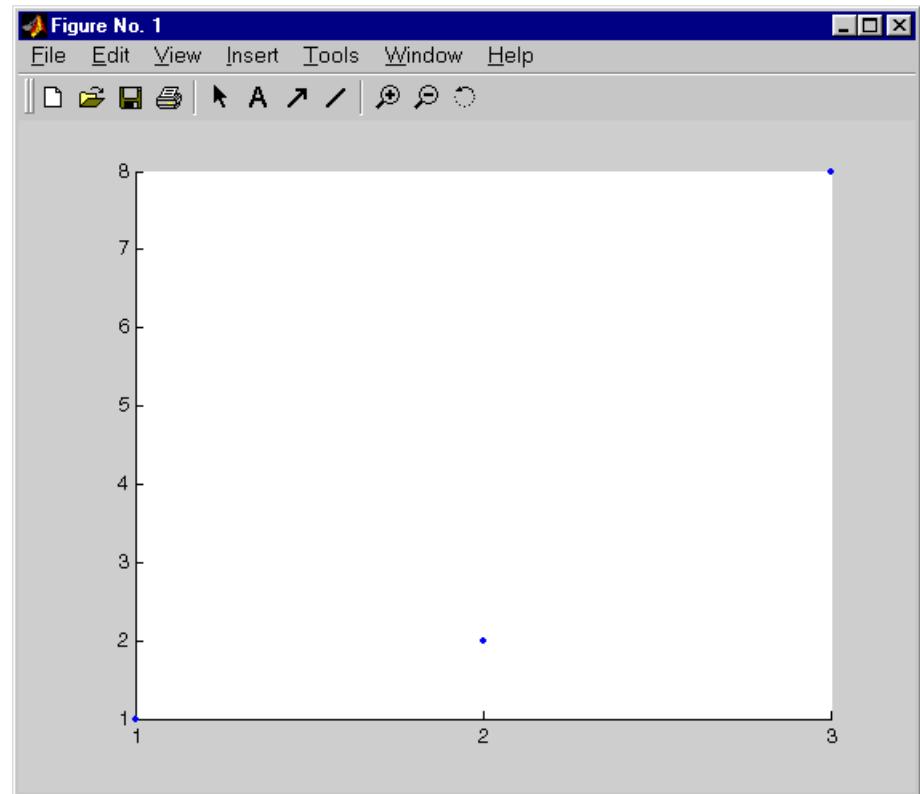
4 Set breakpoints, if desired.

5 Run the M-file again to be sure it produces the expected results.

Completing the Example. To correct the problem in the example, do the following:

1 End the debugging session. One way to do this is to select **Exit Debug Mode** from the **Debug** menu.

2 In `collatzplot.m` line 12, change the string `plot_seq` to `seq_length(m)` and save the file.

3 Clear the breakpoints in `collatzplot.m`. One way to do this is by typing

    dbclear all in collatzplot

in the Command Window.

4 Run `collatzplot` for n = 3 by typing

    collatzplot(3)

in the Command Window.

5 Verify the result. The figure shows that the length of the Collatz series is 1 when n = 1, 2 when n = 2, and 8 when n = 3, as expected.

6 Test the function for a slightly larger value of n, such as 6, to be sure the results are still accurate. To make it easier to verify `collatzplot` for n = 6 as well as the results for `collatz`, add this line at the end of `collatz.m`

```
sequence
```

which displays the series in the Command Window. The results for when n = 6 are

```
sequence =

     6     3    10     5    16     8     4     2     1
```

Then run `collatzplot` for n = 6 by typing

```
collatzplot(6)
```
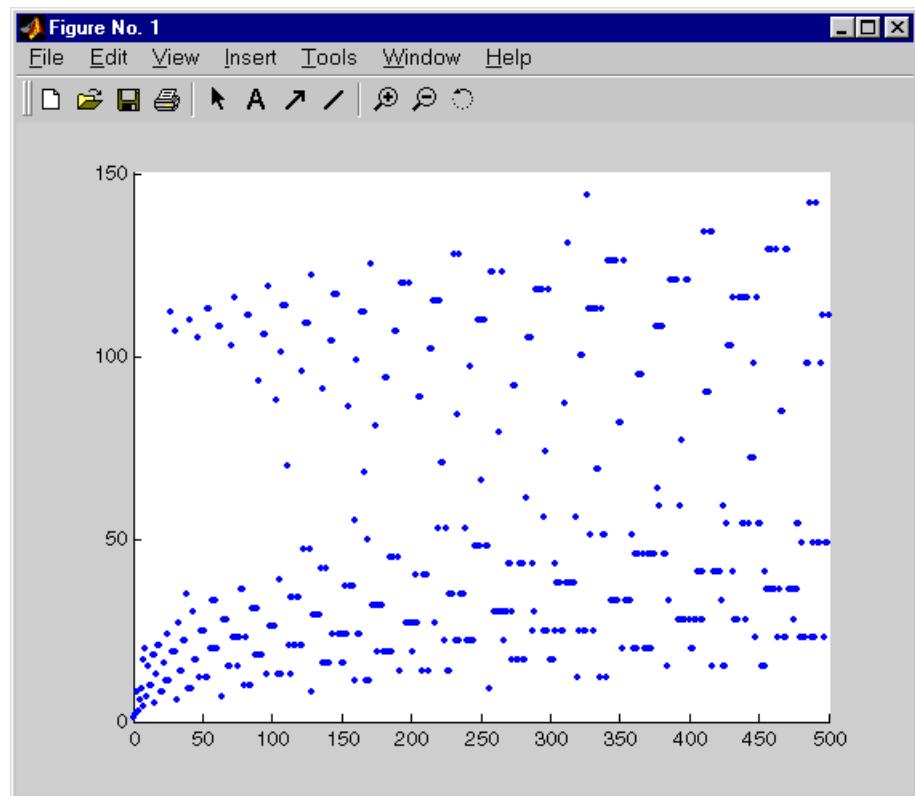
7 To make debugging easier, you ran `collatzplot` for a small value of n. Now that you know it works correctly, run `collatzplot` for a larger value to produce more interesting results. Before doing so, you might want to suppress output for the line you just added in step 6, line 19 of `collatz.m`, by adding a semicolon to the end of the line so it appears as

```
sequence;
```

Then run

```
collatzplot(500)
```
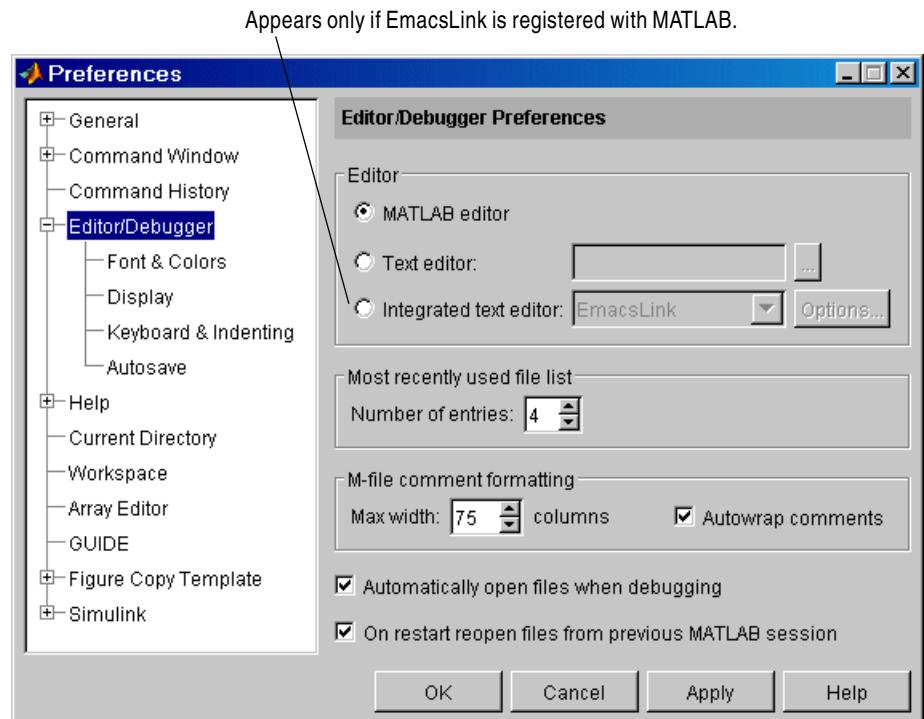
The following figure shows the lengths of the Collatz series for n = 1 through
n = 500.

Preferences for the Editor/Debugger

Using preferences, you can specify the default behavior for various aspects of the Editor/Debugger.

To set preferences for the Editor/Debugger, select **Preferences** from the **File** menu in the Editor/Debugger. The **Preferences** dialog box opens showing **Editor/Debugger Preferences**.

Appears only if EmacsLink is registered with MATLAB.

You can specify the following Editor/Debugger preferences:

- "General Preferences for the Editor/Debugger" on page 7-47 (on the first panel, including the **Editor** preference)
- "Font & Colors Preferences for the Editor/Debugger" on page 7-49
- "Display Preferences for the Editor/Debugger" on page 7-50
- "Keyboard and Indenting Preferences for the Editor/Debugger" on page 7-52
- "Autosave Preferences for the Editor/Debugger" on page 7-56

General Preferences for the Editor/Debugger

When you select **File -> Preferences** for the Editor/Debugger, you can specify the general preferences described here.

Editor

MATLAB editor. Selecting **MATLAB editor** means that MATLAB uses the built-in Editor/Debugger.

Text editor. To specify a text editor other than the MATLAB Editor, such as Emacs or vi, to be used when you open an M-file from within MATLAB, select **Text editor**. Specify the full pathname for the editor application you want to use.

For example, specify `c:/Applications/Emacs.exe` in the **Text editor** field, and then open a file using **Open** from the **File** menu in the MATLAB desktop. The file opens in Emacs instead of in the MATLAB Editor/Debugger.

Integrated text editor. This option appears only if you correctly registered EmacsLink with MATLAB. Select this if you want to use EmacsLink, a tool that allows you to use the Emacs editor with MATLAB debugging capabilities. EmacsLink is not supported by The MathWorks. For details on installing, registering, and using EmacsLink, see "Installing EmacsLink".

Most recently used file list

Use this preference to specify the number of files that appear in the list of most recently used files at the bottom of the **File** menu.

7-47

M-file comment formatting

Specify the **Max width**, that is, the maximum width, in number of columns, for M-file comments when you select the **Autowrap comments** preference.

For example, assume you select **Autowrap comments** and set the maximum width to be 75 characters, which is the width that will fit on a printed page using the default font for the Editor. When typing a comment line, as you reach the 75th column, the comment automatically continues on the next line.

The maximum width also applies when you use the **Format Selected Comment**s feature—see "Formatting Comments" on page 7-11.

Automatically open files when debugging

By default, the item **Automatically open files when debugging** is selected. The result is that when you run an M-file containing breakpoints, the MATLAB Editor/Debugger opens when it encounters a breakpoint.

If you use debugging functions, you might want to clear the item so that the Editor/Debugger does *not* open when a breakpoint is encountered.

On restart

To start MATLAB and automatically open the files that were open when you last shut down MATLAB, select the item **On restart open files from previous MATLAB session**. If the item is not selected and you close MATLAB when there are files open in the Editor/Debugger, the next time you start MATLAB, the Editor/Debugger is not opened upon startup.

Font & Colors Preferences for the Editor/Debugger

Use **Font & Colors** preferences to specify the font and colors used in files in the Editor/Debugger.

Font

Editor/Debugger font preferences specify the characteristics of the font used in files in the Editor/Debugger. Select **Use desktop font** if you want the font in the files to be the same as that specified under **General - Font & Colors**. If you want the font for Editor/Debugger files to be different, select **Use custom font** and specify the font characteristics:

- Type, for example, Lucida Console
- Style, for example, Plain
- Size in points, for example, 12 points

After you make a selection, the **Sample** area shows how the font will look. If you include tabs in lines of code, use a fixed width font, such as Monospaced, to align the tab positions on different lines.

You can specify different font characteristics for printing files from the Editor—see "Page Setup Options for Printing" on page 2-25.

Colors

Specify the colors used in files in the Editor/Debugger:

- **Text color**—The color of nonspecial text; special text uses colors specified for **Syntax highlighting**.
- **Background color**—The color of the background in the window.
- **Syntax highlighting**—The colors used to highlight syntax. Click **Set Colors** to specify them. For a description of syntax highlighting, see "Syntax Highlighting and Parentheses Matching" on page 3-6.

Display Preferences for the Editor/Debugger
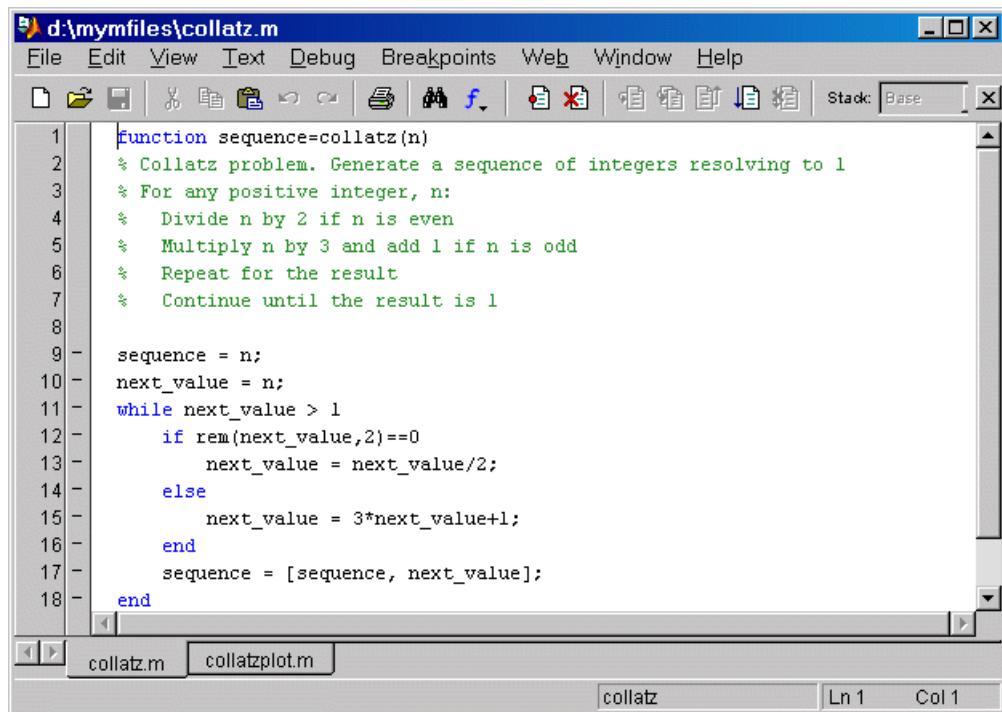
Use **Display** preferences to specify how the Editor/Debugger window should look.

Opening files in editor

This preference controls how files are arranged when you open them in the Editor/Debugger. When you change this preference, it applies to files you open after making the change. Currently opened files are not rearranged to match the preference.
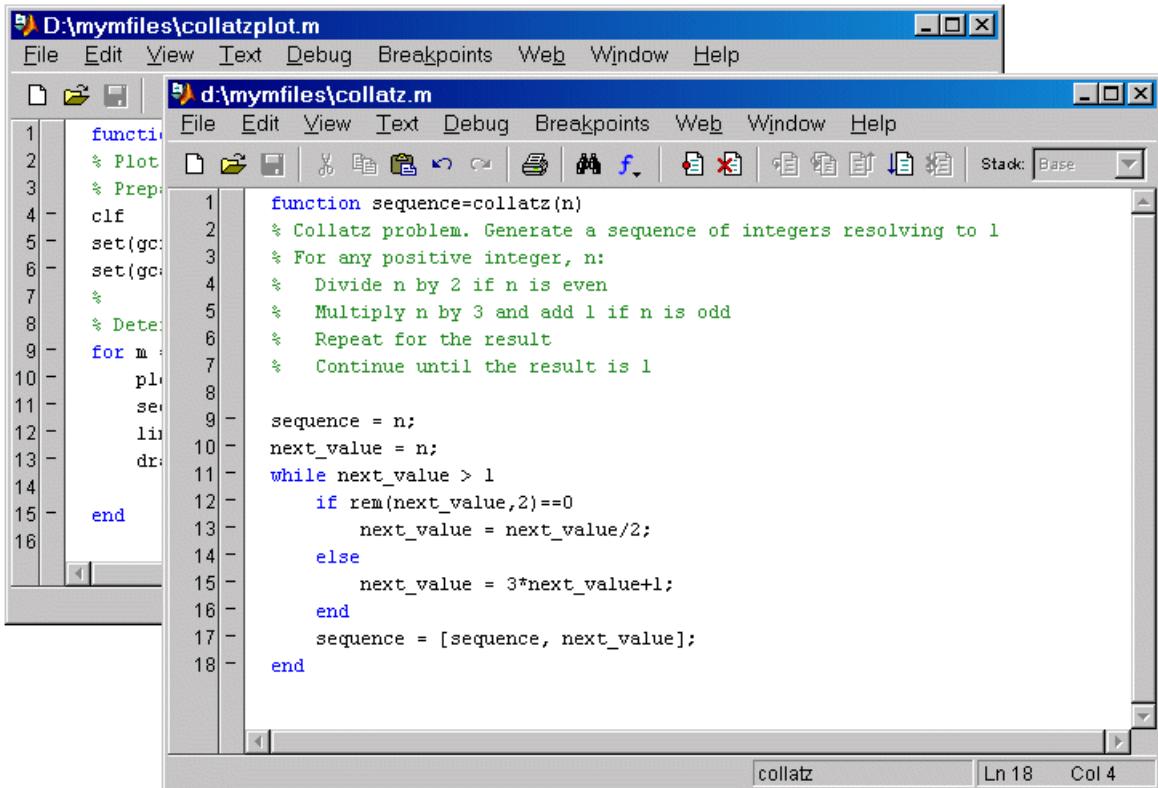
Select **Single window contains all files (tabbed style)** to have a single Editor/Debugger window for all open files, as shown in the following illustration. Click the tab for a file to make it the current file.

Files are tabbed within one window.

Select **Each file is displayed in its own window** to have a separate Editor/Debugger window for each open file, as shown in the following illustration.

Each file is in its own window.

Display

Use display options to specify what is shown and what is hidden in the Editor/Debugger.

Show toolbar. Select this item to display the toolbar. Clear it to hide the toolbar.

Show line numbers. Select this item to show line numbers. They appear along the left side of the window. When you clear this item, line numbers are not shown.

Enable datatips in edit mode. Select this item to see datatips while in edit mode. In edit mode, the datatips display the values of variables in the base workspace, so this is useful for script M-files rather than function M-files. Datatips are always enabled in debug mode.

Prompt

When you type `edit filename` and `filename` does not exist, MATLAB displays a prompt asking if you want to create a new file named `filename.m`, as described in "Creating a New M-File in the Editor/Debugger" on page 7-4. The prompt does *not* appear if you previously selected the check box to not show the prompt again. To again show the prompt, select the check box for **Show dialog prompt when editing files that do not exist**.

Keyboard and Indenting Preferences for the Editor/Debugger

Use keyboard and indenting preferences to specify the key binding conventions MATLAB should follow, how the Editor/Debugger indents lines, and how the Editor/Debugger alerts you to matches and mismatches in parentheses and other delimiters.

M-file indenting for Enter key

Select the style of indenting you want the Editor/Debugger to use when you press the **Enter** key. Examples follow, illustrating the different styles.

- **No indent**—No lines are indented. Use this if you want lines to be aligned on the left or want to insert line indents manually.
- **Block indent**—Indents a line the same amount as the line above it.
- **Smart indent**—Automatically indents lines that start with keyword functions or that follow certain keyword functions. Smart indenting can help you to follow the code sequence.

The indenting style only applies to lines you enter after changing the preference; it does not affect the indenting of existing lines. To change the indenting for existing lines, use the **Text** menu entries for "Indenting" (p. 7-10).

For any indenting style, you can manually insert tabs at the start of a line.

Example of No Indent Without Tabs.

```
sequence = n
next_value = n;
while next_value > 1
if rem(next_value,2)==0
next_value = next_value/2;
else
next_value = 3*next_value+1;
end
sequence = [sequence, next_value]
end
```

Created using **No indent** preference.
Did not manually insert any tabs.

Example of No Indent with Tabs.

```
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
    next_value = next_value/2;
    else
    next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value]
end
```

Created using **No indent** preference.
Created indentation by manually
inserting a tab before each indented line.

Example of Block Indent.

```
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
    next_value = next_value/2;
    else
    next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value]
    end
```

Created using **Block indent** preference.
Inserted a tab before the `if` statement.
Subsequent lines automatically indented
one tab.

Example of Smart Indent.

```
sequence = n
next_value = n;
while next_value > 1
    if rem(next_value,2)==0
        next_value = next_value/2;
    else
        next_value = 3*next_value+1;
    end
    sequence = [sequence, next_value]
end
```

Created using **Smart indent** preference.
Did not manually insert any tabs.
Indented lines were automatically
indented.

Key bindings

Select **Windows** or **Emacs** depending on which convention you want the Editor/Debugger to follow for accelerators and shortcuts. The accelerators on the menus change after you change this option.

For example, when you select Windows key bindings, the shortcut to paste a selection is **Ctrl+V**. When you select Emacs key bindings, the shortcut to paste a selection is **Ctrl+Y**. You can see the accelerator on the **Edit** menu for the **Paste** item.

See also "Arrow and Control Keys to Navigate in the Editor" on page 7-13.

Tabs and indents

Tab size. Specify the amount of space inserted when you press the **Tab** key. When you change the **Tab size**, it changes the tab size for existing lines in that file, unless you inserted the tabs with the preference for **Tab key inserts spaces** selected.

Tab key inserts spaces. Select this item if you want a series of spaces to be inserted when you press the **Tab** key. If the item is not selected, a tab acts as one space whose length is determined by **Tab size**.

Indent size. Specify the indent size for smart indenting.

Emacs style Tab key smart indenting. This indenting convention is based on the style used by the Emacs editor and is similar to the **Smart indent** preference. When you select the style, lines are indented according to smart indenting practices not as you type but when you position the cursor in that line or select a group of lines, and then press the **Tab** key. With this preference selected, you cannot use tabs within a line.

Parentheses Matching Preferences for the Editor/Debugger

The Editor/Debugger alerts you to matches and mismatches in pairs of delimiters, that is, in parentheses ( ), brackets [ ], and braces { }, based upon the MATLAB language syntax rules.

Match parentheses while typing. Select the check box if you want to be alerted to matches and mismatches in pairs of delimiters as you type them. Then choose how you want the Editor/Debugger to alert you to matches using **Show match with**. When you type a closing (or opening) delimiter, the Editor/Debugger alerts you based on the option you choose:

- **Balance**—The corresponding delimiter is highlighted briefly.
- **Underline**—Both delimiters in the pair are underlined briefly.
- **Highlight**—Both delimiters in the pair are highlighted briefly.
- **Bold**—Both delimiters in the pair briefly appear in bold.

Also choose how you want MATLAB to alert you to mismatches using **Show mismatch with**. When you type a closing delimiter that does not have an opening match, MATLAB alerts you based on the option you choose:

- **Beep**—MATLAB beeps.
- **Strikethrough**—The delimiter you typed is briefly crossed out.
- **Gray**—The delimiter you typed is briefly grayed out.
- **None**—There is no action.

Match parentheses on arrow key movement. Select the check box if you want to be alerted to matches and mismatches in pairs of delimiters when you move the cursor over a delimiter using forward and back arrow keys. Then choose how you want the Editor/Debugger to alert you to matches using **Show match with**. When you move the cursor over a closing (or opening) delimiter, the Editor/Debugger alerts you based on the option you choose:

- **Underline**—Both delimiters in the pair are underlined briefly.
- **Highlight**—Both delimiters in the pair are highlighted briefly.
- **Bold**—Both delimiters in the pair are bolded briefly.

Also choose how you want the Editor/Debugger to alert you to mismatches using **Show mismatch with**. When you move the cursor over a delimiter that does not have a match, the Editor/Debugger alerts you based on the option you choose:

• **Beep**—The Editor/Debugger beeps.
• **Strikethrough**—The delimiter is briefly crossed out.
• **Gray**—The delimiter briefly appears in gray.
• **None**—There is no alert.

Autosave Preferences for the Editor/Debugger

You can specify options for the Editor's "Autosave" feature (p. 7-22).

Autosave on

The Editor automatically saves the current version of the file you are editing. Clear this check box if you do not want the Editor to automatically save the file.

Save every n minutes. Specify how often you want the Editor to automatically save the file you are editing.

Save untitled files. Clear this check box if you want the Editor to automatically save new files that you have not yet saved and given a name to. If selected, the first autosaved file is `Untitled.asv`. If the directory already contains a file named `Untitled.asv`, the autosaved file is named `Untitled2.asv`, and so on for additional unnamed files.

If autosave creates `Untitled.asv` and you subsequently save the file as `filename.m`, the next autosave version is `filename.asv`. `Untitled.asv` remains until you delete it.

Filename

Specify the extension used for autosaved files. The default setting for Windows platforms is the extension `.asv` (for *autosave*), making the autosaved file `filename.asv`. For Windows and UNIX platforms, you can select **Replace extension with** and specify any extension.

For UNIX platforms, the default is **Append file with** the tilde (~) character, making the autosaved file `filename.m~`. For Windows and UNIX platforms, you can select **Append file with** and specify a different string to append to the file.

Location

Specify the location for autosaved files. By default, each autosaved file is stored in the same directory as the source file, that is, the file you are editing. You can specify a single directory for all autosaved files, such as a directory you create called `autosaved_files`.

8

Interfacing with Source Control Systems

Interfacing with Source Control Systems is divided into two sections:

Source Control Interface on PC Platforms (p. 8-2)

Select and view the source control system, add files, check files into and out of source control, undo a check-out, remove files, view file history, compare file versions, and more.

Source Control Interface on UNIX Platforms (p. 8-33)

Select and view the source control system, check files into and out of source control, and undo a check-out.

Source Control Interface on PC Platforms

If you use a source control system (SCS) to manage your files, you can perform source control interface actions on M-files and Simulink and Stateflow files within MATLAB, Simulink, and Stateflow. You can interface to your source control system by using menus from a graphical user interface (GUI), or by using functions from the MATLAB command window.

MATLAB, Simulink, and Stateflow do not perform source control functions, but only provide an interface to your own source control system. This means, for example, that you can open a file in the MATLAB Editor and modify it without checking it out. However, the file will remain read-only so that you cannot accidentally overwrite the source control version of the file.

The Source Control Interface works with any source control system that conforms to the Microsoft Common Source Control standard.

Note Files must first be added to the source control system before any of the other source control system interface actions can be performed on the file.

This section includes the following topics:

Selecting and Viewing the Source Control System

To select the source control system to interface, follow these steps:

1 From the MATLAB Desktop, select **Preferences** from the **File** menu. You can also select this from Simulink and Stateflow model and library windows.

The **Preferences** dialog box opens.

2 Click the **+** for **General** and then select **Source Control**.

The currently selected system is shown. The list box will be populated by the systems installed in the machine that support the Microsoft Common Source Control standard. The default selection is None.

3 Select the system you want to use from the **Source control system** list.

4 Click **OK**.

Function Alternative for Viewing the Source Control System

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

---

1 To view the currently selected system, type

```
cmopts
```

MATLAB displays the current source control system. For example:

```
ans =

Microsoft Visual SourceSafe
```

2 To view all of the source control systems installed on your computer, type

```
verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed in your computer. For example:

```
ans =

'Microsoft Visual SourceSafe'

'Jalindi Igloo'
```

3 To open the currently selected source control system, type

```
verctrl ('runscc', winhandle)
```

Adding Files to the Source Control System
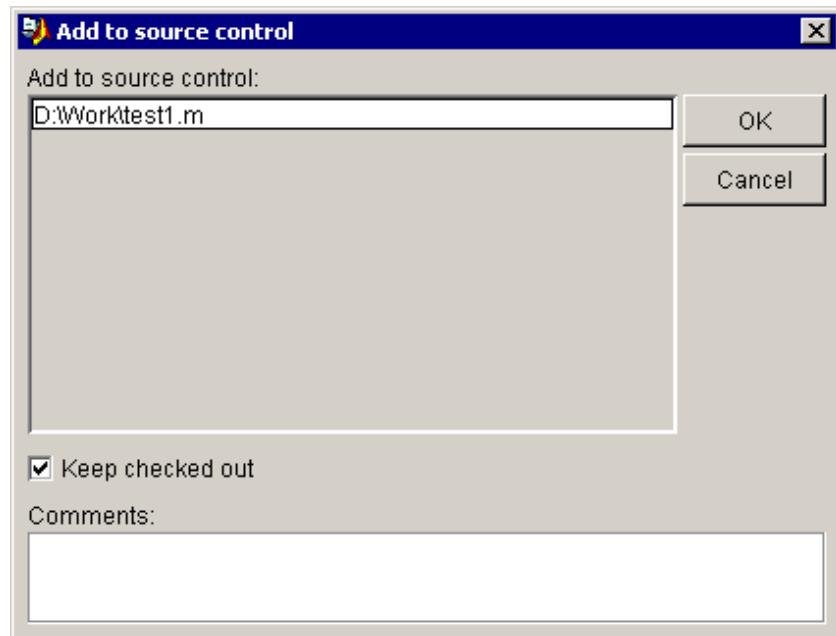
You can add a single file or multiple files to the source control system. Note that
the file is first added to the source control system using the **Add** command, not
the **Check In** command.

Adding a Single File

To add a file to the source control system:

1 Select **Source Control -> Add to Source Control** from the **File** menu in the
MATLAB Editor, Simulink model, or Stateflow model.

The MATLAB **Add to source control** dialog box opens.

2 If you want to add the file to the source control system and keep it checked out so you can continue making changes, select **Keep checked out**. This is selected by default. If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**.

3 Click **OK**.
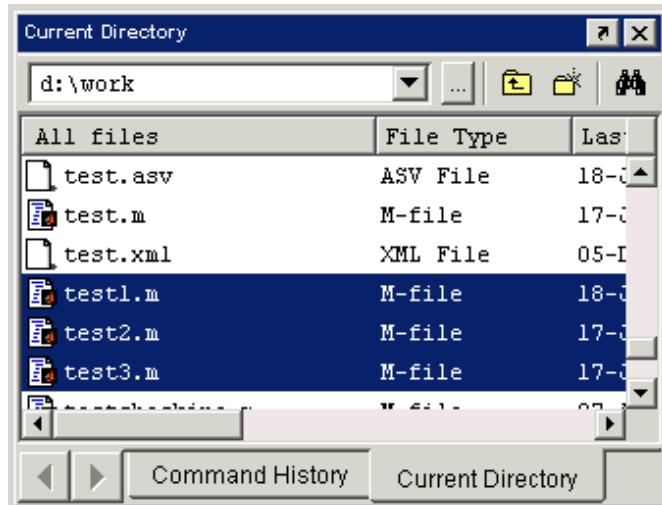
The file is added to the source control system. If you did not save the file before adding it to the source control system, it is automatically saved when it is added.

If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

Adding Multiple Files

To add multiple files to the source control system:
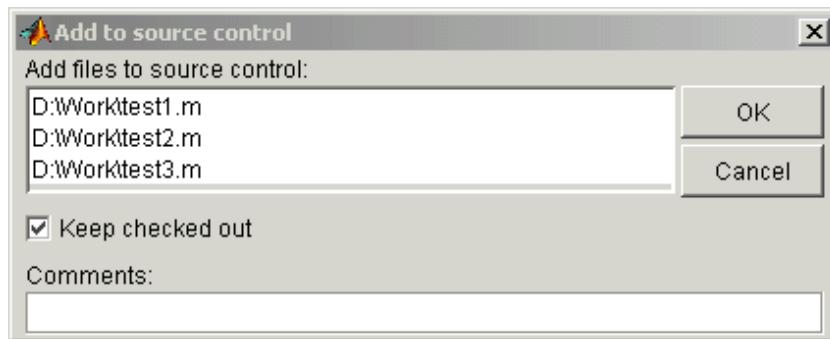
1 Select the files in the MATLAB **Current Directory** window.

2 Right click on the selected files.

3 Select **Source Control -> Add to Source Control** from the pop-up menu.

The MATLAB **Add to source control** dialog box opens.

4 If you want to add the files to the source control system and keep them checked out so you can continue making changes, select **Keep checked out**. This is selected by default. If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**.

5 Click **OK.**

The files are added to the source control system. If you did not save the files before adding them to the source control system, they are automatically saved when they are added.

If you did not keep the files checked out and you keep the files open, note that they are read-only versions.

Function Alternative for Adding File to Source Control

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use add as the first argument in the verctrl function to add a file to the source control system. Note that the verctrl function with the add argument returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. You can add a single file or multiple files. The verctrl function with the add argument has this form:

```
fileChange = verctrl('add',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Checking Files Out of the Source Control System
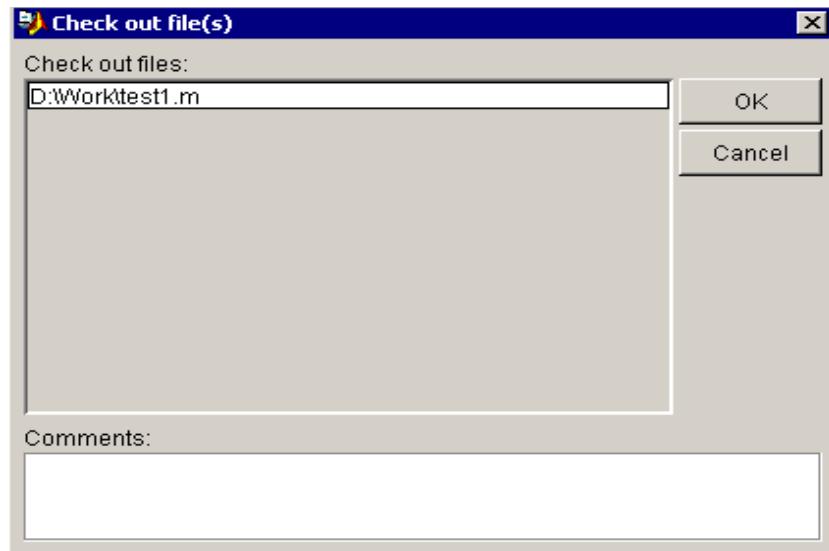
You can check out a single file or multiple files.

Checking out a Single File

To check out a single file from the source control system:

1 Select **Source Control -> Check Out** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The MATLAB **Check out file(s)** dialog box opens

.

2 Click **OK**.
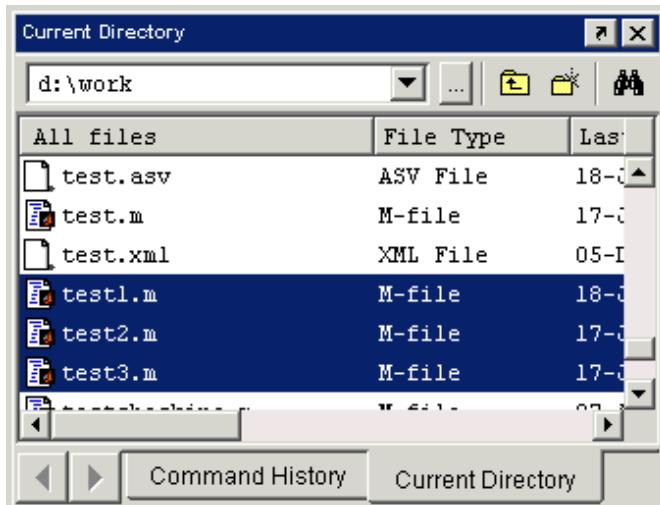
The file is checked out from the source control system and is available to you for editing.

Note The **Comments** text area will not be included in the **Check out file(s)** dialog box if the source control system does not support "comments" on file check-out.

Checking out Multiple Files

To check out multiple files from the source control system:

1 Select the files in the MATLAB **Current Directory** window,

2 Right-click on the selected files.

3 Select **Source Control -> Check Out** from the pop-up menu.

The MATLAB **Check out file(s)** dialog box opens.

4 Click **OK**.

The files are checked out from the source control system and are available to you for editing.

Function Alternative for Checking Out Files

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use checkout as the first argument in the verctrl function to check a file out of the source control system. Note that the verctrl function with the checkout argument returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. You can check out a single file or multiple files. The verctrl function with the checkout argument has this form:

```
fileChange = verctrl('checkout',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Checking Files into the Source Control System
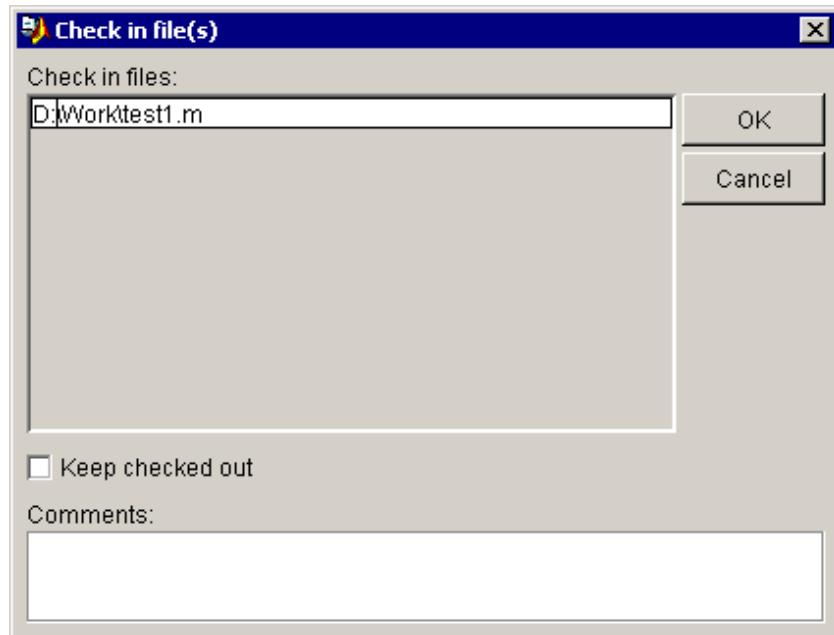
You can check in one or more MATLAB M-files, Simulink models, or Stateflow models.

Checking in a Single File

To check in a single file into the source control system:

1 Select **Source Control -> Check In** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model:

The MATLAB **Check in file(s)** dialog box opens.

.

2 If you want to check in the file to the source control system and keep it checked out so you can continue making changes, select **Keep checked out**. If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**.

3 Click **OK**.
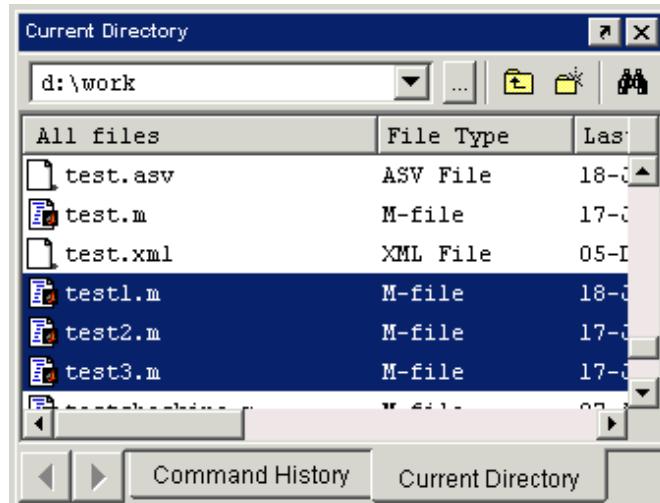
The file is checked into the source control system. If you did not save the file before checking it in, it is automatically saved when it is checked in.

If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

Checking in Multiple Files
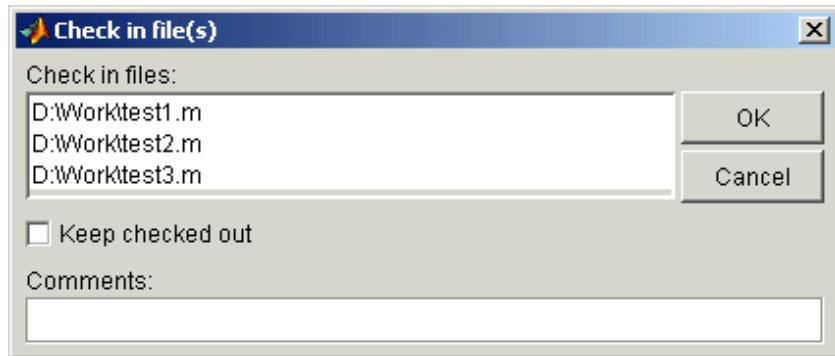
To check in multiple files into the source control system:

1 Select the files in the MATLAB **Current Directory** window,

2 Right click on the selected files.

3 Select **Source Control -> Check In** from the pop-up menu.

The MATLAB **Check in file(s)** dialog box opens.

4 If you want to check in the files to the source control system and keep them checked out so you can continue making changes, select **Keep checked out**. If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**.

5 Click **OK**.

The files are checked into the source control system. If you did not save the files before checking them in, they are automatically saved when they are checked in.

If you did not keep the files checked out and you keep the files open, note that they are read-only versions.

Function Alternative for Checking In Files

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use `checkin` as the first argument in the `verctrl` function to check files into the source control system. Note that the `verctrl` function with the `checkin` argument returns a logical `1` to the workspace if the file has changed on disk or a logical `0` to the workspace if the file has not changed on disk. You can check in a single file or multiple files. The files can be open or closed when you use `checkin`. The `verctrl` function with the `checkin` argument has this form:

```
fileChange = verctrl('checkin',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Getting the Latest Version of Files from the Source Control System
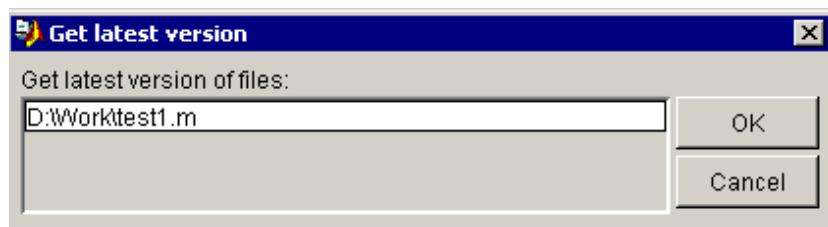
You can get the latest version of a file from the source control system for viewing and compiling, but not editing. You can get a single file, a single directory, multiple files, or multiple directories. The file or files will be tagged read only. The list of files should contain either files or directories but not both.

Getting the Latest Version of a Single File

To get the latest version of a single file:

1 Select **Source Control -> Get Latest Version** from the **File** menu in the MATLAB editor, Simulink model, or Stateflow model.

The MATLAB **Get latest version** dialog box opens.
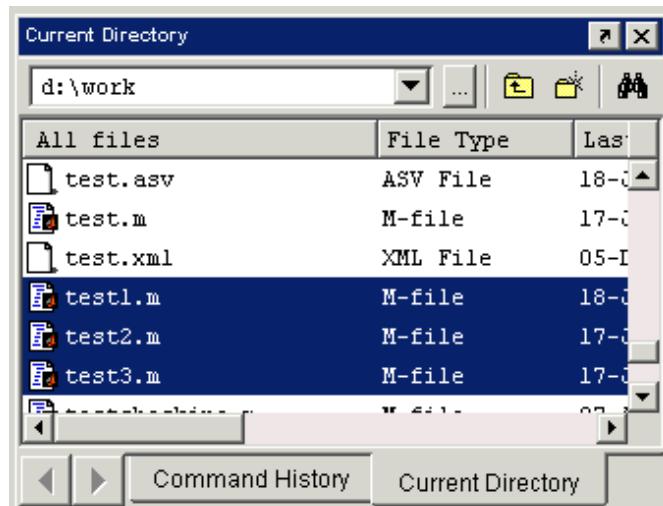
2 Click **OK**.

Getting the Latest Version of Multiple Files
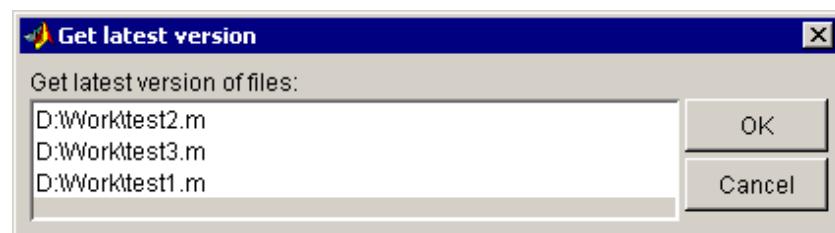
To get the latest version of multiple files:

1 Select the files in the MATLAB **Current Directory** window.

2 Right click on the selected files.

3 Select **Source Control -> Get Latest Version** from the pop-up menu.

The MATLAB **Get latest version** dialog box opens.

4 Click **OK**.

Function Alternative for Getting Latest Version

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use `get` as the first argument in the `verctrl` function to get a file from the source control system. Note that the `verctrl` function with the `get` argument returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. You can get a single file or multiple files. The `verctrl` function with the `get` argument has this form:

```
fileChange = verctrl('get',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```
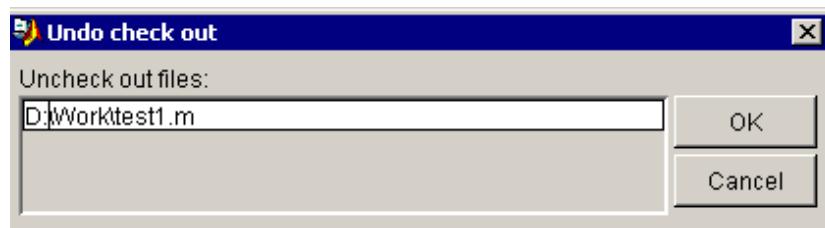
Undoing the Check-Out

You can undo the check-out for a file. The file remains checked in, without any of the changes you made since you checked them out. You can undo the check-out of a single file or multiple files.

Note You will lose the changes you have made since you checked out the file. To save these changes when undoing the check-out, use the **Save As** item from the **File** menu.

---

1 Select **Source Control -> Undo Check-Out** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The MATLAB **Undo check out** dialog box opens.

2 Click **OK**.

Function Alternative for Undoing a Check-Out

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use the uncheckout as the first argument in the verctrl function to undo a check-out. Note that the verctrl function with the uncheckout argument

returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. The verctrl function with the uncheckout argument has this form:
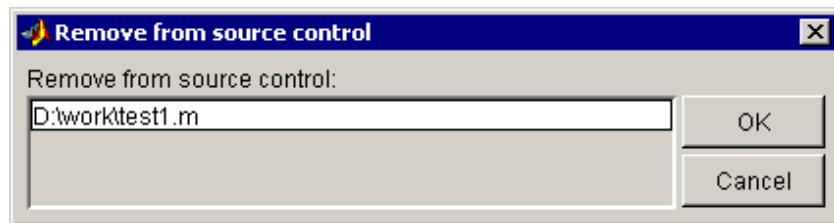
```
fileChange = verctrl('uncheckout',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Removing Files from the Source Control System

You can remove a single file or multiple files from the source control system. To remove a file from the source control system:

1 Select **Source Control -> Remove from Source Control** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The MATLAB **Remove from source control** dialog box opens.

2 Click **OK**.

Function Alternative for Removing File from Source Control

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use remove as the first argument in the verctrl function to remove a file or group of files from source control system. Note that the verctrl function with the remove argument does not return anything. The verctrl function with the remove argument has this form:
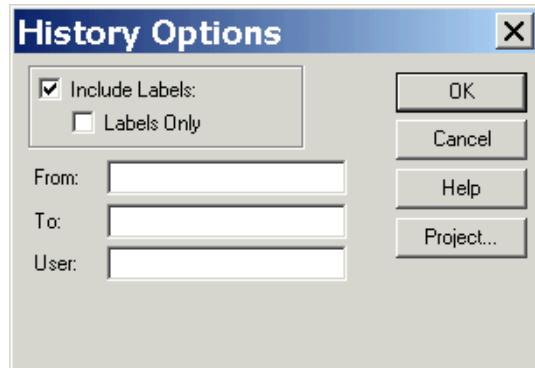
```
verctrl('remove',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Showing File History

You can show the history of a single file or multiple files in the source control system. To show the history of a file:
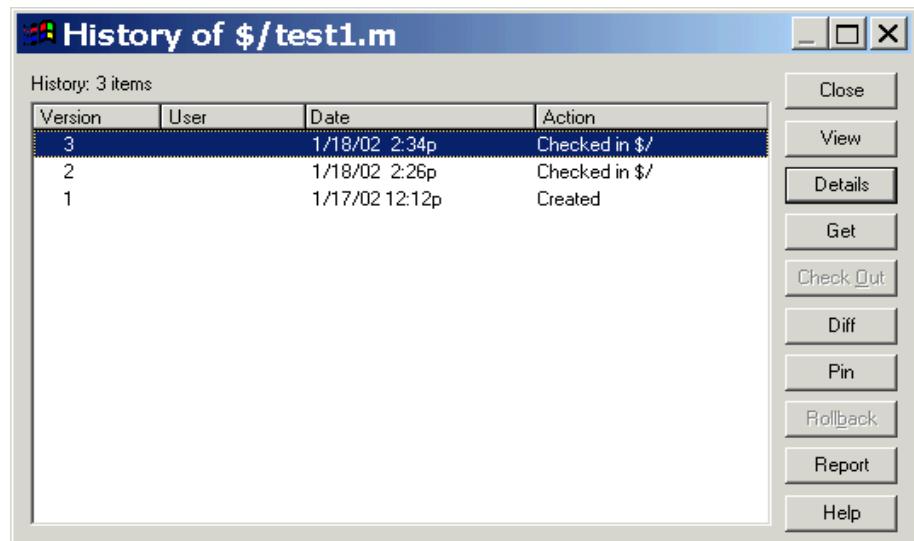
1 Select **Source Control -> Show History** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The dialog boxes returned are specific to the source control system being used. For example, if Microsoft Visual SourceSafe is the currently selected source control system, then the **History Options** dialog box will be returned.

2 Enter the appropiate label, date, and user information and Click **OK.**

The Microsoft Visual SourceSafe **History** dialog box opens

.

Function Alternative for Showing File History

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use history as the first argument in the verctrl function to show the history of the file in the source control system. Note that the verctrl function with the history argument returns a logical 1 to the workspace if the file has changed on disk or a logical 0 to the workspace if the file has not changed on disk. The verctrl function with the history argument has this form:

```
fileChange = verctrl('history',{'D:\file1.ext',..,'D:\filen.ext'},winhandle);
```

Comparing the Working Copy of a File to the Latest Version in Source Control

You can use the **Differences** option to compare the current working copy of a file on disk with the latest checked-in version of the file in the source control system. Note that you can only show differences on one file at a time, not multiple files.

Comparing M-Files
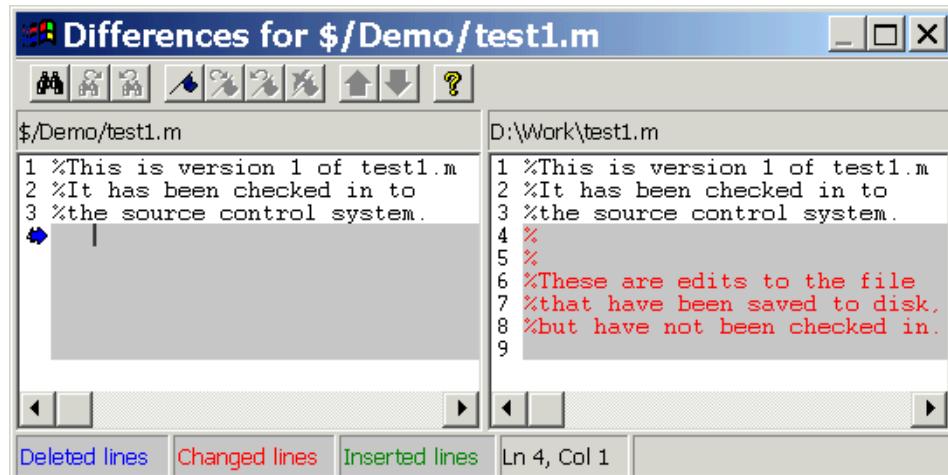
To show differences of an M-file:

1 Select **Source Control -> Differences** from the **File** menu in the MATLAB Editor.

A dialog box from the currently selected source control system opens. For example, if Microsoft Visual SourceSafe is the currently selected source control system, then the **Difference Options** dialog box opens.

2 Click **OK**.

The Microsoft Visual SourceSafe **Differences** dialog box opens. This compares the working copy of the file to the latest checked-in version of the file.
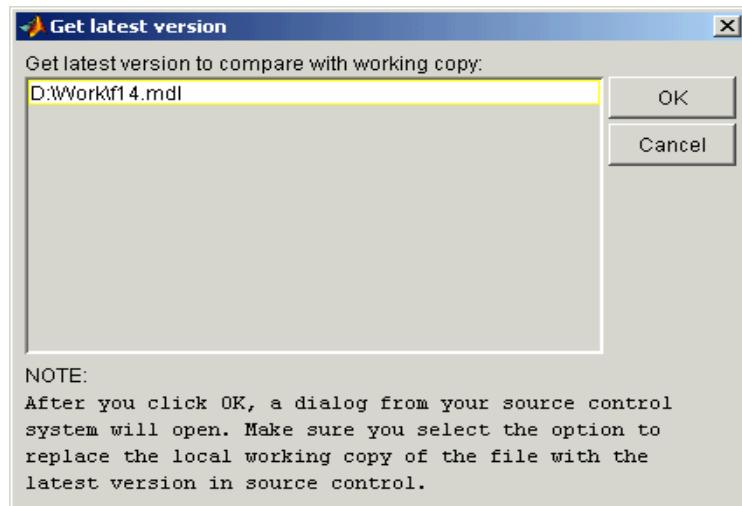
Comparing Simulink and Stateflow Model Files

Note The Simulink Graphical Model Merge tool is used to show the differences between versions of Simulink models. You need to have Simulink Performance Tools installed on your computer to do this.

To show differences of a Simulink or Stateflow model with Simulink Performance Tools installed on your system:

1 Select **Source Control -> Differences** from the **File** menu in the Simulink or Stateflow model.
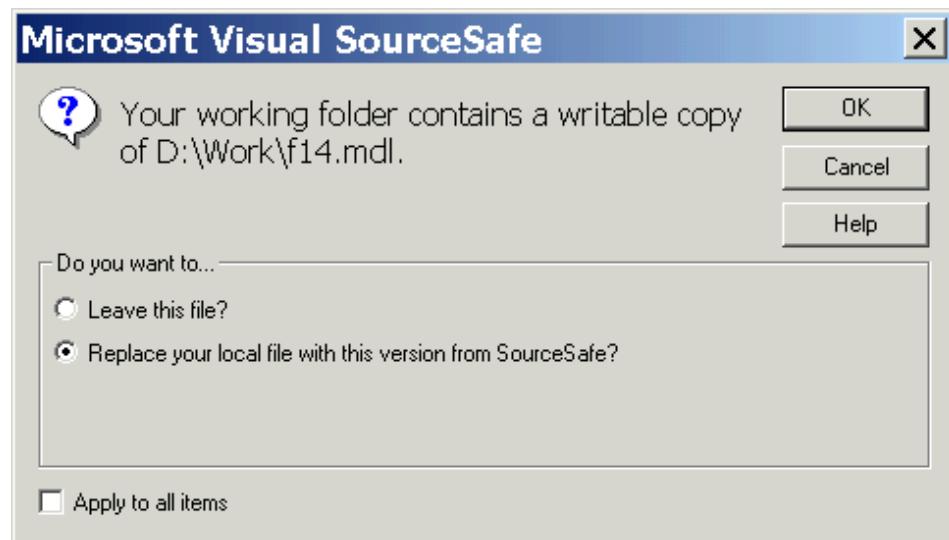
The MATLAB **Get latest version** dialog box opens.

2 Click **OK**.
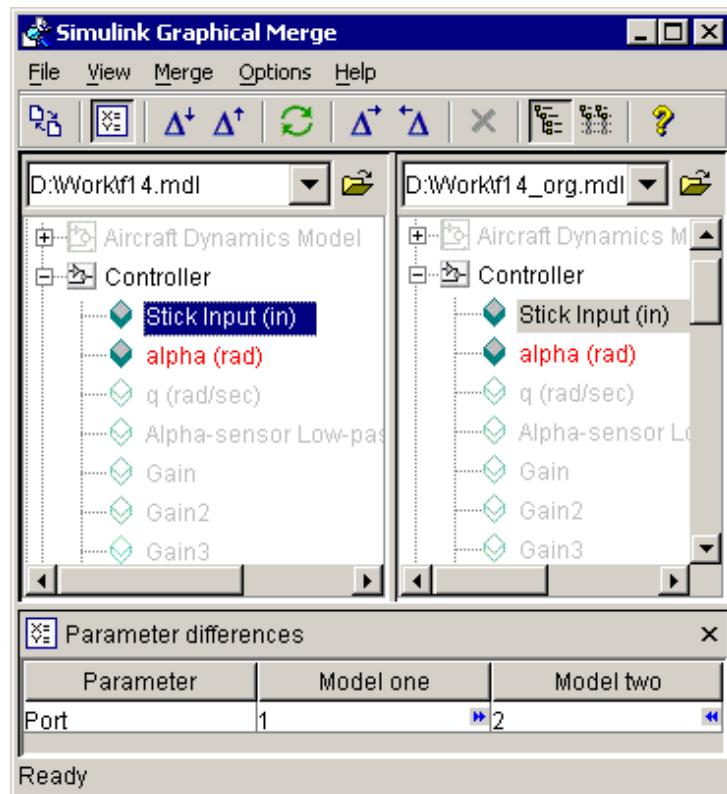
A dialog box from the currently selected source control system opens stating that there is a writable, working copy of the file. The dialog box will give you the option of either keeping the writable, working copy or replacing it with the latest version from the source control system. For example, if Microsoft

Visual SourceSafe is the currently selected source control system, then the following dialog box opens.

3 Select **Replace your local file with this version from SourceSafe?** or the equivalent option in other source control systems.

The Simulink Graphical Merge tool opens. For example, the following dialog shows that the Stick Input and alpha (rad) parameters in the Controller subsystem of the F14 model have been changed.

Function Alternative for Showing File Differences

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use `isdiff` as the first argument in the `verctrl` function to return a Boolean value to the window, which indicates whether or not there are any differences between the current file on disk and the latest checked-in version of the file. The `verctrl` function with the `isdiff` argument has this form:

```
fileChange = verctrl('isdiff','D:\file.ext',winhandle);
```

This will return the following in the command window if the two copies of the file are different:

```
fileChange =

1
```

Use `showdiff` as the first argument in the `verctrl` function to show the differences between the disk copy of a file and the latest checked-in version in the source control system. Note that the `verctrl` function with the `showdiff` argument does not return anything. The `verctrl` function with the `showdiff` argument has this form:
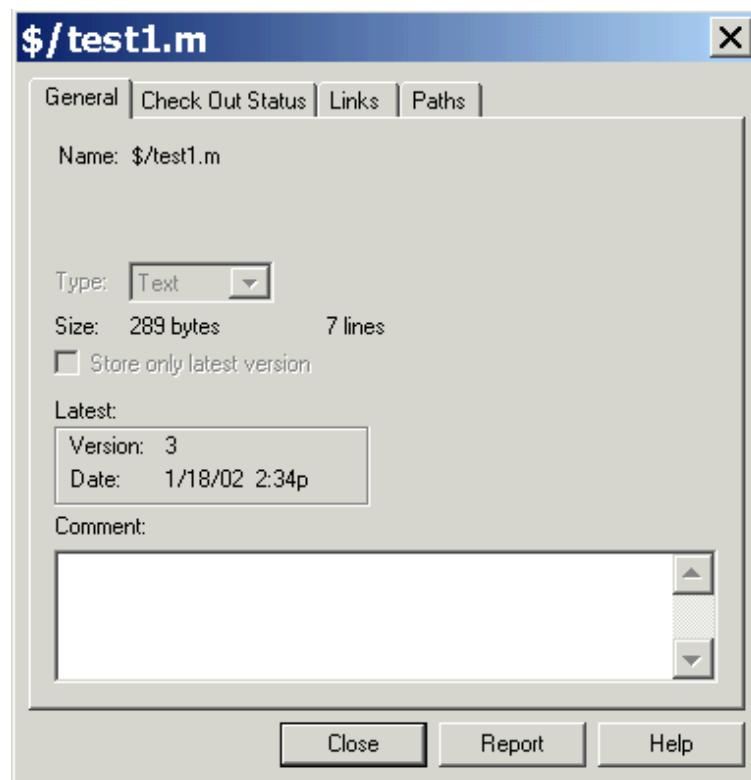
```
verctrl('showdiff','D:\file.ext',winhandle);
```

Displaying Source Control Properties of a File

You can display the properties of a single file from the source control system. Note that you cannot display the properties of multiple files. To display the properties of a file:

1 Select **Source Control -> Show Properties** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

A dialog box from the source control system being used opens. Below is a Microsoft Visual SourceSafe properties dialog box.

Function Alternative for Displaying File Properties

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use `properties` as the first argument in the `verctrl` function to display the properties of a file. Note that the `verctrl` function with the `properties` argument returns a logical `1` to the workspace if the file has changed on disk or a logical `0` to the workspace if the file has not changed on disk. The `verctrl` function with the `properties` argument has this form:
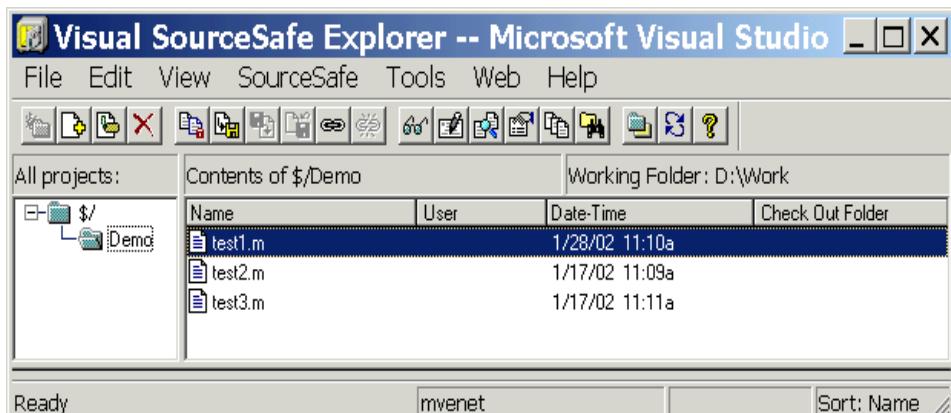
```
fileChange = verctrl('properties','D:\file.ext',winhandle);
```

Starting the Source Control System

To start the source control system:

1 Select **Source Control -> Start Source Control** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The dialog box from the currently selected source control system opens. Below is the Microsoft **Visual SourceSafe Explorer** dialog box.

Function Alternative for Starting Source Control

Note For MATLAB command window access to the source control interface, you must first create a window and get its handle. See "Using the Source Control Interface from the MATLAB Command Window" on page 8-31 for instructions on doing this.

Use runscc as the first argument in the verctrl function to start the currently selected source control system. The verctrl function with the runscc argument has this form:

```
verctrl('runscc',winhandle);
```

Using the Source Control Interface from the MATLAB Command Window

For MATLAB command window access to the source control interface, you must first create a window and get its handle:

1 To create a window and get its handle, enter the following in the MATLAB command window:

```
import java.awt.*;

frame = Frame('Test frame');

frame.setVisible(1);

winhandle=com.mathworks.util.NativeJava.hWndFromComponent(frame)
```

The MATLAB command window will return a handle:

```
winhandle =

919892
```

2 Perform source control operations using the verctrl function. Refer to the information on the specific source control operation for instructions on using the verctrl function:

- "Function Alternative for Viewing the Source Control System" on page 8-4
- "Function Alternative for Adding File to Source Control" on page 8-8
- "Function Alternative for Checking Out Files" on page 8-11
- "Function Alternative for Checking In Files" on page 8-15
- "Function Alternative for Getting Latest Version" on page 8-18
- "Function Alternative for Undoing a Check-Out" on page 8-19
- "Function Alternative for Removing File from Source Control" on page 8-20
- "Function Alternative for Showing File History" on page 8-22
- "Function Alternative for Showing File Differences" on page 8-27
- "Function Alternative for Displaying File Properties" on page 8-29
- "Function Alternative for Starting Source Control" on page 8-30

For addtional help on command window access to source control operations, enter the following in the MATLAB command window:

```
help verctrl.m
```

Source Control Interface on UNIX Platforms

If you use a source control system (SCS) to manage your files, you can check M-files and Simulink and Stateflow files into and out of the source control system from within MATLAB, Simulink, and Stateflow.

MATLAB, Simulink, and Stateflow do not perform source control functions, but only provide an interface to your own source control system. This means, for example, that you can open a file in the MATLAB Editor and modify it without checking it out. However, the file will remain read only so that you cannot accidentally overwrite the source control version of the file.

The Source Control Interface supports three popular source control systems, as well as a custom option:

- ClearCase from Rational Software
- PVCS Version Manager from Merant
- Revision Control System (RCS)
- Custom option — Allows you to build your own interface if you use a different source control system

You can interface to your source control system by using menus from a graphical user interface (GUI), or by using functions from the command window. There are some options you can perform using the MATLAB functions that are not available with the GUIs — these are noted in the instructions.

This section includes the following topics:

- "Selecting and Viewing the Source Control System" on page 8-34.
- "Setting a View and Checking Out a Directory — For ClearCase on UNIX Only" on page 8-35
- "Checking Files into the Source Control System" on page 8-36
- "Checking Files Out of the Source Control System" on page 8-38
- "Undoing the Check-Out" on page 8-40

Selecting and Viewing the Source Control System
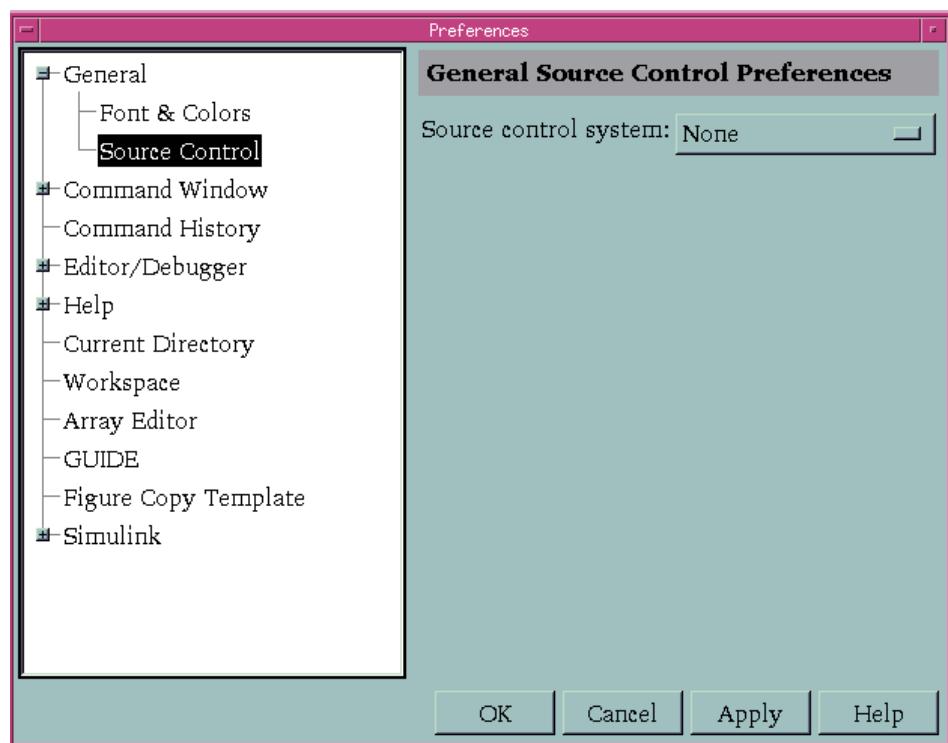
Specify the source control system using these steps:

1 Select **Preferences** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

The **Preferences** dialog box opens.

2 Click the **+** for **General** and then select **Source Control**.

The currently selected system is shown. The default selection is None.

.

3 Select the system you want to use from the **Source control system** list.

Function Alternative for Viewing the Source Control System

1 To view the currently selected system, type `cmopts` in the command window.

MATLAB displays the current source control system. For example:

```
ans =

PVCS Source Control
```

Setting a View and Checking Out a Directory — For ClearCase on UNIX Only

If you use ClearCase on a UNIX platform, do the following using ClearCase:
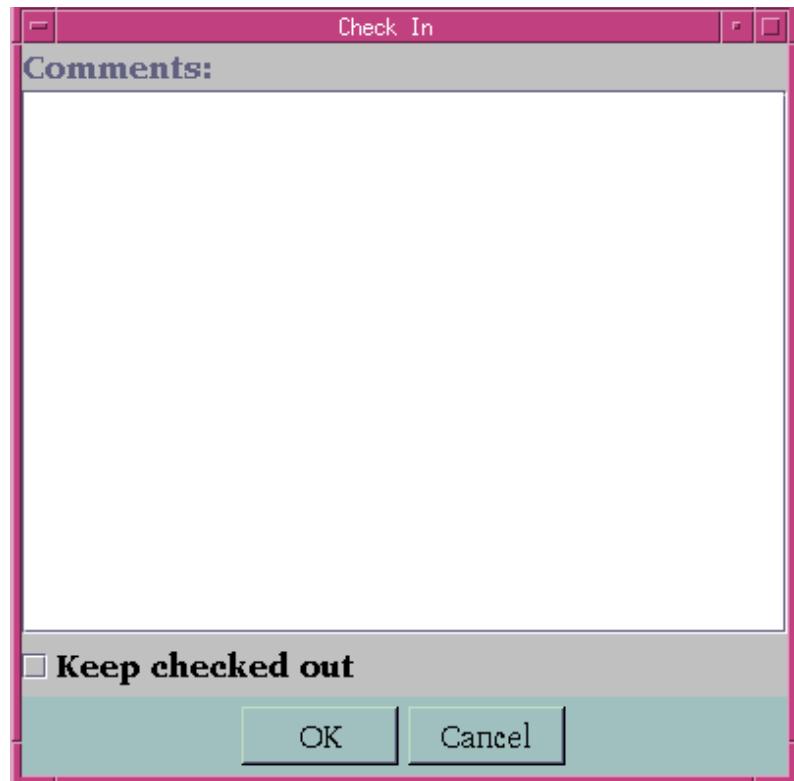
1 Set a view.

2 Check out the directory in which you want to save files, check files into, or check files out of.

You can now use the MATLAB, Simulink, or Stateflow interfaces to ClearCase to check files into and out of the directory you checked out in step 2.

Checking Files into the Source Control System

After creating or editing a file in the MATLAB Editor, Simulink, or Stateflow, save it, and then check in the file by following these steps:

1 Select **Source Control -> Check In** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model. The **Check In** dialog box opens.

2 If you want to check in the file but keep it checked out so you can continue making changes, select **Keep checked out**. If you have comments, type them in the **Comments** area.

Your comments will be submitted whether or not you select **Keep checked out**.

3 Click **OK**.

The file is checked into the source control system. If you did not save the file before checking it in, it is automatically saved when it is checked in.

If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

Function Alternative for Checking In Files

Use `checkin` to check files into the source control system. The files can be open or closed when you use `checkin`. The `checkin` function has this form:

```
checkin({'D:\file1.ext',..'D:\filen.ext'},'comments','string','option',...'value')
```

For `file`, use the complete path. You must supply the **comments** argument and a comments string with `checkin`.

Use the `option` argument to

• Check in a file and keep it checked out — set the `lock` option to `on`.
• Check in a file even though it has not changed since the previous check in — set the `force` option to `on`.

The **comments** argument and the `lock` and `force` options apply to all files checked in.

After checking in the file, if you did not keep it checked out and have it open, note that it is a read-only version.

Example – Check in a File with Comments

To check in the file `clock.m` with a comment `Adjustment for Y2K`, type

```
checkin('\matlabr12\mymfiles\clock.m','comments','Adjustment ... for Y2K')
```

For other examples, see the reference page for `checkin`.
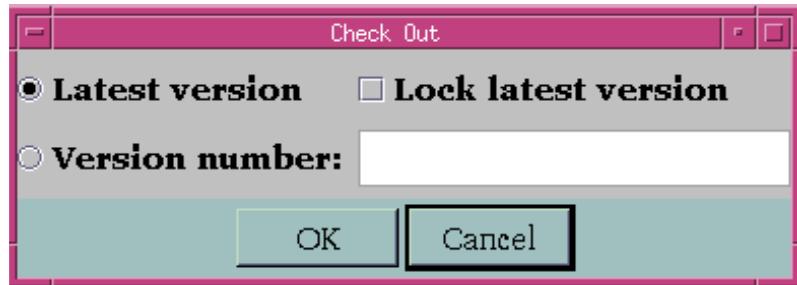
Checking Files Out of the Source Control System

To check files out of the source control system using MATLAB, follow these steps:

1 Open the M-file, Simulink file, or Stateflow file you want to check out.

The file opens and the title bar indicates it is read only.

2 Select **Source Control -> Check Out** from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model. The **Check Out** dialog box opens

3 To check out the version that was most recently checked in, select the **Latest version** option. To check out a specific version of the file, select the **Version number** option and type the version number in the field

To prevent others from checking out the file while you have it checked out, select **Lock latest version**. To check out a read-only version of the file, clear **Lock latest version**.

4 Click **OK**.

The file is checked out from the source control system and is available to you for editing.

Function Alternative for Checking Out Files

Use `checkout` to check a file out of the source control system. You can check out multiple files at once and specify check-out options. The `checkout` function has this form:

```
checkout({'D:\file1.ext',..'D:\filen.ext'},'option','value')
```

For `file`, use the complete path.

Use the `option` argument to:

- Check out a read-only version of the file — set the `lock` option to `off`.
- Check out the file even if you already have it checked out — set the `force` option to `on`.
- Check out a specific version of the file — use the `revision` option, and assign the version number to the `value` argument.

The options apply to all files checked out. The file can be open or closed when you use `checkout`.

Example — Check Out a Specific Version of a File

To check out the 1.1 version of the file `clock.m`, type

```
checkout('\matlab\mymfiles\clock.m', 'revision', '1.1')
```

For other examples, see the reference page for `checkout`.

Undoing the Check-Out

You can undo the check-out for a file. The files remain checked in, without any of the changes you made since you checked them out. If you want to keep a local copy of your changes, use the **Save As** item from the **File** menu:

1 Select **Source Control -> Undo Check-Out**. from the **File** menu in the MATLAB Editor, Simulink model, or Stateflow model.

There is no return dialog.

Function Alternative for Undoing a Check-Out

The undocheckout function has this form:

```
undocheckout({'D:\file1.ext',..'D:\filen.ext'})
```

Use the complete path for file.

Example – Undo the Check-Out for Two Files. To undo the check-out for the files clock.m and calendar.m, type

```
undocheckout({'\matlab\mymfiles\clock.m', '\matlab\mymfiles\calendar.m'})
```

9

Using Notebook

Notebook allows you to access the numeric computation and visualization software of MATLAB from within the word processing environment, Microsoft Word.

Notebook Basics

Notebook allows you to access the numeric computation and visualization software of MATLAB from within the word processing environment, Microsoft Word. Using Notebook, you can create a document, called an *M-book*, that contains text, MATLAB commands, and the output from MATLAB commands.

You can think of an M-book as a record of an interactive MATLAB session annotated with text, or as a document embedded with live MATLAB commands and output. Notebook is useful for creating electronic or printed records of MATLAB sessions, class notes, textbooks or technical reports. This section introduces basic Notebook capabilities:

- "Creating an M-Book" on page 9-2
- "Entering MATLAB Commands in an M-Book" on page 9-5
- "Protecting the Integrity of Your Workspace" on page 9-5
- "Ensuring Data Consistency" on page 9-6

Creating an M-Book

This section includes

- "Creating an M-Book from MATLAB" on page 9-2
- "Creating an M-Book While Running Notebook" on page 9-3
- "Opening an Existing M-Book" on page 9-3
- "Converting a Word Document to an M-Book" on page 9-4

Creating an M-Book from MATLAB

To create a new M-book from within MATLAB, type

```
notebook
```
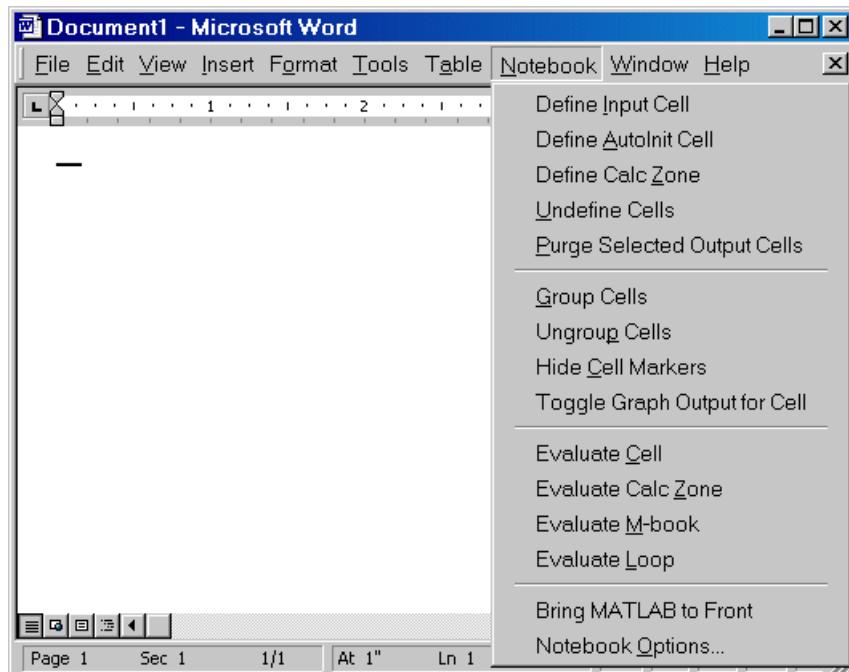
at the prompt. If you are running Notebook for the first time, you may need to configure it. See "Configuring Notebook" on page 9-23 for more information.

Notebook starts Microsoft Word on your system and creates a new M-book, called Document1.

When Word is opening, if a dialog box appears asking you to enable or disable macros, choose to enable macros. Notebook defines Microsoft Word macros that enable MATLAB to interpret the different types of cells that hold MATLAB

commands and their output. For more information on macro security, see "Configuring Notebook" on page 9-23.

Notebook adds the **Notebook** menu to the Word menu bar. Use this menu, illustrated below, to access Notebook features.
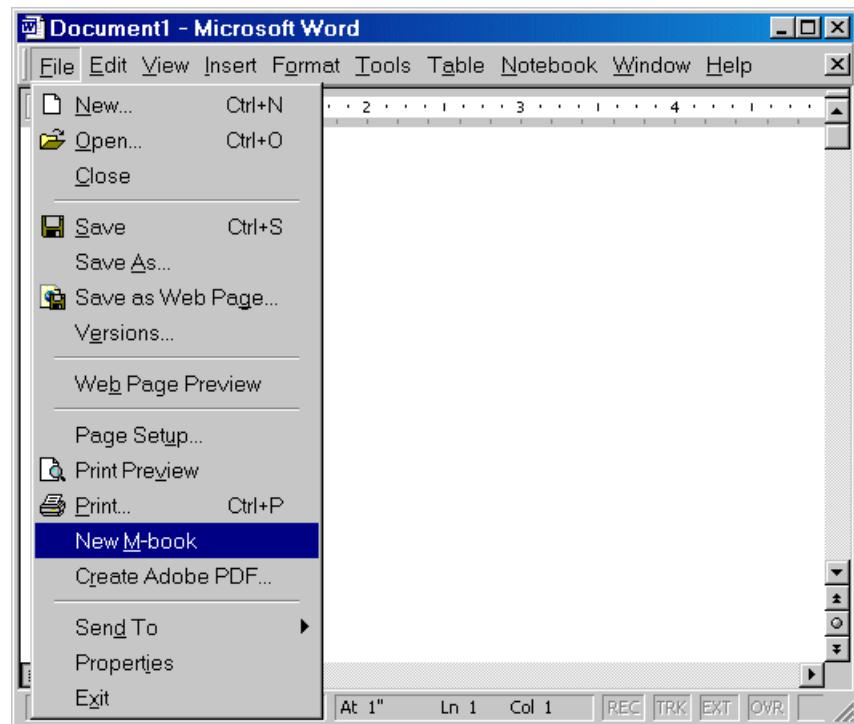
Creating an M-Book While Running Notebook
With Notebook running, you can create a new M-book by selecting **New M-book** from the Word **File** menu.

Opening an Existing M-Book
You can use the notebook command to open an existing M-book

```
notebook filename
```

where filename is the M-book you want to open, or you can simply double-click on an M-book file in a Windows file management tool, such as Explorer.

When you double-click on an M-book, Microsoft Word opens the M-book and starts MATLAB if it is not already running. Notebook adds the **Notebook** menu to the Word menu bar and adds **New M-book** to the **File** menu.

Converting a Word Document to an M-Book

To convert a Word document to an M-book, follow these steps:

1 Create a new M-book.

2 From the **Insert** menu, select the **File**.

3 Select the file you want to convert.

4 Click **OK**.

Entering MATLAB Commands in an M-Book

Note A good way to learn how to use Notebook is to open the sample M-book, `Readme.doc`, and try out the various techniques described in this section. You can find this file in the `$matlabroot/notebook/pc` directory.

You enter MATLAB commands in an M-book the same way you enter text in any other Word document. For example, you can enter the following text in a Word document. The example uses text in Courier Font but you can use any font:

```
Here is a sample M-book.

a = magic(3)
```

To execute the MATLAB `magic` command in this document, you must

- Define the command as an input cell
- Evaluate the input cell

MATLAB displays the output of the command in the Word document in an output cell.

Protecting the Integrity of Your Workspace

When you work on more than one M-book in a single word processing session, note that:

- Each M-book uses the same "copy" of MATLAB.
- All M-books share the same workspace.

If you use the same variable names in more than one M-book, data used in one M-book can be affected by another M-book. You can protect the integrity of your workspace by specifying the `clear` command as the first autoinit cell in the M-book.

Ensuring Data Consistency

An M-book can be thought of as a sequential record of a MATLAB session. When executed in order, from the first MATLAB command to the last, the M-book accurately reflects the relationships among these commands.

If, however, you change an input cell or output cell as you refine your M-book, Notebook does not automatically recalculate input cells that depend on either the contents or the results of the changed cells. As a result, the M-book may contain inconsistent data.

When working on an M-book, you might find it useful to select **Evaluate M-book** periodically to ensure that your M-book data is consistent. You could also use calc zones to isolate related commands in a section of the M-book. You can then use **Evaluate Calc Zone** to execute only those input cells contained in the calc zone.

Defining MATLAB Commands as Input Cells

To define a MATLAB command in a Word document as an input cell:

1 Type the command into the M-book as text. For example,

```
This is a sample M-book.

a = magic(3)
```

2 Position the cursor anywhere in the command and select **Notebook -> Define Input Cell** or press **Alt+D**. If the command is embedded in a line of text, use the mouse to select it. Notebook defines the MATLAB command as an input cell:

```
This is a sample M-book.

[a = magic(3)]
```

Note how Notebook changes the character font of the text in the input cell to a bold, dark green color and encloses it within *cell markers*. Cell markers are bold, gray brackets. They differ from the brackets used to enclose matrices by their size and weight. For information about changing these default formats, see "Modifying Styles in the M-Book Template" on page 9-17.

For information about defining other types of input cells, see

- "Defining Cell Groups" on page 9-7
- "Defining Autoinit Input Cells" on page 9-9
- "Defining Calc Zones" on page 9-9
- "Converting an Input Cell to Text" on page 9-10

For information about evaluating the input cells you define, see "Evaluating MATLAB Commands" on page 9-11.

Defining Cell Groups

You can collect several input cells into a single input cell. This is called a *cell group*. Because all the output from a cell group appears in a single output cell that Notebook places immediately after the group, cell groups are useful when several MATLAB commands are needed, such as, to fully define a graphic.

For example, if you define all the MATLAB commands that produce a graphic as a cell group and then evaluate the cell group, Notebook generates a single graphic that includes all the graphic components defined in the commands. If instead you define all the MATLAB commands that generate the graphic as separate input cells, evaluating the cells generates multiple graphic output cells.

See "Evaluating Cell Groups" on page 9-12 for information about evaluating a cell group. For information about undefining a cell group, see "Ungroup Cells" on page 9-31.

Creating a Cell Group

To create a cell group:

1 Use the mouse to select the input cells that are to make up the group.

2 Select **Notebook -> Group Cells** or press **Alt+G**.

Notebook converts the selected cells into a cell group and replaces cell markers with a single pair that surrounds the group:

```
This is a sample cell group.

[date
a = magic(3) ]
```

Note the following:

- A cell group cannot contain output cells. If the selection includes output cells, Notebook deletes them.
- A cell group cannot contain text. If the selection includes text, Notebook places the text after the cell group. However, if the text precedes the first input cell in the selection, Notebook leaves it where it is.
- If you select part or all of an output cell but not its input cell, Notebook includes the input cell in the cell group.

When you create a cell group, Notebook defines it as an input cell unless its first line is an autoinit cell, in which case Notebook defines the group as an autoinit cell.

Defining Autoinit Input Cells

You can use *autoinit cells* to specify MATLAB commands to be automatically evaluated each time an M-book is opened. This is a quick and easy way to initialize the workspace. *Autoinit cells* are simply input cells with the following additional characteristics:

- Notebook evaluates the autoinit cells when it opens the M-book.
- Notebook displays the commands in autoinit cells using dark blue characters.

Autoinit cells are otherwise identical to input cells.

Creating an Autoinit Cell

You can create an autoinit cell in two ways:

- Enter the MATLAB command as text, then convert the command to an autoinit cell by selecting **Notebook -> Define AutoInit Cell**.
- If you already entered the MATLAB command as an input cell, you can convert the input cell to an autoinit cell. Either select the input cell or position the cursor in the cell, then select **Notebook -> Define AutoInit Cell**.

See "Evaluating MATLAB Commands" on page 9-11 for information about evaluating autoinit cells.

Defining Calc Zones

You can partition an M-book into self-contained sections, called *calc zones*. A calc zone is a contiguous block of text, input cells, and output cells. Notebook inserts Microsoft Word section breaks before and after the section to define the calc zone. The section break indicators include bold, gray brackets to distinguish them from standard Word section breaks.

You can use calc zones to prepare problem sets, making each problem a separate calc zone that can be created and tested on its own. An M-book can contain any number of calc zones.

Note Using calc zones does not affect the scope of the variables in an M-book. Variables used in one calc zone are accessible to all calc zones.

Creating a Calc Zone

After you create the text and cells you want to include in the calc zone, you define the calc zone by following these steps:

1 Select the input cells and text to be included in the calc zone.

2 Select **Notebook -> Define Calc Zone**.

Note You must select an input cell and its output cell in their entirety to include them in the calc zone.

See "Evaluating a Calc Zone" on page 9-13 for information about evaluating a calc zone.

Converting an Input Cell to Text

To convert an input cell (or an autoinit cell or a cell group) to text:

1 Select the input cell with the mouse or position the cursor in the input cell.

2 Select **Notebook -> Undefine Cells** or press **Alt+U**.

When Notebook converts the cell to text, it reformats the cell contents according to the Microsoft Word Normal style. For more information about M-book styles, see "Modifying Styles in the M-Book Template" on page 9-17. When you convert an input cell to text, Notebook also converts the corresponding output cell to text.

Evaluating MATLAB Commands

After you define a MATLAB command as an input cell, or as an autoinit cell, you can evaluate it in your M-book. Use the following steps to define and evaluate a MATLAB command:

1 Type the command into the M-book as text. For example:

```
This is a sample M-book

a = magic(3)
```

2 Position the cursor anywhere in the command. If the command is embedded in a line of text, use the mouse to select it. Then select **Notebook -> Define Input Cell** or press **Alt+D**.

Notebook defines the MATLAB command as an input cell. For example:

```
This is a sample M-book

[a = magic(3)]
```

3 Specify the input cell to be evaluated by selecting it with the mouse or by placing the cursor in it. Then select **Notebook -> Evaluate Cell** or press **Ctrl+Enter**.

Notebook evaluates the input cell and displays the results in a output cell immediately following the input cell. If there is already an output cell, Notebook replaces its contents, wherever it is in the M-book. For example:

```
This is a sample M-book.

[a = magic(3) ]

[a =
     8     1     6
     3     5     7
     4     9     2 ]
```

The text in the output cell is blue and is enclosed within cell markers. Cell markers are bold, gray brackets. They differ from the brackets used to enclose matrices by their size and weight. Error messages appear in red. For

information about changing these default formats, see "Modifying Styles in the M-Book Template" on page 9-17.

For more information about evaluating MATLAB commands in an M-book, see

- "Evaluating Cell Groups" on page 9-12
- "Evaluating a Range of Input Cells" on page 9-13
- "Evaluating a Calc Zone" on page 9-13
- "Evaluating an Entire M-Book" on page 9-14
- "Using a Loop to Evaluate Input Cells Repeatedly" on page 9-14.
- "Converting Output Cells to Text" on page 9-15
- "Deleting Output Cells" on page 9-16

Evaluating Cell Groups

You evaluate a cell group the same way you evaluate an input cell (because a cell group is an input cell):

1 Position the cursor anywhere in the cell or in its output cell.

2 Select **Notebook -> Evaluate Cell** or press **Ctrl+Enter**.

For information about creating a cell group, see "Defining Cell Groups" on page 9-7.
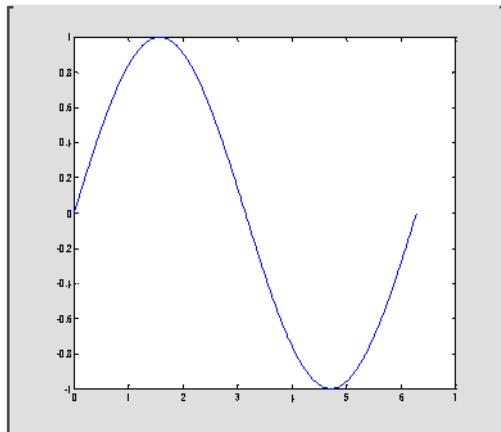
When MATLAB evaluates a cell group, the output for all commands in the group appears in a single output cell. By default, Notebook places the output cell immediately after the cell group the first time the cell group is evaluated. If you evaluate a cell group with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Note Text or numeric output always comes first, regardless of the order of the commands in the group.

The illustration shows a cell group and the figure created when you evaluate the cell group.

This is a sample M-book with a cell group.

```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y) ]
```


Evaluating a Range of Input Cells

To evaluate more than one MATLAB command contained in different but contiguous input cells:

1 Select the range of cells that includes the input cells you want to evaluate. You can include text that surrounds input cells in your selection.

2 Select **Notebook -> Evaluate Cell** or press **Ctrl+Enter**.

Notebook evaluates each input cell in the selection, inserting new output cells or replacing existing ones.

Evaluating a Calc Zone

To evaluate a calc zone:

1 Position the cursor anywhere in the calc zone.

2 Select **Notebook -> Evaluate Calc Zone** or press **Alt+Enter**.

9-13

For information about creating a calc zone, see "Defining Calc Zones" on page 9-9.

By default, Notebook places the output cell immediately after the calc zone the first time the calc zone is evaluated. If you evaluate a calc zone with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Evaluating an Entire M-Book

To evaluate the entire M-book, either select **Notebook -> Evaluate M-book** or press **Alt+R**.

Notebook begins at the top of the M-book regardless of the cursor position and evaluates each input cell in the M-book. As it evaluates the M-book, Notebook inserts new output cells or replaces existing output cells.

Controlling Execution of Multiple Commands

When you evaluate an entire M-book, and an error occurs, evaluation continues. If you want to stop evaluation if an error occurs, follow this procedure:

1 Select **Notebook -> Notebook Options**.
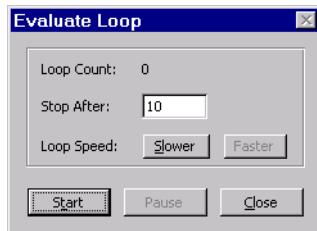
The **Notebook Options** dialog box opens.

2 Select the **Stop evaluating on error** check box and click **OK**.

Using a Loop to Evaluate Input Cells Repeatedly

To evaluate a sequence of MATLAB commands repeatedly:

1 Use the mouse to select the input cells, including any text or output cells located between them.

2 Select **Notebook -> Evaluate Loop** or press **Alt+L**. Notebook displays the **Evaluate Loop** dialog box.

3 Enter the number of times you want MATLAB to evaluate the selected
commands in the **Stop After** field, then click **Start**. The button changes to
Stop. Notebook begins evaluating the commands and indicates the number
of completed iterations in the **Loop Count** field.

You can increase or decrease the delay at the end of each iteration by clicking
Slower or **Faster**. Slower increases the delay. Faster decreases the delay.

To suspend evaluation of the commands, click **Pause**. The button changes to
Resume. Click **Resume** to continue evaluation.

To stop processing the commands, click **Stop**. To close the **Evaluate Loop**
dialog box, click **Close**.

Converting Output Cells to Text

You can convert an output cell to text by undefining cells. If the output is
numeric or textual, Notebook removes the cell markers and converts the cell
contents to text according to the Microsoft Word Normal style. If the output is
graphical, Notebook removes the cell markers and dissociates the graphic from
its input cell, but does not alter its contents.

Note Undefining an output cell does not affect the associated input cell.

To undefine an output cell:

1 Select the output cell you want to undefine.

2 Select **Notebook -> Undefine Cells** or press **Alt+U**.

Deleting Output Cells

To delete output cells:

1 Select an output cell, using the mouse, or place the cursor in the output cell.

2 Select **Notebook -> Purge Selected Output Cells** or press **Alt+P**.

If you select a range of cells, Notebook deletes all the output cells in the selected range, but any associate input cells remain intact.

Printing and Formatting an M-Book

This section describes

- "Printing an M-Book" on page 9-17
- "Modifying Styles in the M-Book Template" on page 9-17
- "Choosing Loose or Compact Format" on page 9-18
- "Controlling Numeric Output Format" on page 9-19
- "Controlling Graphic Output" on page 9-19

Printing an M-Book

You can print all or part of an M-book by selecting **File -> Print**. Word follows these rules when printing M-book cells and graphics:

- Cell markers are not printed.
- Input cells, autoinit cells, and output cells (including error messages) are printed according to their defined styles. If you prefer to print these cells using black type instead of colors or shades of gray, you can modify the styles.

Modifying Styles in the M-Book Template

You can control the appearance of the text in your M-book by modifying the predefined styles stored in the M-book template. These styles control the appearance of text and cells. By default, M-books use the Word Normal style for all other text.

For example, if you print an M-book on a color printer, input cells appear dark green, output and autoinit cells appear dark blue, and error messages appear red. If you print the M-book on a grayscale printer, these cells appear as shades of gray. To print these cells using black type, you need to modify the color of the Input, Output, AutoInit, and Error styles in the M-book template.

The table below describes the default styles used by Notebook. If you modify styles, you can use the information in the tables below to help you return the styles to their original settings. For general information about using styles in Word documents, see the Word documentation.

Style	Font	Size	Weight	Color
Normal	Times New Roman	10 points		Black
AutoInit	Courier New	10 points	Bold	Dark blue
Error	Courier New	10 points	Bold	Red
Input	Courier New	10 points	Bold	Dark green
Output	Courier New	10 points		Blue

When you change a style, Word applies the change to all characters in the M-book that use that style and gives you the option to change the template. Be cautious about making changes to the template. If you choose to apply the changes to the template, you will affect all new M-books you create using the template. See the Word documentation for more information.

Choosing Loose or Compact Format

You can specify whether a blank line appears between the input and output cells by selecting the loose or compact format:

1 Select **Notebook -> Notebook Options**.

2 In the **Notebook Options** dialog box, select either **Loose** or **Compact**. Loose format adds an empty line. Compact format does not.
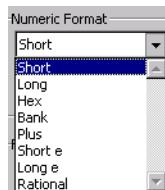
3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

Controlling Numeric Output Format

To change how Notebook displays numeric output:

1 Select **Notebook -> Notebook Options**.

2 In the **Notebook Options** dialog box, select a format from the **Numeric Format** list. These settings correspond to the choices available with the MATLAB `format` command.

3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

Controlling Graphic Output
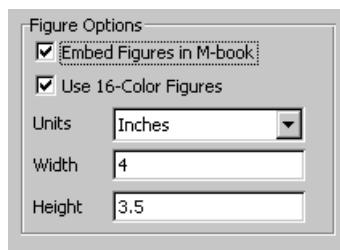
This section describes how to control several aspects of the graphic output produced by MATLAB commands in an M-book, including

- "Embedding Graphic Output in the M-Book" on page 9-20
- "Suppressing Graphic Output for Individual Input Cells" on page 9-20
- "Sizing Graphic Output" on page 9-21
- "Cropping Graphic Output" on page 9-21
- "Adding White Space Around Graphic Output" on page 9-22
- "Specifying Color Mode" on page 9-22

Embedding Graphic Output in the M-Book

By default, graphic output is embedded in an M-book. To display graphic output in a separate figure window:

1 Select **Notebook -> Notebook Options**.

2 In the **Notebook Options** dialog box, clear the **Embed Figures in M-book** check box.

3 Click **OK**.

Note Embedded figures do not include Handle Graphics objects generated by the `uicontrol` and `uimenu` functions.

Notebook determines whether to embed a figure in the M-book by examining the value of the figure object's `Visible` property. If the value of the property is `off`, Notebook embeds the figure. If the value of this property is `on`, all graphic output is directed to the current figure window.

Suppressing Graphic Output for Individual Input Cells

If an input or autoinit cell generates figure output that you want to suppress:

1 Place the cursor in the input cell.

2 Select **Notebook -> Toggle Graph Output for Cell**.

Notebook suppresses graphic output from the cell, inserting the string (no graph) after the input cell.

To allow graphic output for a cell, repeat the procedure. Notebook removes the (no graph) marker and allows graphic output from the cell.

Note **Toggle Graph Output for Cell** overrides the **Embed Figures in M-book** option, if that option is set.

Sizing Graphic Output

To set the default size of embedded graphics in an M-book:

1 Select **Notebook -> Notebook Options**.

2 In the **Notebook Options** dialog box, use the **Units**, **Height**, and **Width** fields to set the size of graphics generated by the M-book.

3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for graphic output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

You change the size of an existing embedded figure by selecting the figure, clicking the left mouse button anywhere in the figure, and dragging the resize handles of the figure. If you resize an embedded figure using its resize handles and then regenerate the figure, its size reverts to its original size.

Cropping Graphic Output

To crop an embedded figure to cut off areas you do not want to show:

1 Select the graphic by clicking the left mouse button anywhere in the figure.

2 Hold down the **Shift** key.

3 Drag a sizing handle toward the center of the graphic.

Adding White Space Around Graphic Output

You can add white space around an embedded figure by moving the boundaries of a graphic outward. Select the graphic, then hold down the **Shift** key and drag a sizing handle away from the graphic.

Specifying Color Mode

If you print graphic output that includes surfaces or patches, the output uses 16-color mode by default. To use 256-color mode:

1 Select **Notebook -> Notebook Options**.

2 Clear the **Use 16-Color Figures** check box in the **Notebook Options** dialog box.

3 Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for graphic output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

Configuring Notebook

After you install Notebook but before you begin using it, you must configure it. (Notebook is installed as part of the MATLAB installation process. For more information, see the MATLAB installation documentation for your platform.)

In Word versions for Office 2000 or higher, before configuring Notebook, you must specify that Word can use the Notebook macros. Do either of the following:

- Set the macro security level to medium. In Word, select **Tools -> Macros -> Security**, and in the resulting dialog box, choose **Medium**.

- After launching Notebook, when Word first opens, a security warning dialog box appears. In the dialog box, select **Always trust macros from this source**. This allows you to use Notebook, but still maintain a high security level for other macros you use in Word.

To configure Notebook:

1 In the MATLAB command window, type

```
notebook -setup
```

Notebook prompts you to specify which version of Microsoft Word you are using.

```
Welcome to the utility for setting up the MATLAB Notebook for
interfacing MATLAB to Microsoft Word

Choose your version of Microsoft Word:
[1] Microsoft Word 97
[2] Microsoft Word 2000
[3] Microsoft Word 2002 (XP)
[4] Exit, making no changes

Microsoft Word Version:
```

2 Type the number that corresponds to your version. For example, type 3 if you have Microsoft Word 2000 XP.

Notebook performs the setup. If Notebook cannot find all of the necessary files, it will prompt you to specify the locations of the files, including the

Microsoft Word executable (`winword.exe`) and the template file (`normal.dot`).

When setup is complete, the following message appears:

```
Notebook setup is complete.
```

Notebook Feature Reference

This section provides reference information about each of the Notebook features, listed alphabetically. To use these features, select them from the **Notebook** menu:

Bring MATLAB to Front

Bring MATLAB to Front brings the MATLAB Command Window to the foreground.

Define Autoinit Cell

Define AutoInit Cell creates an autoinit cell by converting the current paragraph, selected text, or input cell. An autoinit cell is an input cell that is automatically evaluated whenever you open an M-book.

Result

If you select this feature while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an autoinit cell. If you select this feature while

text is selected, Notebook converts the text to an autoinit cell. If you select this feature while the cursor is in an input cell, Notebook converts the input cell to an autoinit cell.

Format

Notebook formats the autoinit cell using the AutoInit style, defined as bold, dark blue, 10-point Courier New.

See Also

For more information about autoinit cells, see "Defining Autoinit Input Cells" on page 9-9.

Define Calc Zone

Define Calc Zone defines the selected text, input cells, and output cells as a calc zone. A calc zone is a contiguous block of related text, input cells, and output cells that describes a specific operation or problem.

Result

Notebook defines a calc zone as a Word document section, placing section breaks before and after the calc zone. However, Word does not display section breaks at the beginning or end of a document.

See Also

For information about evaluating calc zones, see "Evaluating a Calc Zone" on page 9-13. For more information about document sections, see the Microsoft Word documentation.

Define Input Cell

Define Input Cell creates an input cell by converting the current paragraph, selected text, or autoinit cell. An input cell contains a MATLAB command.

Result

If you select this feature while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an input cell. If you select this feature while text is selected, Notebook converts the text to an input cell.

If you select this feature while the cursor is in an autoinit cell, Notebook converts the autoinit cell to an input cell.

Format

Notebook encloses the text in cell markers and formats the cell using the Input style, defined as bold, dark green, 10-point Courier New.

See Also

For more information about creating input cells, see "Defining MATLAB Commands as Input Cells" on page 9-7. For information about evaluating input cells, see "Evaluating MATLAB Commands" on page 9-11.

Evaluate Calc Zone

Evaluate Calc Zone sends the input cells in the current calc zone to MATLAB to be evaluated. A calc zone is a contiguous block of related text, input cells, and output cells that describes a specific operation or problem.

The current calc zone is the Word section that contains the cursor.

Result

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see "Evaluating a Calc Zone" on page 9-13.

Evaluate Cell

Evaluate Cell sends the current input cell or cell group to MATLAB to be evaluated. An input cell contains a MATLAB command. A cell group is a single, multiline input cell that contains more than one MATLAB command. Notebook displays the output or an error message in an output cell.

Result

If you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book. If you evaluate a cell group, all output for the cell appears in a single output cell.

An input cell or cell group is the current input cell or cell group if

• The cursor is in the input cell or cell group.
• The cursor is at the end of the line that contains the closing cell marker for the input cell or cell group.
• The cursor is in the output cell for the input cell or cell group.
• The input cell or cell group is selected.

Note Evaluating a cell that involves a lengthy operation may cause a time-out. If this happens, Word displays a time-out message and asks whether you want to continue waiting for a response or terminate the request. If you choose to continue, Word resets the time-out value and continues waiting for a response. Word sets the time-out value; you cannot change it.

See Also

For more information, see "Evaluating MATLAB Commands" on page 9-11. For information about evaluating the entire M-book, see "Evaluating an Entire M-Book" on page 9-14.

Evaluate Loop

Evaluate Loop evaluates the selected input cells repeatedly.

For more information, see "Using a Loop to Evaluate Input Cells Repeatedly" on page 9-14.

Evaluate M-Book

Evaluate M-book evaluates the entire M-book, sending all input cells to MATLAB to be evaluated. Notebook begins at the top of the M-book regardless of the cursor position.

Result

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see "Evaluating an Entire M-Book" on page 9-14.

Group Cells

Group Cells converts the input cells in the selection into a single multiline input cell called a cell group. You evaluate a cell group using **Evaluate Cell**. When you evaluate a cell group, all of its output follows the group and appears in a single output cell.

Result

If you include text in the selection, Notebook moves it after the cell group. However, if text precedes the first input cell in the group, the text will remain before the group.

If you include output cells in the selection, Notebook deletes them. If you select all or part of an output cell before selecting this feature, Notebook includes its input cell in the cell group.

If the first line in the cell group is an autoinit cell, the entire group acts as a sequence of autoinit cells. Otherwise, the group acts as a sequence of input cells. You can convert an entire cell group to an autoinit cell by using **Define AutoInit Cell**.

See Also

For more information, see "Defining Cell Groups" on page 9-7. For information about converting a cell group to individual input cells, see the description of the "Ungroup Cells" on page 9-31.

Hide Cell Markers

Hide Cell Markers hides cell markers in the M-book.

When you select this feature, it changes to **Show Cell Markers**.

Note Notebook does not print cell markers whether you choose to hide them or show them on the screen.

Notebook Options

Notebook Options allows you to examine and modify display options for numeric and graphic output.

See Also

See "Printing and Formatting an M-Book" on page 9-17 for more information.

Purge Selected Output Cells

Purge Selected Output Cells deletes all output cells from the current selection.

See Also

For more information, see "Deleting Output Cells" on page 9-16.

Toggle Graph Output for Cell

Toggle Graph Output for Cell suppresses or allows graphic output from an input cell.

If an input or autoinit cell generates figure output that you want to suppress, place the cursor in the input cell and choose this feature. The string `(no graph)`

will be placed after the input cell to indicate that graph output for that cell will be suppressed.

To allow graphic output for that cell, place the cursor inside the input cell and choose **Toggle Graph Output for Cell** again. The (no graph) marker will be removed. This feature overrides the **Embed Graphic Output** in the M-book option, if that option is set in the **Notebook Options** dialog box.

See Also

See "Embedding Graphic Output in the M-Book" on page 9-20 and "Suppressing Graphic Output for Individual Input Cells" on page 9-20 for more information.

Undefine Cells

Undefine Cells converts the selected cells to text. If no cells are selected but the cursor is in a cell, Notebook undefines that cell. Notebook removes the cell markers and reformats the cell according to the Normal style.

If you undefine an input cell, Notebook automatically undefines its output cell. However, if you undefine an output cell, Notebook does not undefine its input cell. If you undefine an output cell containing an embedded graphic, the graphic remains in the M-book but is no longer associated with an input cell.

See Also

For information about the Normal style, see "Modifying Styles in the M-Book Template" on page 9-17. For information about deleting output cells, see the description of the "Purge Selected Output Cells" on page 9-30.

Ungroup Cells

Ungroup Cells converts the current cell group into a sequence of individual input cells or autoinit cells. If the cell group is an input cell, Notebook converts the cell group to input cells. If the cell group is an autoinit cell, Notebook converts the cell group to autoinit cells. Notebook deletes the output cell for the cell group.

A cell group is the current cell group if

• The cursor is in the cell group.

- The cursor is at the end of a line that contains the closing cell marker for the cell group.
- The cursor is in the output cell for the cell group.
- The cell group is selected.

See Also

For information about creating cell groups, see the description of the "Defining Cell Groups" on page 9-7.

Mathematics

MATLAB provides many functions for performing mathematical operations and analyzing data. The following list summarizes the contents of this collection:

10

Matrices and Linear Algebra

This chapter describes basic matrix operations in MATLAB and explains their use in solving problems. It includes:

Function Summary

The linear algebra functions are located in the MATLAB `matfun` directory.

Function Summary

Category	Function	Description
Matrix analysis	norm	Matrix or vector norm.
	normest	Estimate the matrix 2-norm.
	rank	Matrix rank.
	det	Determinant.
	trace	Sum of diagonal elements.
	null	Null space.
	orth	Orthogonalization.
	rref	Reduced row echelon form.
	subspace	Angle between two subspaces.
Linear equations	\ and /	Linear equation solution.
	inv	Matrix inverse.
	cond	Condition number for inversion.
	condest	1-norm condition number estimate.
	chol	Cholesky factorization.
	cholinc	Incomplete Cholesky factorization.
	lu	LU factorization.
	luinc	Incomplete LU factorization.
	qr	Orthogonal-triangular decomposition.
	lsqnonneg	Nonnegative least-squares.

Function Summary (Continued)

Category	Function	Description
	pinv	Pseudoinverse.
	lscov	Least squares with known covariance.
Eigenvalues and singular values	eig	Eigenvalues and eigenvectors.
	svd	Singular value decomposition.
	eigs	A few eigenvalues.
	svds	A few singular values.
	poly	Characteristic polynomial.
	polyeig	Polynomial eigenvalue problem.
	condeig	Condition number for eigenvalues.
	hess	Hessenberg form.
	qz	QZ factorization.
	schur	Schur decomposition.
Matrix functions	expm	Matrix exponential.
	logm	Matrix logarithm.
	sqrtm	Matrix square root.
	funm	Evaluate general matrix function.

Matrices in MATLAB

A *matrix* is a two-dimensional array of real or complex numbers. *Linear algebra* defines many matrix operations that are directly supported by MATLAB. Linear algebra includes matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.

This section describes these matrix operations:

- Creation
- Addition and subtraction
- Vector products
- Matrix multiplication

It also describes the MATLAB functions you use to produce:

- An identity matrix
- The Knonecker Tensor product of two matrices
- Vector and matrix norms

Creation

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields. (The optional Symbolic Math Toolbox extends the capabilities of MATLAB to operations on various types of nonnumeric matrices.)

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric.

```
A = pascal(3)

A =
        1    1    1
        1    2    3
        1    3    6
```

The second example is not symmetric.

```
B = magic(3)

B =
      8     1     6
      3     5     7
      4     9     2
```

Another example is a 3-by-2 rectangular matrix of random integers.

```
C = fix(10*rand(3,2))

C =
      9     4
      2     8
      6     7
```

A c*olumn vector* is an *m*-by-1 matrix, a *row vector* is a 1-by-*n* matrix and a *scalar* is a 1-by-1 matrix. The statements

```
u = [3; 1; 4]

v = [2 0 -1]

s = 7
```

produce a column vector, a row vector, and a scalar.

```
u =
      3
      1
      4

v =
      2     0    -1

s =
      7
```

Addition and Subtraction

Addition and subtraction of matrices is defined just as it is for arrays, element-by-element. Adding A to B and then subtracting A from the result recovers B.

```
A = pascal(3);
B = magic(3);
X = A + B

X =
      9      2      7
      4      7     10
      5     12      8

Y = X - A

Y =
      8      1      6
      3      5      7
      4      9      2
```

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results.

```
C = fix(10*rand(3,2))
X = A + C
Error using ==> +
Matrix dimensions must agree.

w = v + s

w =
      9      7      6
```

Vector Productsand Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the *inner* product, or a matrix, the *outer* product.

```
u = [3; 1; 4];
```

```
v = [2 0 -1];
x = v*u

x =
      2

X = u*v

X =
      6      0     -3
      2      0     -1
      8      0     -4
```

For real matrices, the *transpose* operation interchanges a_{ij} and a_{ji}. MATLAB uses the apostrophe (or single quote) to denote transpose. Our example matrix A is *symmetric*, so A' is equal to A. But B is not symmetric.

```
B = magic(3);
X = B'

X =
      8      3      4
      1      5      9
      6      7      2
```

Transposition turns a row vector into a column vector.

```
x = v'

x =
      2
      0
     -1
```

If x and y are both real column vectors, the product x*y is not defined, but the two products

```
x'*y
```

and

```
y'*x
```

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

For a complex vector or matrix, z, the quantity z' denotes the *complex conjugate transpose*, where the sign of the complex part of each element is reversed. The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by z.'. So if

```
z = [1+2i 3+4i]
```

then z' is

```
1-2i
3-4i
```

while z.' is

```
1+2i
3+4i
```

For complex vectors, the two scalar products x'*y and y'*x are complex conjugates of each other and the scalar product x'*x of a complex vector with itself is real.

Matrix Multiplication

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product $C = AB$ is defined when the column dimension of A is equal to the row dimension of B, or when one of them is a scalar. If A is m-by-p and B is p-by-n, their product C is m-by-n. The product can actually be defined using MATLAB for loops, colon notation, and vector dot products.

```
A = pascal(3);
B = magic(3);
m = 3; n = 3;
for i = 1:m
    for j = 1:n
        C(i,j) = A(i,:)*B(:,j);
    end
end
```

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; *AB* is usually not equal to *BA*.

```
X = A*B

X =
      15     15     15
      26     38     26
      41     70     39

Y = B*A

Y =
      15     28     47
      15     34     60
      15     28     43
```

A matrix can be multiplied on the right by a column vector and on the left by a row vector.

```
u = [3; 1; 4];
x = A*u

x =
       8
      17
      30

v = [2 0 -1];
y = v*B

y =
      12     -7     10
```

Rectangular matrix multiplications must satisfy the dimension compatibility conditions.

```
C = fix(10*rand(3,2));
X = A*C
```

```
X =
      17     19
      31     41
      51     70

Y = C*A

Error using ==> *
Inner matrix dimensions must agree.
```

Anything can be multiplied by a scalar.

```
s = 7;
w = s*v

w =
      14      0     -7
```

The Identity Matrix

Generally accepted mathematical notation uses the capital letter I to denote *identity* matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that $AI = A$ and $IA = A$ whenever the dimensions are compatible. The original version of MATLAB could not use I for this purpose because it did not distinguish between upper and lowercase letters and i already served double duty as a subscript and as the complex unit. So an English language pun was introduced. The function

```
eye(m,n)
```

returns an *m*-by-*n* rectangular identity matrix and eye(n) returns an *n*-by-*n* square identity matrix.

The Kronecker Tensor Product

The Kronecker product, kron(X,Y), of two matrices is the larger matrix formed from all possible products of the elements of X with those of Y. If X is *m*-by-*n* and Y is *p*-by-*q*, then kron(X,Y) is *mp*-by-*nq*. The elements are arranged in the following order.

```
[X(1,1)*Y  X(1,2)*Y  . . .  X(1,n)*Y
                       . . .
 X(m,1)*Y  X(m,2)*Y  . . .  X(m,n)*Y]
```

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if X is the 2-by-2 matrix

```
X =
        1       2
        3       4
```

and I = eye(2,2) is the 2-by-2 identity matrix, then the two matrices

```
kron(X,I)
```

and

```
kron(I,X)
```

are

```
        1       0       2       0
        0       1       0       2
        3       0       4       0
        0       3       0       4
```

and

```
        1       2       0       0
        3       4       0       0
        0       0       1       2
        0       0       3       4
```

Vector and Matrix Norms

The *p*-norm of a vector *x*

$$\|x\|_p = \left(\sum |x_i|^p \right)^{1/p}$$

is computed by norm(x,p). This is defined by any value of $p > 1$, but the most common values of p are 1, 2, and ∞. The default value is $p = 2$, which corresponds to *Euclidean length*.

```
v = [2 0 -1];
[norm(v,1) norm(v) norm(v,inf)]

ans =
      3.0000    2.2361    2.0000
```

The *p*-norm of a matrix *A*,

$$\|A\|_p = \max_x \frac{\|Ax\|_p}{\|x\|_p}$$

can be computed for $p = 1, 2$, and ∞ by norm(A,p). Again, the default value is
$p = 2$.

```
C = fix(10*rand(3,2));
[norm(C,1) norm(C) norm(C,inf)]

ans =
      19.0000   14.8015   13.0000
```

Solving Linear Systems of Equations

This section describes:

- Computational considerations
- The general solution to a system

It also discusses particular solutions to:

- Square systems
- Overdetermined systems
- Underdetermined systems

Computational Considerations

One of the most important problems in technical computing is the solution of simultaneous linear equations. In matrix notation, this problem can be stated as follows.

Given two matrices A and B, does there exist a unique matrix X so that $AX = B$ or $XA = B$?

It is instructive to consider a 1-by-1 example.

Does the equation

$$7x \ = \ 21$$

have a unique solution ?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by *division*.

$$x \ = \ 21/7 \ = \ 3$$

The solution is *not* ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857\ldots$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, *slash*, */*, and

backslash, \, are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix.

X = A\B   Denotes the solution to the matrix equation $AX = B$.

X = B/A   Denotes the solution to the matrix equation $XA = B$.

You can think of "dividing" both sides of the equation $AX = B$ or $XA = B$ by A. The coefficient matrix A is always in the "denominator."

The dimension compatibility conditions for X = A\B require the two matrices A and B to have the same number of rows. The solution X then has the same number of columns as B and its row dimension is equal to the column dimension of A. For X = B/A, the roles of rows and columns are interchanged.

In practice, linear equations of the form $AX = B$ occur more frequently than those of the form $XA = B$. Consequently, backslash is used far more frequently than slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity

    (B/A)' = (A'\B')

The coefficient matrix A need not be square. If A is m-by-n, there are three cases.

$m = n$ Square system. Seek an exact solution.

$m > n$ Overdetermined system. Find a least squares solution.

m < n   Underdetermined system. Find a basic solution with at most m nonzero components.

The backslash operator employs different algorithms to handle different kinds of coefficient matrices. The various cases, which are diagnosed automatically by examining the coefficient matrix, include:

• Permutations of triangular matrices

• Symmetric, positive definite matrices

• Square, nonsingular matrices

• Rectangular, overdetermined systems

• Rectangular, underdetermined systems

General Solution

The general solution to a system of linear equations $AX = b$ describes all possible solutions. You can find the general solution by:

1 Solving the corresponding homogeneous system $AX = 0$. Do this using the null command, by typing null(A). This returns a basis for the solution space to $AX = 0$. Any solution is a linear combination of basis vectors.

2 Finding a particular solution to the non-homogeneous system $AX = b$.

You can then write any solution to $AX = b$ as the sum of the particular solution to $AX = b$, from step 2, plus a linear combination of the basis vectors from step 1.

The rest of this section describes how to use MATLAB to find a particular solution to $AX = b$, as in step 2.

Square Systems

The most common situation involves a square coefficient matrix A and a single right-hand side column vector b.

Nonsingular Coefficient Matrix

If the matrix A is nonsingular, the solution, x = A\b, is then the same size as b. For example,

```
A = pascal(3);
u = [3; 1; 4];
x = A\u

x =
      10
     -12
       5
```

It can be confirmed that A*x is exactly equal to u.

If A and B are square and the same size, then X = A\B is also that size.

```
B = magic(3);
X = A\B
```

```
X =
        19      -3      -1
       -17       4      13
         6       0      -6
```

It can be confirmed that A*X is exactly equal to B.

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be pascal(3), which has a determinant equal to one. A later section considers the effects of roundoff error inherent in more realistic computations.

Singular Coefficient Matrix

A square matrix *A* is *singular* if it does not have linearly independent columns. If *A* is singular, the solution to *AX = B* either does not exist, or is not unique. The backslash operator, A\B, issues a warning if A is nearly singular and raises an error condition if it detects exact singularity.

If *A* is singular and *AX = b* has a solution, you can find a particular solution that is not unique, by typing

```
P = pinv(A)*b
```

P is a pseudoinverse of *A*. If *AX = b* does not have an exact solution, pinv(A) returns a least-squares solution.

For example,

```
A = [ 1      3      7
     -1      4      4
      1     10     18 ]
```

is singular, as you can verify by typing

```
det(A)

ans =
    0
```

Note For information about using `pinv` to solve systems with rectangular coefficient matrices, see "Pseudoinverses" on page 10-23.

Exact Solutions. For b =[5;2;12], the equation $AX = b$ has an exact solution, given by

```
pinv(A)*b

ans =
     0.3850
    -0.1103
     0.7066
```

You can verify that `pinv(A)*b` is an exact solution by typing

```
A*pinv(A)*b

ans =
     5.0000
     2.0000
    12.0000
```

Least Squares Solutions. On the other hand, if b = [3;6;0], then $AX = b$ does not have an exact solution. In this case, `pinv(A)*b` returns a least squares solution. If you type

```
A*pinv(A)*b

ans =
    -1.0000
     4.0000
     2.0000
```

you do not get back the original vector b.

You can determine whether $AX = b$ has an exact solution by finding the row reduced echelon form of the augmented matrix [A b]. To do so for this example, type

```
rref([A b])
```

```
ans =
    1.0000          0    2.2857          0
         0     1.0000    1.5714          0
         0          0         0     1.0000
```

Since the bottom row contains all zeros except for the last entry, the equation does not have a solution. In this case, `pinv(A)` returns a least-squares solution.

Overdetermined Systems

Overdetermined systems of simultaneous linear equations are often encountered in various kinds of curve fitting to experimental data. Here is a hypothetical example. A quantity y is measured at several different values of time, t, to produce the following observations.

t	y
0.0	0.82
0.3	0.72
0.8	0.63
1.1	0.60
1.6	0.55
2.3	0.50

Enter the data into MATLAB with the statements

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]';
```

Try modeling the data with a decaying exponential function.

$$y(t) \approx c_1 + c_2 e^{-t}$$

This equation says that the vector y should be approximated by a linear combination of two other vectors, one the constant vector containing all ones and the other the vector with components e^{-t}. The unknown coefficients, c_1 and c_2, can be computed by doing a *least squares fit*, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by the 6-by-2 matrix.

```
E = [ones(size(t)) exp(-t)]

E =
      1.0000    1.0000
      1.0000    0.7408
      1.0000    0.4493
      1.0000    0.3329
      1.0000    0.2019
      1.0000    0.1003
```

Use the backslash operator to get the least squares solution.

```
c = E\y

c =
      0.4760
      0.3413
```
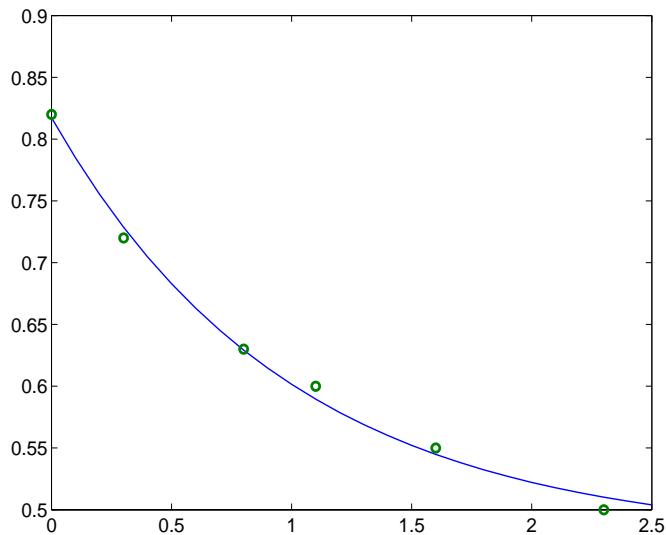
In other words, the least squares fit to the data is

$$y(t) \approx 0.4760 + 0.3413 \ e^{-t}$$

The following statements evaluate the model at regularly spaced increments in *t*, and then plot the result, together with the original data.

```
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
```

You can see that E*c is not exactly equal to y, but that the difference might well be less than measurement errors in the original data.

A rectangular matrix *A* is *rank deficient* if it does not have linearly independent columns. If *A* is rank deficient, the least squares solution to *AX = B* is not unique. The backslash operator, A\B, issues a warning if A is rank deficient and produces a least squares solution that has at most rank(A) nonzeros.

Underdetermined Systems

Underdetermined linear systems involve more unknowns than equations. When they are accompanied by additional constraints, they are the purview of *linear programming*. By itself, the backslash operator deals only with the unconstrained system. The solution is never unique. MATLAB finds a *basic* solution, which has at most *m* nonzero components, but even this may not be unique. The particular solution actually computed is determined by the QR factorization with column pivoting (see a later section on the QR factorization).

Here is a small, random example.

```
R = fix(10*rand(2,4))
```

```
R =
        6       8       7       3
        3       5       4       1

b = fix(10*rand(2,1))
b =
        1
        2
```

The linear system $Rx = b$ involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to use the format command to display the solution in *rational* format. The particular solution is obtained with

```
format rat
p = R\b
p =
        0
      5/7
        0
    -11/7
```

One of the nonzero components is p(2) because R(:,2) is the column of R with largest norm. The other nonzero component is p(4) because R(:,4) dominates after R(:,2) is eliminated.

The complete solution to the underdetermined system can be characterized by adding an arbitrary vector from the null space, which can be found using the null function with an option requesting a "rational" basis.

```
Z = null(R,'r')
Z =
      -1/2            -7/6
      -1/2             1/2
         1               0
         0               1
```

It can be confirmed that R*Z is zero and that any vector x where

```
x = p + Z*q
```

for an arbitrary vector q satisfies R*x = b.

Inverses and Determinants

This section provides:

- An overview of the use of inverses and determinants for solving square nonsingular systems of linear equations
- A discussion of the Moore-Penrose pseudoinverse for solving rectangular systems of linear equations

Overview

If A is square and nonsingular, the equations $AX = I$ and $XA = I$ have the same solution, X. This solution is called the *inverse* of A, is denoted by A^{-1}, and is computed by the function inv. The *determinant* of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and roundoff error properties make it far less satisfactory for numeric computation. Nevertheless, the function det computes the determinant of a square matrix.

```
A = pascal(3)

A =
      1      1      1
      1      2      3
      1      3      6

d = det(A)
X = inv(A)

d =
      1

X =
      3     -3      1
     -3      5     -2
      1     -2      1
```

Again, because A is symmetric, has integer elements, and has determinant equal to one, so does its inverse. On the other hand,

```
B = magic(3)
```

```
B =
       8       1       6
       3       5       7
       4       9       2

d = det(B)
X = inv(B)

d =
     -360

X =
      0.1472    -0.1444     0.0639
     -0.0611     0.0222     0.1056
     -0.0194     0.1889    -0.1028
```

Closer examination of the elements of X, or use of `format rat`, would reveal that they are integers divided by 360.

If A is square and nonsingular, then without roundoff error, X = inv(A)*B would theoretically be the same as X = A\B and Y = B*inv(A) would theoretically be the same as Y = B/A. But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error detection properties.

Pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations *AX* = *I* and *XA* = *I* does not have a solution. A partial replacement for the inverse is provided by the *Moore-Penrose pseudoinverse*, which is computed by the `pinv` function.

```
C = fix(10*rand(3,2));
X = pinv(C)

X =
    0.0401    -0.1492     0.1050
    0.0110     0.1657    -0.0055
```

The matrix

```
Q = X*C

Q =
     1.0000    0.0000
     0.0000    1.0000
```

is the 2-by-2 identity, but the matrix

```
P = C*X

P =
    0.2044    0.0663    0.3978
    0.0663    0.9945   -0.0331
    0.3978   -0.0331    0.8011
```

is not the 3-by-3 identity. However, P acts like an identity on a portion of the space in the sense that P is symmetric, P*C is equal to C and X*P is equal to X.

Solving a Rank-Deficient System

If A is *m*-by-*n* with $m > n$ and full rank *n*, then each of the three statements

```
x = A\b
x = pinv(A)*b
x = inv(A'*A)*A'*b
```

theoretically computes the same least squares solution x, although the backslash operator does it faster.

However, if A does not have full rank, the solution to the least squares problem is not unique. There are many vectors x that minimize

```
norm(A*x -b)
```

The solution computed by x = A\b is a *basic* solution; it has at most *r* nonzero components, where *r* is the rank of A. The solution computed by x = pinv(A)*b is the *minimal norm* solution because it minimizes norm(x). An attempt to compute a solution with x = inv(A'*A)*A'*b fails because A'*A is singular.

Here is an example that illustrates the various solutions.

```
A = [ 1   2   3
      4   5   6
      7   8   9
     10  11  12 ]
```

does not have full rank. Its second column is the average of the first and third columns. If

```
b = A(:,2)
```

is the second column, then an obvious solution to A*x = b is x = [0 1 0]'. But none of the approaches computes that x. The backslash operator gives

```
x = A\b

Warning: Rank deficient, rank = 2.

x =
     0.5000
     0
     0.5000
```

This solution has two nonzero components. The pseudoinverse approach gives

```
y = pinv(A)*b

y =
     0.3333
     0.3333
     0.3333
```

There is no warning about rank deficiency. But norm(y) = 0.5774 is less than norm(x) = 0.7071. Finally

```
z = inv(A'*A)*A'*b
```

fails completely.

```
Warning: Matrix is singular to working precision.

z =
     Inf
     Inf
     Inf
```

Cholesky, LU, and QR Factorizations

The MATLAB linear equation capabilities are based on three basic matrix factorizations:

- Cholesky factorization for symmetric, positive definite matrices
- LU factorization (Gaussian elimination) for general square matrices
- QR (orthogonal) for rectangular matrices

These three factorizations are available through the `chol`, `lu`, and `qr` functions.

All three of these factorizations make use of *triangular* matrices where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either *forward* or *back substitution*.

Cholesky Factorization

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose

$$A = R'R$$

where R is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be *positive definite*. This implies that all the diagonal elements of A are positive and that the offdiagonal elements are "not too big." The Pascal matrices provide an interesting example. Throughout this chapter, our example matrix A has been the 3-by-3 Pascal matrix. Let's temporarily switch to the 6-by-6.

```
A = pascal(6)

A =
       1     1     1     1     1     1
       1     2     3     4     5     6
       1     3     6    10    15    21
       1     4    10    20    35    56
       1     5    15    35    70   126
       1     6    21    56   126   252
```

The elements of A are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

```
R = chol(A)

R =
     1     1     1     1     1     1
     0     1     2     3     4     5
     0     0     1     3     6    10
     0     0     0     1     4    10
     0     0     0     0     1     5
     0     0     0     0     0     1
```

The elements are again binomial coefficients. The fact that R'*R is equal to A demonstrates an identity involving sums of products of binomial coefficients.

Note The Cholesky factorization also applies to complex matrices. Any complex matrix which has a Cholesky factorization satisfies $A' = A$ and is said to be *Hermitian positive definite*.

The Cholesky factorization allows the linear system

$$Ax = b$$

to be replaced by

$$R'Rx = b$$

Because the backslash operator recognizes triangular systems, this can be solved in MATLAB quickly with

```
x = R\(R'\b)
```

If A is n-by-n, the computational complexity of chol(A) is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

LU Factorization

LU factorization, or Gaussian elimination, expresses any square matrix A as the product of a permutation of a lower triangular matrix and an upper triangular matrix

$$A = LU$$

where L is a permutation of a lower triangular matrix with ones on its diagonal and U is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

cannot be expressed as the product of triangular matrices without interchanging its two rows. Although the matrix

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix}$$

can be expressed as the product of triangular matrices, when ε is small the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. *Partial pivoting* ensures that the elements of L are bounded by one in magnitude and that the elements of U are not much larger than those of A.

For example

```
[L,U] = lu(B)

L =
    1.0000         0         0
    0.3750    0.5441    1.0000
    0.5000    1.0000         0

U =
    8.0000    1.0000    6.0000
         0    8.5000   -1.0000
         0         0    5.2941
```

The LU factorization of A allows the linear system

```
A*x = b
```

to be solved quickly with

```
x = U\(L\b)
```

Determinants and inverses are computed from the LU factorization using

```
det(A) = det(L)*det(U)
```

and

```
inv(A) = inv(U)*inv(L)
```

You can also compute the determinants using `det(A) = prod(diag(U))`, though the signs of the determinants may be reversed.

QR Factorization

An *orthogonal* matrix, or a matrix with *orthonormal columns*, is a real matrix whose columns all have unit length and are perpendicular to each other. If Q is orthogonal, then

$$Q'Q = 1$$

The simplest orthogonal matrices are two-dimensional coordinate rotations.

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

For complex matrices, the corresponding term is *unitary*. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation may also be involved.

$$A = QR$$

or

$$AP = QR$$

where Q is orthogonal or unitary, R is upper triangular, and P is a permutation.

There are four variants of the QR factorization– full or economy size, and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is m-by-n with $m > n$. The *full* size QR factorization produces a square, m-by-m orthogonal Q and a rectangular m-by-n upper triangular R.

```
[Q,R] = qr(C)

Q =
    -0.8182     0.3999    -0.4131
    -0.1818    -0.8616    -0.4739
    -0.5455    -0.3126     0.7777

R =
  -11.0000     -8.5455
         0     -7.4817
         0           0
```

In many cases, the last $m - n$ columns of Q are not needed because they are multiplied by the zeros in the bottom portion of R. So the *economy* size QR factorization produces a rectangular, m-by-n Q with orthonormal columns and a square n-by-n upper triangular R. For our 3-by-2 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important.

```
[Q,R] = qr(C,0)

Q =
    -0.8182     0.3999
    -0.1818    -0.8616
    -0.5455    -0.3126

R =
  -11.0000     -8.5455
         0     -7.4817
```

In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of R occur in decreasing order and that any

linear dependence among the columns is almost certainly be revealed by examining these elements. For our small example, the second column of C has a larger norm than the first, so the two columns are exchanged.

```
[Q,R,P] = qr(C)

Q =
   -0.3522     0.8398    -0.4131
   -0.7044    -0.5285    -0.4739
   -0.6163     0.1241     0.7777

R =
  -11.3578    -8.2762
        0     7.2460
        0           0

P =
      0       1
      1       0
```

When the economy size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix.

```
[Q,R,p] = qr(C,0)

Q =
   -0.3522     0.8398
   -0.7044    -0.5285
   -0.6163     0.1241

R =
  -11.3578    -8.2762
        0     7.2460


p =
      2       1
```

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

```
norm(A*x - b)
```

is equal to

```
norm(Q*R*x - b)
```

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

```
norm(R*x - y)
```

where y = Q'*b. Since the last $m - n$ rows of R are zero, this expression breaks into two pieces

```
norm(R(1:n,1:n)*x - y(1:n))
```

and

```
norm(y(n+1:m))
```

When A has full rank, it is possible to solve for x so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When A does not have full rank, the triangular structure of R makes it possible to find a basic solution to the least squares problem.

Matrix Powers and Exponentials

This section tells you how to obtain the following matrix powers and exponentials in MATLAB:

- Positive integer
- Inverse and fractional
- Element-by-element
- Exponentials

Positive Integer Powers

If A is a square matrix and p is a positive integer, then A^p effectively multiplies A by itself p-1 times.

```
X = A^2

X =
     3      6     10
     6     14     25
    10     25     46
```

Inverse and Fractional Powers

If A is square and nonsingular, then A^(-p) effectively multiplies inv(A) by itself p-1 times.

```
Y = B^(-3)

Y =
    0.0053    -0.0068     0.0018
   -0.0034     0.0001     0.0036
   -0.0016     0.0070    -0.0051
```

Fractional powers, like A^(2/3), are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers

The .^ operator produces element-by-element powers. For example,

```
X = A.^2
```

```
A =
       1       1       1
       1       4       9
       1       9      36
```

Exponentials

The function

```
sqrtm(A)
```

computes A^(1/2) by a more accurate algorithm. The m in sqrtm distinguishes this function from sqrt(A) which, like A.^(1/2), does its job element-by-element.

A system of linear, constant coefficient, ordinary differential equations can be written

$$dx/dt = Ax$$

where $x = x(t)$ is a vector of functions of t and A is a matrix independent of t. The solution can be expressed in terms of the *matrix exponential*,

$$x(t) = e^{tA}x(0)$$

The function

```
expm(A)
```

computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix

```
A =
       0      -6      -1
       6       2     -16
      -5      20     -10
```

and the initial condition, $x(0)$

```
x0 =
       1
       1
       1
```
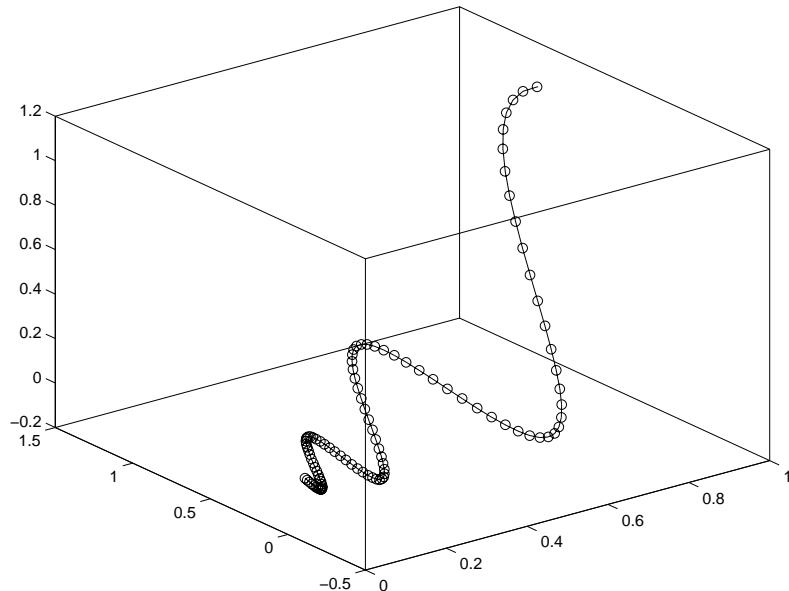
The matrix exponential is used to compute the solution, *x(t)*, to the differential equation at 101 points on the interval $0 \leq t \leq 1$ with

```
X = [];
for t = 0:.01:1
    X = [X expm(t*A)*x0];
end
```

A three-dimensional phase plane plot obtained with

```
plot3(X(1,:),X(2,:),X(3,:),'-o')
```

shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix, which are discussed in the next section.

Eigenvalues

An *eigenvalue* and *eigenvector* of a square matrix A are a scalar λ and a nonzero vector v that satisfy

$$Av = \lambda v$$

This section explains:

- Eigenvalue decomposition
- Problems associated with defective (not diagonalizable) matrices
- The use of Schur decomposition to avoid problems associated with eigenvalue decomposition

Eigenvalue Decomposition

With the eigenvalues on the diagonal of a diagonal matrix Λ and the corresponding eigenvectors forming the columns of a matrix V, we have

$$AV = V\Lambda$$

If V is nonsingular, this becomes the *eigenvalue decomposition*

$$A = V\Lambda V^{-1}$$

A good example is provided by the coefficient matrix of the ordinary differential equation in the previous section.

```
A =
     0    -6    -1
     6     2   -16
    -5    20   -10
```

The statement

```
lambda = eig(A)
```

produces a column vector containing the eigenvalues. For this matrix, the eigenvalues are complex.

```
lambda =
    -3.0710
    -2.4645+17.6008i
    -2.4645-17.6008i
```

The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as t increases. The nonzero imaginary part of two of the eigenvalues, $\pm\omega$, contributes the oscillatory component, $\sin(\omega t)$, to the solution of the differential equation.

With two output arguments, `eig` computes the eigenvectors and stores the eigenvalues in a diagonal matrix.

```
[V,D] = eig(A)

V =
   -0.8326        0.2003 - 0.1394i   0.2003 + 0.1394i
   -0.3553       -0.2110 - 0.6447i  -0.2110 + 0.6447i
   -0.4248       -0.6930            -0.6930

D =
   -3.0710                0                0
        0         -2.4645+17.6008i         0
        0                0        -2.4645-17.6008i
```

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, `norm(v,2)`, equal to one.

The matrix `V*D*inv(V)`, which can be written more succinctly as `V*D/V`, is within roundoff error of A. And, `inv(V)*A*V`, or `V\A*V`, is within roundoff error of D.

Defective Matrices

Some matrices do not have an eigenvector decomposition. These matrices are *defective*, or *not diagonalizable*. For example,

```
A = [ 6    12    19
     -9   -20   -33
      4     9    15 ]
```

For this matrix

```
[V,D] = eig(A)
```

produces

```
V =

   -0.4741    -0.4082    -0.4082
    0.8127     0.8165     0.8165
   -0.3386    -0.4082    -0.4082

D =

   -1.0000         0          0
         0    1.0000          0
         0         0     1.0000
```

There is a double eigenvalue at $\lambda = 1$. The second and third columns of V are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

The optional Symbolic Math Toolbox extends the capabilities of MATLAB by connecting to Maple, a powerful computer algebra system. One of the functions provided by the toolbox computes the Jordan Canonical Form. This is appropriate for matrices like our example, which is 3-by-3 and has exactly known, integer elements.

```
[X,J] = jordan(A)

X =
   -1.7500     1.5000     2.7500
    3.0000    -3.0000    -3.0000
   -1.2500     1.5000     1.2500

J =
   -1     0     0
    0     1     1
    0     0     1
```

The Jordan Canonical Form is an important theoretical concept, but it is not a reliable computational tool for larger matrices, or for matrices whose elements are subject to roundoff errors and other uncertainties.

Schur Decomposition in MATLAB Matrix Computations

The MATLAB advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the *Schur decomposition*

$$A = U \, S \, U^T$$

where *U* is an orthogonal matrix and *S* is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of *S*, while the columns of *U* provide a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of our defective example is

```
[U,S] = schur(A)

U =
   -0.4741     0.6648     0.5774
    0.8127     0.0782     0.5774
   -0.3386    -0.7430     0.5774

S =
   -1.0000    20.7846   -44.6948
        0     1.0000    -0.6096
        0          0     1.0000
```

The double eigenvalue is contained in the lower 2-by-2 block of S.

Note If A is complex, schur returns the complex Schur form, which is upper triangular with the eigenvalues of A on the diagonal.

Singular Value Decomposition

A *singular value* and corresponding *singular vectors* of a rectangular matrix A are a scalar σ and a pair of vectors u and v that satisfy

$$Av = \sigma u$$
$$A^T u = \sigma v$$

With the singular values on the diagonal of a diagonal matrix Σ and the corresponding singular vectors forming the columns of two orthogonal matrices U and V, we have

$$A V = U \Sigma$$
$$A^T U = V \Sigma$$

Since U and V are orthogonal, this becomes the *singular value decomposition*

$$A = U \Sigma V^T$$

The full singular value decomposition of an m-by-n matrix involves an m-by-m U, an m-by-n Σ, and an n-by-n V. In other words, U and V are both square and Σ is the same size as A. If A has many more rows than columns, the resulting U can be quite large, but most of its columns are multiplied by zeros in Σ. In this situation, the *economy* sized decomposition saves both time and storage by producing an m-by-n U, an n-by-n Σ and the same V.

The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. On the other hand, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If A is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as A departs from symmetry and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.

For the example matrix

```
A =
     9     4
     6     8
     2     7
```

the full singular value decomposition is

```
[U,S,V] = svd(A)

U =
    -0.6105     0.7174     0.3355
    -0.6646    -0.2336    -0.7098
    -0.4308    -0.6563     0.6194

S =
   14.9359          0
         0     5.1883
         0          0

V =
    -0.6925     0.7214
    -0.7214    -0.6925
```

You can verify that U*S*V' is equal to A to within roundoff error. For this small problem, the economy size decomposition is only slightly smaller.

```
[U,S,V] = svd(A,0)

U =
    -0.6105     0.7174
    -0.6646    -0.2336
    -0.4308    -0.6563

S =
   14.9359          0
         0     5.1883

V =
    -0.6925     0.7214
    -0.7214    -0.6925
```

Again, U*S*V' is equal to A to within roundoff error.

Polynomials and Interpolation

This chapter introduces MATLAB functions that enable you to work with polynomials and interpolate one-, two-, and multi-dimensional data. It includes the following:

Polynomials

This section provides:

- A summary of the MATLAB polynomial functions
- Instructions for representing polynomials in MATLAB

It also describes the MATLAB polynomial functions that:

- Calculate the roots of a polynomial
- Calculate the coefficients of the characteristic polynomial of a matrix
- Evaluate a polynomial at a specified value
- Convolve (multiply) and deconvolve (divide) polynomials
- Compute the derivative of a polynomial
- Fit a polynomial to a set of data
- Convert between partial fraction expansion and polynomial coefficients

Polynomial Function Summary

MATLAB provides functions for standard polynomial operations, such as polynomial roots, evaluation, and differentiation. In addition, there are functions for more advanced applications, such as curve fitting and partial fraction expansion.

The polynomial functions reside in the MATLAB `polyfun` directory.

Polynomial Function Summary

Function	Description
conv	Multiply polynomials.
deconv	Divide polynomials.
poly	Polynomial with specified roots.
polyder	Polynomial derivative.
polyfit	Polynomial curve fitting.
polyval	Polynomial evaluation.

Polynomial Function Summary (Continued)

Function	Description
polyvalm	Matrix polynomial evaluation.
residue	Partial-fraction expansion (residues).
roots	Find polynomial roots.

The Symbolic Math Toolbox contains additional specialized support for polynomial operations.

Representing Polynomials

MATLAB represents polynomials as row vectors containing coefficients ordered by descending powers. For example, consider the equation

$$p(x) = x^3 - 2x - 5$$

This is the celebrated example Wallis used when he first represented Newton's method to the French Academy. To enter this polynomial into MATLAB, use

```
p = [1 0 -2 -5];
```

Polynomial Roots

The roots function calculates the roots of a polynomial.

```
r = roots(p)

r =
     2.0946
    -1.0473 +      1.1359i
    -1.0473 -      1.1359i
```

By convention, MATLAB stores roots in column vectors. The function poly returns to the polynomial coefficients.

```
p2 = poly(r)

p2 =
    1    8.8818e-16    -2    -5
```

poly and roots are inverse functions, up to ordering, scaling, and roundoff error.

Characteristic Polynomials

The poly function also computes the coefficients of the characteristic polynomial of a matrix.

```
A = [1.2 3 -0.9; 5 1.75 6; 9 0 1];
poly(A)

ans =
    1.0000   -3.9500   -1.8500  -163.2750
```

The roots of this polynomial, computed with roots, are the *characteristic roots*, or eigenvalues, of the matrix A. (Use eig to compute the eigenvalues of a matrix directly.)

Polynomial Evaluation

The polyval function evaluates a polynomial at a specified value. To evaluate p at $s = 5$, use

```
polyval(p,5)

ans =
   110
```

It is also possible to evaluate a polynomial in a matrix sense. In this case $p(s) = x^3 - 2x - 5$ becomes $p(X) = X^3 - 2X - 5I$, where X is a square matrix and I is the identity matrix. For example, create a square matrix X and evaluate the polynomial p at X.

```
X = [2 4 5; -1 0 3; 7 1 5];
Y = polyvalm(p,X)

Y =
   377   179   439
   111    81   136
   490   253   639
```

Convolution and Deconvolution

Polynomial multiplication and division correspond to the operations convolution and deconvolution. The functions `conv` and `deconv` implement these operations.

Consider the polynomials $a(s) = s^2 + 2s + 3$ and $b(s) = 4s^2 + 5s + 6$. To compute their product,

```
a = [1 2 3]; b = [4 5 6];
c = conv(a,b)

c =
     4    13    28    27    18
```

Use deconvolution to divide $a(s)$ back out of the product.

```
[q,r] = deconv(c,a)

q =
     4     5     6

r =
     0     0     0     0     0
```

Polynomial Derivatives

The `polyder` function computes the derivative of any polynomial. To obtain the derivative of the polynomial `p = [1 0 -2 -5]`,

```
q = polyder(p)

q =
     3     0    -2
```

`polyder` also computes the derivative of the product or quotient of two polynomials. For example, create two polynomials `a` and `b`.

```
a = [1 3 5];
b = [2 4 6];
```

Calculate the derivative of the product `a*b` by calling `polyder` with a single output argument.

```
c = polyder(a,b)

c =
    8    30    56    38
```

Calculate the derivative of the quotient a/b by calling `polyder` with two output arguments.

```
[q,d] = polyder(a,b)

q =
   -2    -8    -2

d =
    4    16    40    48    36
```

q/d is the result of the operation.

Polynomial Curve Fitting

`polyfit` finds the coefficients of a polynomial that fits a set of data in a least-squares sense.

```
p = polyfit(x,y,n)
```

x and y are vectors containing the *x* and *y* data to be fitted, and n is the order of the polynomial to return. For example, consider the *x-y* test data.

```
x = [1 2 3 4 5]; y = [5.5 43.1 128 290.7 498.4];
```
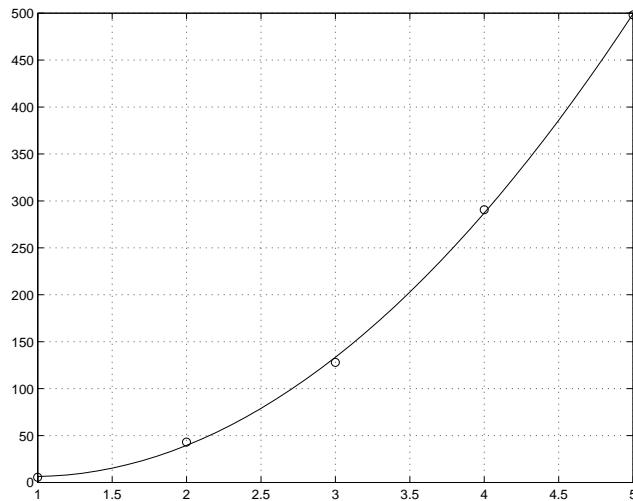
A third order polynomial that approximately fits the data is

```
p = polyfit(x,y,3)

p =
   -0.1917    31.5821    -60.3262    35.3400
```

Compute the values of the `polyfit` estimate over a finer range, and plot the estimate over the real data values for comparison.

```
x2 = 1:.1:5;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)
grid on
```

To use these functions in an application example, see the "Data Analysis and Statistics" chapter.

Partial Fraction Expansion

residue finds the partial fraction expansion of the ratio of two polynomials. This is particularly useful for applications that represent systems in transfer function form. For polynomials b and a, if there are no multiple roots,

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k_s$$

where r is a column vector of residues, p is a column vector of pole locations, and k is a row vector of direct terms. Consider the transfer function

$$\frac{-4s+8}{s^2+6s+8}$$

```
b = [-4 8];
a = [1 6 8];
```

```
[r,p,k] = residue(b,a)

r =
   -12
     8

p =
   -4
   -2

k =
   []
```

Given three input arguments (r, p, and k), residue converts back to polynomial form.

```
[b2,a2] = residue(r,p,k)

b2 =
   -4     8

a2 =
    1     6     8
```

Interpolation

Interpolation is a process for estimating values that lie between known data points. It has important applications in areas such as signal and image processing.

This section:

- Provides a summary of the MATLAB interpolation functions
- Discusses one-dimensional interpolation
- Discusses two-dimensional interpolation
- Uses an example to compare nearest neighbor, bilinear, and bicubic interpolation methods
- Discusses interpolation of multidimensional data
- Discusses triangulation and interpolation of scattered data

Interpolation Function Summary

MATLAB provides a number of interpolation techniques that let you balance the smoothness of the data fit with speed of execution and memory usage.

The interpolation functions reside in the MATLAB `polyfun` directory.

Interpolation Function Summary

Function	Description
`griddata`	Data gridding and surface fitting.
`griddata3`	Data gridding and hypersurface fitting for three-dimensional data.
`griddatan`	Data gridding and hypersurface fitting (dimension >= 3).
`interp1`	One-dimensional interpolation (table lookup).
`interp2`	Two-dimensional interpolation (table lookup).
`interp3`	Three-dimensional interpolation (table lookup).
`interpft`	One-dimensional interpolation using FFT method.

Interpolation Function Summary (Continued)

Function	Description
interpn	N-D interpolation (table lookup).
mkpp	Make a piecewise polynomial
pchip	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP).
ppval	Piecewise polynomial evaluation
spline	Cubic spline data interpolation
unmkpp	Piecewise polynomial details

One-Dimensional Interpolation

There are two kinds of one-dimensional interpolation in MATLAB:

- Polynomial interpolation
- FFT-based interpolation

Polynomial Interpolation

The function interp1 performs one-dimensional interpolation, an important operation for data analysis and curve fitting. This function uses polynomial techniques, fitting the supplied data with polynomial functions between data points and evaluating the appropriate function at the desired interpolation points. Its most general form is

```
yi = interp1(x,y,xi,method)
```

y is a vector containing the values of a function, and x is a vector of the same length containing the points for which the values in y are given. xi is a vector containing the points at which to interpolate. *method* is an optional string specifying an interpolation method:

- *Nearest neighbor interpolation* (method = 'nearest'). This method sets the value of an interpolated point to the value of the nearest existing data point.
- *Linear interpolation* (method = 'linear'). This method fits a different linear function between each pair of existing data points, and returns the value of

the relevant function at the points specified by `xi`. This is the default method for the `interp1` function.

- *Cubic spline interpolation* (`method = 'spline'`). This method fits a different cubic function between each pair of existing data points, and uses the `spline` function to perform cubic spline interpolation at the data points.

- *Cubic interpolation* (`method = 'pchip'` or `'cubic'`). These methods are identical. They use the `pchip` function to perform piecewise cubic Hermite interpolation within the vectors `x` and `y`. These methods preserve monotonicity and the shape of the data.

If any element of `xi` is outside the interval spanned by `x`, the specified interpolation method is used for extrapolation. Alternatively, `yi = interp1(x,Y,xi,method,extrapval)` replaces extrapolated values with `extrapval`. `NaN` is often used for `extrapval`.

All methods work with nonuniformly spaced data.

Speed, Memory, and Smoothness Considerations

When choosing an interpolation method, keep in mind that some require more memory or longer computation time than others. However, you may need to trade off these resources to achieve the desired smoothness in the result.

- Nearest neighbor interpolation is the fastest method. However, it provides the worst results in terms of smoothness.

- Linear interpolation uses more memory than the nearest neighbor method, and requires slightly more execution time. Unlike nearest neighbor interpolation its results are continuous, but the slope changes at the vertex points.

- Cubic spline interpolation has the longest relative execution time, although it requires less memory than cubic interpolation. It produces the smoothest results of all the interpolation methods. You may obtain unexpected results, however, if your input data is non-uniform and some points are much closer together than others.

- Cubic interpolation requires more memory and execution time than either the nearest neighbor or linear methods. However, both the interpolated data and its derivative are continuous.

The relative performance of each method holds true even for interpolation of two-dimensional or multidimensional data. For a graphical comparison of

interpolation methods, see the section "Comparing Interpolation Methods" on page 11-13.

FFT-Based Interpolation

The function `interpft` performs one-dimensional interpolation using an FFT-based method. This method calculates the Fourier transform of a vector that contains the values of a periodic function. It then calculates the inverse Fourier transform using more points. Its form is

```
y = interpft(x,n)
```

`x` is a vector containing the values of a periodic function, sampled at equally spaced points. `n` is the number of equally spaced points to return.

Two-Dimensional Interpolation

The function `interp2` performs two-dimensional interpolation, an important operation for image processing and data visualization. Its most general form is

```
ZI = interp2(X,Y,Z,XI,YI,method)
```

`Z` is a rectangular array containing the values of a two-dimensional function, and `X` and `Y` are arrays of the same size containing the points for which the values in `Z` are given. `XI` and `YI` are matrices containing the points at which to interpolate the data. `method` is an optional string specifying an interpolation method.

There are three different interpolation methods for two-dimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method fits a piecewise constant surface through the data values. The value of an interpolated point is the value of the nearest point.

- *Bilinear interpolation* (`method = 'linear'`). This method fits a bilinear surface through existing data points. The value of an interpolated point is a combination of the values of the four closest points. This method is piecewise bilinear, and is faster and less memory-intensive than bicubic interpolation.

- *Bicubic interpolation* (`method = 'cubic'`). This method fits a bicubic surface through existing data points. The value of an interpolated point is a combination of the values of the sixteen closest points. This method is piecewise bicubic, and produces a much smoother surface than bilinear interpolation. This can be a key advantage for applications like image

processing. Use bicubic interpolation when the interpolated data and its derivative must be continuous.
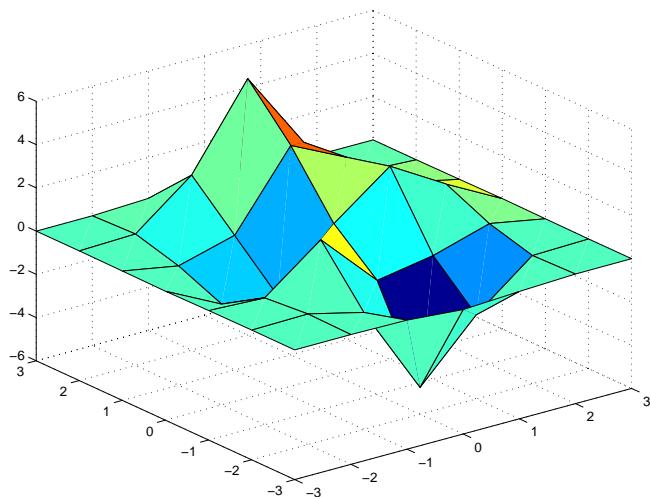
All of these methods require that X and Y be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the meshgrid function, or else be sure that the "pattern" of the points emulates the output of meshgrid. In addition, each method automatically maps the input to an equally spaced domain before interpolating. If X and Y are already equally spaced, you can speed execution time by prepending an asterisk to the method string, for example, '*cubic'.

Comparing Interpolation Methods

This example compares two-dimensional interpolation methods on a 7-by-7 matrix of data.

1 Generate the peaks function at low resolution.

```
[x,y] = meshgrid(-3:1:3);
z = peaks(x,y);
surf(x,y,z)
```

2 Generate a finer mesh for interpolation.

```
[xi,yi] = meshgrid(-3:0.25:3);
```

3 Interpolate using nearest neighbor interpolation.

```
zi1 = interp2(x,y,z,xi,yi,'nearest');
```
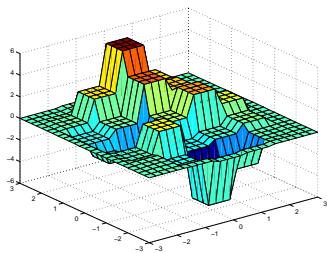
4 Interpolate using bilinear interpolation:

```
zi2 = interp2(x,y,z,xi,yi,'bilinear');
```
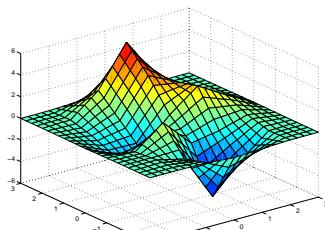
5 Interpolate using bicubic interpolation.

```
zi3 = interp2(x,y,z,xi,yi,'bicubic');
```
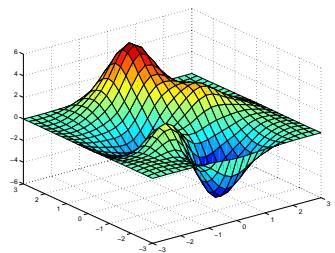
6 Compare the surface plots for the different interpolation methods.
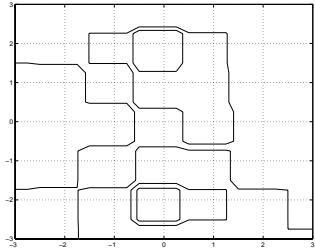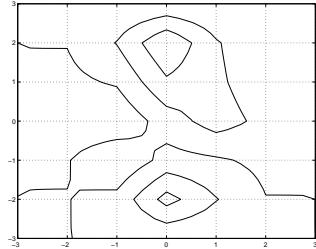

```
surf(xi,yi,zi1)          surf(xi,yi,zi2)          surf(xi,yi,zi3)
% nearest                % bilinear               % bicubic
```
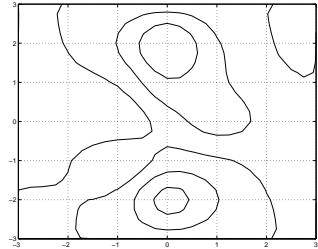
7 Compare the contour plots for the different interpolation methods.


```
contour(xi,yi,zi1)
% nearest
```



```
contour(xi,yi,zi2)
% bilinear
```



```
contour(xi,yi,zi3)
% bicubic
```

Notice that the bicubic method, in particular, produces smoother contours. This is not always the primary concern, however. For some applications, such as medical image processing, a method like nearest neighbor may be preferred because it doesn't generate any "new" data values.

Interpolation and Multidimensional Arrays

Several interpolation functions operate specifically on multidimensional data.

Interpolation Functions for Multidimensional Data

Function	Description
interp3	Three-dimensional data interpolation.
interpn	Multidimensional data interpolation.
ndgrid	Multidimensional data gridding (ndfun directory).

This section discusses:

- Interpolation of three-dimensional data
- Interpolation of higher dimensional data
- Multidimensional data gridding

Interpolation of Three-Dimensional Data

The function `interp3` performs three-dimensional interpolation, finding interpolated values between points of a three-dimensional set of samples `V`. You must specify a set of known data points:

- `X`, `Y`, and `Z` matrices specify the points for which values of `V` are given.
- A matrix `V` contains values corresponding to the points in `X`, `Y`, and `Z`.

The most general form for `interp3` is

```
VI = interp3(X,Y,Z,V,XI,YI,ZI,method)
```

`XI`, `YI`, and `ZI` are the points at which `interp3` interpolates values of `V`. For out-of-range values, `interp3` returns `NaN`.

There are three different interpolation methods for three-dimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Trilinear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest eight points.
- *Tricubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest sixty-four points.

All of these methods require that `X`, `Y`, and `Z` be *monotonic*, that is, either always increasing or always decreasing in a particular direction. In addition, you should prepare these matrices using the `meshgrid` function, or else be sure that the "pattern" of the points emulates the output of `meshgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If `x` is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string, for example, `'*cubic'`.

Interpolation of Higher Dimensional Data

The function `interpn` performs multidimensional interpolation, finding interpolated values between points of a multidimensional set of samples `V`. The most general form for `interpn` is

```
VI = interpn(X1,X2,X3...,V,Y1,Y2,Y3,...,method)
```

1, 2, 3, ... are matrices that specify the points for which values of `V` are given. `V` is a matrix that contains the values corresponding to these points. 1, 2, 3, ...

are the points for which `interpn` returns interpolated values of `V`. For out-of-range values, `interpn` returns `NaN`.

`Y1`, `Y2`, `Y3`, ... must be either arrays of the same size, or vectors. If they are vectors of different sizes, `interpn` passes them to `ndgrid` and then uses the resulting arrays.

There are three different interpolation methods for multidimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Linear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest two points in each dimension.
- *Cubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest four points in each dimension.

All of these methods require that `X1`, `X2,X3` be monotonic. In addition, you should prepare these matrices using the `ndgrid` function, or else be sure that the "pattern" of the points emulates the output of `ndgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If `X` is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string; for example, `'*cubic'`.

Multidimensional Data Gridding

The `ndgrid` function generates arrays of data for multidimensional function evaluation and interpolation. `ndgrid` transforms the domain specified by a series of input vectors into a series of output arrays. The `i`th dimension of these output arrays are copies of the elements of input vector x_i.

The syntax for `ndgrid` is
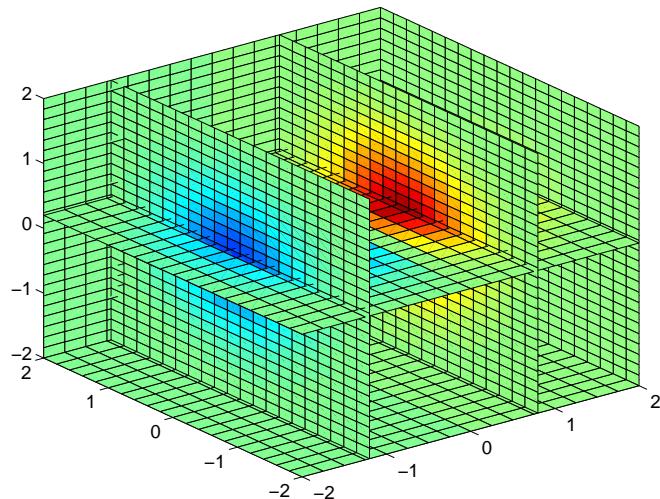
```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
```

For example, assume that you want to evaluate a function of three variables over a given range. Consider the function

$$z = x_2 e^{(-x_1^2 - x_2^2 - x_3^2)}$$

for $-2\pi \le x_1 \le 0$, $2\pi \le x_2 \le 4\pi$, and $0 \le x_3 \le 2\pi$. To evaluate and plot this function:

```
x1 = -2:0.2:2;
x2 = -2:0.25:2;
x3 = -2:0.16:2;
[X1,X2,X3] = ndgrid(x1,x2,x3);
z = X2.*exp(-X1.^2 -X2.^2 -X3.^2);
slice(X2,X1,X3,z,[-1.2 0.8 2],2,[-2 0.2])
```


Triangulation and Interpolation of Scattered Data

MATLAB provides routines that aid in the analysis of closest-point problems and geometric analysis.

Functions for Analysis of Closest-Point Problems and Geometric Analysis

Function	Description
convhull	Convex hull.
delaunay	Delaunay triangulation.

Functions for Analysis of Closest-Point Problems and Geometric Analysis

Function	Description
delaunay3	3-D Delaunay tessellation.
dsearch	Nearest point search of Delaunay triangulation.
inpolygon	True for points inside polygonal region.
polyarea	Area of polygon.
rectint	Area of intersection for two or more rectangles.
tsearch	Closest triangle search.
voronoi	Voronoi diagram.

This section applies the following techniques to the seamount data set supplied with MATLAB:

- Convex hulls
- Delaunay triangulation
- Voronoi diagrams

See also "Tessellation and Interpolation of Scattered Data in Higher Dimensions" on page 11-26.

Note Examples in this section use the MATLAB seamount data set. Seamounts are underwater mountains. They are valuable sources of information about marine geology. The seamount data set represents the surface, in 1984, of the seamount designated LR148.8W located at 48.2°S, 148.8°W on the Louisville Ridge in the South Pacific. For more information about the data and its use, see Parker [2].
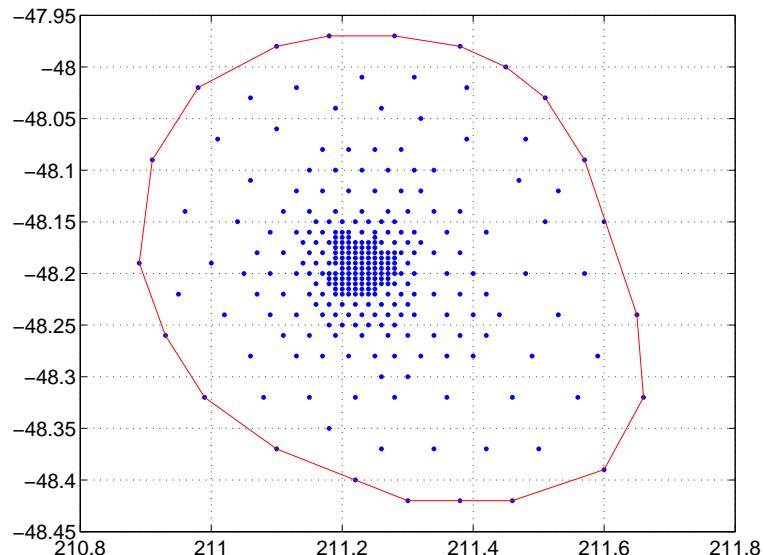
The seamount data set provides longitude (x), latitude (y) and depth-in-feet (z) data for 294 points on the seamount LR148.8W.

Convex Hulls

The `convhull` function returns the indices of the points in a data set that comprise the convex hull for the set. Use the `plot` function to plot the output of `convhull`.

This example loads the `seamount` data and plots the longitudinal (x) and latitudinal (y) data as a scatter plot. It then generates the convex hull and uses `plot` to plot the convex hull.

```
load seamount
plot(x,y,'.','markersize',10)
k = convhull(x,y);
hold on, plot(x(k),y(k),'-r'), hold off
grid on
```
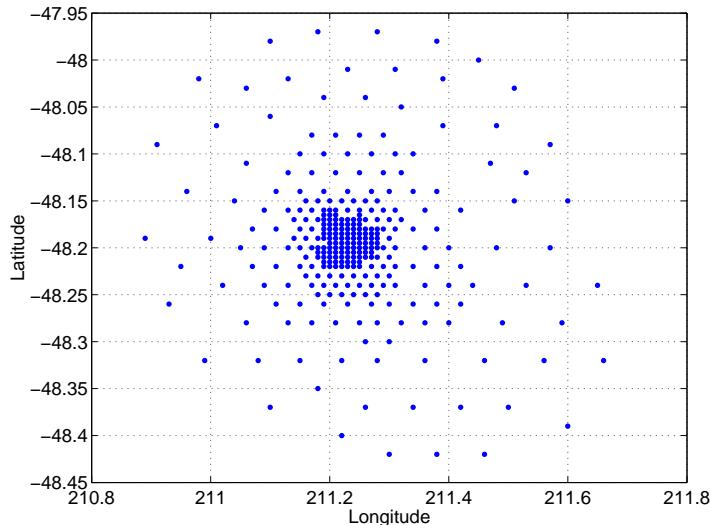

Delaunay Triangulation

Given a set of coplanar data points, *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The `delaunay` function returns a Delaunay triangulation as a set of triangles such that no data points are contained in any triangle's circumcircle.

You can use `triplot` to print the resulting triangles in two-dimensional space. You can also add data for a third dimension to the output of `delaunay` and plot the result as a surface with `trisurf`, or as a mesh with `trimesh`.
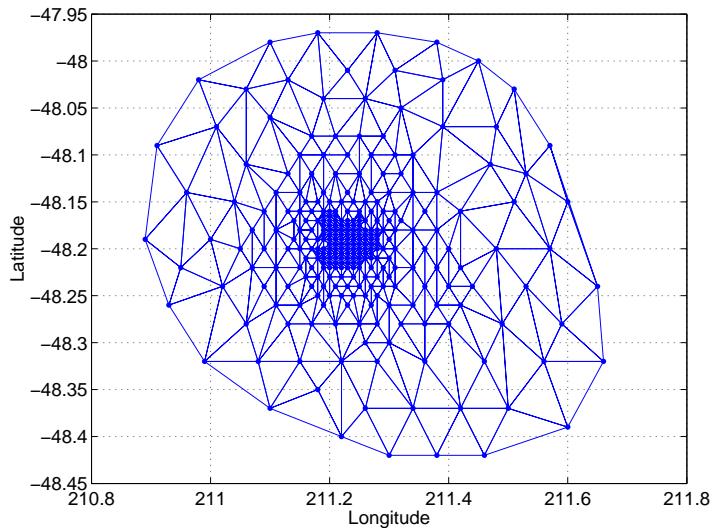
Plotting a Delaunay Triangulation. To try `delaunay`, load the `seamount` data set and view the longitude (x) and latitude (y) data as a scatter plot.

```
load seamount
plot(x,y,'.','markersize',12)
xlabel('Longitude'), ylabel('Latitude')
grid on
```
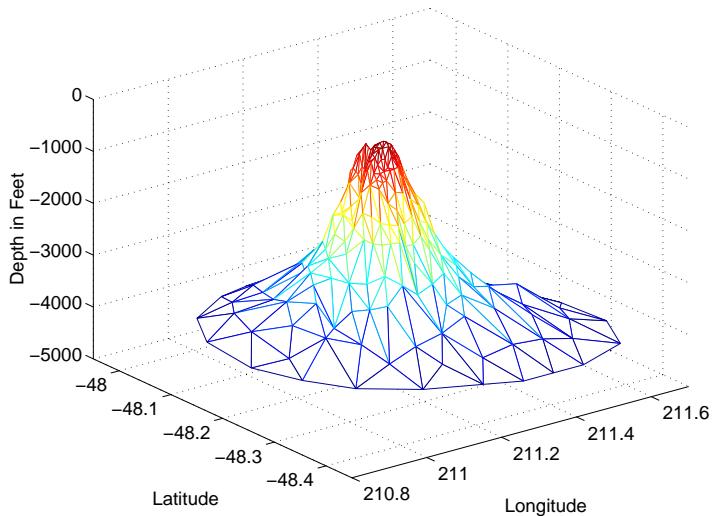

Apply Delaunay triangulation and use `triplot` to overplot the resulting triangles on the scatter plot.

```
tri = delaunay(x,y);
hold on, triplot(tri,x,y), hold off
```
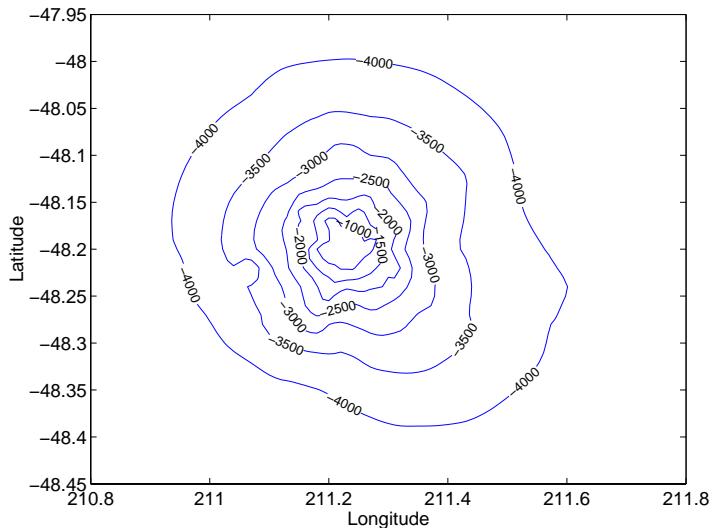
Mesh and Surface Plots. Add the depth data (z) from seamount, to the Delaunay triangulation, and use trimesh to produce a mesh in three-dimensional space. Similarly, you can use trisurf to produce a surface.

```
figure
hidden on
trimesh(tri,x,y,z)
grid on
xlabel('Longitude'); ylabel('Latitude'); zlabel('Depth in Feet')
```

Contour Plots. This code uses `meshgrid`, `griddata`, and `contour` to produce a contour plot of the `seamount` data.

```
figure
[xi,yi] = meshgrid(210.8:.01:211.8,-48.5:.01:-47.9);
zi = griddata(x,y,z,xi,yi,'cubic');
[c,h] = contour(xi,yi,zi,'b-');
clabel(c,h)
xlabel('Longitude'), ylabel('Latitude')
```

11-23

The arguments for meshgrid encompass the largest and smallest x and y values in the original seamount data. To obtain these values, use min(x), max(x), min(y), and max(y).

Closest-Point Searches. You can search through the Delaunay triangulation data with two functions:

- dsearch finds the indices of the (x,y) points in a Delaunay triangulation closest to the points you specify. This code searches for the point closest to (211.32, -48.35) in the triangulation of the seamount data.

```
xi = 211.32; yi = -48.35;
p = dsearch(x,y,tri,xi,yi);
[x(p), y(p)]

ans =
    211.3400  -48.3700
```

- tsearch, finds the indices into the delaunay output that specify the enclosing triangles of the points you specify. This example uses the index of the enclosing triangle for the point (211.32, -48.35) to obtain the coordinates of the vertices of the triangle.

```
xi = 211.32; yi = -48.35;
t = tsearch(x,y,tri,xi,yi);
r = tri(t,:);
A = [x(r) y(r)]

A =
  211.3000  -48.3000
  211.3400  -48.3700
  211.2800  -48.3200
```
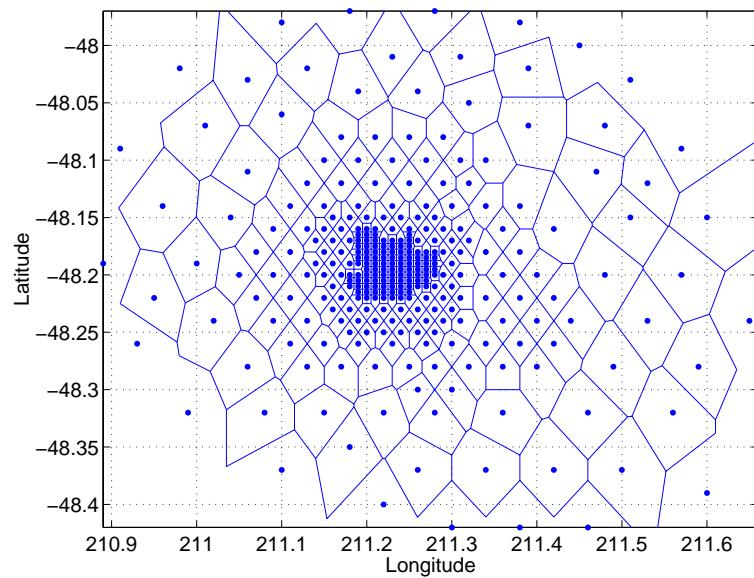
Voronoi Diagrams

Voronoi diagrams are a closest-point plotting technique related to Delaunay triangulation.

For each point in a set of coplanar points, you can draw a polygon that encloses all the intermediate points that are closer to that point than to any other point in the set. Such a polygon is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

The voronoi function can plot the cells of the Voronoi diagram, or return the vertices of the edges of the diagram. This example loads the seamount data, then uses the voronoi function to produce the Voronoi diagram for the longitudinal (x) and latitudinal (y) dimensions. Note that voronoi plots only the bounded cells of the Voronoi diagram.

```
load seamount
voronoi(x,y)
grid on
xlabel('Longitude'), ylabel('Latitude')
```

11-25

Note See the voronoi function for an example that uses the vertices of the edges to plot a Voronoi diagram.

Tessellation and Interpolation of Scattered Data in Higher Dimensions

Many applications in science, engineering, statistics, and mathematics require structures like convex hulls, Voronoi diagrams, and Delaunay tessellations. Using Qhull [1], MATLAB functions enable you to geometrically analyze data sets in any dimension.

Functions for Multidimensional Geometrical Analysis

Function	Description
convhulln	n-D convex hull.
delaunayn	n-D Delaunay tessellation.
dsearchn	n-D nearest point search.
griddatan	n-D data gridding and hypersurface fitting.
tsearchn	n-D closest simplex search.
voronoin	n-D Voronoi diagrams.

This section demonstrates these geometric analysis techniques:

- Convex hulls
- Delaunay triangulations
- Voronoi diagrams
- Interpolation of scattered multidimensional data

Convex Hulls

The convex hull of a data set in n-dimensional space is defined as the smallest convex region that contains the data set.

Computing a Convex Hull. The convhulln function returns the indices of the points in a data set that comprise the facets of the convex hull for the set. For example, suppose X is an 8-by-3 matrix that consists of the 8 vertices of a cube. The convex hull of X then consists of 12 facets.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];        % 8 corner points of a cube
C = convhulln(X)

C =
     3     1     5
     1     2     5
```
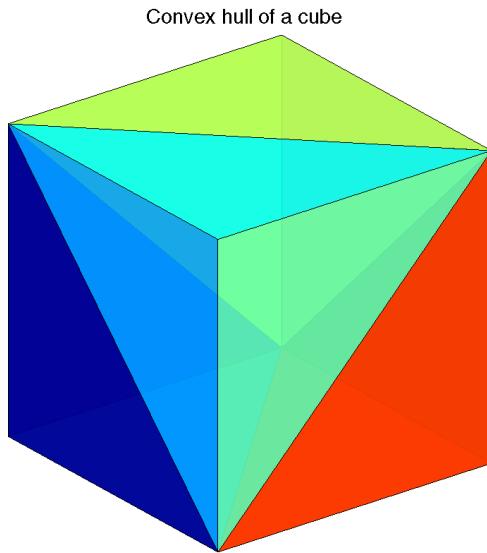
2	1	3
7	3	5
8	7	5
7	8	3
6	8	5
2	6	5
6	2	8
8	4	3
4	2	3
2	4	8

Because the data is three-dimensional, the facets that make up the convex hull are triangles. The 12 rows of C represent 12 triangles. The elements of C are indices of points in X. For example, the first row, 3 1 5, means that the first triangle has X(3,:), X(1,:), and X(5,:) as its vertices.

For three-dimensional convex hulls, you can use trisurf to plot the output. However, using patch to plot the output gives you more control over the color of the facets. Note that you cannot plot convhulln output for n > 3.

This code plots the convex hull by drawing the triangles as three-dimensional patches.

```
figure, hold on
d = [1 2 3 1        % Index into C column.
for i = 1:size(C,1) % Draw each triangle.
    j= C(i,d);      % Get the ith C to make a patch.
    h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9);
end                 % 'FaceAlpha' is used to make it transparent.
hold off
view(3), axis equal, axis off
camorbit(90,-5);    % To view it from another angle
title('Convex hull of a cube')
```

Convex hull of a cube

Delaunay Tessellations

A Delaunay tessellation is a set of simplices such that no data points are contained in any simplex's circumsphere. In two-dimensional space, a simplex is a triangle. In three-dimensional space, a simplex is a tetrahedron.

Computing a Delaunay Tessellation. The `delaunayn` function returns the indices of the points in a data set that comprise the simplices of an n-dimensional Delaunay tessellation of the data set.

This example uses the same X as in the convex hull example, i.e. the 8 corner points of a cube, with the addition of a center point.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];   % 8 corner points of a cube
X(9,:) = [0 0 0];       % Add center to the vertex list.
T = delaunayn(X)        % Generate Delaunay tessellation.

T =
```

9	1	5	6
3	9	1	5
2	9	1	6
2	3	9	4
2	3	9	1
7	9	5	6
7	3	9	5
8	7	9	6
8	2	9	6
8	2	9	4
8	3	9	4
8	7	3	9

The 12 rows of T represent the 12 simplices, in this case irregular tetrahedrons, that partition the cube. Each row represents one tetrahedron, and the row elements are indices of points in X.

For three-dimensional tessellations, you can use tetramesh to plot the output. However, using patch to plot the output gives you more control over the color of the facets. Note that you cannot plot delaunayn output for n > 3.
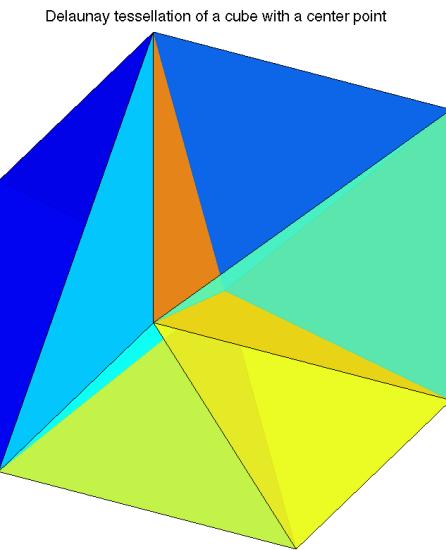
This code plots the tessellation T by drawing the tetrahedrons using three-dimensional patches.

```
figure, hold on
d = [1 1 1 2; 2 2 3 3; 3 4 4 4];  % Index into T
for i = 1:size(T,1)        % Draw each tetrahedron.
   y = T(i,d);             % Get the ith T to make a patch.
   x1 = reshape(X(y,1),3,4);
   x2 = reshape(X(y,2),3,4);
   x3 = reshape(X(y,3),3,4);
   h(i)=patch(x1,x2,x3,(1:4)*i,'FaceAlpha',0.9);
end
hold off
view(3), axis equal
axis off
camorbit(65,120)          % To view it from another angle
title('Delaunay tessellation of a cube with a center point')
```

You can use cameramenu to rotate the figure in any direction.

Delaunay tessellation of a cube with a center point

Voronoi Diagrams

Given *m* data points in *n*-dimensional space, a *Voronoi diagram* is the partition of *n*-dimensional space into *m* polyhedral regions, one region for each data point. Such a region is called a *Voronoi cell*. A Voronoi cell satisfies the condition that it contains all points that are closer to its data point than any other data point in the set.

Computing a Voronoi Diagram. The voronoin function returns two outputs:

- V is an *m*-by-*n* matrix of *m* points in *n*-space. Each row of V represents a Voronoi vertex.
- C is a cell array of vectors. Each vector in the cell array C represents a Voronoi cell. The vector contains indices of the points in V that are the vertices of the Voronoi cell. Each Voronoi cell may have a different number of points.

Because a Voronoi cell can be unbounded, the first row of V is a point at infinity. Then any unbounded Voronoi cell in C includes the point at infinity, i.e., the first point in V.

This example uses the same X as in the Delaunay example, i.e., the 8 corner points of a cube and its center. Random noise is added to make the cube less regular. The resulting Voronoi diagram has 9 Voronoi cells.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];      % 8 corner points of a cube
X(9,:) = [0 0 0];          % Add center to the vertex list.
X = X+0.01*rand(size(X));  % Make the cube less regular.
[V,C] = voronoin(X);

V =
       Inf       Inf       Inf
    0.0055    1.5054    0.0004
    0.0037    0.0101   -1.4990
    0.0052    0.0087   -1.4990
    0.0030    1.5054    0.0030
    0.0072    0.0072    1.4971
   -1.7912    0.0000    0.0044
   -1.4886    0.0011    0.0036
   -1.4886    0.0002    0.0045
    0.0101    0.0044    1.4971
    1.5115    0.0074    0.0033
    1.5115    0.0081    0.0040
    0.0104   -1.4846   -0.0007
    0.0026   -1.4846    0.0071

C =
    [1x8 double]
    [1x6 double]
    [1x4 double]
    [1x6 double]
    [1x6 double]
    [1x6 double]
    [1x6 double]
    [1x6 double]
    [1x12 double]
```

In this example, V is a 13-by-3 matrix, the 13 rows are the coordinates of the 13 Voronoi vertices. The first row of V is a point at infinity. C is a 9-by-1 cell array,
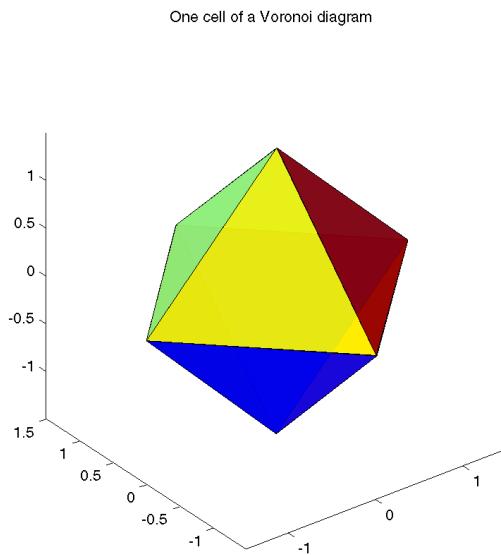
where each cell in the array contains an index vector into V corresponding to one of the 9 Voronoi cells. For example, the 9th cell of the Voronoi diagram is

```
C{9} = 2 3 4 5 6 7 8 9 10 11 12 13
```

If any index in a cell of the cell array is 1, then the corresponding Voronoi cell contains the first point in V, a point at infinity. This means the Voronoi cell is unbounded.

To view a *bounded* Voronoi cell, i.e., one that does not contain a point at infinity, use the convhulln function to compute the vertices of the facets that make up the Voronoi cell. Then use patch and other plot functions to generate the figure. For example, this code plots the Voronoi cell defined by the 9th cell in C.

```
X = V(C{9},:);        % View 9th Voronoi cell.
K = convhulln(X);
figure
hold on
d = [1 2 3 1];        % Index into K
for i = 1:size(K,1)
    j = K(i,d);
    h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9);
end
hold off
view(3)
axis equal
title('One cell of a Voronoi diagram')
```

One cell of a Voronoi diagram

Interpolating N-Dimensional Data

Use the `griddatan` function to interpolate multidimensional data, particularly scattered data. `griddatan` uses the `delaunayn` function to tessellate the data, and then interpolates based on the tessellation.

Suppose you want to visualize a function that you have evaluated at a set of n scattered points. In this example, X is an n-by-3 matrix of points, each row containing the (x,y,z) coordinates for one of the points. The vector v contains the n function values at these points. The function for this example is the squared distance from the origin, `v = x.^2 + y.^2 + z.^2`.

Start by generating n = 5000 points at random in three-dimensional space, and computing the value of a function on those points.

```
n = 5000;
X = 2*rand(n,3)-1;
v = sum(X.^2,2);
```

The next step is to use interpolation to compute function values over a grid. Use meshgrid to create the grid, and griddatan to do the interpolation.
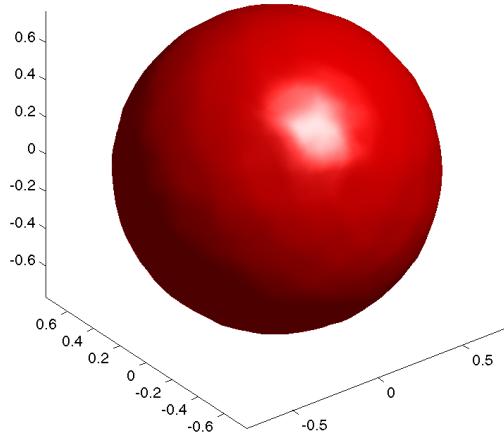
```
delta = 0.05;
d = -1:delta:1;
[x0,y0,z0] = meshgrid(d,d,d);
X0 = [x0(:), y0(:), z0(:)];
v0 = griddatan(X,v,X0);
v0 = reshape(v0, size(x0));
```

Then use isosurface and related functions to visualize the surface that consists of the (x,y,z) values for which the function takes a constant value. You could pick any value, but the example uses the value 0.6. Since the function is the squared distance from the origin, the surface at a constant value is a sphere.

```
p = patch(isosurface(x0,y0,z0,v0,0.6));
isonormals(x0,y0,z0,v0,p);
set(p,'FaceColor','red','EdgeColor','none');
view(3);
camlight;
lighting phong
axis equal
title('Interpolated sphere from scattered data')
```

Note A smaller delta produces a smoother sphere, but increases the compute time.

Interpolated sphere from scattered data

Selected Bibliography

[1] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993. For information about `qhull`, see `http://www.geom.umn.edu/software/qhull/`.

[2] Parker, Robert. L., Loren Shure, & John A. Hildebrand, "The Application of Inverse Theory to Seamount Magnetism." *Reviews of Geophysics*. Vol. 25, No. 1, 1987.

12

Data Analysis and Statistics

This chapter introduces the MATLAB data analysis capabilities. It includes the following topics:

Column-Oriented Data Sets (p. 12-3)   Organizing arrays for data analysis.

Basic Data Analysis Functions   Basic data analysis functions and an example that uses
(p. 12-7)   some of the functions. This section also discusses functions
                                       for the computation of correlation coefficients and
                                       covariance, and for finite difference calculations.

Data Preprocessing (p. 12-13)   Working with missing values, and outliers or misplaced
                                       data points in a data set.

Regression and Curve Fitting   Investigates the use of different regression methods to find
(p. 12-16)   functions that describe the relationship among observed
                                       variables.

Case Study: Curve Fitting (p. 12-21)   Uses a case study to look at some of the MATLAB basic
                                       data analysis capabilities. This section also provides
                                       information about the Basic Fitting interface.

Difference Equations and Filtering   Discusses MATLAB functions for working with difference
(p. 12-38)   equations and filters.

Fourier Analysis and the Fast Fourier Discusses Fourier analysis in MATLAB.
Transform (FFT) (p. 12-41)

Data Analysis and Statistics Functions

The data analysis and statistics functions are in the directory `datafun` in the MATLAB Toolbox. Use online help to get a complete list of functions.

Related Toolboxes

A number of related toolboxes provide advanced functionality for specialized data analysis applications.

Toolbox	Data Analysis Application
Optimization	Nonlinear curve fitting and regression.
Signal Processing	Signal processing, filtering, and frequency analysis.
Spline	Curve fitting and regression.
Statistics	Advanced statistical analysis, nonlinear curve fitting, and regression.
System Identification	Parametric / ARMA modeling.
Wavelet	Wavelet analysis.

Column-Oriented Data Sets

Univariate statistical data is typically stored in individual vectors. The vectors can be either 1-by-n or n-by-1. For multivariate data, a matrix is the natural representation but there are, in principle, two possibilities for orientation. By MATLAB convention, however, the different variables are put into columns, allowing observations to vary down through the rows. Therefore, a data set consisting of twenty four samples of three variables is stored in a matrix of size 24-by-3.

Vehicle Traffic Sample Data Set

Consider a sample data set comprising vehicle traffic count observations at three locations over a 24-hour period.

Vehicle Traffic Sample Data Set

Time	Location 1	Location 2	Location 3
01h00	11	11	9
02h00	7	13	11
03h00	14	17	20
04h00	11	13	9
05h00	43	51	69
06h00	38	46	76
07h00	61	132	186
08h00	75	135	180
09h00	38	88	115
10h00	28	36	55
11h00	12	12	14
12h00	18	27	30
13h00	18	19	29

Vehicle Traffic Sample Data Set (Continued)

Time	Location 1	Location 2	Location 3
14h00	17	15	18
15h00	19	36	48
16h00	32	47	10
17h00	42	65	92
18h00	57	66	151
19h00	44	55	90
20h00	114	145	257
21h00	35	58	68
22h00	11	12	15
23h00	13	9	15
24h00	10	9	7

Loading and Plotting the Data

The raw data is stored in the file, count.dat.

```
11    11     9
 7    13    11
14    17    20
11    13     9
43    51    69
38    46    76
61   132   186
75   135   180
38    88   115
28    36    55
12    12    14
18    27    30
18    19    29
17    15    18
19    36    48
```

```
 32    47    10
 42    65    92
 57    66   151
 44    55    90
114   145   257
 35    58    68
 11    12    15
 13     9    15
 10     9     7
```

Use the `load` command to import the data.

```
load count.dat
```

This creates the matrix `count` in the workspace.

For this example, there are 24 observations of three variables. This is confirmed by

```
[n,p] = size(count)
n =
    24
p =
     3
```
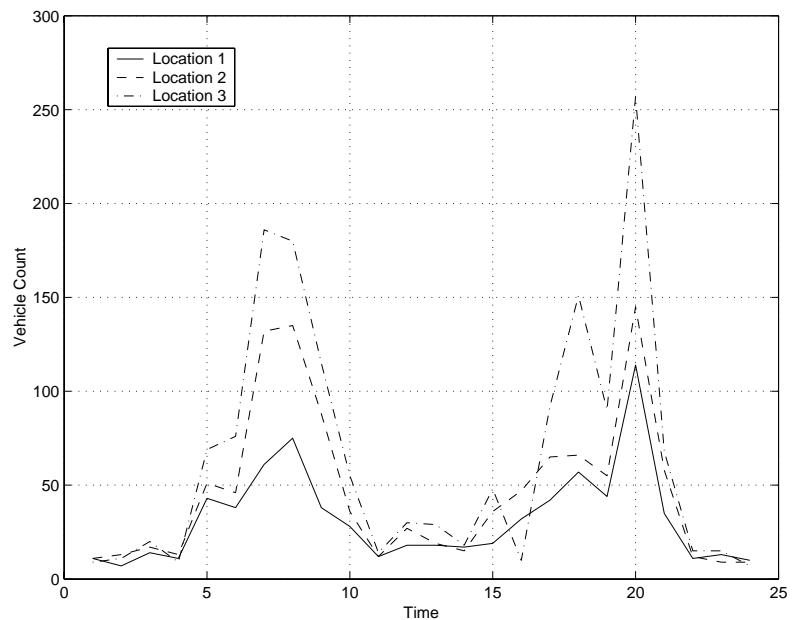
Create a time vector, `t`, of integers from 1 to n.

```
t = 1:n;
```

Now plot the counts versus time and annotate the plot.

```
set(0,'defaultaxeslinestyleorder','-|--|-.')
set(0,'defaultaxescolororder',[0 0 0])
plot(t,count), legend('Location 1','Location 2','Location 3',2)
xlabel('Time'), ylabel('Vehicle Count'), grid on
```

The plot shows the vehicle counts at three locations over a 24-hour period.

Basic Data Analysis Functions

This section introduces functions for:

- Basic column-oriented data analysis
- Computation of correlation coefficients and covariance
- Calculating finite differences

Function Summary

A collection of functions provides basic column-oriented data analysis capabilities. These functions are located in the MATLAB `datafun` directory.

This section also gives you some hints about using row and column data, and provides some basic examples. This table lists the functions.

Basic Data Analysis Function Summary

Function	Description
cumprod	Cumulative product of elements.
cumsum	Cumulative sum of elements.
cumtrapz	Cumulative trapezoidal numerical integration.
diff	Difference function and approximate derivative.
max	Largest component.
mean	Average or mean value.
median	Median value.
min	Smallest component.
prod	Product of elements.
sort	Sort in ascending order.
sortrows	Sort rows in ascending order.
std	Standard deviation.

Basic Data Analysis Function Summary (Continued)

Function	Description
sum	Sum of elements.
trapz	Trapezoidal numerical integration.

To use the Data Statistics Tool to calculate the maximum, minimum, mean, median, range, and standard deviation on plotted data, and create plots of these statistics, see "Using the Data Statistics Tool" in the MATLAB graphics documentation.

Working with Row and Column Data

For vector input arguments to these functions, it does not matter whether the vectors are oriented in row or column direction. For array arguments, however, the functions operate column by column on the data in the array. This means, for example, that if you apply max to an array, the result is a row vector containing the maximum values over each column.

Note You can add more functions to this list using M-files, but when doing so, you must exercise care to handle the row-vector case. If you are writing your own column-oriented M-files, check other M-files; for example, mean.m and diff.m.

Basic Examples

Continuing with the vehicle traffic count example, the statements

```
mx = max(count)
mu = mean(count)
sigma = std(count)
```

result in

```
mx =
          114            145            257

mu =
       32.0000       46.5417       65.5833
```

```
sigma =
      25.3703      41.4057      68.0281
```

To locate the index at which the minimum or maximum occurs, a second output parameter can be specified. For example,

```
[mx,indx] = min(count)

mx =
      7     9     7

indx =
      2    23    24
```

shows that the lowest vehicle count is recorded at 02h00 for the first observation point (column one) and at 23h00 and 24h00 for the other observation points.

You can subtract the mean from each column of the data using an outer product involving a vector of n ones.

```
[n,p] = size(count)
e = ones(n,1)
x = count - e*mu
```

Rearranging the data may help you evaluate a vector function over an entire data set. For example, to find the smallest value in the entire data set, use

```
min(count(:))
```

which produces

```
ans =
       7
```

The syntax count(:) rearranges the 24-by-3 matrix into a 72-by-1 column vector.

Covariance and Correlation Coefficients

The MATLAB statistical capabilities include two functions for the computation of correlation coefficients and covariance.

Covariance and Correlation Coefficient Function Summary

Function	Description
cov	Variance of vector – measure of spread or dispersion of sample variable.
	Covariance of matrix – measure of strength of linear relationships between variables.
corrcoef	Correlation coefficient – normalized measure of linear relationship strength between variables.

Covariance

cov returns the variance for a vector of data. The variance of the data in the first column of count is

```
cov(count(:,1))

ans =
     643.6522
```

For an array of data, cov calculates the covariance matrix. The variance values for the array columns are arranged along the diagonal of the covariance matrix. The remaining entries reflect the covariance between the columns of the original array. For an m-by-n matrix, the covariance matrix has size n-by-n. For example, the covariance matrix for count, cov(count), is arranged as

$$\begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 \\ \sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2 \end{bmatrix}$$

$$\sigma_{ij}^2 = \sigma_{ji}^2$$

Correlation Coefficients

corrcoef produces a matrix of correlation coefficients for an array of data where each row is an observation and each column is a variable. The *correlation coefficient* is a normalized measure of the strength of the linear relationship between two variables. Uncorrelated data results in a correlation coefficient of 0; equivalent data sets have a correlation coefficient of 1.

For an *m*-by-*n* matrix, the correlation coefficient matrix has size *n*-by-*n*. The arrangement of the elements in the correlation coefficient matrix corresponds to the location of the elements in the covariance matrix described above.

For our traffic count example

```
corrcoef(count)
```

results in

```
ans =
     1.0000    0.9331    0.9599
     0.9331    1.0000    0.9553
     0.9599    0.9553    1.0000
```

Clearly there is a strong linear correlation between the three traffic counts observed at the three locations, as the results are close to 1.

Finite Differences

MATLAB provides three functions for finite difference calculations.

Function	Description
diff	Difference between successive elements of a vector. Numerical partial derivatives of a vector.
gradient	Numerical partial derivatives a matrix.
del2	Discrete Laplacian of a matrix.

The diff function computes the difference between successive elements in a numeric vector. That is, diff(X) is [X(2)-X(1) X(3)-X(2)... X(n)-X(n-1)]. So, for a vector A,

```
A = [9 -2 3 0 1 5 4];
```

```
diff(A)

ans =
   -11     5    -3     1     4    -1
```

Besides computing the first difference, `diff` is useful for determining certain characteristics of vectors. For example, you can use `diff` to determine if a vector is monotonic (elements are always either increasing or decreasing), or if a vector has equally spaced elements. This table describes a few different ways to use `diff` with a vector `x`.

Test	Description
`diff(x)==0`	Tests for repeated elements.
`all(diff(x)>0)`	Tests for monotonicity.
`all(diff(diff(x))==0)`	Tests for equally spaced vector elements.

Data Preprocessing

This section tells you how to work with

- Missing values
- Outliers and misplaced data points

Missing Values

The special value, NaN, stands for Not-a-Number in MATLAB. IEEE floating-point arithmetic convention specifies NaN as the result of undefined expressions such as 0/0.

The correct handling of missing data is a difficult problem and often varies in different situations. For data analysis purposes, it is often convenient to use NaNs to represent missing values or data that are *not available.*

MATLAB treats NaNs in a uniform and rigorous way. They propagate naturally through to the final result in any calculation. Any mathematical calculation involving NaNs produces NaNs in the results.

For example, consider a matrix containing the 3-by-3 magic square with its center element set to NaN.

```
a = magic(3); a(2,2) = NaN

a =
     8     1     6
     3   NaN     7
     4     9     2
```

Compute a sum for each column in the matrix.

```
sum(a)

ans =
    15   NaN    15
```

Any mathematical calculation involving NaNs propagates NaNs through to the final result as appropriate.

You should remove NaNs from the data before performing statistical computations. Here are some ways to use isnan to remove NaNs from data.

Code	Description
`i = find(~isnan(x));` `x = x(i)`	Find indices of elements in vector that are not NaNs, then keep only the non-NaN elements.
`x = x(find(~isnan(x)))`	Remove NaNs from vector.
`x = x(~isnan(x));`	Remove NaNs from vector (faster).
`x(isnan(x)) = [];`	Remove NaNs from vector.
`X(any(isnan(X)'),:) = [];`	Remove any rows of matrix X containing NaNs.

Note You must use the special function isnan to find NaNs because, by IEEE arithmetic convention, the logical comparison, NaN == NaN always produces 0. You *cannot* use x(x==NaN) = [] to remove NaNs from your data.

If you frequently need to remove NaNs, write a short M-file function.

```
function X = excise(X)
X(any(isnan(X)'),:) = [];
```

Now, typing

```
X = excise(X);
```

accomplishes the same thing.

Removing Outliers

You can remove outliers or misplaced data points from a data set in much the same manner as NaNs. For the vehicle traffic count data, the mean and standard deviations of each column of the data are

```
mu = mean(count)
```

```
sigma = std(count)

mu =
      32.0000    46.5417    65.5833

sigma =
      25.3703    41.4057    68.0281
```

The number of rows with outliers greater than three standard deviations is obtained with

```
[n,p] = size(count)
outliers = abs(count - mu(ones(n, 1),:)) > 3*sigma(ones(n, 1),:);
nout = sum(outliers)
nout =
      1    0    0
```

There is one outlier in the first column. Remove this entire observation with
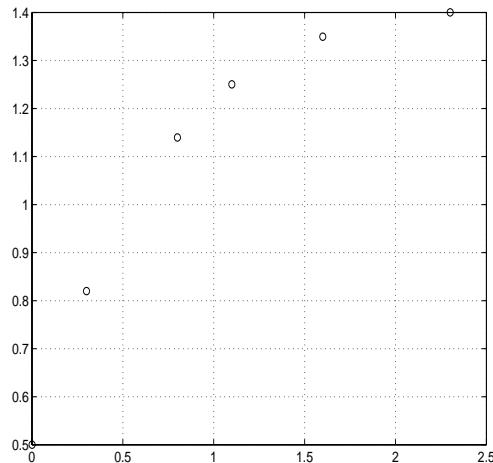
```
count(any(outliers'),:) = [];
```

Regression and Curve Fitting

It is often useful to find functions that describe the relationship between some variables you have observed. Identification of the coefficients of the function often leads to the formulation of an overdetermined system of simultaneous linear equations. You can find these coefficients efficiently by using the MATLAB backslash operator.

Suppose you measure a quantity y at several values of time t.

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [0.5 0.82 1.14 1.25 1.35 1.40]';
plot(t,y,'o'), grid on
```


The following sections look at three ways of modeling the data:

- Polynomial regression
- Linear-in-the-parameters regression
- Multiple regression

Polynomial Regression

Based on the plot, it is possible that the data can be modeled by a polynomial function

$$y = a_0 + a_1 t + a_2 t^2$$

The unknown coefficients a_0, a_1, and a_2 can be computed by doing a *least squares fit*, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in three unknowns,

$$
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}
=
\begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \\ 1 & t_6 & t_6^2 \end{bmatrix}
\times
\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}
$$

represented by the 6-by-3 matrix

```
X = [ones(size(t))  t  t.^2]

X =
    1.0000         0         0
    1.0000    0.3000    0.0900
    1.0000    0.8000    0.6400
    1.0000    1.1000    1.2100
    1.0000    1.6000    2.5600
    1.0000    2.3000    5.2900
```

The solution is found with the backslash operator.
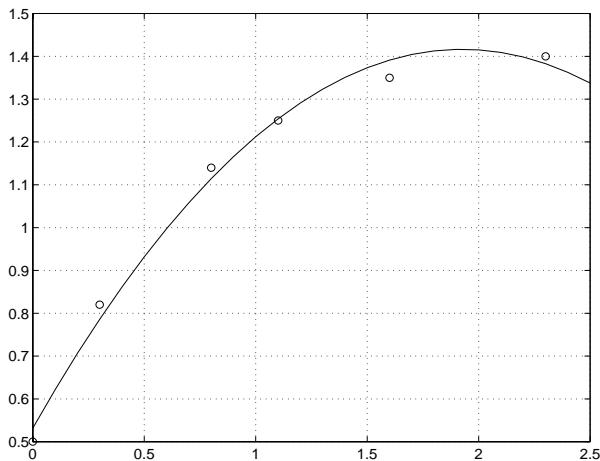
```
a = X\y

a =
    0.5318
    0.9191
  - 0.2387
```

The second-order polynomial model of the data is therefore

$$y = 0.5318 + 0.919(1)t - 0.2387t^2$$

Now evaluate the model at regularly spaced points and overlay the original data in a plot.

```
T = (0:0.1:2.5)';
Y = [ones(size(T))  T  T.^2]*a;
plot(T,Y,'-',t,y,'o'), grid on
```


Clearly this fit does not perfectly approximate the data. We could either increase the order of the polynomial fit, or explore some other functional form to get a better approximation.

Linear-in-the-Parameters Regression

Instead of a polynomial function, we could try using a function that is linear-in-the-parameters. In this case, consider the exponential function

$$y = a_0 + a_1 e^{-t} + a_2 t e^{-t}$$

The unknown coefficients a_0, a_1, and a_2, are computed by performing a *least squares fit*. Construct and solve the set of simultaneous equations by forming the regression matrix, X, and solving for the coefficients using the backslash operator.

```
X = [ones(size(t))  exp(-t)  t.*exp(-t)];
a = X\y
```
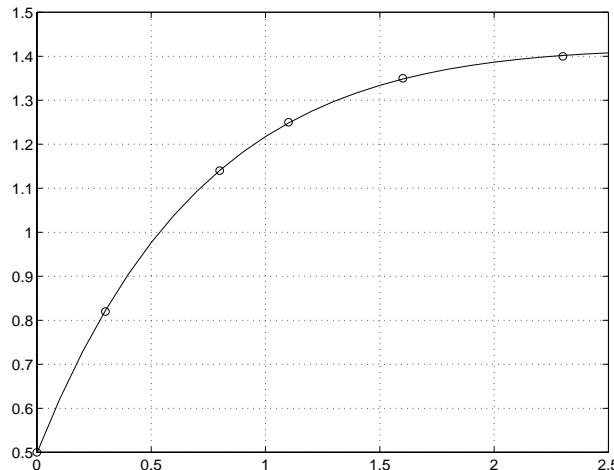
```
a =
    1.3974
  - 0.8988
    0.4097
```

The fitted model of the data is, therefore,

$$y = 1.3974 - 0.8988 \ e^{-t} + 0.4097 \ te^{-t}$$

Now evaluate the model at regularly spaced points and overlay the original data in a plot.

```
T = (0:0.1:2.5)';
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;
plot(T,Y,'-',t,y,'o'), grid on
```


This is a much better fit than the second-order polynomial function.

Multiple Regression

If y is a function of more than one independent variable, the matrix equations that express the relationships among the variables can be expanded to accommodate the additional data.

Suppose we measure a quantity y for several values of parameters x_1 and x_2. The observations are entered as

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';
```

A multivariate model of the data is

$$y = a_0 + a_1 x_1 + a_2 x_2$$

Multiple regression solves for unknown coefficients a_0, a_1, and a_2, by performing a *least squares fit*. Construct and solve the set of simultaneous equations by forming the regression matrix, X, and solving for the coefficients using the backslash operator.

```
X = [ones(size(x1))  x1  x2];
a = X\y

a =
    0.1018
    0.4844
   -0.2847
```

The least squares fit model of the data is

$$y = 0.1018 + 0.4844 \ x_1 - 0.2847 \ x_2$$

To validate the model, find the maximum of the absolute value of the deviation of the data from the model.

```
Y = X*a;
MaxErr = max(abs(Y - y))

MaxErr =
     0.0038
```

This is sufficiently small to be confident the model reasonably fits the data.

Case Study: Curve Fitting

This section provides an overview of some of the MATLAB basic data analysis capabilities in the form of a case study. The examples that follow work with a collection of census data, using MATLAB functions to experiment with fitting curves to the data:

- Polynomial fit
- Analyzing residuals
- Exponential fit
- Error bounds

This section also tells you how to use the Basic Fitting interface to perform curve fitting tasks.

Loading the Data

The file `census.mat` contains U.S. population data for the years 1790 through 1990. Load it into MATLAB:

```
load census
```

Your workspace now contains two new variables, `cdate` and `pop`:

- `cdate` is a column vector containing the years from 1790 to 1990 in increments of 10.
- `pop` is a column vector with the U.S. population figures that correspond to the years in `cdate`.

Polynomial Fit

A first try in fitting the census data might be a simple polynomial fit. Two MATLAB functions help with this process.

Curve Fitting Function Summary

Function	Description
polyfit	Polynomial curve fit.
polyval	Evaluation of polynomial fit.

The MATLAB `polyfit` function generates a "best fit" polynomial (in the least squares sense) of a specified order for a given set of data. For a polynomial fit of the fourth-order

```
p = polyfit(cdate,pop,4)
Warning: Polynomial is badly conditioned. Remove repeated data
points or try centering and scaling as described in HELP POLYFIT.

p =
   1.0e+05 *

     0.0000   -0.0000    0.0000   -0.0126 6.0020
```

The warning arises because the `polyfit` function uses the `cdate` values as the basis for a matrix with very large values (it creates a Vandermonde matrix in its calculations – see the `polyfit` M-file for details). The spread of the `cdate` values results in scaling problems. One way to deal with this is to normalize the `cdate` data.

Preprocessing: Normalizing the Data

Normalization is a process of scaling the numbers in a data set to improve the accuracy of the subsequent numeric computations. A way to normalize `cdate` is to center it at zero mean and scale it to unit standard deviation:

```
sdate = (cdate - mean(cdate))./std(cdate)
```
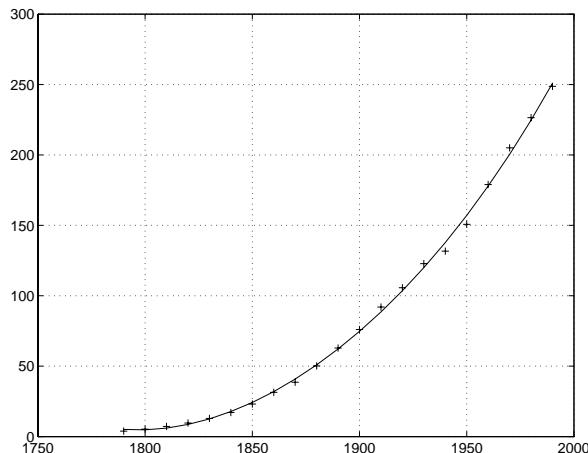
Now try the fourth-degree polynomial model using the normalized data:

```
p = polyfit(sdate,pop,4)

p =
    0.7047    0.9210   23.4706   73.8598   62.2285
```

Evaluate the fitted polynomial at the normalized year values, and plot the fit against the observed data points:

```
pop4 = polyval(p,sdate);
plot(cdate,pop4,'-',cdate,pop,'+'), grid on
```

Another way to normalize data is to use some knowledge of the solution and units. For example, with this data set, choosing 1790 to be year zero would also have produced satisfactory results.
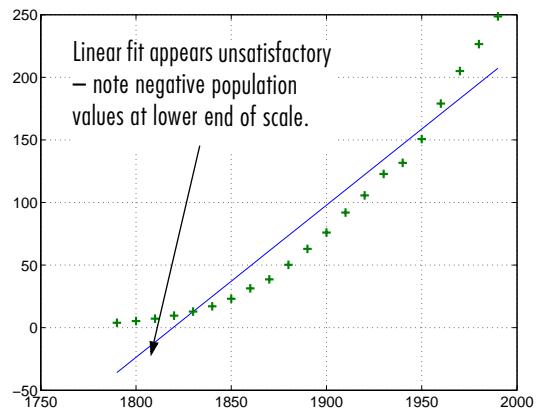
Analyzing Residuals

A measure of the "goodness" of fit is the *residual*, the difference between the observed and predicted data. Compare the residuals for the various fits, using normalized cdate values. It's evident from studying the fit plots and residuals that it should be possible to do better than a simple polynomial fit with this data set.

Comparison Plots of Fit and Residual
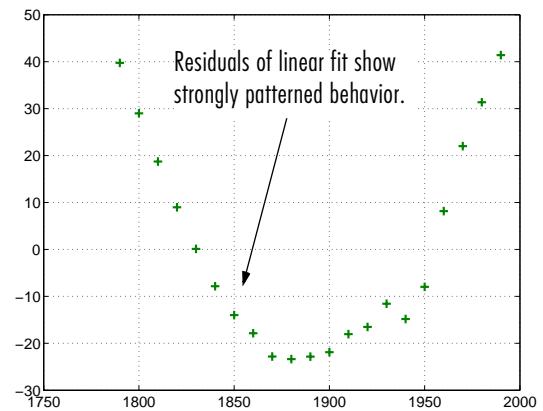
Fit	Residuals

```p1 = polyfit(sdate,pop,1);```
```pop1 = polyval(p1,sdate);```
```plot(cdate,pop1,'b-',cdate,pop,'g+')```
	```res1 = pop - pop1;```
	```figure, plot(cdate,res1,'g+')```
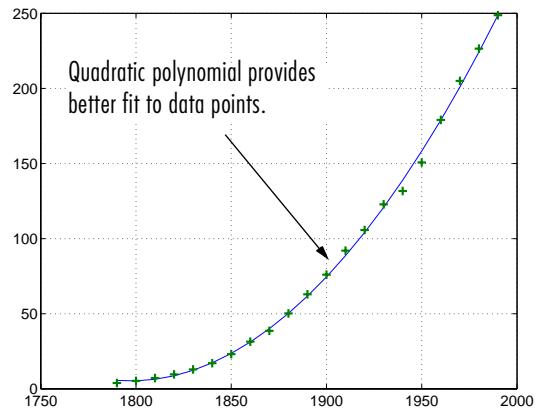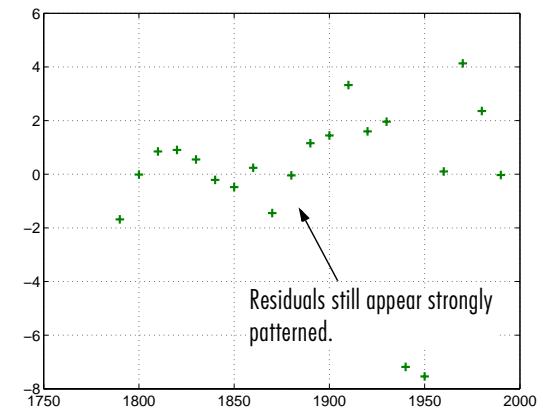
Linear fit appears unsatisfactory – note negative population values at lower end of scale.

Residuals of linear fit show strongly patterned behavior.

| ```p = polyfit(sdate,pop,2);```
```pop2 = polyval(p,sdate);```
```plot(cdate,pop2,'b-',cdate,pop,'g+')``` | ```res2 = pop - pop2;```
```figure, plot(cdate,res2,'g+')``` |

Quadratic polynomial provides better fit to data points.

Residuals still appear strongly patterned.
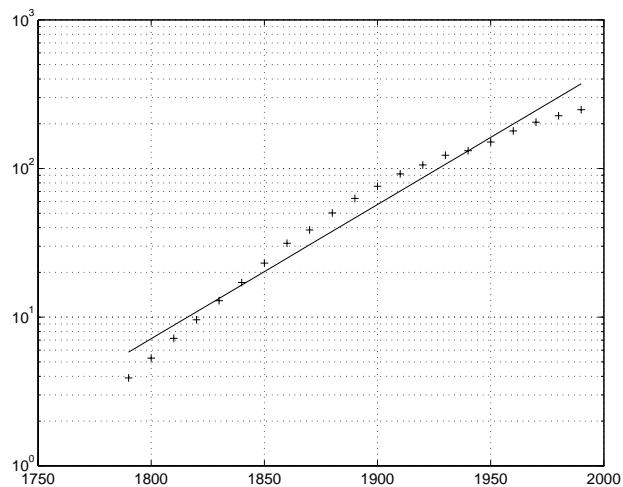
Comparison Plots of Fit and Residual (Continued)

Fit	Residuals

``
p = polyfit(sdate,pop,4);
pop4 = polyval(p,sdate);
plot(cdate,pop4,'b-',cdate,pop,'g+')
``
	``
	res4 = pop - pop4;
	figure, plot(cdate,res4,'g+')
	``

Fourth-degree model provides little improvement — note that curve still begins to turn upward at lower end of plot.

Residuals still appear strongly patterned.

Exponential Fit
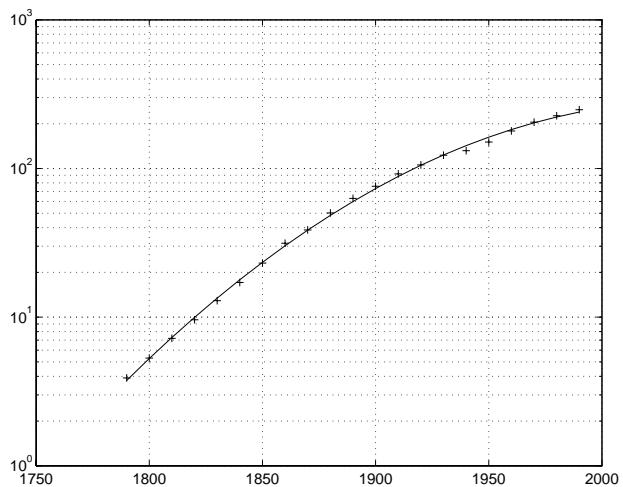
By looking at the population data plots on the previous pages, the population data curve is somewhat exponential in appearance. To take advantage of this, let's try to fit the logarithm of the population values, again working with normalized year values.

```
logp1 = polyfit(sdate,log10(pop),1);
logpred1 = 10.^polyval(logp1,sdate);
semilogy(cdate,logpred1,'-',cdate,pop,'+');
grid on
```
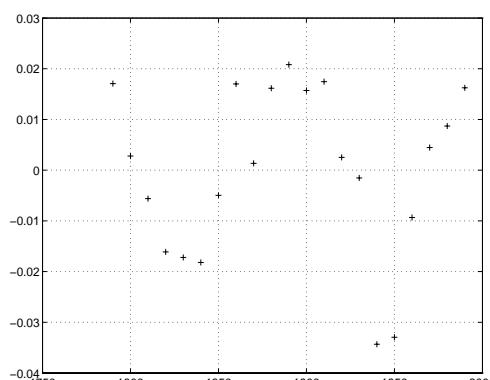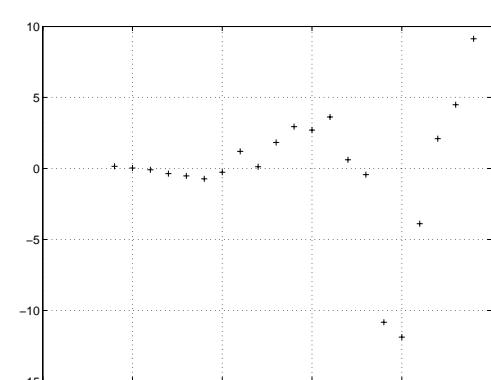
Now try the logarithm analysis with a second-order model.

```
logp2 = polyfit(sdate,log10(pop),2);
logpred2 = 10.^polyval(logp2,sdate);
semilogy(cdate,logpred2,'-',cdate,pop,'+'); grid on
```

This is a more accurate model. The upper end of the plot appears to taper off, while the polynomial fits in the previous section continue, concave up, to infinity.

Compare the residuals for the second-order logarithmic model.

Residuals in Log Population Scale	Residuals in Population Scale
| ```
logres2 = log10(pop)
polyval(logp2,sdate);
plot(cdate,logres2,'+')
``` | ```
r = pop  10.^(polyval(logp2,sdate));
plot(cdate,r,'+')
``` |

The residuals are more random than for the simple polynomial fit. As might be expected, the residuals tend to get larger in magnitude as the population increases. But overall, the logarithmic model provides a more accurate fit to the population data.

Error Bounds

Error bounds are useful for determining if your data is reasonably modeled by the fit. You can obtain the error bounds by passing an optional second output parameter from `polyfit` as an input parameter to `polyval`.

This example uses the `census` demo data and normalizes the data by centering it at zero mean and scaling it to unit standard deviation. The example then

uses `polyfit` and `polyval` to produce error bounds for a second-order polynomial model. Year values are normalized. This code uses an interval of $\pm 2\Delta$, corresponding to a 95% confidence interval.

```
load census
sdate = (cdate - mean(cdate))./std(cdate)
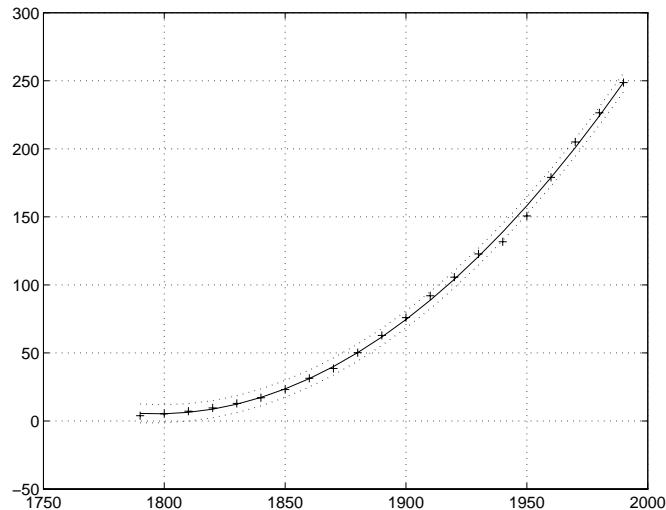
[p2,S2] = polyfit(sdate,pop,2);
[pop2,del2] = polyval(p2,sdate,S2);
plot(cdate,pop,'+',cdate,pop2,'g-',cdate,pop2+2*del2,'r:',...
    cdate,pop2-2*del2,'r:'), grid on
```


The Basic Fitting Interface

MATLAB supports curve fitting through the Basic Fitting interface. Using this interface, you can quickly perform many curve fitting tasks within the same easy-to-use environment. The interface is designed so that you can:

• Fit data using a spline interpolant, a shape-preserving interpolant, or a polynomial up to degree 10.

• Plot multiple fits simultaneously for a given data set.

- Plot the fit residuals.

- Examine the numerical results of a fit.

- Evaluate (interpolate or extrapolate) a fit.

- Annotate the plot with the numerical fit results and the norm of residuals.

- Save the fit and evaluated results to the MATLAB workspace.

Depending on your specific curve fitting application, you can use the Basic Fitting interface, the command line functionality, or both.

You can use the Basic Fitting interface only with 2-D data. However, if you plot multiple data sets as a subplot, and at least one data set is 2-D, then the interface is enabled.

Note For the HP, IBM, and SGI platforms, the Basic Fitting interface is not supported for Release 13.

Overview of the Basic Fitting Interface

The full Basic Fitting interface is shown below. To reproduce this state, follow these three steps:

1 Plot some data.

2 Select **Basic Fitting** from the **Tools** menu.

3 Click the ➔ button twice.

Select data – This parameter list is populated with the names of all the data sets you display in the figure window associated with the Basic Fitting interface.

Use this list to select the current data set. The current data set is defined as the data set that is to be fit. You can fit only one data set at a time. However, you can perform multiple fits for the current data set. Use the Plot Editor to change the name of a data set.

Center and scale X data – If checked, the data is centered at zero mean and scaled to unit standard deviation. You may need to center and scale your data to improve the accuracy of the subsequent numerical computations. A warning is displayed if a fit produces results that may be inaccurate.

Plot fits – This panel allows you to visually explore one or more fits to the current data set:

• **Check to display fits on figure** – Select the fits you want to display for the current data set. There are two types of fits to choose from: interpolants and polynomials. The spline interpolant uses the spline function, while the

shape-preserving interpolant uses the `pchip` function. Refer to the `pchip` online help for a comparison of these two functions. The polynomial fits use the `polyfit` function. You can choose as many fits for a given data set as you want.

If your data set has N points, then you should use polynomials with, at most, N coefficients. If your fit uses polynomials with more than N coefficients, the interface automatically sets a sufficient number of coefficients to `0` during the calculation so that the system is not underdetermined.

- **Show equations** – If checked, the fit equation is displayed on the plot.

  - **Significant digits** – Select the significant digits associated with the equation display.

- **Plot residuals** – If checked, the fit residuals are displayed. The fit residuals are defined as the difference between the ordinate data point and the resulting fit for each abscissa data point. You can display the residuals as a bar plot, as a scatter plot, or as a line plot in the same figure window as the data or in a separate figure window. If you use subplots to plot multiple data sets, then residuals can be plotted only in a separate figure window.

  - **Show norm of residuals** – If checked, the norm of residuals are displayed. The norm of residuals is a measure of the goodness of fit, where a smaller value indicates a better fit than a larger value. It is calculated using the `norm` function, `norm(V,2)`, where `V` is the vector of residuals.

Numerical results – This panel allows you to explore the numerical results of a single fit to the current data set without plotting the fit:

- **Fit** – Select the equation to fit to the current data set. The fit results are displayed in the list box below the menu. Note that selecting an equation in this menu does not affect the state of the **Plot fits** panel. Therefore, if you want to display the fit in the data plot, you may need to select the associated check box in **Plot fits**.

- **Coefficients and norm of residuals** – Display the numerical results for the equation selected in **Fit**. Note that when you first open the **Numerical Results** panel, the results of the last fit you selected in **Plot fits** are displayed.

- **Save to workspace** – Launch a dialog box that allows you to save the fit results to workspace variables.

- **Find Y = f(X)** – Interpolate or extrapolate the current fit.

- **Enter value(s)** – Enter a MATLAB expression to evaluate for the current fit. The expression is evaluated after you press the **Evaluate** button, and the results are displayed in the associated table. The current fit is displayed in the **Fit** menu.

- **Save to workspace** – Launch a dialog box that allows you to save the evaluated results to workspace variables.

- **Plot results** – If checked, the evaluated results are displayed on the data plot.

Example: Using the Basic Fitting Interface

This example illustrates the features of the Basic Fitting interface by fitting a cubic polynomial to the census data. You may want to repeat this example using different equations and compare results. To launch the interface:

1 Plot some data.

```
plot(cdate,pop,'ro')
```

2 Select **Basic Fitting** from the **Tools** menu in the figure.

Configure the Basic Fitting interface to:

- Fit a cubic polynomial to the data.
- Display the equation in the data plot.
- Plot the fit residuals as a bar plot, and display the residuals as a subplot of the data figure window.
- Display the norm of the residuals.

This configuration is shown below.

The **Plot fits** panel allows you to visually explore multiple fits to the current data set. For comparison, try fitting additional equations to the census data by selecting the appropriate check boxes. If an equation produces results that may be numerically inaccurate, a warning is displayed. In this case, you should select the **Center and scale X data** check box to improve the numerical accuracy.

The resulting fit and the residuals are shown below.

The plot legend indicates the name of the data set and the equation. The legend is automatically updated as you add or remove data sets or fits. Additionally, fits are displayed using a default set of line styles and colors. You can change any of the default plot settings using the Plot Editor. However, any changes you make are undone if you subsequently perform another fit. To retain changes, you should wait until after you have finished fitting your data.

Note If you change the name of a data set in the legend, then the name is automatically changed in the **Select data** menu.

By selecting the [→] button, you can examine the fit coefficients and the norm of the residuals.

The **Fit** menu allows you to explore numerical fit results for the current data set without plotting the fit. For comparison, you can display the numerical results for other fits by selecting the desired equation. Note that if you want to display a fit in the data plot, you have to select the associated check box in **Plot fits**.

You can save the fit results to the MATLAB workspace by selecting the **Save to workspace** button.

The fit structure is shown below.

```
fit1
```

```
fit1 =
     type: 'polynomial degree 3'
    coeff: [3.8555e-006 -0.0153 17.7815 -4.8519e+003]
```

You may want to use this structure for subsequent display or analysis. For example, you can use the saved coefficients and the `polyval` function to evaluate the cubic polynomial at the command line.

By selecting the ➡ button again, you can evaluate the current fit at the specified abscissa values. The current fit is displayed in the **Fit** menu. In this example, the population for the years 2000 to 2050 is evaluated in increments of 10, and then displayed in the data plot.

The evaluated data is shown below.

You can save the evaluated data to the MATLAB workspace by selecting the **Save to workspace** button.

Difference Equations and Filtering

MATLAB has functions for working with difference equations and filters. These functions operate primarily on vectors.

Vectors are used to hold sampled-data signals, or sequences, for signal processing and data analysis. For multi-input systems, each row of a matrix corresponds to a sample point with each input appearing as columns of the matrix.

The function

```
y = filter(b,a,x)
```

processes the data in vector x with the filter described by vectors a and b, creating filtered data y.

The `filter` command can be thought of as an efficient implementation of the difference equation. The filter structure is the general tapped delay-line filter described by the difference equation below, where n is the index of the current sample, na is the order of the polynomial described by vector a and nb is the order of the polynomial described by vector b. The output $y(n)$, is a linear combination of current and previous inputs, $x(n)\ x(n\text{-}1)$ ..., and previous outputs, $y(n\text{-}1)\ y(n\text{-}2)$ ...

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n - nb + 1)$$
$$- a(2)y(n-1) - \ldots - a(na)y(n - na + 1)$$

Suppose, for example, we want to smooth our traffic count data with a moving average filter to see the average traffic flow over a 4-hour window. This process is represented by the difference equation

$$y(n) = \frac{1}{4}x(n) + \frac{1}{4}x(n-1) + \frac{1}{4}x(n-2) + \frac{1}{4}x(n-3)$$

The corresponding vectors are

```
a = 1;
b = [1/4 1/4 1/4 1/4];
```

Note Enter the `format` command, `format rat`, to display and enter data using the rational format.

Executing the command

```
load count.dat
```

creates the matrix `count` in the workspace.

For this example, extract the first column of traffic counts and assign it to the vector `x`.

```
x = count(:,1);
```

The 4-hour moving-average of the data is efficiently calculated with

```
y = filter(b,a,x);
```

Compare the original data and the smoothed data with an overlaid plot of the two curves.

```
t = 1:length(x);
plot(t,x,'-.',t,y,'-'), grid on
legend('Original Data','Smoothed Data',2)
```

The filtered data represented by the solid line is the 4-hour moving average of the observed traffic count data represented by the dashed line.

For practical filtering applications, the Signal Processing Toolbox includes numerous functions for designing and analyzing filters.

Fourier Analysis and the Fast Fourier Transform (FFT)

Fourier analysis is extremely useful for data analysis, as it breaks down a signal into constituent sinusoids of different frequencies. For sampled vector data, Fourier analysis is performed using the discrete Fourier transform (DFT).

The fast Fourier transform (FFT) is an efficient algorithm for computing the DFT of a sequence; it is not a separate transform. It is particularly useful in areas such as signal and image processing, where its uses range from filtering, convolution, and frequency analysis to power spectrum estimation.

This section:

- Summarizes the Fourier transform functions
- Introduces Fourier transform analysis with an example about sunspot activity
- Calculates magnitude and phase of transformed data
- Discusses the dependence of execution time on length of the transform

Function Summary

MATLAB provides a collection of functions for computing and working with Fourier transforms.

FFT Function Summary

Function	Description
fft	Discrete Fourier transform.
fft2	Two-dimensional discrete Fourier transform.
fftn	N-dimensional discrete Fourier transform.
ifft	Inverse discrete Fourier transform.
ifft2	Two-dimensional inverse discrete Fourier transform.
ifftn	N-dimensional inverse discrete Fourier transform.
abs	Magnitude.

FFT Function Summary (Continued)

Function	Description
angle	Phase angle.
unwrap	Unwrap phase angle in radians.
fftshift	Move zeroth lag to center of spectrum.
cplxpair	Sort numbers into complex conjugate pairs.
nextpow2	Next higher power of two.

Introduction

For length N input sequence x, the DFT is a length N vector, X. `fft` and `ifft` implement the relationships

$$X(k) = \sum_{n=1}^{N} x(n) e^{-j2\pi(k-1)\left(\frac{n-1}{N}\right)} \qquad 1 \le k \le N$$

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} X(k) e^{j2\pi(k-1)\left(\frac{n-1}{N}\right)} \qquad 1 \le n \le N$$

Note Since the first element of a MATLAB vector has an index 1, the summations in the equations above are from 1 to N. These produce identical results as traditional Fourier equations with summations from 0 to N-1.

If $x(n)$ is real, we can rewrite the above equation in terms of a summation of sine and cosine functions with real coefficients

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} a(k)\cos\left(\frac{2\pi(k-1)(n-1)}{N}\right) + b(k)\sin\left(\frac{2\pi(k-1)(n-1)}{N}\right)$$

where

$$a(k) = \text{real}(X(k)), \ b(k) = -\text{imag}(X(k)), \ 1 \leq n \leq N$$

Finding an FFT

The FFT of a column vector x

```
x = [4 3 7 -9 1 0 0 0]' ;
```

is found with

```
y = fft(x)
```

which results in

```
y =
6.0000
11.4853 - 2.7574i
-2.0000 -12.0000i
-5.4853 +11.2426i
18.0000
-5.4853 -11.2426i
-2.0000 +12.0000i
11.4853 + 2.7574i
```

Notice that although the sequence x is real, y is complex. The first component of the transformed data is the constant contribution and the fifth element corresponds to the Nyquist frequency. The last three values of y correspond to negative frequencies and, for the real sequence x, they are complex conjugates of three components in the first half of y.

Example: Using FFT to Calculate Sunspot Periodicity

Suppose, we want to analyze the variations in sunspot activity over the last 300 years. You are probably aware that sunspot activity is cyclical, reaching a maximum about every 11 years. Let's confirm that.

Astronomers have tabulated a quantity called the Wolfer number for almost 300 years. This quantity measures both number and size of sunspots.

Load and plot the sunspot data.

```
load sunspot.dat
year = sunspot(:,1);
wolfer = sunspot(:,2);
plot(year,wolfer)
```

```
title('Sunspot Data')
```


Sunspot Data

Now take the FFT of the sunspot data.

```
Y = fft(wolfer);
```

The result of this transform is the complex vector, Y. The magnitude of Y squared is called the power and a plot of power versus frequency is a "periodogram." Remove the first component of Y, which is simply the sum of the data, and plot the results.

```
N = length(Y);
Y(1) = [];
power = abs(Y(1:N/2)).^2;
nyquist = 1/2;
freq = (1:N/2)/(N/2)*nyquist;
plot(freq,power), grid on
xlabel('cycles/year')
title('Periodogram')
```

The scale in cycles/year is somewhat inconvenient. Let's plot in years/cycle and estimate what one cycle is. For convenience, plot the power versus period (where period = 1./freq) from 0 to 40 years/cycle.

```
period = 1./freq;
plot(period,power), axis([O 4O O 2e7]), grid on
ylabel('Power')
xlabel('Period(Years/Cycle)')
```

In order to determine the cycle more precisely,

```
[mp,index] = max(power);
period(index)
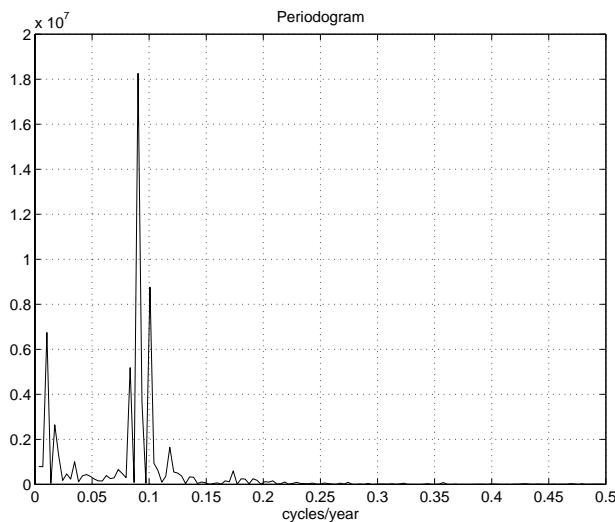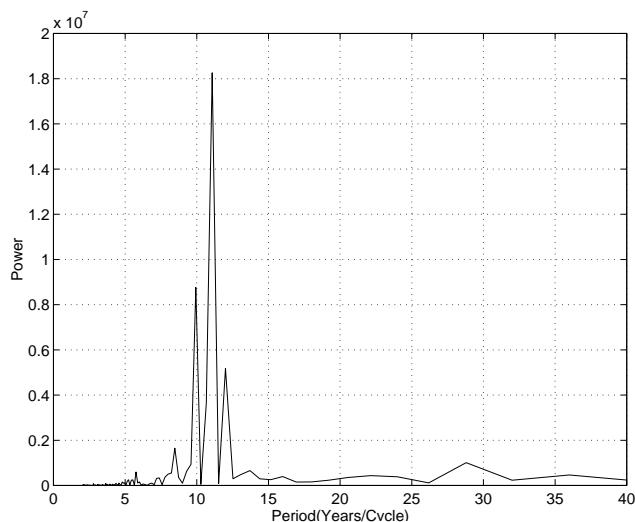
ans =
     11.0769
```

Magnitude and Phase of Transformed Data

Important information about a transformed sequence includes its magnitude and phase. The MATLAB functions abs and angle calculate this information.

To try this, create a time vector t, and use this vector to create a sequence x consisting of two sinusoids at different frequencies.

```
t = 0:1/100:10-1/100;
x = sin(2*pi*15*t) + sin(2*pi*40*t);
```

Now use the fft function to compute the DFT of the sequence. The code below calculates the magnitude and phase of the transformed sequence. It uses the abs function to obtain the magnitude of the data, the angle function to obtain the phase information, and unwrap to remove phase jumps greater than pi to their 2*pi complement.

```
y = fft(x);
m = abs(y);
p = unwrap(angle(y));
```

Now create a frequency vector for the *x*-axis and plot the magnitude and phase.

```
f = (0:length(y)-1)'*100/length(y);
subplot(2,1,1), plot(f,m),
ylabel('Abs. Magnitude'), grid on
subplot(2,1,2), plot(f,p*180/pi)
ylabel('Phase [Degrees]'), grid on
xlabel('Frequency [Hertz]')
```

The magnitude plot is perfectly symmetrical about the Nyquist frequency of 50 hertz. The useful information in the signal is found in the range 0 to 50 hertz.

FFT Length Versus Speed

You can add a second argument to fft to specify a number of points n for the transform

```
y = fft(x,n)
```

With this syntax, `fft` pads x with zeros if it is shorter than n, or truncates it if it is longer than n. If you do not specify n, `fft` defaults to the length of the input sequence.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

The inverse FFT function `ifft` also accepts a transform length argument.

For practical application of the FFT, the Signal Processing Toolbox includes numerous functions for spectral analysis.

13

Function Functions

This chapter introduces and explores the use of *function functions*, i.e., functions that accept one or more functions as input arguments. You can pass such a function either as a function handle or as an inline object that defines a mathematical function. The function that is passed in is referred to as the *objective function*. This chapter includes:

Function Summary (p. 13-2)	A summary of some function functions
Representing Functions in MATLAB (p. 13-3)	Some guidelines for representing functions in MATLAB
Plotting Mathematical Functions (p. 13-5)	A discussion about using `fplot` to plot mathematical functions
Minimizing Functions and Finding Zeros (p. 13-8)	A discussion of high-level function functions that perform optimization-related tasks
Numerical Integration (Quadrature) (p. 13-18)	A discussion of the MATLAB quadrature functions

See the "Differential Equations" and "Sparse Matrices" chapters for information about the use of other function functions.

For information about function handles, see the `function_handle` (@), `func2str`, and `str2func` reference pages, and the "Function Handles" section of "Programming and Data Types" in the MATLAB documentation.

Function Summary

The function functions are located in the MATLAB `funfun` directory.

This table provides a brief description of the functions discussed in this chapter. Related functions are grouped by category.

Function Summary

Category	Function	Description
Plotting	`fplot`	Plot function.
Optimization and zero finding	`fminbnd`	Minimize function of one variable with bound constraints.
	`fminsearch`	Minimize function of several variables.
	`fzero`	Find zero of function of one variable.
Numerical integration	`quad`	Numerically evaluate integral, adaptive Simpson quadrature.
	`quadl`	Numerically evaluate integral, adaptive Lobatto quadrature.
	`dblquad`	Numerically evaluate double integral.
	`triplequad`	Numerically evaluate triple integral.

Representing Functions in MATLAB

MATLAB can represent mathematical functions by expressing them as MATLAB functions in M-files or as inline objects. For example, consider the function

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6$$

This function can be used as input to any of the function functions.

As MATLAB Functions

You can find the function above in the M-file named humps.m.

```
function y = humps(x)
y = 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;
```

To evaluate the function humps at 2.0, use @ to obtain a function handle for humps, and then pass the function handle to feval.

```
fh = @humps;
feval(fh,2.0)

ans =
    -4.8552
```

As Inline Objects

A second way to represent a mathematical function at the command line is by creating an *inline* object from a string expression. For example, you can create an inline object of the humps function

```
f = inline( 1./((x-0.3).^2 + 0.01) + 1./((x-0.9).^2 + 0.04)-6 );
```

You can then evaluate f at 2.0.

```
f(2.0)
ans =
    -4.8552
```

You can also create functions of more than one argument with inline by specifying the names of the input arguments along with the string expression. For example, the following function has two input arguments x and y.

```
f= inline('y*sin(x)+x*cos(y)','x','y')
f(pi,2*pi)
ans =
    3.1416
```

Plotting Mathematical Functions

The fplot function plots a mathematical function between a given set of axes limits. You can control the *x*-axis limits only, or both the *x*- and *y*-axis limits. For example, to plot the humps function over the *x*-axis range [-5 5], use

```
fplot(@humps,[-5 5])
grid on
```


You can zoom in on the function by selecting *y*-axis limits of -10 and 25, using

```
fplot(@humps,[-5 5 -10 25])
grid on
```

You can also pass an inline for `fplot` to graph, as in

```
fplot(inline('2*sin(x+3)'),[-1 1])
```

You can plot more than one function on the same graph with one call to `fplot`. If you use this with a function, then the function must take a column vector `x` and return a matrix where each column corresponds to each function, evaluated at each value of `x`.

If you pass an inline object of several functions to `fplot`, the inline object also must return a matrix where each column corresponds to each function evaluated at each value of `x`, as in

```
fplot(inline('[2*sin(x+3), humps(x)]'),[-5 5])
```

which plots the first and second functions on the same graph.

Note that the inline

```
f= inline('[2*sin(x+3), humps(x)]')
```

evaluates to a matrix of two columns, one for each function, when x is a column vector.

```
f([1;2;3])
```

returns

```
-1.5136    16.0000
-1.9178    -4.8552
-0.5588    -5.6383
```

Minimizing Functions and Finding Zeros

MATLAB provides a number of high-level function functions that perform optimization-related tasks. This section describes:

• Minimizing a function of one variable
• Minimizing a function of several variables
• Setting minimization options
• Finding a zero of a function of one variable
• Converting your code to MATLAB Version 5 syntax

The MATLAB optimization functions are:

fminbnd   Minimize a function of one variable on a fixed interval

fminsearch   Minimize a function of several variables

fzero   Find zero of a function of one variable

lsqnonneg   Linear least squares with nonnegativity constraints

optimget   Get optimization options structure parameter values

optimset   Create or edit optimization options parameter structure

For more optimization capabilities, see the Optimization Toolbox.

Minimizing Functions of One Variable

Given a mathematical function of a single variable coded in an M-file, you can use the fminbnd function to find a local minimizer of the function in a given interval. For example, to find a minimum of the humps function in the range (0.3, 1), use

```
x = fminbnd(@humps,0.3,1)
```

which returns

```
x =
    0.6370
```

You can ask for a tabular display of output by passing a fourth argument created by the optimset command to fminbnd

```
x = fminbnd(@humps,0.3,1,optimset('Display','iter'))
```

which gives the output

```
Func-count       x          f(x)         Procedure
    1         0.567376     12.9098        initial
    2         0.732624     13.7746        golden
    3         0.465248     25.1714        golden
    4         0.644416     11.2693        parabolic
    5          0.6413      11.2583        parabolic
    6         0.637618     11.2529        parabolic
    7         0.636985     11.2528        parabolic
    8         0.637019     11.2528        parabolic
    9         0.637052     11.2528        parabolic

x =
    0.6370
```

This shows the current value of x and the function value at f(x) each time a function evaluation occurs. For fminbnd, one function evaluation corresponds to one iteration of the algorithm. The last column shows what procedure is being used at each iteration, either a golden section search or a parabolic interpolation.

Minimizing Functions of Several Variables

The fminsearch function is similar to fminbnd except that it handles functions of many variables, and you specify a starting vector x_0 rather than a starting interval. fminsearch attempts to return a vector x that is a local minimizer of the mathematical function near this starting vector.

To try fminsearch, create a function three_var of three variables, x, y, and z.

```
function b = three_var(v)
x = v(1);
y = v(2);
z = v(3);
b = x.^2 + 2.5*sin(y) - z^2*x^2*y^2;
```

Now find a minimum for this function using x = -0.6, y = -1.2, and z = 0.135 as the starting values.

```
v = [-0.6 -1.2 0.135];
```

```
a = fminsearch(@three_var,v)

a =
    0.0000    -1.5708     0.1803
```

Setting Minimization Options

You can specify control options that set some minimization parameters by calling fminbnd with the syntax

```
x = fminbnd(fun,x1,x2,options)
```

or fminsearch with the syntax

```
x = fminsearch(fun,x0,options)
```

options is a structure used by the optimization functions. Use optimset to set the values of the options structure.

```
options = optimset('Display','iter');
```

fminbnd and fminsearch use only the options parameters shown in the following table. See the optimset reference page for a complete list of the parameters that are used in the Optimization Toolbox.

options.Display	A flag that determines if intermediate steps in the minimization appear on the screen. If set to 'iter', intermediate steps are displayed; if set to 'off', no intermediate solutions are displayed, if set to final, displays just the final output.
options.TolX	The termination tolerance for x. Its default value is 1.e-4.
options.TolFun	The termination tolerance for the function value. The default value is 1.e-4. This parameter is used by fminsearch, but not fminbnd.

options.MaxIter   Maximum number of iterations allowed.

options.MaxFunEvals   The maximum number of function evaluations
                       allowed. The default value is 500 for fminbnd and
                       200*length(x0) for fminsearch.

The number of function evaluations, the number of iterations, and the
algorithm are returned in the structure output when you provide fminbnd or
fminsearch with a fourth output argument, as in

```
[x,fval,exitflag,output] = fminbnd(@humps,0.3,1);
```

or

```
[x,fval,exitflag,output] = fminsearch(@three_var,v);
```

Finding Zeros of Functions

The fzero function attempts to find a zero of one equation with one variable.
You can call this function with either a one-element starting point or a
two-element vector that designates a starting interval. If you give fzero a
starting point x_0, fzero first searches for an interval around this point where
the function changes sign. If the interval is found, then fzero returns a value
near where the function changes sign. If no such interval is found, fzero
returns NaN. Alternatively, if you know two points where the function value
differs in sign, you can specify this starting interval using a two-element
vector; fzero is guaranteed to narrow down the interval and return a value
near a sign change.

Use fzero to find a zero of the humps function near -0.2

```
a = fzero(@humps,-0.2)

a =
-0.1316
```

For this starting point, fzero searches in the neighborhood of -0.2 until it finds
a change of sign between -0.10949 and -0.264. This interval is then narrowed
down to -0.1316. You can verify that -0.1316 has a function value very close to
zero using

```
humps(a)
```

```
ans =
    8.8818e -16
```

Suppose you know two places where the function value of humps differs in sign such as x = 1 and x = -1. You can use

```
humps(1)

ans =
    16

humps(-1)

ans =
    -5.1378
```

Then you can give fzero this interval to start with and fzero then returns a point near where the function changes sign. You can display information as fzero progresses with

```
options = optimset('Display','iter');
a = fzero(@humps,[-1 1],options)

 Func-count        x              f(x)             Procedure

     1              -1          -5.13779           initial
     1               1               16            initial
     2        -0.513876         -4.02235           interpolation
     3         0.243062          71.6382           bisection
     4        -0.473635         -3.83767           interpolation
     5        -0.115287          0.414441          bisection
     6        -0.150214         -0.423446          interpolation
     7        -0.132562         -0.0226907         interpolation
     8        -0.131666         -0.0011492         interpolation
     9        -0.131618          1.88371e-07       interpolation
    10        -0.131618         -2.7935e-11        interpolation
    11        -0.131618          8.88178e-16       interpolation
    12        -0.131618         -9.76996e-15       interpolation

a =
    -0.1316
```

The steps of the algorithm include both bisection and interpolation under the `Procedure` column. If the example had started with a scalar starting point instead of an interval, the first steps after the initial function evaluations would have included some search steps while `fzero` searched for an interval containing a sign change.

You can specify a relative error tolerance using `optimset`. In the call above, passing in the empty matrix causes the default relative error tolerance of `eps` to be used.

Tips

Optimization problems may take many iterations to converge. Most optimization problems benefit from good starting guesses. Providing good starting guesses improves the execution efficiency and may help locate the global minimum instead of a local minimum.

Sophisticated problems are best solved by an evolutionary approach, whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

Troubleshooting

Below is a list of typical problems and recommendations for dealing with them.

Problem	Recommendation
The solution found by fminbnd or fminsearch does not appear to be a global minimum.	There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points (or intervals in the case of fminbnd) may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.
Sometimes an optimization problem has values of x for which it is impossible to evaluate f.	Modify your function to include a penalty function to give a large positive value to f when infeasibility is encountered.
The minimization routine appears to enter an infinite loop or returns a solution that is not a minimum (or not a zero in the case of fzero).	Your objective function (fun) may be returning Inf, NaN, or complex values. The optimization routines expect only real numbers to be returned. Any other values may cause unexpected results. Insert code into your objective function M-file to verify that only real numbers are returned (use the functions isreal and isfinite).

Converting Your Optimization Code to MATLAB Version 5 Syntax

Most of the function names and calling sequences changed in Version 5 to accommodate new functionality and to clarify the roles of the input and output variables.

This table lists the optimization functions provided by MATLAB and indicates the functions whose names have changed in Version 5.

Old (Version 4) Name	New (Version 5) Name
fmin	fminbnd
fmins	fminsearch

Old (Version 4) Name	New (Version 5) Name
`foptions`	`optimget`, `optimset`
`fzero`	`fzero` (name unchanged)
`nnls`	`lsqnonneg`

This section:

- Tells you how to override default parameter settings with the new `optimset` and `optimget` functions.
- Explains the reasons for the new calling sequences and explains how to convert your code.

In addition to the information in this section, consult the individual function reference pages for information about the new functions and about the arguments they take.

Using optimset and optimget

The `optimset` function replaces `foptions` for overriding default parameter settings. `optimset` creates an `options` structure that contains parameters used in the optimization routines. If, on the first call to an optimization routine, the `options` structure is not provided, or is empty, a set of default parameters is generated. See the `optimset` reference page for details.

New Calling Sequences

Version 5 of MATLAB makes these changes in the calling sequences:

- Each function takes an `options` structure to adjust parameters to the optimization functions (see `optimset`, `optimget`).
- The new default output gives information if the function does not converge. (the Version 4 default was no output, Version 5 used `'final'` as the default, the new default is `options.display = 'notify'`).
- Each function returns an `exitflag` that describes the termination state.
- Each function now has an `output` structure that contains information about the problem solution relevant to that function.

The sections below describe how to convert from the old function names and calling sequences to the new ones. The calls shown are the most general cases,

involving all possible input and output arguments. Note that many of these arguments are optional. See the function reference pages for more information.

Converting from fmin to fminbnd. In Version 4, you used this call to fmin.

```
[X,OPTIONS] = fmin('FUN',x1,x2,OPTIONS,P1,P2,...);
```

In Version 5, you call fminbnd like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fminbnd(@FUN,x1,x2,...
                                   OPTIONS,P1,P2,...);
```

Converting from fmins to fminsearch. In Version 4, you used this call to fmins.

```
[X,OPTIONS] = fmins('FUN',x0,OPTIONS,[],P1,P2,...);
```

In Version 5, you call fminsearch like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fminsearch(@FUN,x0,...
                                   OPTIONS,P1,P2,...);
```

Converting to the new form of fzero. In Version 4, you used this call to fzero.

```
X = fzero('F',X,TOL,TRACE,P1,P2,...);
```

In Version 5, replace the TRACE and TOL arguments with

```
if TRACE == 0,
   val = 'none';
elseif TRACE == 1
   val = 'iter';
end
OPTIONS = optimset('Display',val,'TolX',TOL);
```

Now call fzero like this.

```
[X,FVAL,EXITFLAG,OUTPUT] = fzero(@F,X,OPTIONS,P1,P2,...);
```

Converting from nnls to lsqnonneg. In Version 4, you used this call to nnls.

```
[X,LAMBDA] = nnls(A,b,tol);
```

In Version 5, replace the tol argument with

```
OPTIONS = optimset('Display','none','TolX',tol);
```

Now call lsqnonneg like this.

```
[X,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA] =
    lsqnonneg(A,b,XO,OPTIONS);
```

Numerical Integration (Quadrature)

The area beneath a section of a function $F(x)$ can be determined by numerically integrating $F(x)$, a process referred to as *quadrature*. The MATLAB quadrature functions are:

`quad`	Use adaptive Simpson quadrature
`quadl`	Use adaptive Lobatto quadrature
`dblquad`	Numerically evaluate double integral
`triplequad`	Numerically evaluate triple integral

To integrate the function defined by `humps.m` from `0` to `1`, use

```
q = quad(@humps,0,1)

q =
    29.8583
```

Both `quad` and `quadl` operate recursively. If either method detects a possible singularity, it prints a warning.

You can include a fourth argument for `quad` or `quadl` that specifies a relative error tolerance for the integration. If a nonzero fifth argument is passed to `quad` or `quadl`, the function evaluations are traced.

Two examples illustrate use of these functions:

- Computing the length of a curve
- Double integration

Example: Computing the Length of a Curve

You can use `quad` or `quadl` to compute the length of a curve. Consider the curve parameterized by the equations

$$x(t) = \sin(2t), \qquad y(t) = \cos(t), \qquad z(t) = t$$

where $t \in [0, 3\pi]$.

A three-dimensional plot of this curve is

```
t = 0:0.1:3*pi;
```

```
plot3(sin(2*t),cos(t),t)
```

The arc length formula says the length of the curve is the integral of the norm of the derivatives of the parameterized equations

$$\int_{0}^{3\pi} \sqrt{4\cos(2t)^2 + \sin(t)^2 + 1}\ \ dt$$

The function hcurve computes the integrand

```
function f = hcurve(t)
f = sqrt(4*cos(2*t).^2 + sin(t).^2 + 1);
```

Integrate this function with a call to quad

```
len = quad(@hcurve,0,3*pi)

len =
    1.7222e+01
```

The length of this curve is about 17.2.

Example: Double Integration

Consider the numerical solution of

$$\int_{ymin}^{ymax} \int_{xmin}^{xmax} f(x,y)\ \ dxdy$$

For this example $f(x,y) = y\sin(x) + x\cos(y)$. The first step is to build the function to be evaluated. The function must be capable of returning a vector output when given a vector input. You must also consider which variable is in the inner integral, and which goes in the outer integral. In this example, the inner variable is x and the outer variable is y (the order in the integral is $dxdy$). In this case, the integrand function is

```
function out = integrnd(x,y)
out = y*sin(x) + x*cos(y);
```

To perform the integration, two functions are available in the funfun directory. The first, dblquad, is called directly from the command line. This M-file

evaluates the outer loop using quad. At each iteration, quad calls the second helper function that evaluates the inner loop.

To evaluate the double integral, use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax);
```

The first argument is a string with the name of the integrand function. The second to fifth arguments are

xmin   Lower limit of inner integral

xmax   Upper limit of the inner integral

ymin   Lower limit of outer integral

ymax   Upper limit of the outer integral

Here is a numeric example that illustrates the use of dblquad.

```
xmin = pi;
xmax = 2*pi;
ymin = 0;
ymax = pi;
result = dblquad(@integrnd,xmin,xmax,ymin,ymax)
```

The result is -9.8698.

By default, dblquad calls quad. To integrate the previous example using quadl (with the default values for the tolerance argument), use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax,[],@quadl);
```

Alternatively, you can pass any user-defined quadrature function name to dblquad as long as the quadrature function has the same calling and return arguments as quad.

14

Differential Equations

This chapter treats the numerical solution of differential equations using MATLAB. It includes:

Initial Value Problems for ODEs and DAEs (p. 14-2)	Describes the solution of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs), where the solution of interest satisfies initial conditions at a given initial value of the independent variable.
Initial Value Problems for DDEs (p. 14-57)	Describes the solution of delay differential equations (DDEs) where the solution of interest satisfies initial conditions at a given initial value of the independent variable.
Boundary Value Problems for ODEs (p. 14-77)	Describes the solution of ODEs, where the solution of interest satisfies certain boundary conditions . The boundary conditions specify a relationship between the values of the solution at the initial and final values of the independent variable.
Partial Differential Equations (p. 14-108)	Describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs) in one spatial variable and time.
Selected Bibliography (p. 14-125)	Lists published materials that support concepts described in this chapter.

Note In function tables, commonly used functions are listed first, followed by more advanced functions. The same is true of property tables.

Initial Value Problems for ODEs and DAEs

This section describes how to use MATLAB to solve initial value problems (IVPs) of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs). It provides:

- A summary of the ODE solvers, related functions, and examples
- An introduction to ODEs and initial value problems
- Descriptions of the ODE solvers and their basic syntax
- General instructions for representing an IVP
- A discussion about changing default integration properties
- Examples of the kinds of IVP problems you can solve
- Answers to some frequently asked questions about the ODE solvers, and some troubleshooting suggestions

ODE Function Summary

Initial Value ODE Problem Solvers

These are the initial value problem solvers. The table lists the kind of problem you can solve with each solver, and the method each solver uses.

Solver	Solves These Kinds of Problems	Method
ode45	Nonstiff differential equations	Runge-Kutta
ode23	Nonstiff differential equations	Runge-Kutta
ode113	Nonstiff differential equations	Adams
ode15s	Stiff differential equations and DAEs	NDFs (BDFs)
ode23s	Stiff differential equations	Rosenbrock
ode23t	Moderately stiff differential equations and DAEs	Trapezoidal rule
ode23tb	Stiff differential equations	TR-BDF2

ODE Solution Evaluation

If you call an ODE solver with one output argument, it returns a structure that you can use to evaluate the solution at any point on the interval of integration.

Function	Description
deval	Evaluate the numerical solution using output of ODE solvers.

ODE Solver Properties Handling

An options structure contains named integration properties whose values are passed to the solver, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
odeset	Create or alter options structure for input to ODE solvers.
odeget	Extract properties from options structure created with odeset.

ODE Solver Output Functions

If an output function is specified, the solver calls the specified function after every successful integration step. You can use odeset to specify one of these sample functions as the OutputFcn property, or you can modify them to create your own functions.

Function	Description
odeplot	Time-series plot
odephas2	Two-dimensional phase plane plot
odephas3	Three-dimensional phase plane plot
odeprint	Print to command window

ODE Initial Value Problem Examples

These examples illustrate the kinds of problems you can solve in MATLAB. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the ODE examples and DAE examples. You can also run the examples from the browser. Click on these links to invoke the browser, or type `odeexamples('ode')`or `odeexamples('dae')`at the command line.

Example	Description
amp1dae	Stiff DAE – electrical circuit
ballode	Simple event location – bouncing ball
batonode	ODE with time- and state-dependent mass matrix – motion of a baton
brussode	Stiff large problem – diffusion in a chemical reaction (the Brusselator)
burgersode	ODE with strongly state-dependent mass matrix – Burger's equation solved using a moving mesh technique
fem1ode	Stiff problem with a time-dependent mass matrix – finite element method
fem2ode	Stiff problem with a constant mass matrix – finite element method
hb1dae	Stiff DAE from a conservation law
hb1ode	Stiff problem solved on a very long interval – Robertson chemical reaction
orbitode	Advanced event location – restricted three body problem

Example	Description
rigidode	Nonstiff problem – Euler equations of a rigid body without external forces
vdpode	Parameterizable van der Pol equation (stiff for large μ)

Introduction to Initial Value ODE Problems

What Is an Ordinary Differential Equation?

The ODE solvers are designed to handle *ordinary differential equations*. An ordinary differential equation contains one or more derivatives of a dependent variable y with respect to a single independent variable t, usually referred to as *time*. The derivative of y with respect to t is denoted as y', the second derivative as y'', and so on. Often $y(t)$ is a vector, having elements $y_1, y_2, \ldots, y_n$.

Using Initial Conditions to Specify the Solution of Interest

Generally there are many functions $y(t)$ that satisfy a given ODE, and additional information is necessary to specify the solution of interest. In an *initial value problem,* the solution of interest satisfies a specific *initial condition*, that is, y is equal to y_0 at a given initial time t_0. An initial value problem for an ODE is then

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

(14-1)

If the function $f(t, y)$ is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate $y(t)$ by numerical means, such as using one of the ODE solvers.

Working with Higher-Order ODEs

The ODE solvers accept only first-order differential equations. However, ODEs often involve a number of dependent variables, as well as derivatives of order higher than one. To use the ODE solvers, you must rewrite such equations as an equivalent system of first-order differential equations of the form

$$y' = f(t, y)$$

You can write any ordinary differential equation

$$y^{(n)} = f(t, y, y', ..., y^{(n-1)})$$

as a system of first-order equations by making the substitutions

$$y_1 = y, \quad y_2 = y', \quad ..., \quad y_n = y^{(n-1)}$$

The result is an equivalent system of n first-order ODEs.

$$y_1' = y_2$$
$$y_2' = y_3$$
$$\vdots$$
$$y_n' = f(t, y_1, y_2, ..., y_n)$$

"Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 14-11 rewrites the second-order van der Pol equation

$$y_1'' - \mu(1 - y_1^2) y_1' + y_1 = 0$$

as a system of first-order ODEs.

Initial Value Problem Solvers

The ODE solver functions implement numerical integration methods for solving IVPs for ODEs (Equation 14-1). Beginning at the initial time with initial conditions, they step through the time interval, computing a solution at each time step. If the solution for a time step satisfies the solver's error tolerance criteria, it is a successful step. Otherwise, it is a failed attempt; the solver shrinks the step size and tries again.

This section describes:

• Solvers for nonstiff ODE problems and solvers for stiff ODE problems
• ODE solver basic syntax
• Additional ODE solver arguments

"Mass Matrix and DAE Properties" on page 14-27 explains how to solve more general problems.

Solvers for Nonstiff Problems

There are three solvers designed for nonstiff problems:

ode45   Based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, ode45 is the best function to apply as a "first try" for most problems.

ode23   Based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of mild stiffness. Like ode45, ode23 is a one-step solver.

ode113   Variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances and when the ODE function is particularly expensive to evaluate. ode113 is a *multistep* solver – it normally needs the solutions at several preceding time points to compute the current solution.

Solvers for Stiff Problems

Not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Solvers for stiff problems can be used exactly like the other solvers. However, you can often significantly improve the efficiency of these solvers by providing them with additional information about the problem. (See "Changing ODE Integration Properties" on page 14-16.)

There are four solvers designed for stiff problems:

ode15s   Variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs, (also known as Gear's method). Like `ode113`, `ode15s` is a multistep solver. If you suspect that a problem is stiff or if `ode45` failed or was very inefficient, try `ode15s`.

ode23s   Based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective.

ode23t   An implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.

ode23tb   An implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order 2. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances.

ODE Solver Basic Syntax

All of the ODE solver functions share a syntax that makes it easy to try any of the different numerical methods if it is not apparent which is the most appropriate. To apply a different method to the same problem, simply change the ODE solver function name. The simplest syntax, common to all the solver functions, is

```
[t,y] = solver(odefun,tspan,y0)
```

where *solver* is one of the ODE solver functions listed previously.

The basic input arguments are:

odefun   Function that evaluates the system of ODEs. It has the form

              dydt = odefun(t,y)

          where t is a scalar, and dydt and y are column vectors.

tspan   Vector specifying the interval of integration. The solver imposes
          the initial conditions at tspan(1), and integrates from tspan(1) to
          tspan(end).

          For tspan vectors with two elements [t0 tf], the solver returns
          the solution evaluated at every integration step. For tspan vectors
          with more than two elements, the solver returns solutions
          evaluated at the given time points. The time values must be in
          order, either increasing or decreasing.

          Specifying tspan with more than two elements does not affect the
          internal time steps that the solver uses to traverse the interval
          from tspan(1) to tspan(end). All solvers in the ODE suite obtain
          output values by means of continuous extensions of the basic
          formulas. Although a solver does not necessarily step precisely to a
          time point specified in tspan, the solutions produced at the
          specified time points are of the same order of accuracy as the
          solutions computed at the internal time points.

          Specifying tspan with more than two elements has little effect on
          the efficiency of computation, but for large systems, affects
          memory management.

y0   Vector of initial conditions for the problem

          See also "Introduction to Initial Value ODE Problems" on
          page 14-5.

The output arguments are:

t   Column vector of time points

y   Solution array. Each row in y corresponds to the solution at a time
          returned in the corresponding row of t.

Additional ODE Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional problem parameters.

options   Structure of optional parameters that change the default integration properties. This is the fourth input argument.

```
[t,y] = solver(odefun,tspan,y0,options)
```

"Changing ODE Integration Properties" on page 14-16 tells you how to create the structure and describes the properties you can specify.

p1,p2...   Parameters that the solver passes to odefun.

```
[t,y] = solver(odefun,tspan,y0,options,p1,p2...)
```

The solver passes any input parameters that follow the options argument to odefun and any functions you specify in options. Use options = [] as a placeholder if you set no options. The function odefun must have the form

```
dydt = odefun(t,y,p1,p2,...)
```

See "Passing Additional Parameters to an ODE Function" on page 14-13 for an example.

Solving ODE Problems

This section uses the van der Pol equation

$$y_1'' - \mu(1 - y_1^2) y_1' + y_1 = 0$$

to describe the process for solving initial value ODE problems using the ODE solvers.

- "Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 14-11 describes each step of the process. Because the van der Pol equation is a second-order equation, the example must first rewrite it as a system of first order equations.

- "Example: The van der Pol Equation, $\mu = 1000$ (Stiff)" on page 14-14 demonstrates the solution of a stiff problem.
- "Evaluating the Solution at Specific Points" on page 14-15 tells you how to evaluate the solution at specific points.

Note See "ODE Solver Basic Syntax" on page 14-8 for more information.

Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)

This example explains and illustrates the steps you need to solve an initial value ODE problem.

1 Rewrite the problem as a system of first-order ODEs. Rewrite the van der Pol equation (second-order)

$$y_1'' - \mu(1 - y_1^2)y_1' + y_1 = 0$$

where $\mu > 0$ is a scalar parameter, by making the substitution $y'_1 = y_2$. The resulting system of first-order ODEs is

$$y'_1 = y_2$$
$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

See "Working with Higher-Order ODEs" on page 14-5 for more information.

2 Code the system of first-order ODEs. Once you represent the equation as a system of first-order ODEs, you can code it as a function that an ODE solver can use. The function must be of the form

```
dydt = odefun(t,y)
```

Although t and y must be the function's first two arguments, the function does not need to use them. The output dydt, a column vector, is the derivative of y.

The code below represents the van der Pol system in the function, vdp1. The vdp1 function assumes that $\mu = 1$. y_1 and y_2 become elements y(1) and y(2) of a two-element vector.

```
function dydt = vdp1(t,y)
dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Note that, although vdp1 must accept the arguments t and y, it does not use t in its computations.

3 Apply a solver to the problem. Decide which solver you want to use to solve the problem. Then call the solver and pass it the function you created to describe the first-order system of ODEs, the time interval on which you want to solve the problem, and an initial condition vector. See "Initial Value Problem Solvers" on page 14-6 and the ODE solver reference page for descriptions of the ODE solvers.

For the van der Pol system, you can use ode45 on time interval [0 20] with initial values y(1) = 2 and y(2) = 0.

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

This example uses @ to pass vdp1 as a function handle to ode45. The resulting output is a column vector of time points t and a solution array y. Each row in y corresponds to a time returned in the corresponding row of t. The first column of y corresponds to y_1, and the second column to y_2.

Note See the function_handle (@), func2str, and str2func reference pages, and the Function Handles chapter of "Programming and Data Types" in the MATLAB documentation for information about function handles.

---

4 View the solver output. You can simply use the plot command to view the solver output.

```
plot(t,y(:,1),'-',t,y(:,2),'--')
title('Solution of van der Pol Equation, \mu = 1');
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2')
```

As an alternative, you can use a solver output function to process the output. The solver calls the function specified in the integration property OutputFcn after each successful time step. Use odeset to set OutputFcn to the desired function. See "OutputFcn" on page 14-20 for more information.

Passing Additional Parameters to an ODE Function

The solver passes any input parameters that follow the options argument to the ODE function and any function you specify in options. For example:

1 Generalize the van der Pol function by passing it a mu parameter, instead of specifying a value for mu explicitly in the code.

```
function dydt = vdp1(t,y,mu)
dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
```

2 Pass the parameter mu to the function vdp1 by specifying it after the options argument in the call to the solver. This example uses options = [] as a placeholder.

```
[t,y] = ode45(@vdp1,tspan,y0,[],mu)
```

calls

```
vdp1(t,y,mu)
```

See the vdpode code for a complete example based on these functions.

Example: The van der Pol Equation, μ = 1000 (Stiff)

This example presents a *stiff* problem. For a stiff problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change.

When μ is increased to 1000, the solution to the van der Pol equation changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes a more difficult task. Because this particular problem is stiff, a solver intended for nonstiff problems, such as ode45, is too inefficient to be practical. A solver such as ode15s is intended for such stiff problems.

The vdp1000 function evaluates the van der Pol system from the previous example, but with μ = 1000.

```
function dydt = vdp1000(t,y)
dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

Note This example hardcodes μ in the ODE function. The vdpode example solves the same problem, but passes a user-specified μ as an additional argument to the ODE function. See "Additional ODE Solver Arguments" on page 14-10.

Now use the ode15s function to solve the problem with the initial condition vector of [2; 0], but a time interval of [0 3000]. For scaling purposes, plot just the first component of y(t).

```
[t,y] = ode15s(@vdp1000,[0 3000],[2; 0]);
plot(t,y(:,1),'-');
```

```
title('Solution of van der Pol Equation, \mu = 1000');
xlabel('time t');
ylabel('solution y_1');
```


Note For detailed instructions for solving an initial value ODE problem, see "Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 14-11.

Evaluating the Solution at Specific Points

The numerical methods implemented in the ODE solvers produce a continuous solution over the interval of integration $[a, b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a, b]$ using the function deval and the structure sol returned by the solver.

```
Sxint = deval(sol,xint)
```

The ODE solvers return the structure `sol` when called with a single output argument.

The `deval` function is vectorized. For a vector `xint`, the `i`th column of `Sxint` approximates the solution $y(\texttt{xint(i)})$.

Changing ODE Integration Properties

The default integration properties in the ODE solvers are selected to handle common problems. In some cases, you can improve ODE solver performance by overriding these defaults. You do this by supplying the solvers with one or more property values in an options structure.

```
[t,y] = solver(odefun,tspan,y0,options)
```

This section:

- Explains how to create, modify, and query an options structure.
- Describes the properties that you can use in an options structure.

In this and subsequent property tables, the most commonly used properties are listed first, followed by more advanced properties.

ODE Property Categories

Properties Category	Property Name
Error control	RelTol, AbsTol, NormControl
Solver output	OutputFcn, OutputSel, Refine, Stats
Jacobian matrix	Jacobian, JPattern, Vectorized
Step-size	InitialStep, MaxStep
Mass matrix and DAEs	Mass, MStateDependence, MvPattern, MassSingular, InitialSlope
Event location	Events
ode15s-specific	MaxOrder, BDF

Creating and Maintaining an ODE Options Structure

Creating an Options Structure. The `odeset` function creates an options structure that you can pass as an argument to any of the ODE solvers. To create an options structure, `odeset` accepts property name/property value pairs using the syntax

```
options = odeset('name1',value1,'name2',value2,...)
```

In the resulting structure, `options`, the named properties have the specified values. Any unspecified properties contain default values in the solvers. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. `odeset` ignores case for property names.

With no input arguments, `odeset` displays all property names and their possible values. It indicates defaults with {}.

Modifying an Existing Options Structure. To modify an existing options structure, `oldopts`, use

```
options = odeset(oldopts,'name1',value1,...)
```

This sets `options` equal to the existing structure `oldopts`, overwrites any values in `oldopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument. In the same way, the command

```
options = odeset(oldopts,newopts)
```

combines the structures `oldopts` and `newopts`. In the output argument, any values in the second argument overwrite those in the first argument.

Querying Options. The `odeget` function extracts property values from an options structure created with `odeset`.

```
o = odeget(options,'name')
```

This functions returns the value of the specified property, or an empty matrix [], if the property value is unspecified in the `options` structure.

As with `odeset`, it is sufficient to type only the leading characters that uniquely identify the property name. Case is ignored for property names.

Error Control Properties

At each step, the solver estimates the local error e in the ith component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, RelTol, and the specified absolute tolerance, AbsTol.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust RelTol. For the absolute error tolerance, the scaling of the solution components is important: if |y| is somewhat smaller than AbsTol, the solver is not constrained to obtain any correct digits in y. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want RelTol correct digits in all solution components except those smaller than thresholds AbsTol(i). Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components

The following table describes the error control properties. Use `odeset` to set the properties.

ODE Error Control Properties

Property	Value	Description
RelTol	Positive scalar {1e-3}	A relative error tolerance that applies to all components of the solution vector y. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds AbsTol(i). The default, 1e-3, corresponds to 0.1% accuracy.
AbsTol	Positive scalar or vector {1e-6}	Absolute error tolerances that apply to the individual components of the solution vector. AbsTol(i) is a threshold below which the value of the ith solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components. If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector y. If AbsTol is a scalar, the value applies to all components of y.
NormControl	on | {off}	Control error relative to norm of solution. Set this property on to request that the solvers control the error in each integration step with norm(e) <= max(RelTol*norm(y),AbsTol). By default the solvers use a more stringent component-wise error control.

Solver Output Properties

The solver output properties let you control the output that the solvers generate. Use odeset to set these properties.

ODE Solver Output Properties

Property	Value	Description
OutputFcn	Function {odeplot}	Installable output function. The solver calls this function after every successful integration step.
		For example,
		```options = odeset('OutputFcn',@myfun)```
		sets the OutputFcn property to an output function, myfun, that can be passed to an ODE solver.
		The output function must be of the form
		```status = myfun(t,y,flag,p1,p2,...)```
		The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:
		init The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan and y0 are the input arguments to the ODE solver.

ODE Solver Output Properties (Continued)

Property	Value	Description
	{none}	The solver calls status = myfun(t,y) after each integration step on which output is requested. t contains points where output was generated during the step, and y is the numerical solution at the points in t. If t is a vector, the ith column of y corresponds to the ith element of t.
		When length(tspan) > 2 the output is produced at every point in tspan. When length(tspan) = 2 the output is produced according to the Refine option.
		myfun must return a status output value of 0 or 1. If status = 1, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
	done	The solver calls myfun([],[],'done') when integration is complete to allow the output function to perform any cleanup chores.
		You can use these general purpose output functions or you can edit them to create your own. Type help function at the command line for more information. • odeplot – time series plotting (default when you call the solver with no output arguments and you have not specified an output function) • odephas2 – two-dimensional phase plane plotting • odephas3 – three-dimensional phase plane plotting • odeprint – print solution as it is computed **Note** If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

ODE Solver Output Properties (Continued)

Property	Value	Description
OutputSel	Vector of indices	Vector of indices specifying which components of the solution vector are to be passed to the output function. For example, if you want to use the odeplot output function, but you want to plot only the first and third components of the solution, you can do this using ```options = odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);``` By default, the solver passes all components of the solution to the output function.
Refine	Positive integer	Increases the number of output points by a factor of Refine. If Refine is 1, the solver returns solutions only at the end of each time step. If Refine is n >1, the solver subdivides each time step into n smaller intervals, and returns solutions at each time point. Refine does not apply when length(tspan)>2. **Note** In all the solvers, the default value of Refine is 1. Within ode45, however, the default is 4 to compensate for the solver's large step sizes. To override this and see only the time steps chosen by ode45, set Refine to 1. The extra values produced for Refine are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

ODE Solver Output Properties (Continued)

Property	Value	Description
Stats	on | {off}	Specifies whether the solver should display statistics about its computations. By default, Stats is off. If it is on, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the ODE function was called to evaluate $f(t, y)$ • The number of times that the partial derivatives matrix $\partial f/\partial y$ was formed • The number of LU decompositions • The number of solutions of linear systems

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\partial f/\partial y$, a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \dfrac{\partial f_1}{\partial y_1} & \dfrac{\partial f_1}{\partial y_2} & \cdots \\[2mm] \dfrac{\partial f_2}{\partial y_1} & \dfrac{\partial f_2}{\partial y_2} & \cdots \\[2mm] \vdots & \vdots & \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (ode15s, ode23s, ode23t, and ode23tb) for which the Jacobian matrix $\partial f/\partial y$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you may want to use the Vectorized, or JPattern properties.

The following table describes the Jacobian matrix properties. Use `odeset` to set these properties.

ODE Jacobian Matrix Properties

Property	Value	Description
Jacobian	Function | constant matrix	A constant matrix or a function that evaluates the Jacobian. Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function FJac, where FJac(t,y) computes $\partial f / \partial y$, or to the constant value of $\partial f / \partial y$.
		The Jacobian for the stiff van der Pol problem shown above can be coded as
		``` function J = vdp1000jac(t,y) J = [0 1 (-2000*y(1)*y(2)-1) (1000*(1-y(1)^2))]; ```
JPattern	Sparse matrix of {0,1}	Sparsity pattern with 1s where there might be nonzero entries in the Jacobian. It is used to generate a sparse Jacobian matrix numerically.
		Set this property to a sparse matrix S with $S(i,j) = 1$ if component i of $f(t, y)$ depends on component j of y, and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly accelerate execution. For an example using the JPattern property, see "Example: Large, Stiff Sparse Problem" on page 14-38 (brussode).

ODE Jacobian Matrix Properties (Continued)

Property	Value	Description
Vectorized	on | {off}	Set on to inform the solver that you have coded the ODE function F so that F(t,[y1 y2 ...]) returns [F(t,y1) F(t,y2) ...]. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and may significantly reduce solution time.
		With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, the stiff van der Pol example shown previously can be vectorized by introducing colon notation into the subscripts and by using the array power (.^) and array multiplication (.*) operators.
		``` function dydt = vdp1000(t,y) dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)]; ```
		Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the BVP solver, bvp4c. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while bvp4c requires vectorization with respect the first and second arguments.

Step-Size Properties

The step-size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Use `odeset` to set these properties.

ODE Step Size Properties

Property	Value	Description
InitialStep	Positive scalar	Suggested initial step size. `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.
MaxStep	Positive scalar {0.1*abs(t0-tf)}	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep`: • To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost. • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. (See "Error Control Properties" on page 14-18 for a description of the error tolerance properties.)

ODE Step Size Properties (Continued)

Property	Value	Description
		• To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solvers twice. If you do not know the time at which the change occurs, try reducing the error tolerances RelTol and AbsTol. Use MaxStep as a last resort.

Mass Matrix and DAE Properties

The solvers of the ODE suite can solve ODEs of the form

$$M(t, y)\, y' \,=\, f(t, y) \tag{14-2}$$

with a mass matrix $M(t, y)$ that can be sparse.

When $M(t, y)$ is nonsingular, the equation above is equivalent to $y' \,=\, M^{-1} f(t, y)$ and the ODE has a solution for any initial values y_0 at t_0. The more general form (Equation 14-2) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M(t, y)$, solving Equation 14-2 directly reduces the storage and runtime needed to solve the problem.

When $M(t, y)$ is singular, then $M(t, y)\, y' \,=\, f(t, y)$ is a differential-algebraic equation (DAE). A DAE has a solution only when y_0 is consistent, that is, there exists an initial slope yp_0 such that $M(t_0, y_0) yp_0 \,=\, f(t_0, y_0)$. If y_0 and yp_0 are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The ode15s and ode23t solvers can solve DAEs of index 1. For examples of DAE problems, see hb1dae ("Example: Differential-Algebraic Problem" on page 14-47) and amp1dae.

The following table describes the mass matrix and DAE properties. Use odeset to set these properties.

ODE Mass Matrix and DAE Properties

Property	Value	Description
Mass	Constant matrix | function	Constant mass matrix or a function that evaluates the mass matrix $M(t, y)$. For problems $My' = f(t, y)$ set this property to the value of the constant mass matrix M. For problems $M(t, y) y' = f(t, y)$, set this property to a function Mfun, where Mfun(t,y) evaluates the mass matrix $M(t, y)$. When solving DAEs, it is advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE). The ode23s solver can only solve problems with a constant mass matrix M. For example problems, see fem1ode ("Example: Finite Element Discretization" on page 14-35), fem2ode, or batonode.
MStateDependence	none | {weak} | strong	Dependence of the mass matrix on y. Set this property to none for problems $M(t) y' = f(t, y)$. Both weak and strong indicate $M(t, y)$, but weak results in implicit solvers using approximations when solving algebraic equations.
MvPattern	Sparse matrix	$\partial(M(t, y)v)/\partial y$ sparsity pattern. Set this property to a sparse matrix S with $S(i, j) = 1$ if for any k, the (i, k) component of $M(t, y)$ depends on component j of y, and 0 otherwise. For use with the ode15s, ode23t, and ode23tb solvers when MStateDependence is strong. See burgersode as an example.

ODE Mass Matrix and DAE Properties (Continued)

Property	Value	Description
MassSingular	yes | no | {maybe}	Indicates whether the mass matrix is singular. Set this property to no if the mass matrix is not singular and you are using either the ode15s or ode23t solver. The default value of maybe causes the solver to test whether the problem is a DAE, i.e., whether $M(t_0, y_0)$ is singular.
		For an example of a problem with a mass matrix, see "Example: Finite Element Discretization" on page 14-35 (fem1ode).
InitialSlope	Vector | {zero vector}	Vector representing the consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0) yp_0 = f(t_0, y_0)$. The default is the zero vector.
		This property is for use with the ode15s and ode23t solvers when solving DAEs.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the Events property. Use odeset to set this property.

ODE Events Property

String	Value	Description

Events	Function	Function that includes one or more event functions. The function is of the form
		
		 `[value,isterminal,direction] = events(t,y)`
		
		 `value`, `isterminal`, and `direction` are vectors for which the ith element corresponds to the ith event function:
		
		 • `value(i)` is the value of the ith event function.
		 • `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
		 • `direction(i) = 0` if all zeros are to be located (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.
		
		 If you specify an events function and events are detected, the solver returns three additional outputs:
		
		 • A column vector of times at which events occur
		 • Solution values corresponding to these times
		 • Indices into the vector returned by the events function. The values indicate which event the solver detected.
		
		 If you call the solver as
		
		 `[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)`
		
		 the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as
		
		 `sol = solver(odefun,tspan,y0,options)`
		
		 the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie` respectively.
		
		 For examples that use an event function, see "Example: Simple Event Location" on page 14-41 (`ballode`) and "Example: Advanced Event Location" on page 14-44 (`orbitode`).

ode15s Properties

ode15s is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The ode15s properties let you choose between these formulas, as well as specifying the maximum order for the formula used.

The following table describes the ode15s properties. Use odeset to set these properties.

ode15s Properties

Property	Value	Description
MaxOrder	1 | 2 | 3 | 4 | {5}	The maximum order formula used to compute the solution.
BDF	on | {off}	Specifies whether you want to use the BDFs instead of the default NDFs. Set BDF on to have ode15s use the BDFs.
		For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if MaxOrder is reduced (for example to 2) so that only the most stable formulas are used.

Examples: Applying the ODE Initial Value Problem Solvers

This section contains several examples that illustrate the kinds of problems you can solve:

- Simple nonstiff problem (rigidode)
- Stiff problem (vdpode)
- Finite element discretization (fem1ode)
- Large, stiff sparse problem (brussode)
- Simple event location (ballode)
- Advanced event location (orbitode)
- Differential-algebraic problem (hb1dae)

Example: Simple Nonstiff Problem

`rigidode` illustrates the solution of a standard test problem proposed by Krogh for solvers intended for nonstiff problems [8].

The ODEs are the Euler equations of a rigid body without external forces.

$$y_1' = y_2 \, y_3$$

$$y_2' = -y_1 y_3$$

$$y_3' = -0.51 \, y_1 y_2$$

For your convenience, the entire problem is defined and solved in a single M-file. The differential equations are coded as a subfunction f. Because the example calls the ode45 solver without output arguments, the solver uses the default output function odeplot to plot the solution components.

To run this example, click on the example name, or type rigidode at the command line.

```
function rigidode
%RIGIDODE  Euler equations of a rigid body without external forces
tspan = [0 12];
y0 = [0; 1; 1];
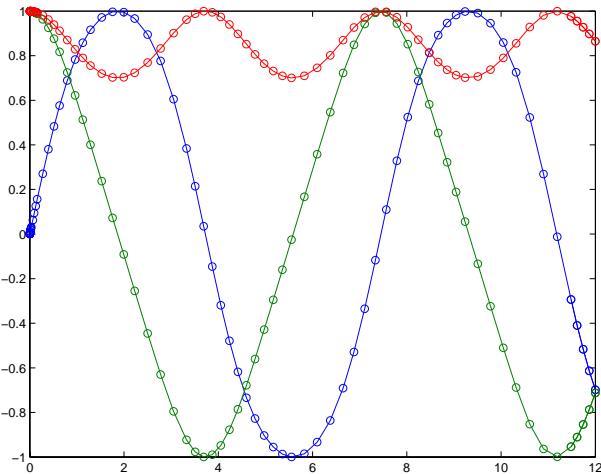
% Solve the problem using ode45
ode45(@f,tspan,y0);
% -----------------------------------------------------------
function dydt = f(t,y)
dydt = [ y(2)*y(3)
        -y(1)*y(3)
        -0.51*y(1)*y(2) ];
```

Example: Stiff Problem (van der Pol Equation)

vdpode illustrates the solution of the van der Pol problem described in "Example: The van der Pol Equation, $\mu = 1000$ (Stiff)" on page 14-14. The differential equations

$$y'_1 = y_2$$
$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

involve a constant parameter μ.

As μ increases, the problem becomes more stiff, and the period of oscillation becomes larger. When μ is 1000 the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff (quasi-discontinuities).

By default, the solvers in the ODE suite that are intended for stiff problems approximate Jacobian matrices numerically. However, this example provides a subfunction J(t,y,mu) to evaluate the Jacobian matrix $\partial f / \partial y$ analytically at

(t,y) for μ = mu. The use of an analytic Jacobian can improve the reliability and efficiency of integration.

To run this example, click on the example name, or type vdpode at the command line. From the command line, you can specify a value of μ as an argument to vdpode. The default is $\mu = 1000$.

```
function vdpode(MU)
%VDPODE  Parameterizable van der Pol equation (stiff for large MU)
if nargin < 1
  MU = 1000;     % default
end
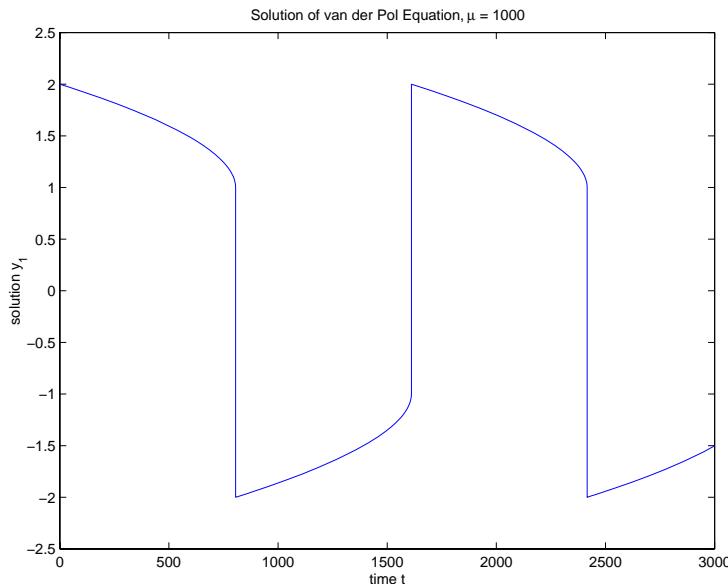
tspan = [0; max(20,3*MU)];              % Several periods
y0 = [2; 0];
options = odeset('Jacobian',@J);

[t,y] = ode15s(@f,tspan,y0,options,MU);

plot(t,y(:,1));
title(['Solution of van der Pol Equation, \mu = ' num2str(MU)]);
xlabel('time t');
ylabel('solution y_1');

axis([tspan(1) tspan(end) -2.5 2.5]);
--------------------------------------------------------------
function dydt = f(t,y,mu)
dydt = [              y(2)
         mu*(1-y(1)^2)*y(2)-y(1) ];
--------------------------------------------------------------
function dfdy = J(t,y,mu)
dfdy = [          0                   1
         -2*mu*y(1)*y(2)-1    mu*(1-y(1)^2) ];
```

Solution of van der Pol Equation, $\mu = 1000$

Example: Finite Element Discretization

`fem1ode` illustrates the solution of ODEs that result from a finite element discretization of a partial differential equation. The value of `N` in the call `fem1ode(N)` controls the discretization, and the resulting system consists of `N` equations. By default, `N` is 19.

This example involves a mass matrix. The system of ODEs comes from a method of lines solution of the partial differential equation

$$e^{-t}\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with initial condition $u(0, x) = \sin(x)$ and boundary conditions $u(t, 0) = u(t, \pi) = 0$. An integer N is chosen, h is defined as $\pi/(N+1)$, and the solution of the partial differential equation is approximated at $x_k = kh$ for $k = 0, 1, ..., N+1$ by

$$u(t,x_k) \approx \sum_{k=1}^{N} c_k(t)\phi_k(x)$$

Here $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at all the other x_j. A Galerkin discretization leads to the system of ODEs

$$M(t)c' = Jc \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ c_N(t) \end{bmatrix}$$

and the tridiagonal matrices $M(t)$ and J are given by

$$M_{ij} = \begin{cases} 2h/3\exp(-t) & \text{if } i = j \\ h/6\exp(-t) & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$J_{ij} = \begin{cases} -2/h & \text{if } i = j \\ 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c(0)$ are taken from the initial condition for the partial differential equation. The problem is solved on the time interval $[0, \pi]$.

In `fem1ode` the properties

```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

indicate that the problem is of the form $M(t)y' = Jy$. The subfunction `mass(t,N)` evaluates the time-dependent mass matrix $M(t)$ and J is the constant Jacobian.

To run this example, click on the example name, or type `fem1ode` at the command line. From the command line, you can specify a value of N as an argument to `fem1ode`. The default is $N = 19$.

```
function fem1ode(N)
%FEM1ODE   Stiff problem with a time-dependent mass matrix
```

```
if nargin < 1
  N = 19;
end
h = pi/(N+1);
y0 = sin(h*(1:N)');
tspan = [0; pi];
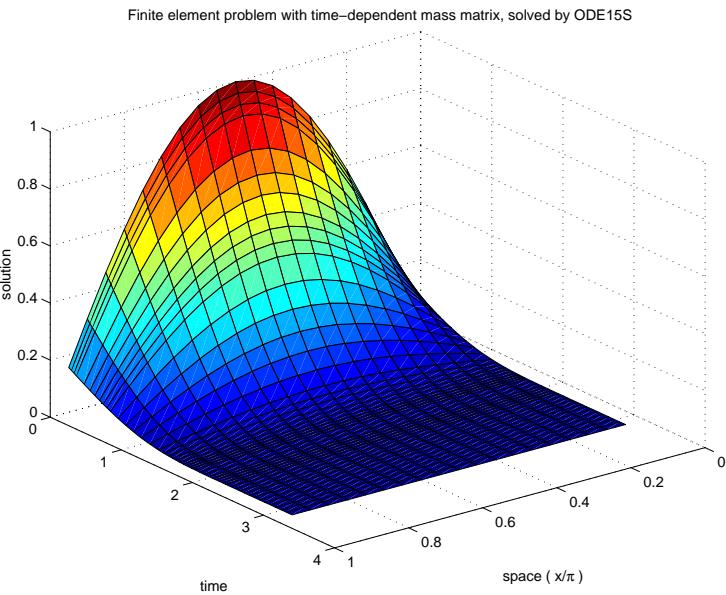
% The Jacobian is constant.
e = repmat(1/h,N,1);     % e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1);    % d=[(-2/h) ... (-2/h)];
J = spdiags([e d e], -1:1, N, N);

options = odeset('Mass',@mass,'MStateDependence','none', ...
                 'Jacobian',J);

[t,y] = ode15s(@f,tspan,y0,options,N);

surf((1:N)/(N+1),t,y);
set(gca,'ZLim',[0 1]);
view(142.5,30);
title(['Finite element problem with time-dependent mass ' ...
       'matrix, solved by ODE15S']);
xlabel('space ( x/\pi )');
ylabel('time');
zlabel('solution');
%----------------------------------------------------------------
function out = f(t,y,N)
h = pi/(N+1);
e = repmat(1/h,N,1);     % e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1);    % d=[(-2/h) ... (-2/h)];
J = spdiags([e d e], -1:1, N, N);
out = J*y;
%----------------------------------------------------------------
function M = mass(t,N)
h = pi/(N+1);
e = repmat(exp(-t)*h/6,N,1);  % e(i)=exp(-t)*h/6
e4 = repmat(4*exp(-t)*h/6,N,1);
M = spdiags([e e4 e], -1:1, N, N);
```

14-37

Finite element problem with time–dependent mass matrix, solved by ODE15S

Example: Large, Stiff Sparse Problem

`brussode` illustrates the solution of a (potentially) large stiff sparse problem. The problem is the classic "Brusselator" system [3] that models diffusion in a chemical reaction

$$u'_i = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1})$$

$$v'_i = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1})$$

and is solved on the time interval [0,10] with $\alpha = 1/50$ and

$$\left. \begin{array}{l} u_i(0) = 1 + \sin(2\pi x_i) \\ v_i(0) = 3 \end{array} \right\} \quad \text{with } x_i = i/(N+1), \text{ for } i = 1, ..., N$$

There are $2N$ equations in the system, but the Jacobian is banded with a constant width 5 if the equations are ordered as $u_1, v_1, u_2, v_2, ...$

In the call brussode(N), where N corresponds to N, the parameter N $\geq$ 2 specifies the number of grid points. The resulting system consists of 2N equations. By default, N is 20. The problem becomes increasingly stiff and the Jacobian increasingly sparse as N increases.

The subfunction f(t,y,N) returns the derivatives vector for the Brusselator problem. The subfunction jpattern(N) returns a sparse matrix of 1s and 0s showing the locations of nonzeros in the Jacobian $\partial f / \partial y$. The example assigns this matrix to the property JPattern, and the solver uses the sparsity pattern to generate the Jacobian numerically as a sparse matrix. Providing a sparsity pattern can significantly reduce the number of function evaluations required to generate the Jacobian and can accelerate integration. For the Brusselator problem, if the sparsity pattern is not supplied, 2N evaluations of the function are needed to compute the 2N-by-2N Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of N.

To run this example, click on the example name, or type brussode at the command line. From the command line, you can specify a value of N as an argument to brussode. The default is $N = 20$.

```
function brussode(N)
%BRUSSODE  Stiff problem modeling a chemical reaction

if nargin<1
  N = 20;
end
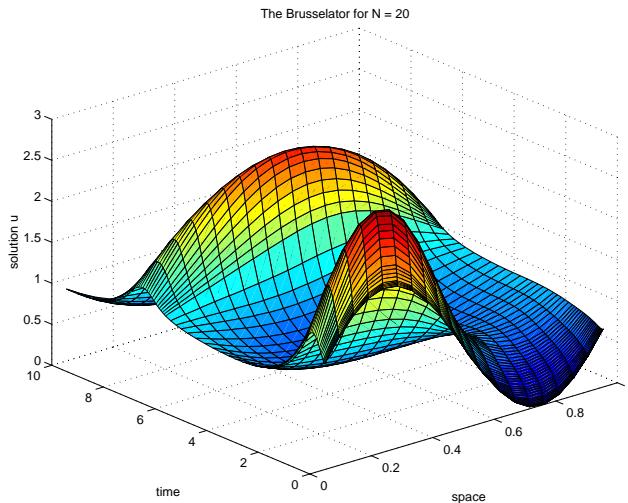
tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N)); repmat(3,1,N)];

options = odeset('Vectorized','on','JPattern',jpattern(N));

[t,y] = ode15s(@f,tspan,y0,options,N);

u = y(:,1:2:end);
x = (1:N)/(N+1);
surf(x,t,u);
view(-40,30);
xlabel('space');
ylabel('time');
zlabel('solution u');
```

```
title(['The Brusselator for N = ' num2str(N)]);
% ---------------------------------------------------------------
function dydt = f(t,y,N)
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt
% Evaluate the two components of the function at one edge of
% the grid (with edge conditions).
i = 1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(1-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(3-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at all interior
% grid points.
i = 3:2:2*N-3;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at the other edge
% of the grid (with edge conditions).
i = 2*N-1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:)-2*y(i,:)+1);
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+3);
% ---------------------------------------------------------------
function S = jpattern(N)
B = ones(2*N,5);
B(2:2:2*N,2) = zeros(N,1);
B(1:2:2*N-1,4) = zeros(N,1);
S = spdiags(B,-2:2,2*N,2*N);
```

The Brusselator for N = 20

Example: Simple Event Location

`ballode` models the motion of a bouncing ball. This example illustrates the event location capabilities of the ODE solvers.

The equations for the bouncing ball are

$$y'_1 = y_2$$
$$y'_2 = -9.8$$

In this example, the event function is coded in a subfunction `events`

```
[value,isterminal,direction] = events(t,y)
```

which returns:

- A value of the event function
- The information whether or not the integration should stop (`isterminal = 1` or `0`, respectively) when `value = 0`
- The desired directionality of the zero crossings:

-1   Detect zero crossings in the negative direction only

0   Detect all zero crossings

1   Detect zero crossings in the positive direction only

The length of value, isterminal, and direction is the same as the number of event functions. The ith element of each vector, corresponds to the ith event function. For an example of more advanced event location, see orbitode ("Example: Advanced Event Location" on page 14-44).

In ballode, setting the Events property to @events causes the solver to stop the integration (isterminal = 1) when the ball hits the ground (the height y(1) is 0) during its fall (direction = -1). The example then restarts the integration with initial conditions corresponding to a ball that bounced.

To run this example, click on the example name, or type ballode at the command line.

```
function ballode
%BALLODE  Run a demo of a bouncing ball.

tstart = 0;
tfinal = 30;
y0 = [0; 20];
refine = 4;
options = odeset('Events',@events,'OutputFcn', @odeplot,...
                 'OutputSel',1,'Refine',refine);

set(gca,'xlim',[0 30],'ylim',[0 25]);
box on
hold on;

tout = tstart;
yout = y0.';
teout = [];
yeout = [];
ieout = [];
for i = 1:10
  % Solve until the first terminal event.
```

```
    [t,y,te,ye,ie] = ode23(@f,[tstart tfinal],y0,options);
    if ~ishold
      hold on
    end
    % Accumulate output.
    nt = length(t);
    tout = [tout; t(2:nt)];
    yout = [yout; y(2:nt,:)];
    teout = [teout; te];    % Events at tstart are never reported.
    yeout = [yeout; ye];
    ieout = [ieout; ie];

    ud = get(gcf,'UserData');
    if ud.stop
      break;
    end
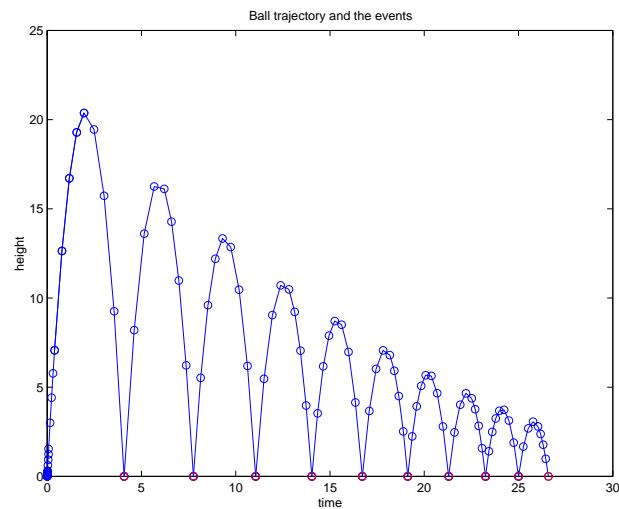
    % Set the new initial conditions, with .9 attenuation.
    y0(1) = 0;
    y0(2) = -.9*y(nt,2);

    % A good guess of a valid first time step is the length of
    % the last valid time step, so use it for faster computation.
    options = odeset(options,'InitialStep',t(nt)-t(nt-refine),...
                              'MaxStep',t(nt)-t(1));
    tstart = t(nt);
  end

plot(teout,yeout(:,1),'ro')
xlabel('time');
ylabel('height');
title('Ball trajectory and the events');
hold off
odeplot([],[],'done');
% -------------------------------------------------------------
function dydt = f(t,y)
dydt = [y(2); -9.8];
% -------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when height passes through zero in a
```

```
% decreasing direction and stop integration.
value = y(1);      % Detect height = 0
isterminal = 1;    % Stop the integration
direction = -1;    % Negative direction only
```


Example: Advanced Event Location

`orbitode` illustrates the solution of a standard test problem for those solvers that are intended for nonstiff problems. It traces the path of a spaceship traveling around the moon and returning to the earth. (Shampine and Gordon [8], p.246).

The `orbitode` problem is a system of the four equations shown below

$$y_1' = y_3$$

$$y_2' = y_4$$

$$y_3' = 2y_4 + y_1 - \frac{\mu^*(y_1 + \mu)}{r_1^3} - \frac{\mu(y_1 - \mu^*)}{r_2^3}$$

$$y_4' = -2y_3 + y_2 - \frac{\mu^* y_2}{r_1^3} - \frac{\mu y_2}{r_2^3}$$

where

$$\mu = 1/82.45$$

$$\mu^* = 1 - \mu$$

$$r_1 = \sqrt{(y_1 + \mu)^2 + y_2^2}$$

$$r_2 = \sqrt{(y_1 - \mu^*)^2 + y_2^2}$$

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body. The initial conditions have been chosen to make the orbit periodic. The value of μ corresponds to a spaceship traveling around the moon and the earth. Moderately stringent tolerances are necessary to reproduce the qualitative behavior of the orbit. Suitable values are `1e-5` for `RelTol` and `1e-4` for `AbsTol`.

The `events` subfunction includes event functions which locate the point of maximum distance from the starting point and the time the spaceship returns to the starting point. Note that the events are located accurately, even though the step sizes used by the integrator are *not* determined by the location of the events. In this example, the ability to specify the direction of the zero crossing is critical. Both the point of return to the initial point and the point of maximum distance have the same event function value, and the direction of the crossing is used to distinguish them.

To run this example, click on the example name, or type `orbitode` at the command line. The example uses the output function `odephase2` to produce the

two-dimensional phase plane plot and let you to see the progress of the
integration.

```
function orbitode
%ORBITODE  Restricted three-body problem
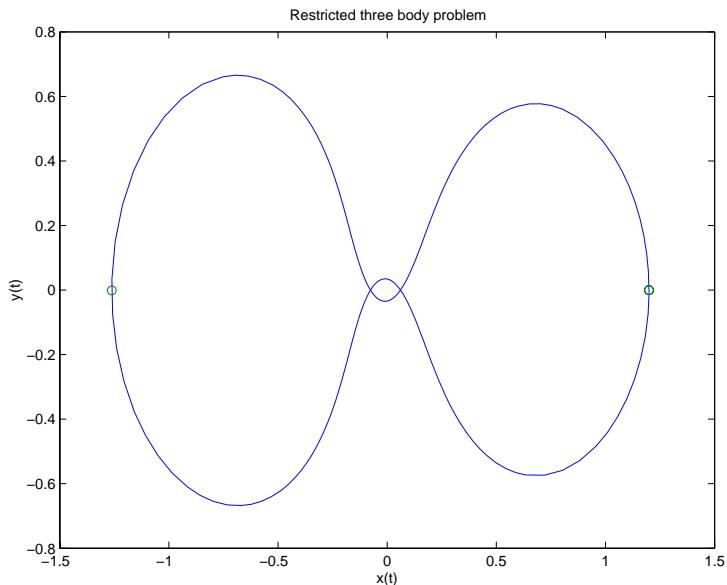
tspan = [0 7];
y0 = [1.2; 0; 0; -1.04935750983031990726];
options = odeset('RelTol',1e-5,'AbsTol',1e-4,...
                 'OutputFcn',@odephas2,'Events',@events);

[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);

plot(y(:,1),y(:,2),ye(:,1),ye(:,2),'o');
title ('Restricted three body problem')
ylabel ('y(t)')
xlabel ('x(t)')
% ------------------------------------------------------------
function dydt = f(t,y)
mu = 1 / 82.45;
mustar = 1 - mu;
r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;
dydt = [ y(3)
         y(4)
         2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - ...
                          mu*((y(1)-mustar)/r23)
        -2*y(3) + y(2) - mustar*(y(2)/r13) - mu*(y(2)/r23) ];
% ------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when the object returns closest to the
% initial point y0 and starts to move away, and stop integration.
% Also locate the time when the object is farthest from the
% initial point y0 and starts to move closer.
%
% The current distance of the body is
%
%   DSQ = (y(1)-y0(1))^2 + (y(2)-y0(2))^2
%       = <y(1:2)-y0,y(1:2)-y0>
%
```

```
% A local minimum of DSQ occurs when d/dt DSQ crosses zero
% heading in the positive direction.  We can compute d(DSQ)/dt as
%
%  d(DSQ)/dt = 2*(y(1:2)-y0)'*dy(1:2)/dt = 2*(y(1:2)-y0)'*y(3:4)
%
y0 = [1.2; 0];
dDSQdt = 2 * ((y(1:2)-y0)' * y(3:4));
value = [dDSQdt; dDSQdt];
isterminal = [1; 0];            % Stop at local minimum
direction = [1; -1];            % [local minimum, local maximum]
```


Restricted three body problem

Example: Differential-Algebraic Problem

hb1dae reformulates the hb1ode example as a *differential-algebraic equation*
(DAE) problem. The Robertson problem coded in hb1ode is a classic test
problem for codes that solve stiff ODEs.

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$y_3' = 3 \cdot 10^7 y_2^2$$

Note The Robertson problem appears as an example in the prolog to LSODI [4].

In hb1ode, the problem is solved with initial conditions $y_1(0) = 1$, $y_2(0) = 0$, $y_3(0) = 0$ to steady state. These differential equations satisfy a linear conservation law that is used to reformulate the problem as the DAE

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$0 = y_1 + y_2 + y_3 - 1$$

Obviously these equations do not have a solution for $y(0)$ with components that do not sum to 1. The problem has the form of $My' = f(t, y)$ with

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

M is obviously singular, but hb1dae does not inform the solver of this. The solver must recognize that the problem is a DAE, not an ODE. Similarly, although consistent initial conditions are obvious, the example uses an inconsistent value $y_3(0) = 10^{-3}$ to illustrate computation of consistent initial conditions.

To run this example, click on the example name, or type hb1dae at the command line. Note that hb1dae:

- Imposes a much smaller absolute error tolerance on y_2 than on the other components. This is because y_2 is much smaller than the other components and its major change takes place in a relatively short time.
- Specifies additional points at which the solution is computed to more clearly show the behavior of y_2.
- Multiplies y_2 by 10^4 to make y_2 visible when plotting it with the rest of the solution.
- Uses a logarithmic scale to plot the solution on the long time interval.

```
function hb1dae
%HB1DAE  Stiff differential-algebraic equation (DAE)

% A constant, singular mass matrix
M = [1 0 0
     0 1 0
     0 0 0];

% Use an inconsistent initial condition to test initialization.
y0 = [1; 0; 1e-3];
tspan = [0 4*logspace(-6,6)];

% Use the LSODI example tolerances. The 'MassSingular' property
% is left at its default 'maybe' to test the automatic detection
% of a DAE.
options = odeset('Mass',M,'RelTol',1e-4,...
                 'AbsTol',[1e-6 1e-10 1e-6],'Vectorized','on');

[t,y] = ode15s(@f,tspan,y0,options);
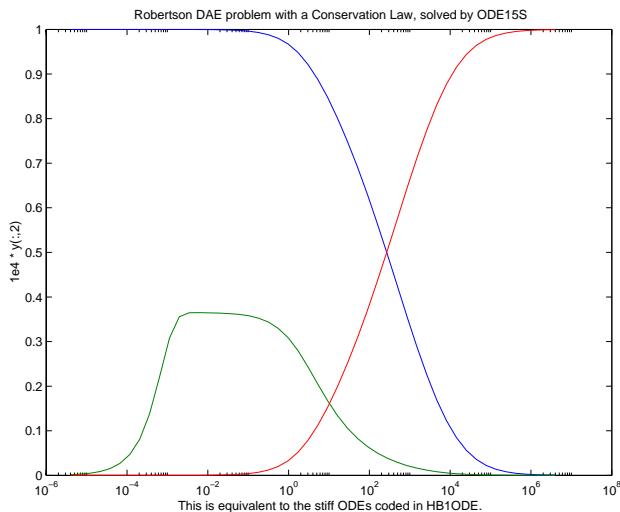
y(:,2) = 1e4*y(:,2);

semilogx(t,y);
ylabel('1e4 * y(:,2)');
title(['Robertson DAE problem with a Conservation Law, '...
       'solved by ODE15S']);
xlabel('This is equivalent to the stiff ODEs coded in HB1ODE.');
% -------------------------------------------------------------
function out = f(t,y)
out = [ -0.04*y(1,:) + 1e4*y(2,:).*y(3,:)
```

```
0.04*y(1,:) - 1e4*y(2,:).*y(3,:) - 3e7*y(2,:).^2
y(1,:) + y(2,:) + y(3,:) - 1 ];
```


Robertson DAE problem with a Conservation Law, solved by ODE15S

This is equivalent to the stiff ODEs coded in HB1ODE.

Questions and Answers, and Troubleshooting

This section contains a number of tables that answer questions about the use and operation of the ODE solvers:

- General ODE solver questions
- Problem size, memory use, and computation speed
- Time steps for integration
- Error tolerance and other options
- Solving different kinds of problems
- Troubleshooting

General ODE Solver Questions

Question	Answer
How do the ODE solvers differ from quad or quadl?	quad and quadl solve problems of the form $y' = f(t)$. The ODE solvers handle more general problems $y' = f(t, y)$, or problems that involve a mass matrix $M(t, y) y' = f(t, y)$.
Can I solve ODE systems in which there are more equations than unknowns, or vice versa?	No.

Problem Size, Memory Use, and Computation Speed

Question	Answer
How large a problem can I solve with the ODE suite?	The primary constraints are memory and time. At each time step, the solvers for nonstiff problems allocate vectors of length n, where n is the number of equations in the system. The solvers for stiff problems allocate vectors of length n, but also an n-by-n Jacobian matrix. For these solvers it may be advantageous to use the sparse option.
	If the problem is nonstiff, or if you are using the sparse option, it may be possible to solve a problem with thousands of unknowns. In this case, however, storage of the result can be problematic. Try asking the solver to evaluate the solution at specific points only, or call the solver with no output arguments and use an output function to monitor the solution.

Problem Size, Memory Use, and Computation Speed (Continued)

Question	Answer
I'm solving a very large system, but only care about a couple of the components of y. Is there any way to avoid storing all of the elements?	Yes. The user-installable output function capability is designed specifically for this purpose. When you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history. Instead, the solver calls `OutputFcn(t,y,flag,p1,p2,...)` at each time step. To keep the history of specific elements, write an output function that stores or plots only the elements you care about.
What is the startup cost of the integration and how can I reduce it?	The biggest startup cost occurs as the solver attempts to find a step size appropriate to the scale of the problem. If you happen to know an appropriate step size, use the `InitialStep` property. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably on scale for the next integration. See `ballode` for an example.

Time Steps for Integration

Question	Answer
The first step size that the integrator takes is too large, and it misses important behavior.	You can specify the first step size with the `InitialStep` property. The integrator tries this value, then reduces it if necessary.
Can I integrate with fixed step sizes?	No.

Error Tolerance and Other Options

Question	Answer
How do I choose `RelTol` and `AbsTol`?	`RelTol`, the relative accuracy tolerance, controls the number of correct digits in the answer. `AbsTol`, the absolute error tolerance, controls the difference between the answer and the solution. At each step, the error `e` in component `i` of the solution satisfies `|e(i)| <= max(RelTol*abs(y(i)),AbsTol(i))` Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.
I want answers that are correct to the precision of the computer. Why can't I simply set `RelTol` to eps?	You can get close to machine precision, but not that close. The solvers do not allow `RelTol` near eps because they try to approximate a continuous function. At tolerances comparable to eps, the machine arithmetic causes all functions to look discontinuous.
How do I tell the solver that I don't care about getting an accurate answer for one of the solution components?	You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

Solving Different Kinds of Problems

Question	Answer
Can the solvers handle partial differential equations (PDEs) that have been discretized by the method of lines?	Yes, because the discretization produces a system of ODEs. Depending on the discretization, you might have a form involving mass matrices – the ODE solvers provide for this. Often the system is stiff. This is to be expected when the PDE is parabolic and when there are phenomena that happen on very different time scales such as a chemical reaction in a fluid flow. In such cases, use one of the four solvers: `ode15s`, `ode23s`, `ode23t`, `ode23tb`. If there are many equations, set the `JPattern` property. This might make the difference between success and failure due to the computation being too expensive. When the system is not stiff, or not very stiff, `ode23` or `ode45` is more efficient than `ode15s`, `ode23s`, `ode23t`, or `ode23tb`. For an example that uses `JPattern`, see "Example: Large, Stiff Sparse Problem" on page 14-38. Parabolic-elliptic partial differential equations in 1-D can be solved directly with the MATLAB PDE solver, `pdepe`. See "Partial Differential Equations" on page 14-108 for more information.
Can I solve differential-algebraic equation (DAE) systems?	Yes. The solvers `ode15s` and `ode23t` can solve some DAEs of the form $M(t, y)y' = f(t, y)$ where $M(t, y)$ is singular. The DAEs must be of index 1. For examples, see `amp1dae` or `hb1dae`.
Can I integrate a set of sampled data?	Not directly. You have to represent the data as a function by interpolation or some other scheme for fitting data. The smoothness of this function is critical. A piecewise polynomial fit like a spline can look smooth to the eye, but rough to a solver; the solver takes small steps where the derivatives of the fit have jumps. Either use a smooth function to represent the data or use one of the lower order solvers (`ode23`, `ode23s`, `ode23t`, `ode23tb`) that is less sensitive to this.

Solving Different Kinds of Problems (Continued)

Question	Answer
What do I do when I have the final and not the initial value?	All the solvers of the ODE suite allow you to solve backwards or forwards in time. The syntax for the solvers is `[t,y] = ode45(odefun,[t0 tf],y0);` and the syntax accepts `t0 > tf`.

Troubleshooting

Question	Answer
The solution doesn't look like what I expected.	If you're right about its appearance, you need to reduce the error tolerances from their default values. A smaller relative error tolerance is needed to compute accurately the solution of problems integrated over "long" intervals, as well as solutions of problems that are moderately unstable. You should check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not asking for any correct digits in these components. This may be acceptable for these components, but failing to compute them accurately may degrade the accuracy of other components that depend on them.
My plots aren't smooth enough.	Increase the value of `Refine` from its default of 4 in `ode45` and 1 in the other solvers. The bigger the value of `Refine`, the more output points. Execution speed is not affected much by the value of `Refine`.
I'm plotting the solution as it is computed and it looks fine, but the code gets stuck at some point.	First verify that the ODE function is smooth near the point where the code gets stuck. If it isn't, the solver must take small steps to deal with this. It may help to break `tspan` into pieces on which the ODE function is smooth. If the function is smooth and the code is taking extremely small steps, you are probably trying to solve a stiff problem with a solver not intended for this purpose. Switch to `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Troubleshooting (Continued)

Question	Answer
My integration proceeds very slowly, using too many time steps.	First, check that your `tspan` is not too long. Remember that the solver uses as many time points as necessary to produce a smooth solution. If the ODE function changes on a time scale that is very short compared to the `tspan`, the solver uses a lot of time steps. Long-time integration is a hard problem. Break `tspan` into smaller pieces.
	If the ODE function does not change noticeably on the `tspan` interval, it could be that your problem is stiff. Try using `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.
	Finally, make sure that the ODE function is written in an efficient way. The solvers evaluate the derivatives in the ODE function many times. The cost of numerical integration depends critically on the expense of evaluating the ODE function. Rather than recompute complicated constant parameters at each evaluation, store them in globals or calculate them once outside the function and pass them in as additional parameters.
I know that the solution undergoes a radical change at time t where t0 ≤ t ≤ tf but the integrator steps past without "seeing" it.	If you know there is a sharp change at time `t`, it might help to break the `tspan` interval into two pieces, `[t0 t]` and `[t tf]`, and call the integrator twice.
	If the differential equation has periodic coefficients or solution, you might restrict the maximum step size to the length of the period so the integrator won't step over periods.

Initial Value Problems for DDEs

This section describes how to use to solve initial value problems (IVPs) for delay differential equations (DDEs). It provides:

- A summary of the DDE functions and examples
- An introduction to DDEs
- A description of the DDE solver and its syntax
- General instructions for representing a DDE
- A discussion and example about discontinuities and restarting
- A discussion about changing default integration properties

DDE Function Summary

DDE Initial Value Problem Solver

Solver	Description
dde23	Solve initial value problems for delay differential equations with constant delays.

DDE Helper Functions

Function	Description
deval	Evaluate the numerical solution using the output of dde23.

DDE Solver Properties Handling

An options structure contains named properties whose values are passed to dde23, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
ddeset	Create/alter the DDE options structure.
ddeget	Extract properties from options structure created with ddeset.

DDE Initial Value Problem Examples

These examples illustrate the kind of problems you can solve using dde23. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the DDE examples, and also run them. Click on the link to invoke the browser, or type odeexamples('dde')at the command line.

Example	Description
ddex1	Straightforward example
ddex2	Cardiovascular model with discontinuities

Additional examples are provided with the tutorial by Shampine, Thompson, and Kierzenka, "Solving Delay Differential Equations with dde23." The tutorial and the examples are available at ftp://ftp.mathworks.com/pub/doc/papers/dde/. This tutorial illustrates techniques for solving nontrivial real-life problems.

Introduction to Initial Value DDE Problems

The DDE solver can solve systems of ordinary differential equations

$$y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$$

where t is the independent variable, y is the dependent variable, and y' represents dy/dt. The delays (lags) $\tau_1, ..., \tau_k$ are positive constants.

Using a History to Specify the Solution of Interest

In an *initial value problem*, we seek the solution on an interval $[t_0, t_f]$. with $t_0 < t_f$. The DDE shows that $y'(t)$ depends on values of the solution at times prior to t. In particular, $y'(t_0)$ depends on $y(t_0 - \tau_1), ..., y(t_0 - \tau_k)$. Because of this, a solution on $[t_0, t_f]$ depends on its values for $t \le t_0$, i.e., its *history* $S(t)$.

Propagation of Discontinuities

Generally, the solution $y(t)$ of an IVP for a system of DDEs has a jump in its first derivative at the initial point t_0 because the first derivative of the history function does not satisfy the DDE there

$$S'(t_0^-) \ne y'(t_0^+) = f(t_0, y(t_0), S(t_0 - \tau_1), ..., S(t_0 - \tau_k))$$

A discontinuity in any derivative propagates into the future at spacings of $\tau_1, \tau_2, ..., \tau_k$.

For reliable and efficient integration of DDEs, a solver must track discontinuities in low order derivatives and deal with them. For DDEs with constant lags, the solution gets smoother as the integration progresses, so after a while the solver can stop tracking a discontinuity. See "Discontinuities" on page 14-65 for more information.

DDE Solver

This section describes:

- The DDE solver, dde23
- DDE solver basic syntax
- Additional DDE solver arguments

The DDE Solver

The function dde23 solves initial value problems for delay differential equations (DDEs) with constant delays. It integrates a system of first-order differential equations

$$y'(t) = f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$$

on the interval $[t_0, t_f]$, with $t_0 < t_f$ and given history $y(t) = S(t)$ for $t \le t_0$.

dde23 produces a solution that is continuous on $[t_0, t_f]$. You can use the function `deval` and the output of dde23 to evaluate the solution at specific points on the interval of integration.

dde23 tracks discontinuities and integrates the differential equations with the explicit Runge-Kutta (2,3) pair and interpolant used by `ode23`. The Runge-Kutta formulas are implicit for step sizes longer than the delays. When the solution is smooth enough that steps this big are justified, the implicit formulas are evaluated by a predictor-corrector iteration.

DDE Solver Basic Syntax

The basic syntax of the DDE solver is

```
sol = dde23(ddefun,lags,history,tspan)
```

The input arguments are

ddefun	A function that evaluates the right side of the differential equations. The function must have the form
	`dydt = ddefun(t,y,Z)`
	where the scalar `t` is the independent variable, the column vector `y` is the dependent variable, and `Z(:,j)` is $y(t - \tau_j)$ for $\tau_j = $ `lags(j)`.
lags	A vector of constant positive delays $\tau_1, ..., \tau_k$.
history	Function of t that evaluates the solution $y(t)$ for $t \leq t_0$. The function must be of the form
	`S = history(t)`
	where `S` is a column vector. Alternatively, if $y(t)$ is constant, you can specify `history` as this constant vector.
	If the current call to dde23 continues a previous integration to t0, use the solution `sol` from that call as the history.
tspan	The interval of integration as a two-element vector `[t0,tf]` with `t0 < tf`.

The output argument `sol` is a structure created by the solver. It has fields:

`sol.x`	Nodes of the mesh selected by `dde23`
`sol.y`	Approximation to $y(t)$ at the mesh points of `sol.x`
`sol.yp`	Approximation to $y'(t)$ at the mesh points of `sol.x`
`sol.solver`	`'dde23'`

To evaluate the numerical solution at any point from `[t0,tf]`, use `deval` with the output structure `sol` as its input.

Additional DDE Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional parameters.

`options`	Structure of optional parameters that change the default integration properties. This is the fifth input argument.

```
sol = dde23(ddefun,lags,history,tspan,options)
```

"Creating and Maintaining a DDE Options Structure" on page 14-68 tells you how to create the structure and describes the properties you can specify.

`p1,p2...`	Parameters that the solver passes to `ddefun` and the `history` function, and all functions specified in `options`.

```
sol = dde23(ddefun,lags,history,tspan,
            options,p1,p2...)
```

The solver passes any input parameters that follow the `options` argument to the functions every time it calls them. Use `options = []` as a placeholder if you set no options. In the `ddefun` argument list, parameters follow the other arguments.

```
dydt = ddefun(t,y,Z,p1,p2,...)
```

Similarly, if `history` is a function, then

```
S = history(t,p1,p2,...).
```

Solving DDE Problems

This section uses an example to describe:

- Using dde23 to solve initial value problems (IVPs) for delay differential equations (DDEs)
- Evaluating the solution at specific points

Example: A Straightforward Problem

This example illustrates the straightforward formulation, computation, and display of the solution of a system of DDEs with constant delays. The history is constant, which is often the case. The differential equations are

$$y_1'(t) = y_1(t-1)$$
$$y_2'(t) = y_1(t-1) + y_2(t-0.2)$$
$$y_3'(t) = y_2(t)$$

The example solves the equations on [0,5] with history

$$y_1(t) = 1$$
$$y_2(t) = 1 \text{ for } t \le 0.$$
$$y_3(t) = 1$$

Note The demo ddex1 contains the complete code for this example. To see the code in an editor, click the example name, or type edit ddex1 at the command line. To run the example type ddex1 at the command line. See "DDE Solver Basic Syntax" on page 14-60 for more information.

---

1 **Rewrite the problem as a first-order system.** To use dde23, you must rewrite the equations as an equivalent system of first-order differential equations. Do this just as you would when solving IVPs and BVPs for ODEs (see "Solving ODE Problems" on page 14-10). However, this example needs no such preparation because it already has the form of a first-order system of equations.

2 Identify the lags. The delays (lags) τ_1, ..., τ_k are supplied to dde23 as a vector. For the example we could use

```
lags = [1,0.2];
```

In coding the differential equations, τ_j = lags(j).

3 Code the system of first-order DDEs. Once you represent the equations as a first-order system, and specify lags, you can code the equations as a function that dde23 can use.

This code represents the system in the function, ddex1de.

```
function dydt = ddex1de(t,y,Z)
ylag1 = Z(:,1);
ylag2 = Z(:,2);
dydt = [ylag1(1)
        ylag1(1) + ylag2(2)
        y(2)                 ];
```

4 Code the history function. The history function for this example is

```
function S = ddex1hist(t)
S = ones(3,1);
```

5 Apply the DDE solver. The example now calls dde23 with the functions ddex1de and ddex1hist.

```
sol = dde23(@ddex1de,lags,@ddex1hist,[0,5]);
```

Here the example supplies the interval of integration [0,5] directly. Because the history is constant, we could also call dde23 by

```
sol = dde23(@ddex1de,lags,ones(3,1),[0,5]);
```

6 View the results. Complete the example by displaying the results. dde23 returns the mesh it selects and the solution there as fields in the solution structure sol. Often, these provide a smooth graph.

```
plot(sol.x,sol.y);
title('An example of Wille'' and Baker');
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2','y_3',2)
```

An example of Wille' and Baker

Evaluating the Solution at Specific Points

The method implemented in dde23 produces a continuous solution over the whole interval of integration $[t_0, t_f]$. You can evaluate the approximate solution, $S(t)$, at any point in $[t_0, t_f]$ using the helper function deval and the structure sol returned by dde23.

```
Sint = deval(sol,tint)
```

The deval function is vectorized. For a vector tint, the ith column of Sint approximates the solution y(tint(i)).

Using the output sol from the previous example, this code evaluates the numerical solution at 100 equally spaced points in the interval [0,5] and plots the result.

```
tint = linspace(0,5);
Sint = deval(sol,tint);
plot(tint,Sint);
```

Discontinuities

`dde23` can solve problems with discontinuities in the history or discontinuities in coefficients of the equations. It provides properties that enable you to supply locations of known discontinuities and a different initial value.

Discontinuity	Property	Comments
At the initial value $t = t_0$	`InitialY`	Generally the intial value $y(t_0)$ is the value $S(t_0)$ returned by the history function, which is to say that the solution is continuous at the initial point. However, if this is not the case, supply a different initial value using the `InitialY` property.
In the history, i.e., the solution at $t < t_0$, or in the equation coefficients for $t > t_0$	`Jumps`	Provide the known locations t of the discontinuities in a vector as the value of the `Jumps` property.
State-dependent	`Events`	`dde23` uses the events function you supply to locate these discontinuities. When `dde23` finds such a discontinuity, restart the integration to continue. Specify the solution structure for the current integration as the history for the new integration. `dde23` extends each element of the solution structure after each restart so that the final structure provides the solution for the whole interval of integration. If the new problem involves a change in the solution, use the `InitialY` property to specify the initial value for the new integration.

Example: Cardiovascular Model

This example solves a cardiovascular model due to J. T. Ottesen [6]. The equations are integrated over the interval [0,1000]. The situation of interest is when the peripheral pressure R is reduced exponentially from its value of 1.05 to 0.84 beginning at $t = 600$.

This is a problem with one delay, a constant history, and three differential equations with fourteen physical parameters. It has a discontinuity in a low order derivative at `t = 600`.

Note The demo ddex2 contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex2` at the command line. To run the example type `ddex2` at the command line. See "DDE Solver Basic Syntax" on page 14-60 for more information.

In ddex2, the fourteen physical parameters are set as fields in a structure p that dde23 passes to ddex2de as an additional argument. The function ddex2de for evaluating the equations begins with

```
function dydt = ddex2de(t,y,Z,p)
if t <= 600
   p.R = 1.05;
else
   p.R = 0.21 * exp(600-t) + 0.84;
end
.
.
.
```

Solve Using the Jumps **Property.** The peripheral pressure R is a continuous function of t, but it does not have a continuous derivative at `t = 600`. The example uses the Jumps property to inform dde23 about the location of this discontinuity.

```
opts = ddeset('Jumps',600);
```

After defining the delay tau and the constant history, the call is

```
sol = dde23(@ddex2de,tau,history,[0, 1000],opts,p);
```

The demo `ddex2` plots only the third component, the heart rate, which shows a sharp change at `t = 600`.

Solve by Restarting. The example could have solved this problem by breaking it into two pieces

```
sol = dde23(@ddex2de,tau,history,[0, 600],[],p);
sol = dde23(@ddex2de,tau,sol,[600, 1000],[],p);
```

The solution structure `sol` on the interval `[0,600]` serves as history for restarting the integration at `t = 600`. In the second call, `dde23` extends `sol` so that on return the solution is available on the whole interval `[0,1000]`. That is, after this second return,

```
Sint = deval(sol,[300,900]);
```

evaluates the solution obtained in the first integration at `t = 300`, and the solution obtained in the second integration at `t = 900`.

When discontinuities occur in low order derivatives at points known in advance, it is better to use the `Jumps` property. When you use event functions to locate such discontinuities, you must restart the integration at discontinuities.

Heart Rate for Baroflex–Feedback Mechanism.

Changing DDE Integration Properties

The default integration properties in the DDE solver dde23 are selected to handle common problems. In some cases, you can improve solver performance by changing these defaults. To do this, create an options structure containing one or more property values and supply it to dde23.

```
sol = dde23(ddefun,lags,history,tspan,options)
```

This section:

- Explains how to create, modify, and query an options structure
- Describes the properties that you can use in an options structure

In this and subsequent property tables, the most commonly used property categories are listed first, followed by more advanced categories.

DDE Property Categories

Properties Category	Property Name
Error control	RelTol, AbsTol, NormControl
Solver output	OutputFcn, OutputSel, Stats
Step-size	InitialStep, MaxStep
Event location	Events
Discontinuities	InitialY, Jumps

Creating and Maintaining a DDE Options Structure

The ddeset function creates an options structure that you can supply to dde23. You can use ddeget to query the options structure for the value of a specific property.

Creating an Options Structure. The ddeset function accepts property name/property value pairs using the syntax

```
options = ddeset('name1',value1,'name2',value2,...)
```

This creates a structure options in which the named properties have the specified values. Unspecified properties retain their default values. For all

properties, it is sufficient to type only the leading characters that uniquely identify the property name. ddeset ignores case for property names.

With no arguments, ddeset displays all property names and their possible values, indicating defaults with braces {}.

Modifying an Existing Options Structure. To modify an existing options argument, use

```
options = ddeset(oldopts,'name1',value1,...)
```

This overwrites any values in oldopts that are specified using name/value pairs. dde23 returns the modified structure as the output argument. In the same way, the command

```
options = ddeset(oldopts,newopts)
```

combines the structures oldopts and newopts. In options, any values set in newopts overwrite those in oldopts.

Querying an Options Structure. The ddeget function extracts a property value from an options structure created with ddeset.

```
o = ddeget(options,'name')
```

This returns the value of the specified property, or an empty matrix [] if you specify no property value in the options structure.

As with ddeset, it is sufficient to type only the leading characters that uniquely identify the property name. ddeget ignores case for property names.

Error Control Properties

At each step, the dde23 solver estimates the local error e in the ith component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, RelTol, and the specified absolute tolerance, AbsTol.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the dde23 solver delivers accuracy roughly equivalent to the accuracy you request. It delivers less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust RelTol. For the absolute error tolerance, the scaling of the solution

components is important: if |y| is somewhat smaller than AbsTol, the solver is not constrained to obtain any correct digits in y. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want RelTol correct digits in all solution components except those smaller than thresholds AbsTol(i). Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components

The following table describes the error control properties. Use ddeset to set the properties.

DDE Error Control Properties

Property	Value	Description
RelTol	Positive scalar {1e-3}	A relative error tolerance that applies to all components of the solution vector y. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds AbsTol(i). The default, 1e-3, corresponds to 0.1% accuracy.
AbsTol	Positive scalar or vector {1e-6}	Absolute error tolerances that apply to the individual components of the solution vector. AbsTol(i) is a threshold below which the value of the ith solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components. If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector y. If AbsTol is a scalar, the value applies to all components of y.

DDE Error Control Properties (Continued)

Property	Value	Description
NormControl	on | {off}	Control error relative to norm of solution. Set this property on to request that the solvers control the error in each integration step with norm(e) <= max(RelTol*norm(y),AbsTol). By default the solvers use a more stringent component-wise error control.

Solver Output Properties

The solver output properties let you control the output that the solvers generate. Use ddeset to set these properties.

DDE Solver Output Properties

Property	Value	Description
OutputFcn	Function {ddeplot}	Installable output function. The solver calls this function after every successful integration step.
		For example,
		``` options = ddeset('OutputFcn',@myfun) ```
		sets the OutputFcn property to an output function, myfun, that can be passed to dde23.
		The output function must be of the form
		``` status = myfun(t,y,flag,p1,p2,...) ```
		The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:
		init The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan and y0 are the input arguments to dde23.

DDE Solver Output Properties (Continued)

Property	Value	Description
	{none}	The solver calls status = myfun(t,y) after each integration step on which output is requested. t contains points where output was generated during the step, and y is the numerical solution at the points in t. If t is a vector, the ith column of y corresponds to the ith element of t.
		myfun must return a status output value of 0 or 1. If status = 1, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
	done	The solver calls myfun([],[],'done') when integration is complete to allow the output function to perform any cleanup chores.
		You can use these general purpose output functions or you can edit them to create your own. Type help functionname at the command line for more information.
		• ddeplot – time series plotting (default when you call the solver with no output argument and you have not specified an output function)
		• ddephas2 – two-dimensional phase plane plotting
		• ddephas3 – three-dimensional phase plane plotting
		• ddeprint – print solution as the solver computes it

DDE Solver Output Properties (Continued)

Property	Value	Description
OutputSel	Vector of indices	Vector of indices specifying which components of the solution vector dde23 passes to the output function. For example, if you want to use the ddeplot output function, but you want to plot only the first and third components of the solution, you can do this using ```options = ddeset('OutputFcn',@ddeplot,'OutputSel',[1 3]);``` By default, the solver passes all components of the solution to the output function.
Stats	on | {off}	Specifies whether the solver should display statistics about its computations. By default, Stats is off. If it is on, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the DDE function was called to evaluate $f(t, y(t), y(t - \tau_1), ..., y(t - \tau_k))$

Step-Size Properties

The step-size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Use ddeset to set these properties.

DDE Step Size Properties

Property	Value	Description
InitialStep	Positive scalar	Suggested initial step size. InitialStep sets an upper bound on the magnitude of the first step size the solver tries. If you do not set InitialStep, the solver bases the initial step size on the slope of the solution at the initial time tspan(1), and the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable InitialStep.
MaxStep	Positive scalar {0.1*abs(t0-tf)}	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set MaxStep to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce MaxStep: • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance RelTol, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector AbsTol. (See "Error Control Properties" on page 14-69 for a description of the error tolerance properties.)
		• To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call dde23 twice. If you do not know the time at which the change occurs, try reducing the error tolerances RelTol and AbsTol. Use MaxStep as a last resort.

Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the dde23 solver can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the Events property. Use ddeset to set this property.

DDE Events Property

String	Value	Description
Events	Function	Function that includes one or more event functions. The function is of the form `[value,isterminal,direction] = events(t,y,Z)` value, isterminal, and direction are vectors for which the ith element corresponds to the ith event function: • value(i) is the value of the ith event function. • isterminal(i) = 1 if you want the integration to terminate at a zero of this event function, and 0 otherwise. • direction(i) = 0 if you want dde23 to locate all zeros (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing. If you specify an events function and events are detected, the solver returns three additional fields in the solution structure sol: • sol.xe is a row vector of times at which events occur. • sol.ye is a matrix whose columns are the solution values corresponding to times in sol.xe. • sol.ie is a vector containing indices that specify which event occurred at the corresponding time in sol.xe. For examples that use an event function while solving ordinary differential equation problems, see "Example: Simple Event Location" on page 14-41 (ballode) and "Example: Advanced Event Location" on page 14-44 (orbitode).

Discontinuity Properties

dde23 can solve problems with discontinuities in the history or discontinuities in coefficients of the equations. These properties enable you to provide dde23 with a different initial value, and locations of known discontinuities. See "Discontinuities" on page 14-65 for more information.

DDE Discontinuity Properties

String	Value	Description
Jumps	Vector	Location of discontinuities. Points t where the history or solution may have a jump discontinuity in a low-order derivative.
InitialY	Vector	Initial value of solution. By default the initial value of the solution is the value returned by history at the initial point. Supply a different initial value as the value of the InitialY property.

Boundary Value Problems for ODEs

This section describes how to use MATLAB to solve boundary value problems (BVPs) of ordinary differential equations (ODEs). It provides:

- A summary of the BVP functions and examples
- An introduction to BVPs
- A description of the BVP solver and its syntax
- General instructions for solving a BVP
- A discussion and examples about using continuation to solve a difficult problem
- Instructions for solving singular BVPs
- A discussion about changing default integration properties

14-77

BVP Function Summary

ODE Boundary Value Problem Solver

Solver	Description
bvp4c	Solve two-point boundary value problems for ordinary differential equations.

BVP Helper Functions

Function	Description
bvpinit	Form the initial guess for bvp4c.
deval	Evaluate the numerical solution using the output of bvp4c.

BVP Solver Properties Handling

An options structure contains named properties whose values are passed to bvp4c, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
bvpset	Create/alter the BVP options structure.
bvpget	Extract properties from options structure created with bvpset.

ODE Boundary Value Problem Examples

These examples illustrate the kind of problems you can solve using the BVP solver. Click the example name to see the code in an editor. Type the example name at the command line to run it.

> **Note** The Differential Equations Examples browser enables you to view the code for the BVP examples, and also run them. Click on the link to invoke the browser, or type `odeexamples('bvp')` at the command line.

Example	Description
emdenbvp	Emden's equation, a singular BVP
fsbvp	Falkner-Skan BVP on an infinite interval
mat4bvp	Fourth eigenfunction of Mathieu's equation
shockbvp	Solution with a shock layer near x = 0
twobvp	BVP with exactly two solutions

Additional examples are provided with the tutorial by Shampine, Reichelt, and Kierzenka, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c." The tutorial and the examples are available at `ftp://ftp.mathworks.com/pub/doc/papers/bvp/`. This tutorial illustrates techniques for solving nontrivial real-life problems.

Introduction to Boundary Value ODE Problems

The BVP solver is designed to handle systems of ordinary differential equations

$$y' = f(x, y)$$

where x is the independent variable, y is the dependent variable, and y' represents dy/dx.

See "What Is an Ordinary Differential Equation?" on page 14-5 for general information about ODEs.

Using Boundary Conditions to Specify the Solution of Interest

In a *boundary value problem,* the solution of interest satisfies certain boundary conditions. These conditions specify a relationship between the values of the

solution at more than one x. bvp4c is designed to solve two-point BVPs, i.e., problems where the solution sought on an interval $[a, b]$ must satisfy the boundary conditions

$$g(y(a), y(b)) = 0$$

Unlike initial value problems, a boundary value problem may not have a solution, may have a finite number of solutions, or may have infinitely many solutions. As an integral part of the process of solving a BVP, you need to provide a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation.

There may be other difficulties when solving BVPs, such as problems imposed on infinite intervals or problems that involve singular coefficients. Often BVPs involve unknown parameters p that have to be determined as part of solving the problem

$$y' = f(x, y, p)$$
$$g(y(a), y(b), p) = 0$$

In this case, the boundary conditions must suffice to determine the value of p.

Boundary Value Problem Solver

This section describes:

• The BVP solver, bvp4c
• BVP solver basic syntax
• Additional BVP solver arguments

The BVP Solver

The function bvp4c solves two-point boundary value problems for ordinary differential equations (ODEs). It integrates a system of first-order ordinary differential equations

$$y' = f(x, y)$$

on the interval $[a, b]$, subject to general two-point boundary conditions

$$bc(y(a), y(b)) = 0$$

It can also accommodate unknown parameters for problems of the form

$$y' = f(x, y, p)$$
$$bc(y(a), y(b), p) = 0$$

In this case, the number of boundary conditions must be sufficient to determine the solution and the unknown parameters. For more information, see "Finding Unknown Parameters" on page 14-87.

bvp4c produces a solution that is continuous on $[a, b]$ and has a continuous first derivative there. You can use the function deval and the output of bvp4c to evaluate the solution at specific points on the interval of integration.

bvp4c is a finite difference code that implements the 3-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in the interval of integration. Mesh selection and error control are based on the residual of the continuous solution.

The collocation technique uses a mesh of points to divide the interval of integration into subintervals. The solver determines a numerical solution by solving a global system of algebraic equations resulting from the boundary conditions, and the collocation conditions imposed on all the subintervals. The solver then estimates the error of the numerical solution on each subinterval. If the solution does not satisfy the tolerance criteria, the solver adapts the mesh and repeats the process. The user *must* provide the points of the initial mesh as well as an initial approximation of the solution at the mesh points.

BVP Solver Basic Syntax

The basic syntax of the BVP solver is

```
sol = bvp4c(odefun,bcfun,solinit)
```

The input arguments are:

odefun   Function that evaluates the differential equations. It has the basic form

```
dydx = odefun(x,y)
```

where x is a scalar, and dydx and y are column vectors. odefun can also accept a vector of unknown parameters and a variable number of known parameters.

bcfun   Function that evaluates the residual in the boundary conditions. It has the basic form

```
res = bcfun(ya,yb)
```

where ya and yb are column vectors representing y(a) and y(b), and res is a column vector of the residual in satisfying the boundary conditions. bcfun can also accept a vector of unknown parameters and a variable number of known parameters.

solinit   Structure with fields x and y:

x   Ordered nodes of the initial mesh. Boundary conditions are imposed at a = solinit.x(1) and b = solinit.x(end).

y   Initial guess for the solution with solinit.y(:,i) a guess for the solution at the node solinit.x(i).

The structure can have any name, but the fields must be named x and y. It can also contain a vector that provides an initial guess for unknown parameters. You can form solinit with the helper function bvpinit. See the bvpinit reference page for details.

The output argument sol is a structure created by the solver. In the basic case the structure has fields x, y, and yp.

sol.x	Nodes of the mesh selected by bvp4c
sol.y	Approximation to $y(x)$ at the mesh points of sol.x
sol.yp	Approximation to $y'(x)$ at the mesh points of sol.x

sol.parameters	Value of unknown parameters, if present, found by the solver.
sol.solver	`'bvp4c'`

The function `deval` uses the output structure `sol` to evaluate the numerical solution at any point from `[a,b]`.

Additional BVP Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional known parameters.

options	Structure of optional parameters that change the default integration properties. This is the fourth input argument.

```
sol = bvp4c(odefun,bcfun,solinit,options)
```

"Creating and Maintaining a BVP Options Structure" on page 14-101 tells you how to create the structure and describes the properties you can specify.

p1,p2...	*Known* parameters that the solver passes to `odefun` and `bcfun`.

```
sol = bvp4c(odefun,bcfun,solinit,options,p1,p2...)
```

The solver passes any input parameters that follow the `options` argument to `odefun` and `bcfun` every time it calls them. Use `options = []` as a placeholder if you set no options. In the `odefun` argument list, known parameters follow `x`, `y`, and a vector of unknown parameters (`parameters`), if present.

```
dydx = odefun(x,y,p1,p2,...)
dydx = odefun(x,y,parameters,p1,p2,...)
```

In the `bcfun` argument list, known parameters follow `ya`, `yb`, and a vector of unknown parameters, if present.

```
res = bcfun(ya,yb,p1,p2,...)
res = bcfun(ya,yb,parameters,p1,p2,...)
```

See "Example: Using Continuation to Solve a Difficult BVP" on page 14-88 for an example.

Solving BVP Problems

This section describes:

- The process for solving boundary value problems (BVPs) using `bvp4c`
- Finding unknown parameters
- Evaluating the solution at specific points

Example: Mathieu's Equation

This example determines the fourth eigenvalue of Mathieu's Equation. It illustrates how to write second-order differential equations as a system of two first-order ODEs and how to use `bvp4c` to determine an unknown parameter λ.

The task is to compute the fourth ($q = 5$) eigenvalue λ of Mathieu's equation

$$y'' + (\lambda - 2q \cos 2x)y = 0$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions

$$y(0) = 1$$
$$y'(0) = 0$$
$$y'(\pi) = 0$$

Note The demo `mat4bvp` contains the complete code for this example. The demo uses subfunctions to place all functions required by `bvp4c` in a single M-file. To run this example type `mat4bvp` at the command line. See "BVP Solver Basic Syntax" on page 14-81 for more information.

---

1 **Rewrite the problem as a first-order system.** To use `bvp4c`, you must rewrite the equations as an equivalent system of first-order differential equations. Using a substitution $y_1 = y$ and $y_2 = y'$, the differential equation is written as a system of two first-order equations

$$y_1' = y_2$$
$$y_2' = -(\lambda - 2\,q\,\cos 2x)y_1$$

Note that the differential equations depend on the unknown parameter λ. The boundary conditions become

$$y_1(0) - 1 = 0$$
$$y_2(0) = 0$$
$$y_2(\pi) = 0$$

2 Code the system of first-order ODEs. Once you represent the equation as a first-order system, you can code it as a function that bvp4c can use. Because there is an unknown parameter, the function must be of the form

```
dydx = odefun(x,y,parameters)
```

The code below represents the system in the function, mat4ode.

```
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [    y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
```

See "Finding Unknown Parameters" on page 14-87 for more information about using unknown parameters with bvp4c.

3 Code the boundary conditions function. You must also code the boundary conditions in a function. Because there is an unknown parameter, the function must be of the form

```
res = bcfun(ya,yb,parameters)
```

The code below represents the boundary conditions in the function, mat4bc.

```
function res = mat4bc(ya,yb,lambda)
res = [   ya(2)
          yb(2)
          ya(1)-1 ];
```

4 Create an initial guess. To form the guess structure `solinit` with `bvpinit`, you need to provide initial guesses for both the solution and the unknown parameter.

The function `mat4init` provides an initial guess for the solution. `mat4init` uses $y = \cos 4x$ because this function satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes).

```
function yinit = mat4init(x)
yinit = [  cos(4*x)
          -4*sin(4*x) ];
```

In the call to `bvpinit`, the third argument, `lambda`, provides an initial guess for the unknown parameter λ.

```
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
```

This example uses `@` to pass `mat4init` as a function handle to `bvpinit`.

Note See the `function_handle` (@), `func2str`, and `str2func` reference pages, and the "Function Handles" chapter of "Programming and Data Types" in the MATLAB documentation for information about function handles.

---

5 Apply the BVP solver. The `mat4bvp` example calls `bvp4c` with the functions `mat4ode` and `mat4bc` and the structure `solinit` created with `bvpinit`.

```
sol = bvp4c(@mat4ode,@mat4bc,solinit);
```

6 View the results. Complete the example by displaying the results:

a Print the value of the unknown parameter λ found by `bvp4c`.

```
fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)
```

b Use `deval` to evaluate the numerical solution at 100 equally spaced points in the interval $[0, \pi]$, and plot its first component. This component approximates $y(x)$.

```
xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
```

```
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
```

See "Evaluating the Solution at Specific Points" on page 14-88 for information about using deval.

The following plot shows the eigenfunction associated with the final eigenvalue $\lambda = 17.097$.

Eigenfunction of Mathieu's equation.

Finding Unknown Parameters

The bvp4c solver can find unknown parameters p for problems of the form

$$y' = f(x, y, p)$$
$$bc(y(a), y(b), p) = 0$$

You must provide bvp4c an initial guess for any unknown parameters in the vector solinit.parameters. When you call bvpinit to create the structure solinit, specify the initial guess as a vector in the additional argument parameters.

```
solinit = bvpinit(x,v,parameters)
```

The `bvp4c` function arguments `odefun` and `bcfun` must each have a third argument.

```
dydx = odefun(x,y,parameters)
res = bcfun(ya,yb,parameters)
```

The `bvp4c` solver calculates intermediate values of unknown parameters at each iteration, and passes the latest values to `odefun` and `bcfun` in the `parameters` arguments. The solver returns the final values of these unknown parameters in `sol.parameters`. See "Example: Mathieu's Equation" on page 14-84.

Evaluating the Solution at Specific Points

The collocation method implemented in `bvp4c` produces a C^1-continuous solution over the whole interval of integration $[a, b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a, b]$ using the helper function `deval` and the structure `sol` returned by `bvp4c`.

```
Sxint = deval(sol,xint)
```

The `deval` function is vectorized. For a vector `xint`, the `i`th column of `Sxint` approximates the solution $y(\text{xint(i)})$.

Using Continuation to Make a Good Initial Guess

To solve a boundary value problem, you need to provide an initial guess for the solution. The quality of your initial guess can be critical to the solver performance, and to being able to solve the problem at all. However, coming up with a sufficiently good guess can be the most challenging part of solving a boundary value problem. Certainly, you should apply the knowledge of the problem's physical origin. Often a problem can be solved as a sequence of relatively simpler problems, i.e., a continuation. This section provides examples that illustrate how to use continuation to:

• Solve a difficult BVP.

• Verify a solution's consistent behavior.

Example: Using Continuation to Solve a Difficult BVP

This example solves the differential equation

$$\varepsilon y'' + xy' = \varepsilon \pi^2 \cos(\pi x) - \pi x \sin(\pi x)$$

for $\varepsilon = 10^{-4}$, on the interval [-1 1], with boundary conditions $y(-1) = -2$ and $y(1) = 0$. For $0 < \varepsilon < 1$, the solution has a transition layer at $x = 0$. Because of this rapid change in the solution for small values of ε, the problem becomes difficult to solve numerically.

The example solves the problem as a sequence of relatively simpler problems, i.e., a continuation. The solution of one problem is used as the initial guess for solving the next problem.

Note The demo shockbvp contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type shockbvp at the command line. See "BVP Solver Basic Syntax" on page 14-81 and "Solving BVP Problems" on page 14-84 for more information.

Note This problem appears in [1] to illustrate the mesh selection capability of a well established BVP code COLSYS.

---

1 **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that bvp4c can use. Because there is an additional known parameter ε, the functions must be of the form

```
dydx = odefun(x,y,p1)
res = bcfun(ya,yb,p1)
```

The code below represents the differential equation and the boundary conditions in the functions shockODE and shockBC. Note that shockODE is vectorized to improve solver performance. The additional parameter ε is represented by e.

```
function dydx = shockODE(x,y,e)
pix = pi*x;
dydx = [   y(2,:)
         -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix) ];
```

```
function res = shockBC(ya,yb,e)
res = [ ya(1)+2
        yb(1)   ];
```

The example passes e as an additional input argument to bvp4c.

```
  sol = bvp4c(@shockODE,@shockBC,sol,options,e);
```

bvp4c then passes this argument to the functions shockODE and shockBC
when it evaluates them. See "Additional BVP Solver Arguments" on
page 14-83 for more information.

2 **Provide analytical partial derivatives.** For this problem, the solver
benefits from using analytical partial derivatives. The code below represents
the derivatives in functions shockJac and shockBCJac.

```
function jac = shockJac(x,y,e)
jac = [ 0    1
        0 -x/e ];

function [dBCdya,dBCdyb] = shockBCJac(ya,yb,e)
dBCdya = [ 1 0
           0 0 ];
dBCdyb = [ 0 0
           1 0 ];
```

shockJac and shockBCJac must accept the additional argument e, because
bvp4c passes the additional argument to all the functions the user supplies.

Tell bvp4c to use these functions to evaluate the partial derivatives by
setting the options FJacobian and BCJacobian. Also set 'Vectorized' to
'on' to indicate that the differential equation function shockODE is
vectorized.

```
options = bvpset('FJacobian',@shockJac,...
                 'BCJacobian',@shockBCJac,...
                 'Vectorized','on');
```

3 **Create an initial guess.** You must provide bvp4c with a guess structure
that contains an initial mesh and a guess for values of the solution at the
mesh points. A constant guess of $y(x) \equiv 1$ and $y'(x) \equiv 0$, and a mesh of five

equally spaced points on [-1 1] suffice to solve the problem for $\varepsilon = 10^{-2}$. Use bvpinit to form the guess structure.

```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

4 **Use continuation to solve the problem.** To obtain the solution for the parameter $\varepsilon = 10^{-4}$, the example uses continuation by solving a sequence of problems for $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$. The solver bvp4c does not perform continuation automatically, but the code's user interface has been designed to make continuation easy. The code uses the output sol that bvp4c produces for one value of e as the guess in the next iteration.

```
e = 0.1;
for i=2:4
    e = e/10;
    sol = bvp4c(@shockODE,@shockBC,sol,options,e);
end
```

5 **View the results.** Complete the example by displaying the final solution

```
plot(sol.x,sol.y(1,:))
axis([-1 1 -2.2 2.2])
title(['There is a shock at x = 0 when \epsilon = '...
      sprintf('%.e',e) '.'])
xlabel('x')
ylabel('solution y')
```

There is a shock at x = 0 when ε =1e−004.

Example: Using Continuation to Verify a Solution's Consistent Behavior

Falkner-Skan BVPs arise from similarity solutions of viscous, incompressible, laminar flow over a flat plate. An example is

$$f''' + ff'' + \beta(1 - (f')^2) = 0$$

for $\beta = 0.5$ on the interval $[0, \infty)$ with boundary conditions $f(0) = 0$, $f'(0) = 0$, and $f'(\infty) = 1$.

The BVP cannot be solved on an infinite interval, and it would be impractical to solve it for even a very large finite interval. So, the example tries to solve a sequence of problems posed on increasingly larger intervals to verify the solution's consistent behavior as the boundary approaches ∞.

The example imposes the infinite boundary condition at a finite point called infinity. The example then uses continuation in this end point to get convergence for increasingly larger values of infinity. It uses bvpinit to extrapolate the solution sol for one value of infinity as an initial guess for the

new value of `infinity`. The plot of each successive solution is superimposed over those of previous solutions so they can easily be compared for consistency.

Note The demo `fsbvp` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type `fsbvp` at the command line. See "BVP Solver Basic Syntax" on page 14-81 and "Solving BVP Problems" on page 14-84 for more information.

---

1 **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use.

```
function dfdeta = fsode(eta,f)
beta = 0.5;
dfdeta = [ f(2)
           f(3)
          -f(1)*f(3) - beta*(1 - f(2)^2) ];

function res = fsbc(f0,finf)
res = [f0(1)
       f0(2)
       finf(2) - 1];
```

2 **Create an initial guess.** You must provide `bvp4c` with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A crude mesh of five points and a constant guess that satisfies the boundary conditions are good enough to get convergence when `infinity` = 3.

```
infinity = 3;
maxinfinity = 6;

solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);
```

3 **Solve on the initial interval.** The example obtains the solution for `infinity` = 3. It then prints the computed value of $f''(0)$ for comparison with the value reported by Cebeci and Keller [2].

```
sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;
```

14-93

```
fprintf('\n');
fprintf('Cebeci & Keller report that f''''(0) = 0.92768.\n')
fprintf('Value computed using infinity = %g is '...
        '%7.5f.\n',Bnew,f(3,1))
```

The example prints

```
Cebeci & Keller report that f''(0) = 0.92768.
Value computed using infinity = 3 is 0.92915.
```

4 **Setup the figure and plot the initial solution.**

```
figure
plot(eta,f(2,:),eta(end),f(2,end),'o');
axis([0 maxinfinity 0 1.4]);
title('Falkner-Skan equation, positive wall shear, \beta = 0.5.')
xlabel('\eta')
ylabel('df/d\eta')
hold on
drawnow
shg
```

5 Use continuation to solve the problem and plot susbsequent solutions.
The example then solves the problem for `infinity` = 4, 5, 6. It uses `bvpinit`
to extrapolate the solution `sol` for one value of `infinity` as an initial guess
for the next value of `infinity`. For each iteration, the example prints the
computed value of $f''(0)$ and superimposes a plot of the solution in the
existing figure.

```
for Bnew = infinity+1:maxinfinity
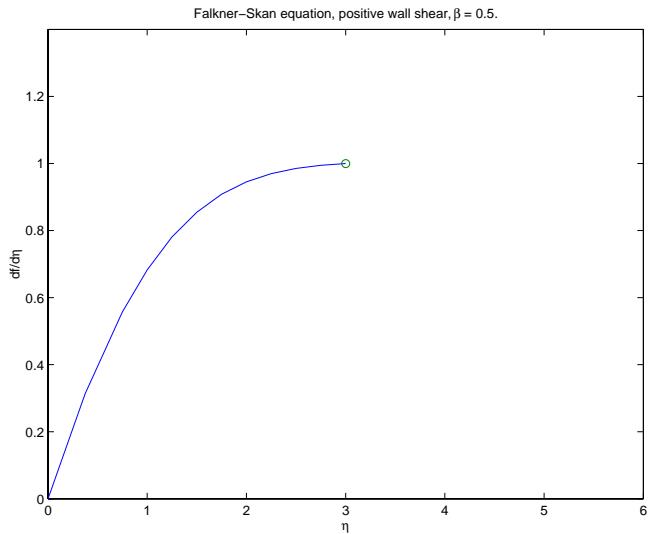
  solinit = bvpinit(sol,[0 Bnew]); % Extend solution to Bnew.
  sol = bvp4c(@fsode,@fsbc,solinit);
  eta = sol.x;
  f = sol.y;
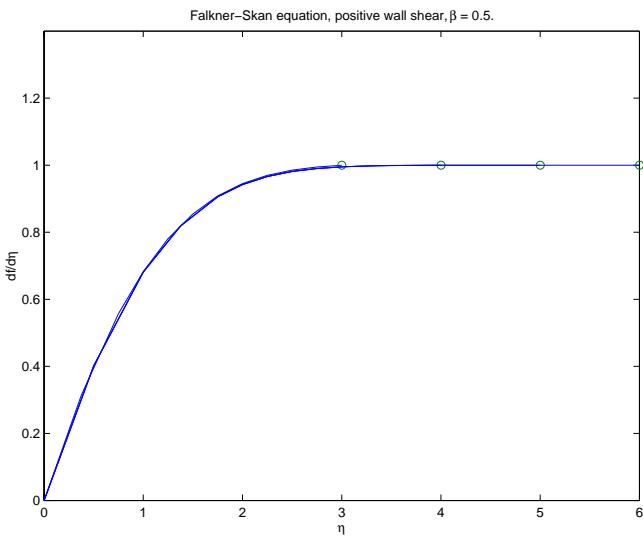
  fprintf('Value computed using infinity = %g is '...
          '%7.5f.\n',Bnew,f(3,1))
  plot(eta,f(2,:),eta(end),f(2,end),'o');
  drawnow

end
hold off
```

The example prints

```
Value computed using infinity = 4 is 0.92774.
Value computed using infinity = 5 is 0.92770.
Value computed using infinity = 6 is 0.92770.
```

Note that the values approach 0.92768 as reported by Cebeci and Keller. The
superimposed plots confirm the consistency of the solution's behavior.

Falkner–Skan equation, positive wall shear, β = 0.5.

Solving Singular BVPs

The function bvp4c solves a class of singular BVPs of the form

$$y' = \frac{1}{x}Sy + f(x, y)$$

$$0 = g(y(0), y(b))$$

(14-3)

It can also accommodate unknown parameters for problems of the form

$$y' = \frac{1}{x}Sy + f(x, y, p)$$

$$0 = g(y(0), y(b), p)$$

Singular problems must be posed on an interval $[0, b]$ with $b > 0$. Use bvpset to pass the constant matrix S to bvp4c as the value of the 'SingularTerm' integration property. Boundary conditions at $x = 0$ must be consistent with the necessary condition for a smooth solution, $Sy(0) = 0$. An initial guess should also satisfy this necessary condition.

When you solve a singular BVP using

```
sol = bvp4c(@odefun,@bcfun,solinit,options)
```

`bvp4c` requires that your function `odefun(x,y)` return only the value of the $f(x, y)$ term in Equation 14-3.

Example: Solving a BVP that Has a Singular Term

Emden's equation arises in modeling a spherical body of gas. The PDE of the model is reduced by symmetry to the ODE

$$y'' + \frac{2}{x}y' + y^5 = 0$$

on an interval $[0, 1]$. The coefficient $2/x$ is singular at $x = 0$, but symmetry implies the boundary condition $y'(0) = 0$. With this boundary condition, the term

$$\frac{2}{x}y'(0)$$

is well-defined as x approaches 0. For the boundary condition $y(1) = \sqrt{3}/2$, this BVP has the analytical solution

$$y(x) = \left(1 + \frac{x^2}{3}\right)^{-1/2}$$

Note The demo `emdenbvp` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type `emdenbvp` at the command line. See "BVP Solver Basic Syntax" on page 14-81 and "Solving BVP Problems" on page 14-84 for more information.

---

1 **Rewrite the problem as a first-order system and identify the singular term.** Using a substitution $y_1 = y$ and $y_2 = y'$, write the differential equation as a system of two first-order equations

$$y_1' = y_2$$

$$y_2' = -\frac{2}{x}y_2 - y_1^5$$

The boundary conditions become

$$y_2(0) = 0$$

$$y_1(1) = \sqrt{3}/2$$

Writing the ODE system in a vector-matrix form

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \frac{1}{x}\begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}$$

the terms of Equation 14-3 are identified as

$$S = \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}$$

and

$$f(x, y) = \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}$$

2 **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use.

```
function dydx = emdenode(x,y)
dydx = [   y(2)
          -y(1)^5 ];

function res = emdenbc(ya,yb)
res = [ ya(2)
        yb(1) - sqrt(3)/2 ];
```

3 **Setup integration properties.** Use the matrix as the value of the `'SingularTerm'` integration property.

```
S = [0,0;0,-2];
options = bvpset('SingularTerm',S);
```

4 Create an initial guess. This example starts with a mesh of five points and a constant guess for the solution.

$$y_1(x) \equiv \sqrt{3}/2$$

$$y_2(x) \equiv 0$$

Use bvpinit to form the guess structure

```
guess = [sqrt(3)/2;0];
solinit = bvpinit(linspace(0,1,5),guess);
```

5 Solve the problem. Use the standard bvp4c syntax to solve the problem.

```
sol = bvp4c(@emdenode,@emdenbc,solinit,options);
```

6 View the results. This problem has an analytical solution

$$y(x) = \left(1 + \frac{x^2}{3}\right)^{-1/2}$$

The example evaluates the analytical solution at 100 equally-spaced points and plots it along with the numerical solution computed using bvp4c.

```
x = linspace(0,1);
truy = 1 ./ sqrt(1 + (x.^2)/3);
plot(x,truy,sol.x,sol.y(1,:),'ro');
title('Emden problem -- BVP with singular term.')
legend('Analytical','Computed');
xlabel('x');
ylabel('solution y');
```

Emden problem –– BVP with singular term.

Changing BVP Integration Properties

The default integration properties in the BVP solver `bvp4c` are selected to handle common problems. In some cases, you can improve solver performance by changing these defaults. To do this, supply `bvp4c` with one or more property values in an options structure.

```
sol = bvp4c(odefun,bcfun,solinit,options)
```

This section:

- Explains how to create, modify, and query an options structure
- Describes the properties that you can use in an options structure

In this and subsequent property tables, the most commonly used property categories are listed first, followed by more advanced categories.

BVP Property Categories

Properties Category	Property Names
Error control	`RelTol, AbsTol`
Vectorization	`Vectorized`

BVP Property Categories (Continued)

Properties Category	Property Names
Analytical partial derivatives	`FJacobian, BCJacobian`
Singular BVPs	`SingularTerm`
Mesh size	`NMax`
Output displayed	`Stats`

Note For other ways to improve solver efficiency, check "Using Continuation to Make a Good Initial Guess" on page 8-88 and the tutorial, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c," available at `ftp://ftp.mathworks.com/pub/doc/papers/bvp/`.

Creating and Maintaining a BVP Options Structure

The `bvpset` function creates an `options` structure that you can supply to `bvp4c`. You can use `bvpget` to query the `options` structure for the value of a specific property.

Creating an Options Structure. The `bvpset` function accepts property name/property value pairs using the syntax

```
options = bvpset('name1',value1,'name2',value2,...)
```

This creates a structure `options` in which the named properties have the specified values. Unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property name. `bvpset` ignores case for property names.

With no arguments, `bvpset` displays all property names and their possible values, indicating defaults with braces {}.

Modifying an Existing Options Structure. To modify an existing `options` argument, use

```
options = bvpset(oldopts,'name1',value1,...)
```

This overwrites any values in `oldopts` that are specified using name/value pairs. The modified structure is returned as the output argument. In the same way, the command

```
options = bvpset(oldopts,newopts)
```

combines the structures `oldopts` and `newopts`. In `options`, any values set in `newopts` overwrite those in `oldopts`.

Querying an Options Structure. The `bvpget` function extracts a property value from an `options` structure created with `bvpset`.

```
o = bvpget(options,'name')
```

This returns the value of the specified property, or an empty matrix `[]` if the property value is unspecified in the `options` structure.

As with `bvpset`, it is sufficient to type only the leading characters that uniquely identify the property name; case is ignored for property names.

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x, S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the ith component of the solution, `res(i)`, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, `RelTol` and `AbsTol`, defined by the user.

```
‖(res(i)/max(abs(f(i)),AbsTol(i)/RelTol))‖ ≤ RelTol
```

The following table describes the error tolerance properties. Use bvpset to set these properties.

BVP Error Tolerance Properties

Property	Value	Description
RelTol	Positive scalar {1e-3}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $f(x, y)$. The default, 1e-3, corresponds to 0.1% accuracy.
AbsTol	Positive scalar or vector {1e-6}	Absolute error tolerances that apply to the corresponding components of the residual vector. AbsTol(i) is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by bvp4c differs from the vectorization used by the ODE solvers:

- For bvp4c, the ODE function must be vectorized with respect to the first argument as well as the second one, so that F([x1 x2 ...],[y1 y2 ...]) returns [F(x1,y1) F(x2,y2) ...].
- bvp4c benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Use bvpset to set this property.

Vectorization Properties

Property	Value	Description
Vectorized	on | {off}	Set on to inform bvp4c that you have coded the ODE function F so that F([x1 x2 ...],[y1 y2 ...]) returns [F(x1,y1) F(x2,y2) ...]. This allows the solver to reduce the number of function evaluations, and may significantly reduce solution time.
		With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the shockbvp example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (.*) operator.
| | | ```
function dydx = shockODE(x,y,e)
pix = pi*x;
dydx = [ y(2,:)
         -x/e.*y(2,:)-pi^2*cos(pix)-pix/e.*sin(pix)
       ];
``` |

Analytical Partial Derivatives

By default, the bvp4c solver approximates all partial derivatives with finite differences. bvp4c can be more efficient if you provide analytical partial derivatives $\partial f/\partial y$ of the differential equations, and analytical partial derivatives, $\partial bc/\partial ya$ and $\partial bc/\partial yb$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties. Use bvpset to set these properties.

BVP Analytical Partial Derivative Properties

Property	Value	Description
FJacobian	Function	The function computes the analytical partial derivatives of $f(x,y)$. When solving $y' = f(x,y)$, set this property to `@fjac` if `dfdy = fjac(x,y)` evaluates the Jacobian $\partial f / \partial y$. If the problem involves unknown parameters p, `[dfdy,dfdp] = fjac(x,y,p)` must also return the partial derivative $\partial f / \partial p$. For problems with constant partial derivatives, set this property to the value of `dfdy` or to a cell array `{dfdy,dfdp}`.
BCJacobian	Function	The function computes the analytical partial derivatives of $bc(ya,yb)$. For boundary conditions $bc(ya,yb)$, set this property to `@bcjac` if `[dbcdya,dbcdyb] = bcjac(ya,yb)` evaluates the partial derivatives $\partial bc / \partial ya$, and $\partial bc / \partial yb$. If the problem involves unknown parameters p, `[dbcdya,dbcdyb,dbcdp] = bcjac(ya,yb,p)` must also return the partial derivative $\partial bc / \partial p$. For problems with constant partial derivatives, set this property to a cell array `{dbcdya,dbcdyb}` or `{dbcdya,dbcdyb,dbcdp}`.

Singular BVPs

bvp4c can solve singular problems of the form

$$y' = S\frac{y}{x} + f(x,y,p)$$

posed on the interval $[0,b]$ where $b > 0$. For such problems, specify the constant matrix S as the value of `SingularTerm`. For equations of this form, odefun evaluates only the $f(x,y,p)$ term, where p represents unknown parameters, if any.

Singular BVP Property

Property	Value	Description
SingularTerm	Constant matrix	Singular term of singular BVPs. Set to the constant matrix S for equations of the form $$y' = S\frac{y}{x} + f(x, y, p)$$ posed on the interval $[0, b]$ where $b > 0$.

Mesh Size Property

bvp4c solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (n) and the number of mesh points in the current mesh (N). When the allowed number of mesh points is exhausted, the computation stops, bvp4c displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an excellent initial guess for computations restarted with relaxed error tolerances or an increased value of NMax.

The following table describes the mesh size property. Use bvpset to set this property.

BVP Mesh Size Property

Property	Value	Description
NMax	positive integer {floor(1000/n)}	Maximum number of mesh points allowed when solving the BVP, where n is the number of differential equations in the problem. The default value of NMax limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of NMax should be sufficient to obtain an accurate solution.

Solution Statistic Property

The Stats property lets you view solution statistics.

The following table describes the solution statistics property. Use `bvpset` to set this property.

BVP Solution Statistic Property

Property	Value	Description
Stats	on | {off}	Specifies whether statistics about the computations are displayed. If the `stats` property is `on`, after solving the problem, `bvp4c` displays: • The number of points in the mesh • The maximum residual of the solution • The number of times it called the differential equation function `odefun` to evaluate $f(x, y)$ • The number of times it called the boundary condition function `bcfun` to evaluate $bc(y(a), y(b))$

Partial Differential Equations

This section describes how to use MATLAB to solve initial-boundary value problems for partial differential equations (PDEs). It provides:

- A summary of the MATLAB PDE functions and examples
- An introduction to PDEs
- A description of the PDE solver and its syntax
- General instructions for representing a PDE in MATLAB, including an example
- A discussion about changing default integration properties
- An example of solving a real-life problem

PDE Function Summary

MATLAB PDE Solver

This is the MATLAB PDE solver.

PDE Initial-Boundary Value Problem Solver	
pdepe	Solve initial-boundary value problems for systems of parabolic and elliptic PDEs in one space variable and time.

PDE Helper Function

PDE Helper Function	
pdeval	Evaluate the numerical solution of a PDE using the output of pdepe.

PDE Examples

These examples illustrate some problems you can solve using the MATLAB PDE solver. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the PDE examples, and also run them. Click on the link to invoke the browser, or type odeexamples('pde') at the command line.

Example	Description
pdex1	Simple PDE that illustrates the straightforward formulation, computation, and plotting of the solution
pdex2	Problem that involves discontinuities
pdex3	Problem that requires computing values of the partial derivative
pdex4	System of two PDEs whose solution has boundary layers at both ends of the interval and changes rapidly for small t
pdex5	System of PDEs with step functions as initial conditions

Introduction to PDE Problems

pdepe solves systems of PDEs in one spatial variable x and time t, of the form

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x}\left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right) \qquad \textbf{(14-4)}$$

The PDEs hold for $t_0 \leq t \leq t_f$ and $a \leq x \leq b$. The interval $[a, b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then $a \geq 0$ must also hold.

In Equation 14-4, $f(x, t, u, \partial u / \partial x)$ is a flux term and $s(x, t, u, \partial u / \partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x, t, u, \partial u / \partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x

if they are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

At the initial time $t = t_0$, for all x the solution components satisfy initial conditions of the form

$$u(x, t_0) = u_0(x) \tag{14-5}$$

At the boundary $x = a$ or $x = b$, for all t the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t)f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0 \tag{14-6}$$

$q(x, t)$ is a diagonal matrix with elements that are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u / \partial x$. Also, of the two coefficients, only p can depend on u.

MATLAB Partial Differential Equation Solver

This section describes:

- The PDE solver, `pdepe`
- PDE solver basic syntax
- Additional PDE solver arguments

The PDE Solver

The MATLAB PDE solver, `pdepe`, solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. There must be at least one parabolic equation in the system.

The pdepe solver converts the PDEs to ODEs using a second-order accurate spatial discretization based on a set of nodes specified by the user. The discretization method is described in [9]. The time integration is done with `ode15s`. The `pdepe` solver exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when Equation 14-4 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern. `ode15s` changes both the time step and the formula dynamically.

After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations

are not "consistent" with the discretization, pdepe tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, pdepe can find consistent initial conditions close to the given ones. If pdepe displays a message that it has difficulty finding consistent initial conditions, try refining the mesh. No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

PDE Solver Basic Syntax

The basic syntax of the solver is

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

Note Correspondences given are to terms used in "Introduction to PDE Problems" on page 14-109.

The input arguments are:

m	Specifies the symmetry of the problem. m can be 0 = slab, 1 = cylindrical, or 2 = spherical. It corresponds to m in Equation 14-4.
pdefun	Function that defines the components of the PDE. It computes the terms c, f, and s in Equation 14-4, and has the form

```
[c,f,s] = pdefun(x,t,u,dudx)
```

where x and t are scalars, and u and dudx are vectors that approximate the solution u and its partial derivative with respect to x. c, f, and s are column vectors. c stores the diagonal elements of the matrix c.

icfun	Function that evaluates the initial conditions. It has the form

```
u = icfun(x)
```

When called with an argument x, icfun evaluates and returns the initial values of the solution components at x in the column vector u.

bcfun   Function that evaluates the terms p and q of the boundary conditions. It has the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

where ul is the approximate solution at the left boundary xl $= a$ and ur is the approximate solution at the right boundary xr $= b$. pl and ql are column vectors corresponding to p and the diagonal of q evaluated at xl. Similarly, pr and qr correspond to xr. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. pdepe imposes this boundary condition automatically and it ignores values returned in pl and ql.

xmesh   Vector [x0, x1, …, xn] specifying the points at which a numerical solution is requested for every value in tspan. x0 and xn correspond to a and b, respectively.

Second-order approximation to the solution is made on the mesh specified in xmesh. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in xmesh. The cost depends strongly on the length of xmesh. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

The elements of xmesh must satisfy x0 < x1 < … < xn. The length of xmesh must be ≥ 3.

tspan   Vector [t0, t1, …, tf] specifying the points at which a solution is requested for every value in xmesh. t0 and tf correspond to t_0 and t_f, respectively.

pdepe performs the time integration with an ODE solver that selects both the time step and formula dynamically. The solutions at the points specified in tspan are obtained using the natural continuous extension of the integration formulas. The elements of tspan merely specify where you want answers and the cost depends weakly on the length of tspan.

The elements of tspan must satisfy t0 < t1 < … < tf. The length of tspan must be ≥ 3.

The output argument sol is a three-dimensional array, such that:

- `sol(:,:,k)` approximates component `k` of the solution u.
- `sol(i,:,k)` approximates component `k` of the solution at time `tspan(i)` and mesh points `xmesh(:)`.
- `sol(i,j,k)` approximates component `k` of the solution at time `tspan(i)` and the mesh point `xmesh(j)`.

Additional PDE Solver Arguments

For more advanced applications, you can also specify as input arguments solver options and additional parameters that are passed to the PDE functions.

`options`	Structure of optional parameters that change the default integration properties. This is the seventh input argument.

```
sol = pdepe(m,pdefun,icfun,bcfun,...
            xmesh,tspan,options)
```

See "Changing PDE Integration Properties" on page 14-119 for more information.

`p1,p2...`	Parameters that the solver passes to `pdefun`, `icfun`, and `bcfun`.

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,...
            options,p1,p2...)
```

The solver passes any input parameters that follow the `options` argument to `pdefun`, `icfun`, and `bcfun` every time it calls them. Use `options = []` as a placeholder if you set no options. In the `pdefun` argument list, parameters follow `x`, `t`, `u`, and `dudx`.

```
f = pdefun(x,t,u,dudx,p1,p2,...)
```

In the `icfun` argument list, parameters follow `x`.

```
res = icfun(x,p1,p2,...)
```

In the `bcfun` argument list, parameters follow `xl`, `ul`, `xr`, `ur`, and `t`.

```
res = bcfun(xl,ul,xr,ur,t,p1,p2,...)
```

See the `pdex3` demo for an example.

Solving PDE Problems

This section describes:

- The process for solving PDE problems using the MATLAB solver, pdepe
- Evaluating the solution at specific points

Example: A Single PDE

This example illustrates the straightforward formulation, solution, and plotting of the solution of a single PDE

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$. At $t = 0$, the solution satisfies the initial condition

$$u(x, 0) = \sin \pi x$$

At $x = 0$ and $x = 1$, the solution satisfies the boundary conditions

$$u(0, t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

Note The demo pdex1 contains the complete code for this example. The demo uses subfunctions to place all functions it requires in a single M-file. To run the demo type pdex1 at the command line. See "PDE Solver Basic Syntax" on page 14-111 for more information.

---

1 Rewrite the PDE. Write the PDE in the form

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) \frac{\partial u}{\partial t} = x^{-m} \frac{\partial}{\partial x} \left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right)$$

This is the form shown in Equation 14-4 and expected by pdepe. See "Introduction to PDE Problems" on page 14-109 for more information. For this example, the resulting equation is

$$\pi^2 \frac{\partial u}{\partial t} = x^0 \frac{\partial}{\partial x}\left(x^0 \frac{\partial u}{\partial x}\right) + 0$$

with parameter $m = 0$ and the terms

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) = \pi^2$$

$$f\left(x, t, u, \frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$$

$$s\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0$$

2 Code the PDE. Once you rewrite the PDE in the form shown above (Equation 14-4) and identify the terms, you can code the PDE in a function that pdepe can use. The function must be of the form

```
[c,f,s] = pdefun(x,t,u,dudx)
```

where c, f, and s correspond to the c, f, and s terms. The code below computes c, f, and s for the example problem.

```
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
```

3 Code the initial conditions function. You must code the initial conditions in a function of the form

```
u = icfun(x)
```

The code below represents the initial conditions in the function pdex1ic.

```
function u0 = pdex1ic(x)
u0 = sin(pi*x);
```

4 Code the boundary conditions function. You must also code the boundary conditions in a function of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The boundary conditions, written in the same form as Equation 14-6, are

$$u(0, t) + 0 \cdot \frac{\partial u}{\partial x}(0, t) = 0 \qquad \text{at } x = 0$$

and

$$\pi e^{-t} + 1 \cdot \frac{\partial u}{\partial x}(1, t) = 0 \qquad \text{at } x = 1$$

The code below evaluates the components $p(x, t, u)$ and $q(x, t)$ of the boundary conditions in the function pdex1bc.

```
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In the function pdex1bc, pl and ql correspond to the left boundary conditions ($x = 0$), and pr and qr correspond to the right boundary condition ($x = 1$).

5 **Select mesh points for the solution.** Before you use the MATLAB PDE solver, you need to specify the mesh points (t, x) at which you want pdepe to evaluate the solution. Specify the points as vectors t and x.

The vectors t and x play different roles in the solver (see "MATLAB Partial Differential Equation Solver" on page 14-110). In particular, the cost and the accuracy of the solution depend strongly on the length of the vector x. However, the computation is much less sensitive to the values in the vector t.

This example requests the solution on the mesh produced by 20 equally spaced points from the spatial interval [0,1] and five values of t from the time interval [0,2].

```
x = linspace(0,1,20);
t = linspace(0,2,5);
```

6 **Apply the PDE solver.** The example calls pdepe with m = 0, the functions pdex1pde, pdex1ic, and pdex1bc, and the mesh defined by x and t at which pdepe is to evaluate the solution. The pdepe function returns the numerical solution in a three-dimensional array sol, where sol(i,j,k) approximates the kth component of the solution, u_k, evaluated at t(i) and x(j).

```
m = 0;
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
```

This example uses `@` to pass `pdex1pde`, `pdex1ic`, and `pdex1bc` as function handles to `pdepe`.

Note See the `function_handle` (@), `func2str`, and `str2func` reference pages, and the "Function Handles" chapter of "Programming and Data Types" in the MATLAB documentation for information about function handles.

7 View the results. Complete the example by displaying the results:

a Extract and display the first solution component. In this example, the solution u has only one component, but for illustrative purposes, the example "extracts" it from the three-dimensional array. The surface plot shows the behavior of the solution.

```
u = sol(:,:,1);
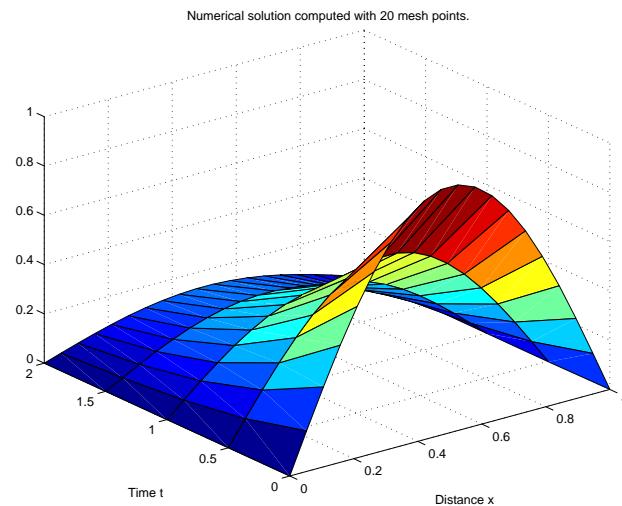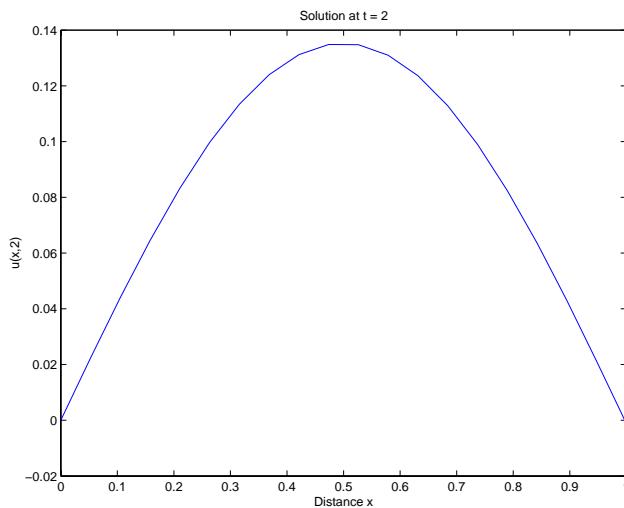
surf(x,t,u)
title('Numerical solution computed with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```

Numerical solution computed with 20 mesh points.

b Display a solution profile at t_f, the final value of t. In this example, t_f = t = 2. See "Evaluating the Solution at Specific Points" on page 14-119 for more information.

```
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
```

Solution at t = 2

Evaluating the Solution at Specific Points

After obtaining and plotting the solution above, you might be interested in a solution profile for a particular value of t, or the time changes of the solution at a particular point x. The kth column u(:,k) (of the solution extracted in step 7) contains the time history of the solution at x(k). The jth row u(j,:) contains the solution profile at t(j).

Using the vectors x and u(j,:), and the helper function pdeval, you can evaluate the solution u and its derivative $\partial u / \partial x$ at any set of points xout

```
[uout,DuoutDx] = pdeval(m,x,u(j,:),xout)
```

The example pdex3 uses pdeval to evaluate the derivative of the solution at xout = 0. See pdeval for details.

Changing PDE Integration Properties

The default integration properties in the MATLAB PDE solver are selected to handle common problems. In some cases, you can improve solver performance by overriding these defaults. You do this by supplying pdepe with one or more property values in an options structure.

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Use `odeset` to create the options structure. Only those options of the underlying ODE solver shown in the following table are available for `pdepe`. The defaults obtained by leaving off the input argument `options` are generally satisfactory. "Changing ODE Integration Properties" on page 14-16 tells you how to create the structure and describes the properties.

PDE Property Categories

Properties Category	Property Name
Error control	`RelTol, AbsTol, NormControl`
Step-size	`InitialStep, MaxStep`

Example: Electrodynamics Problem

This example illustrates the solution of a system of partial differential equations. The problem is taken from electrodynamics. It has boundary layers at both ends of the interval, and the solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$. The equations hold on an interval $0 \leq x \leq 1$ for times $t \geq 0$.

The solution u satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$
$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$

$$u_2(0, t) \equiv 0$$

$$u_1(1, t) \equiv 1$$

$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0$$

Note The demo `pdex4` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type `pdex4` at the command line. See "PDE Solver Basic Syntax" on page 14-111 and "Solving PDE Problems" on page 14-114 for more information.

1 Rewrite the PDE. In the form expected by `pdepe`, the equations are

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} .* \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected by `pdepe`, the left boundary condition is

$$\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

2 Code the PDE. After you rewrite the PDE in the form shown above, you can code it as a function that `pdepe` can use. The function must be of the form

```
[c,f,s] = pdefun(x,t,u,dudx)
```

where c, f, and s correspond to the c, f, and s terms in Equation 14-4.

```
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
```

3 **Code the initial conditions function.** The initial conditions function must be of the form

```
u = icfun(x)
```

The code below represents the initial conditions in the function pdex4ic.

```
function u0 = pdex4ic(x);
u0 = [1; 0];
```

4 **Code the boundary conditions function.** The boundary conditions functions must be of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The code below evaluates the components $p(x, t, u)$ and $q(x, t)$ (Equation 14-6) of the boundary conditions in the function pdex4bc.

```
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
```

5 **Select mesh points for the solution.** The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, output times must be selected accordingly. There are boundary layers in the solution at both ends of [0,1], so mesh points must be placed there to resolve these sharp changes. Often some experimentation is needed to select the mesh that reveals the behavior of the solution.

```
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
```

```
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];
```

6 **Apply the PDE solver.** The example calls pdepe with m = 0, the functions pdex4pde, pdex4ic, and pdex4bc, and the mesh defined by x and t at which pdepe is to evaluate the solution. The pdepe function returns the numerical solution in a three-dimensional array sol, where sol(i,j,k) approximates the kth component of the solution, u_k, evaluated at t(i) and x(j).

```
m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
```

7 **View the results.** The surface plots show the behavior of the solution components.

```
u1 = sol(:,:,1);
u2 = sol(:,:,2);
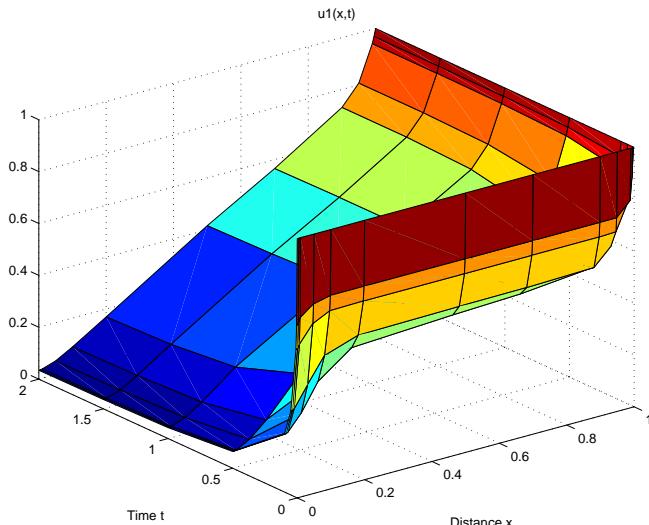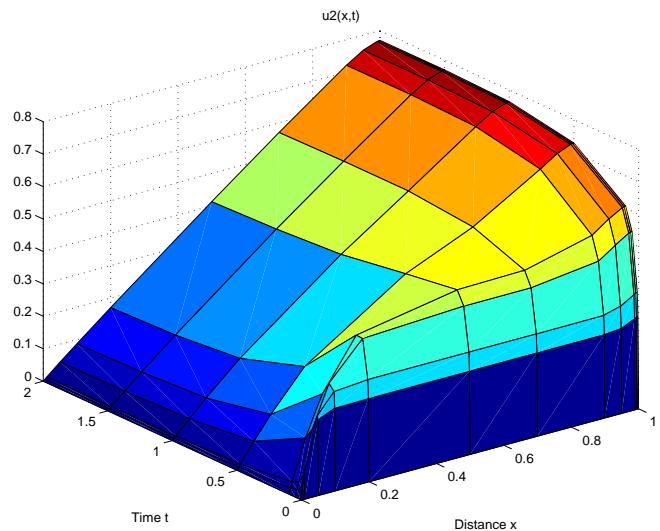
figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')
```

```
figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
```

Selected Bibliography

[1] Ascher, U., R. Mattheij, and R. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM, Philadelphia, PA, 1995, p. 372.

[2] Cebeci, T. and H. B. Keller, "Shooting and Parallel Shooting Methods for Solving the Falkner-Skan Boundary-layer Equation," *J. Comp. Phys.*, Vol. 7, 1971, pp. 289-300.

[3] Hairer, E., and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*, Springer-Verlag, Berlin, 1991, pp. 5-8.

[4] Hindmarsh, A. C., "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers," *SIGNUM Newsletter*, Vol. 15, 1980, pp. 10-11.

[5] Hindmarsh, A. C., and G. D. Byrne, "Applications of EPISODE: An Experimental Package for the Integration of Ordinary Differential Equations," *Numerical Methods for Differential Systems*, L. Lapidus and W. E. Schiesser eds., Academic Press, Orlando, FL, 1976, pp 147-166.

[6] Ottesen, J. T., "Modelling of the Baroflex-Feedback Mechanism with Time-Delay," *J. Math. Biol.*, Vol. 36, 1997.

[7] Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall Mathematics, 1994.

[8] Shampine, L. F., and M. K. Gordon, *Computer Solution of Ordinary Differential Equations*, W.H. Freeman & Co., 1975.

[9] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1-32.

Sparse Matrices

MATLAB supports *sparse matrices*, matrices that contain a small proportion of nonzero elements. This characteristic provides advantages in both matrix storage space and computation time. This chapter explains how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations. It includes:

Function Summary

The sparse matrix functions are located in the `sparfun` directory in the MATLAB `toolbox` directory.

Function Summary

Category	Function	Description
Elementary sparse matrices	speye	Sparse identity matrix.
	sprand	Sparse uniformly distributed random matrix.
	sprandn	Sparse normally distributed random matrix.
	sprandsym	Sparse random symmetric matrix.
	spdiags	Sparse matrix formed from diagonals.
Full to sparse conversion	sparse	Create sparse matrix.
	full	Convert sparse matrix to full matrix.
	find	Find indices of nonzero elements.
	spconvert	Import from sparse matrix external format.
Working with sparse matrices	nnz	Number of nonzero matrix elements.
	nonzeros	Nonzero matrix elements.
	nzmax	Amount of storage allocated for nonzero matrix elements.
	spones	Replace nonzero sparse matrix elements with ones.
	spalloc	Allocate space for sparse matrix.
	issparse	True for sparse matrix.
	spfun	Apply function to nonzero matrix elements.
	spy	Visualize sparsity pattern.

Function Summary (Continued)

Category	Function	Description
Graph theory	gplot	Plot graph, as in "graph theory."
	etree	Elimination tree.
	etreeplot	Plot elimination tree.
	treelayout	Lay out tree or forest.
	treeplot	Plot picture of tree.
Reordering algorithms	colamd	Column approximate minimum degree permutation.
	colmmd	Column minimum degree permutation.
	symamd	Symmetric approximate minimum degree permutation.
	symmmd	Symmetric minimum degree permutation.
	symrcm	Symmetric reverse Cuthill-McKee permutation.
	colperm	Column permutation.
	randperm	Random permutation.
	dmperm	Dulmage-Mendelsohn permutation.
Linear algebra	eigs	A few eigenvalues.
	svds	A few singular values.
	luinc	Incomplete LU factorization.
	cholinc	Incomplete Cholesky factorization.
	normest	Estimate the matrix 2-norm.
	condest	1-norm condition number estimate.
	sprank	Structural rank.

Function Summary (Continued)

Category	Function	Description
Linear equations (iterative methods)	bicg	BiConjugate Gradients Method.
	bicgstab	BiConjugate Gradients Stabilized Method.
	cgs	Conjugate Gradients Squared Method.
	gmres	Generalized Minimum Residual Method.
	lsqr	LSQR implementation of Conjugate Gradients on the Normal Equations.
	minres	Minimum Residual Method.
	pcg	Preconditioned Conjugate Gradients Method.
	qmr	Quasi-Minimal Residual Method.
	symmlq	Symmetric LQ method
Miscellaneous	spaugment	Form least squares augmented system.
	spparms	Set parameters for sparse matrix routines.
	symbfact	Symbolic factorization analysis.

Introduction

Sparse matrices are a special class of matrices that contain a significant number of zero-valued elements. This property allows MATLAB to:

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements.

This section provides information about:

- Sparse matrix storage
- General storage information
- Creating sparse matrices
- Importing sparse matrices

Sparse Matrix Storage

For full matrices, MATLAB stores internally every matrix element. Zero-valued elements require the same amount of storage space as any other matrix element. For sparse matrices, however, MATLAB stores only the nonzero elements and their indices. For large matrices with a high percentage of zero-valued elements, this scheme significantly reduces the amount of memory required for data storage.

MATLAB uses three arrays internally to store sparse matrices with real elements. Consider an m-by-n sparse matrix with nnz nonzero entries stored in arrays of length nzmax:

- The first array contains all the nonzero elements of the array in floating-point format. The length of this array is equal to nzmax.
- The second array contains the corresponding integer row indices for the nonzero elements stored in the first nnz entries. This array also has length equal to nzmax.
- The third array contains n integer pointers to the start of each column in the other arrays and an additional pointer that marks the end of those arrays. The length of the third array is n+1.

This matrix requires storage for nzmax floating-point numbers and nzmax+n+1 integers. At 8 bytes per floating-point number and 4 bytes per integer, the total number of bytes required to store a sparse matrix is

```
8*nzmax + 4*(nzmax+n+1)
```

Sparse matrices with complex elements are also possible. In this case, MATLAB uses a fourth array with nnz elements to store the imaginary parts of the nonzero elements. An element is considered nonzero if either its real or imaginary part is nonzero.

General Storage Information

The whos command provides high-level information about matrix storage, including size and storage class. For example, this whos listing shows information about sparse and full versions of the same matrix.

```
whos
  Name          Size          Bytes  Class

  M_full      1100x1100      9680000  double array
  M_sparse    1100x1100         4404  sparse array

Grand total is 1210000 elements using 9684404 bytes
```

Notice that the number of bytes used is much less in the sparse case, because zero-valued elements are not stored. In this case, the density of the sparse matrix is 4404/9680000, or approximately .00045%.

Creating Sparse Matrices

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques.

The *density* of a matrix is the number of non-zero elements divided by the total number of matrix elements. Matrices with very low density are often good candidates for use of the sparse format.

Converting Full to Sparse

You can convert a full matrix to sparse storage using the sparse function with a single argument.

```
S = sparse(A)
```

For example

```
A = [ 0    0    0    5
      0    2    0    0
      1    3    0    0
      0    0    4    0];

S = sparse(A)
```

produces

```
 S =

   (3,1)        1
   (2,2)        2
   (3,2)        3
   (4,3)        4
   (1,4)        5
```

The printed output lists the nonzero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

You can convert a sparse matrix to full storage using the full function, provided the matrix order is not too large. For example A = full(S) reverses the example conversion.

Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

Creating Sparse Matrices Directly

You can create a sparse matrix from a list of nonzero elements using the sparse function with five arguments.

```
S = sparse(i,j,s,m,n)
```

i and j are vectors of row and column indices, respectively, for the nonzero elements of the matrix. s is a vector of nonzero values whose indices are specified by the corresponding (i,j) pairs. m is the row dimension for the resulting matrix, and n is the column dimension.

The matrix S of the previous example can be generated directly with

```
S = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)

S =

    (3,1)         1
    (2,2)         2
    (3,2)         3
    (4,3)         4
    (1,4)         5
```

The sparse command has a number of alternate forms. The example above uses a form that sets the maximum number of nonzero elements in the matrix to length(s). If desired, you can append a sixth argument that specifies a larger maximum, allowing you to add nonzero elements later without reallocating the sparse matrix.

Example: Generating a Second Difference Operator

The matrix representation of the second difference operator is a good example of a sparse matrix. It is a tridiagonal matrix with -2s on the diagonal and 1s on the super- and subdiagonal. There are many ways to generate it – here's one possibility.

```
D = sparse(1:n,1:n,-2*ones(1,n),n,n);
E = sparse(2:n,1:n-1,ones(1,n-1),n,n);
S = E+D+E'
```

For n = 5, MATLAB responds with

```
S =

    (1,1)        -2
    (2,1)         1
    (1,2)         1
    (2,2)        -2
    (3,2)         1
    (2,3)         1
    (3,3)        -2
    (4,3)         1
    (3,4)         1
    (4,4)        -2
```

```
(5,4)        1
(4,5)        1
(5,5)       -2
```

Now F = full(S) displays the corresponding full matrix.

```
F = full(S)

F =

   -2    1    0    0    0
    1   -2    1    0    0
    0    1   -2    1    0
    0    0    1   -2    1
    0    0    0    1   -2
```

Creating Sparse Matrices from Their Diagonal Elements

Creating sparse matrices based on their diagonal elements is a common operation, so the function spdiags handles this task. Its syntax is

```
S = spdiags(B,d,m,n)
```

To create an output matrix S of size m-by-n with elements on p diagonals:

- B is a matrix of size $\min(m,n)$-by-p. The columns of B are the values to populate the diagonals of S.

- d is a vector of length p whose integer elements specify which diagonals of S to populate.

That is, the elements in column j of B fill the diagonal specified by element j of d.

Note If a column of B is longer than the diagonal it's replacing, super-diagonals are taken from the lower part of the column of B, and sub-diagonals are taken from the upper part of the column of B.

As an example, consider the matrix B and the vector d.

```
B = [ 41    11    0
      52    22    0
```

```
           63       33       13
           74       44       24 ];

    d = [-3
          0
          2];
```

Use these matrices to create a 7-by-4 sparse matrix A.

```
    A = spdiags(B,d,7,4)

    A =

        (1,1)           11
        (4,1)           41
        (2,2)           22
        (5,2)           52
        (1,3)           13
        (3,3)           33
        (6,3)           63
        (2,4)           24
        (4,4)           44
        (7,4)           74
```

In its full form, A looks like this.

```
    full(A)

    ans =

        11        0       13        0
         0       22        0       24
         0        0       33        0
        41        0        0       44
         0       52        0        0
         0        0       63        0
         0        0        0       74
```

spdiags can also extract diagonal elements from a sparse matrix, or replace matrix diagonal elements with new values. Type help spdiags for details.

Importing Sparse Matrices from Outside MATLAB

You can import sparse matrices from computations outside MATLAB. Use the `spconvert` function in conjunction with the `load` command to import text files containing lists of indices and nonzero elements. For example, consider a three-column text file `T.dat` whose first column is a list of row indices, second column is a list of column indices, and third column is a list of nonzero values. These statements load `T.dat` into MATLAB and convert it into a sparse matrix `S`:

```
load T.dat
S = spconvert(T)
```

The `save` and `load` commands can also process sparse matrices stored as binary data in MAT-files.

Viewing Sparse Matrices

MATLAB provides a number of functions that let you get quantitative or graphical information about sparse matrices.

This section provides information about:

- Obtaining information about nonzero elements
- Viewing graphs of sparse matrices
- Finding indices and values of nonzero elements

Information About Nonzero Elements

There are several commands that provide high-level information about the nonzero elements of a sparse matrix:

- nnz returns the number of nonzero elements in a sparse matrix.
- nonzeros returns a column vector containing all the nonzero elements of a sparse matrix.
- nzmax returns the amount of storage space allocated for the nonzero entries of a sparse matrix.

To try some of these, load the supplied sparse matrix west0479, one of the Harwell-Boeing collection.

```
load west0479
whos
  Name          Size          Bytes  Class

   west0479    479x479        24576  sparse array
```

This matrix models an eight-stage chemical distillation column.

Try these commands.

```
nnz(west0479)

ans =
1887

format short e
west0479
```

```
west0479 =

  (25,1)        1.0000e+00
  (31,1)       -3.7648e-02
  (87,1)       -3.4424e-01
  (26,2)        1.0000e+00
  (31,2)       -2.4523e-02
  (88,2)       -3.7371e-01
  (27,3)        1.0000e+00
  (31,3)       -3.6613e-02
  (89,3)       -8.3694e-01
  (28,4)        1.3000e+02
      .
      .
      .

nonzeros(west0479);
    ans =

   1.0000e+00
  -3.7648e-02
  -3.4424e-01
   1.0000e+00
  -2.4523e-02
  -3.7371e-01
   1.0000e+00
  -3.6613e-02
  -8.3694e-01
   1.3000e+02
      .
      .
      .
```

Note Use **Ctrl+C** to stop the nonzeros listing at any time.

Note that initially nnz has the same value as nzmax by default. That is, the number of nonzero elements is equivalent to the number of storage locations

allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of nnz, but not that of nzmax.

However, you can add as many nonzero elements to the matrix as desired. You are not constrained by the original value of nzmax.

Viewing Sparse Matrices Graphically

It is often useful to use a graphical format to view the distribution of the nonzero elements within a sparse matrix. The MATLAB spy function produces a template view of the sparsity structure, where each point on the graph represents the location of a nonzero array element.

For example,

```
spy(west0479)
```

The find Function and Sparse Matrices

For any matrix, full or sparse, the find function returns the indices and values
of nonzero elements. Its syntax is

```
[i,j,s] = find(S)
```

find returns the row indices of nonzero values in vector i, the column indices
in vector j, and the nonzero values themselves in the vector s. The example
below uses find to locate the indices and values of the nonzeros in a sparse
matrix. The sparse function uses the find output, together with the size of the
matrix, to recreate the matrix.

```
[i,j,s] = find(S)
[m,n] = size(S)
S = sparse(i,j,s,m,n)
```

Example: Adjacency Matrices and Graphs

This section includes:

- An introduction to adjacency matrices
- Instructions for graphing adjacency matrices with `gplot`
- A Bucky ball example, including information about using spy plots to illustrate fill-in and distance
- An airflow model example

Introduction to Adjacency Matrices

The formal mathematical definition of a *graph* is a set of points, or nodes, with specified connections between them. An economic model, for example, is a graph with different industries as the nodes and direct economic ties as the connections. The computer software industry is connected to the computer hardware industry, which, in turn, is connected to the semiconductor industry, and so on.

This definition of a graph lends itself to matrix representation. The *adjacency matrix* of an *undirected* graph is a matrix whose `(i,j)`th and `(j,i)`th entries are 1 if node `i` is connected to node `j`, and 0 otherwise. For example, the adjacency matrix for a diamond-shaped graph looks like

Since most graphs have relatively few connections per node, most adjacency matrices are sparse. The actual locations of the nonzero elements depend on how the nodes are numbered. A change in the numbering leads to permutation

of the rows and columns of the adjacency matrix, which can have a significant effect on both the time and storage requirements for sparse matrix computations.

Graphing Using Adjacency Matrices

The MATLAB `gplot` function creates a graph based on an adjacency matrix and a related array of coordinates. To try `gplot`, create the adjacency matrix shown above by entering

```
A = [0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0];
```

The columns of `gplot`'s coordinate array contain the Cartesian coordinates for the corresponding node. For the diamond example, create the array by entering

```
xy = [1 3; 2 1; 3 3; 2 5];
```

This places the first node at location `(1,3)`, the second at location `(2,1)`, the third at location `(3,3)`, and the fourth at location `(2,5)`. To view the resulting graph, enter

```
gplot(A,xy)
```

The Bucky Ball

One interesting construction for graph analysis is the *Bucky ball*. This is composed of 60 points distributed on the surface of a sphere in such a way that the distance from any point to its nearest neighbors is the same for all the points. Each point has exactly three neighbors. The Bucky ball models four different physical objects:

- The geodesic dome popularized by Buckminster Fuller
- The C_{60} molecule, a form of pure carbon with 60 atoms in a nearly spherical configuration
- In geometry, the truncated icosahedron
- In sports, the seams in a soccer ball

The Bucky ball adjacency matrix is a 60-by-60 symmetric matrix B. B has three nonzero elements in each row and column, for a total of 180 nonzero values. This matrix has important applications related to the physical objects listed earlier. For example, the eigenvalues of B are involved in studying the chemical properties of C_{60}.

15-17

To obtain the Bucky ball adjacency matrix, enter

```
B = bucky;
```

At order 60, and with a density of 5%, this matrix does not require sparse techniques, but it does provide an interesting example.

You can also obtain the coordinates of the Bucky ball graph using

```
[B,v] = bucky;
```

This statement generates v, a list of *xyz*-coordinates of the 60 points in 3-space equidistributed on the unit sphere. The function gplot uses these points to plot the Bucky ball graph.

```
gplot(B,v)
axis equal
```


It is not obvious how to number the nodes in the Bucky ball so that the resulting adjacency matrix reflects the spherical and combinatorial symmetries of the graph. The numbering used by bucky.m is based on the pentagons inherent in the ball's structure.

The vertices of one pentagon are numbered 1 through 5, the vertices of an adjacent pentagon are numbered 6 through 10, and so on. The picture on the following page shows the numbering of half of the nodes (one hemisphere); the numbering of the other hemisphere is obtained by a reflection about the equator. Use gplot to produce a graph showing half the nodes. You can add the node numbers using a for loop.

```
k = 1:30;
gplot(B(k,k),v);
axis square
for j = 1:30, text(v(j,1),v(j,2), int2str(j)); end
```


To view a template of the nonzero locations in the Bucky ball's adjacency matrix, use the spy function:

```
spy(B)
```

The node numbering that this model uses generates a spy plot with 12 groups of five elements, corresponding to the 12 pentagons in the structure. Each node is connected to two other nodes within its pentagon and one node in some other pentagon. Since the nodes within each pentagon have consecutive numbers, most of the elements in the first super- and sub-diagonals of B are nonzero. In addition, the symmetry of the numbering about the equator is apparent in the symmetry of the spy plot about the antidiagonal.

Graphs and Characteristics of Sparse Matrices

Spy plots of the matrix powers of B illustrate two important concepts related to sparse matrix operations, fill-in and distance. spy plots help illustrate these concepts.

```
spy(B^2)
spy(B^3)
spy(B^4)
spy(B^8)
```

Fill-in is generated by operations like matrix multiplication. The product of two or more matrices usually has more nonzero entries than the individual terms, and so requires more storage. As p increases, B^p fills in and spy(B^p) gets more dense.

The *distance* between two nodes in a graph is the number of steps on the graph necessary to get from one node to the other. The spy plot of the p-th power of B shows the nodes that are a distance p apart. As p increases, it is possible to get to more and more nodes in p steps. For the Bucky ball, B^8 is almost completely full. Only the antidiagonal is zero, indicating that it is possible to get from any node to any other node, except the one directly opposite it on the sphere, in eight steps.

15-21

An Airflow Model

A calculation performed at NASA's Research Institute for Applications of Computer Science involves modeling the flow over an airplane wing with two trailing flaps.

In a two-dimensional model, a triangular grid surrounds a cross section of the wing and flaps. The partial differential equations are nonlinear and involve several unknowns, including hydrodynamic pressure and two components of velocity. Each step of the nonlinear iteration requires the solution of a sparse linear system of equations. Since both the connectivity and the geometric location of the grid points are known, the gplot function can produce the graph shown above.

In this example, there are 4253 grid points, each of which is connected to between 3 and 9 others, for a total of 28831 nonzeros in the matrix, and a density equal to 0.0016. This spy plot shows that the node numbering yields a definite band structure.

The Laplacian of the mesh.

nz = 28831

Sparse Matrix Operations

Most of the MATLAB standard mathematical functions work on sparse matrices just as they do on full matrices. In addition, MATLAB provides a number of functions that perform operations specific to sparse matrices. This section discusses:

- Computational considerations
- Standard mathematical operations
- Permutation and reordering
- Factorization
- Simultaneous linear equations
- Eigenvalues and singular values

Computational Considerations

The computational complexity of sparse operations is proportional to nnz, the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size m and column size n of the matrix, but is independent of the product m*n, the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities.

Standard Mathematical Operations

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or vector always produce output in full storage format. For example, the size function always returns a full vector, whether its input is full or sparse.
- Functions that accept scalars or vectors and return matrices, such as zeros, ones, rand, and eye, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of zeros(m,n) is simply sparse(m,n). The sparse analogs of rand and eye are sprand and speye, respectively. There is no sparse analog for the function ones.

- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If S is a sparse matrix, then chol(S) is also a sparse matrix, and diag(S) is a sparse vector. Columnwise functions such as max and sum also return sparse vectors, even though these vectors may be entirely nonzero. Important exceptions to this rule are the sparse and full functions.

- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If S is sparse and F is full, then S+F, S*F, and F\S are full, while S.*F and S&F are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.

- Matrix concatenation using either the cat function or square brackets produces sparse results for mixed operands.

- Submatrix indexing on the right side of an assignment preserves the storage format of the operand unless the result is a scalar. T = S(i,j) produces a sparse result if S is sparse and either i or j is a vector. It produces a full scalar if both i and j are scalars. Submatrix indexing on the left, as in T(i,j) = S, does not change the storage format of the matrix on the left.

Permutation and Reordering

A permutation of the rows and columns of a sparse matrix S can be represented in two ways:

- A permutation matrix P acts on the rows of S as P*S or on the columns as S*P'.

- A permutation vector p, which is a full vector containing a permutation of 1:n, acts on the rows of S as S(p,:), or on the columns as S(:,p).

For example, the statements

```
p = [1 3 4 2 5]
I = eye(5,5);
P = I(p,:);
e = ones(4,1);
S = diag(11:11:55) + diag(e,1) + diag(e,-1)
```

produce

```
p =

     1     3     4     2     5

P =

     1     0     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     1     0     0     0
     0     0     0     0     1

S =

    11     1     0     0     0
     1    22     1     0     0
     0     1    33     1     0
     0     0     1    44     1
     0     0     0     1    55
```

You can now try some permutations using the permutation vector p and the permutation matrix P. For example, the statements S(p,:) and P*S produce

```
ans =

    11     1     0     0     0
     0     1    33     1     0
     0     0     1    44     1
     1    22     1     0     0
     0     0     0     1    55
```

Similarly, S(:,p) and S*P' produce

```
ans =

    11     0     0     1     0
     1     1     0    22     0
     0    33     1     1     0
     0     1    44     0     1
     0     0     1     0    55
```

If P is a sparse matrix, then both representations use storage proportional to n and you can apply either to S in time proportional to nnz(S). The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return full row vectors with the exception of the pivoting permutation in LU (triangular) factorization, which returns a matrix compatible with earlier versions of MATLAB.

To convert between the two representations, let I = speye(n) be an identity matrix of the appropriate size. Then,

```
P = I(p,:)
P' = I(:,p)
p = (1:n)*P'
p = (P*(1:n)')'
```

The inverse of P is simply R = P'. You can compute the inverse of p with r(p) = 1:n.

```
r(p) = 1:5

r =

     1     4     2     3     5
```

Reordering for Sparsity

Reordering the columns of a matrix can often make its LU or QR factors sparser. Reordering the rows and columns can often make its Cholesky, factors sparser. The simplest such reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The function p = colperm(S) computes this column-count permutation. The colperm M-file has only a single line.

```
[ignore,p] = sort(full(sum(spones(S))));
```

This line performs these steps:

1 The inner call to spones creates a sparse matrix with ones at the location of every nonzero element in S.

2 The `sum` function sums down the columns of the matrix, producing a vector that contains the count of nonzeros in each column.

3 `full` converts this vector to full storage format.

4 `sort` sorts the values in ascending order. The second output argument from `sort` is the permutation that sorts this vector.

Reordering to Reduce Bandwidth

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix. It is not guaranteed to find the smallest possible bandwidth, but it usually does. The function `symrcm(A)` actually operates on the nonzero structure of the symmetric matrix `A + A'`, but the result is also useful for asymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense "long and thin."

Minimum Degree Ordering

The degree of a node in a graph is the number of connections to that node. This is the same as the number of off-diagonal nonzero elements in the corresponding row of the adjacency matrix. The minimum degree algorithm generates an ordering based on how these degrees are altered during Gaussian elimination or Cholesky factorization. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee.

MATLAB functions implement two methods for each of two types of matrices: `symamd` and `symmmd` for symmetric matrices, and `colamd` and `colmmd` for nonsymmetric matrices. `colamd` and `colmmd` also work for symmetric matrices of the form `A*A'` or `A'*A`.

Because the most time-consuming part of a minimum degree ordering algorithm is keeping track of the degree of each node, all four functions use an approximation to the degree, rather the exact degree. As a result:

• Factorizations obtained using `colmmd` and `symmmd` tend to have more nonzero elements than if the implementation used exact degrees.

• `colamd` and `symamd` use a tighter approximation than `colmmd` and `symmmd`. They generate orderings that are as good as could be obtained using exact degrees.

- colamd and symamd are faster than colmmd and symmmd, respectively. This is true particularly for very large matrices.

You can change various parameters associated with details of the algorithms using the spparms function.

For details on the algorithms used by colmmd and symmmd, see [4]. For details on the algorithms used by colamd and symamd, see [5]. The approximate degree used in colamd and symamd is based on [1].

Factorization

This section discusses four important factorization techniques for sparse matrices:

- LU, or triangular, factorization
- Cholesky factorization
- QR, or orthogonal, factorization
- Incomplete factorizations

LU Factorization

If S is a sparse matrix, the statement below returns three sparse matrices L, U, and P such that P*S = L*U.

```
[L,U,P] = lu(S)
```

lu obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix P has only n nonzero elements. As with dense matrices, the statement [L,U] = lu(S) returns a permuted unit lower triangular matrix and an upper triangular matrix whose product is S. By itself, lu(S) returns L and U in a single matrix without the pivot information.

The sparse LU factorization does not pivot for sparsity, but it does pivot for numerical stability. In fact, both the sparse factorization (line 1) and the full factorization (line 2) below produce the same L and U, even though the time and storage requirements might differ greatly.

```
[L,U] = lu(S)                  % Sparse factorization

[L,U] = sparse(lu(full(S)))  % Full factorization
```

You can control pivoting in sparse matrices using

```
lu(S,thresh)
```

where `thresh` is a pivot threshold in [0,1]. Pivoting occurs when the diagonal entry in a column has magnitude less than `thresh` times the magnitude of any sub-diagonal entry in that column. `thresh = 0` forces diagonal pivoting. `thresh = 1` is the default.

MATLAB automatically allocates the memory necessary to hold the sparse `L` and `U` factors during the factorization. MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data structures in advance.

Reordering and Factorization. If you obtain a good column permutation p that reduces fill-in, perhaps from `symrcm` or `colamd`, then computing `lu(S(:,p))` takes less time and storage than computing `lu(S)`. Two permutations are the symmetric reverse Cuthill-McKee ordering and the symmetric minimum degree ordering.

```
r = symrcm(B);
m = symamd(B);
```

The three spy plots produced by the lines below show the three adjacency matrices of the Bucky Ball graph with these three different numberings. The local, pentagon-based structure of the original numbering is not evident in the other three.

```
spy(B)
spy(B(r,r))
spy(B(m,m))
```

The reverse Cuthill-McKee ordering, `r`, reduces the bandwidth and concentrates all the nonzero elements near the diagonal. The approximate minimum degree ordering, `m`, produces a fractal-like structure with large blocks of zeros.

To see the fill-in generated in the LU factorization of the Bucky ball, use `speye(n,n)`, the sparse identity matrix, to insert -3s on the diagonal of `B`.

```
B = B - 3*speye(n,n);
```

Since each row sum is now zero, this new `B` is actually singular, but it is still instructive to compute its LU factorization. When called with only one output argument, `lu` returns the two triangular factors, `L` and `U`, in a single sparse matrix. The number of nonzeros in that matrix is a measure of the time and storage required to solve linear systems involving `B`. Here are the nonzero counts for the three permutations being considered.

Original	`lu(B)`	1022
Reverse Cuthill-McKee	`lu(B(r,r))`	968
Approximate minimum degree	`lu(B(m,m))`	636

Even though this is a small example, the results are typical. The original numbering scheme leads to the most fill-in. The fill-in for the reverse

15-31

Cuthill-McKee ordering is concentrated within the band, but it is almost as extensive as the first two orderings. For the minimum degree ordering, the relatively large blocks of zeros are preserved during the elimination and the amount of fill-in is significantly less than that generated by the other orderings. The spy plots below reflect the characteristics of each reordering.

Cholesky Factorization

If S is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix R so that R'*R = S.

```
R = chol(S)
```

chol does not automatically pivot for sparsity, but you can compute minimum degree and profile limiting permutations for use with chol(S(p,p)).

Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, chol does a quick calculation of the amount of memory required and allocates all the memory at the start of the factorization. You can use symbfact, which uses the same algorithm as chol, to calculate how much memory is allocated.

QR Factorization

MATLAB computes the complete QR factorization of a sparse matrix S with

```
[Q,R] = qr(S)
```

but this is usually impractical. The orthogonal matrix `Q` often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as "the Q-less QR factorization," is available.

With one sparse input argument and one output argument

```
R = qr(S)
```

returns just the upper triangular portion of the QR factorization. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations,

```
R'*R = S'*S
```

However, the loss of numerical information inherent in the computation of `S'*S` is avoided.

With two input arguments having the same number of rows, and two output arguments, the statement

```
[C,R] = qr(S,B)
```

applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`.

The Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize} \|Ax - b\|$$

with two steps

```
[c,R] = qr(A,b)
x = R\c
```

If `A` is sparse, but not square, MATLAB uses these steps for the linear equation solving backslash operator

```
x = A\b
```

Or, you can do the factorization yourself and examine `R` for rank deficiency.

It is also possible to solve a sequence of least squares linear systems with different right-hand sides, b, that are not necessarily known when `R = qr(A)` is computed. The approach solves the "semi-normal equations"

```
R'*R*x = A'*b
```

with

```
x = R\(R'\(A'*b))
```

and then employs one step of iterative refinement to reduce roundoff error

```
r = b - A*x
e = R\(R'\(A'*r))
x = x + e
```

Incomplete Factorizations

The `luinc` and `cholinc` functions provide approximate, *incomplete* factorizations, which are useful as preconditioners for sparse iterative methods.

The `luinc` function produces two different kinds of incomplete LU factorizations, one involving a drop tolerance and one involving fill-in level. If A is a sparse matrix, and `tol` is a small tolerance, then

```
[L,U] = luinc(A,tol)
```

computes an approximate LU factorization where all elements less than `tol` times the norm of the relevant column are set to zero. Alternatively,

```
[L,U] = luinc(A,'0')
```

computes an approximate LU factorization where the sparsity pattern of L+U is a permutation of the sparsity pattern of A.

For example,

```
load west0479
A = west0479;
nnz(A)
nnz(lu(A))
nnz(luinc(A,1e-6))
nnz(luinc(A,'0'))
```

shows that A has 1887 nonzeros, its complete LU factorization has 16777 nonzeros, its incomplete LU factorization with a drop tolerance of 1e 6 has 10311 nonzeros, and its lu('0') factorization has 1886 nonzeros.

The `luinc` function has a few other options. See the `luinc` reference page for details.

The `cholinc` function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the `cholinc` reference page for more information.

Simultaneous Linear Equations

Systems of simultaneous linear equations can be solved by two different classes of methods:

- Direct methods. These are usually variants of Gaussian elimination and are often expressed as matrix factorizations such as LU or Cholesky factorization. The algorithms involve access to the individual matrix elements.

- Iterative methods. Only an approximate solution is produced after a finite number of steps. The coefficient matrix is involved only indirectly, through a matrix-vector product or as the result of an abstract linear operator.

Direct Methods

Direct methods are usually faster and more generally applicable, if there is enough storage available to carry them out. Iterative methods are usually applicable to restricted cases of equations and depend upon properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of MATLAB and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB M-files and may make use of the direct solution of subproblems or preconditioners.

The usual way to access direct methods in MATLAB is not through the `lu` or `chol` functions, but rather with the matrix division operators `/` and `\`. If A is square, the result of X = A\B is the solution to the linear system A*X = B. If A is not square, then a least squares solution is computed.

If A is a square, full, or sparse matrix, then A\B has the same storage class as B. Its computation involves a choice among several algorithms:

- If A is triangular, perform a triangular solve for each column of B.

- If A is a permutation of a triangular matrix, permute it and perform a sparse triangular solve for each column of B.

- If A is symmetric or Hermitian and has positive real diagonal elements, find a symmetric minimum degree order p = symmmd(A), and attempt to compute

15-35

the Cholesky factorization of A(p,p). If successful, finish with two sparse triangular solves for each column of B.

- Otherwise (if A is not triangular, or is not Hermitian with positive diagonal, or if Cholesky factorization fails), find a column minimum degree order p = colmmd(A). Compute the LU factorization with partial pivoting of A(:,p), and perform two triangular solves for each column of B.

For a square matrix, MATLAB tries these possibilities in order of increasing cost. The tests for triangularity and symmetry are relatively fast and, if successful, allow for faster computation and more efficient memory usage than the general purpose method.

For example, consider the sequence below.

```
[L,U] = lu(A);
y = L\b;
x = U\y;
```

In this case, MATLAB uses triangular solves for both matrix divisions, since L is a permutation of a triangular matrix and U is triangular.

Using a Different Preordering. If A is not triangular or a permutation of a triangular matrix, backslash (\) uses colmmd and symmmd to determine a minimum degree order. Use the function spparms to turn off the minimum degree preordering if you want to use a better preorder for a particular matrix.

If A is sparse and x = A\b can use LU factorization, you can use a column ordering other than colmmd to solve for x, as in the following example.

```
spparms('autommd',0);
q = colamd(A);
x = A (:,q) \ b;
x(q) = x;
spparms('autommd',1);
```

If A can be factorized using Cholesky factorization, then x = A\b can be computed efficiently using

```
spparms('autommd',0);
p = symamd(A);
x = A(p,p) \ b(p);
x (p) = x;
spparms('autommd',1);
```

In the examples above, the `spparms('autommd',0)` statement turns the automatic `colmmd` or `symmmd` ordering off. The `spparms('autommd',1)` statement turns it back on, just in case you use A\b later without specifying an appropriate pre-ordering. `spparms` with no arguments reports the current settings of the sparse parameters.

Iterative Methods

Nine functions are available that implement iterative methods for sparse systems of simultaneous linear systems.

Functions for Iterative Methods for Sparse Systems

Function	Method
bicg	Biconjugate gradient
bicgstab	Biconjugate gradient stabilized
cgs	Conjugate gradient squared
gmres	Generalized minimum residual
lsqr	LSQR implementation of Conjugate Gradients on the Normal Equations
minres	Minimum residual
pcg	Preconditioned conjugate gradient
qmr	Quasiminimal residual
symmlq	Symmetric LQ

These methods are designed to solve $Ax = b$ or $\min\|b - Ax\|$. For the Preconditioned Conjugate Gradient method, `pcg`, A must be a symmetric, positive definite matrix. `minres` and `symmlq` can be used on symmetric indefinite matrices. For `lsqr`, the matrix need not be square. The other five can handle nonsymmetric, square matrices.

All nine methods can make use of preconditioners. The linear system

$$Ax = b$$

is replaced by the equivalent system

$$M^{-1}Ax = M^{-1}b$$

The preconditioner M is chosen to accelerate convergence of the iterative method. In many cases, the preconditioners occur naturally in the mathematical model. A partial differential equation with variable coefficients may be approximated by one with constant coefficients, for example. Incomplete matrix factorizations may be used in the absence of natural preconditioners.

The five-point finite difference approximation to Laplace's equation on a square, two-dimensional domain provides an example. The following statements use the preconditioned conjugate gradient method preconditioner $M = R'*R$, where R is the incomplete Cholesky factor of A.

```
A = delsq(numgrid('S',50));
b = ones(size(A,1),1);
tol = 1.e-3;
maxit = 10;
R = cholinc(A,tol);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,R',R);
```

Only four iterations are required to achieve the prescribed accuracy.

Background information on these iterative methods and incomplete factorizations is available in [2] and [7].

Eigenvalues and Singular Values

Two functions are available which compute a few specified eigenvalues or singular values. svds is based on eigs which uses ARPACK [6].

Functions to Compute a Few Eigenvalues or Singular Values

Function	Description
eigs	Few eigenvalues
svds	Few singular values

These functions are most frequently used with sparse matrices, but they can be used with full matrices or even with linear operators defined by M-files.

The statement

```
[V,lambda] = eigs(A,k,sigma)
```

finds the k eigenvalues and corresponding eigenvectors of the matrix A which are nearest the "shift" sigma. If sigma is omitted, the eigenvalues largest in magnitude are found. If sigma is zero, the eigenvalues smallest in magnitude are found. A second matrix, B, may be included for the generalized eigenvalue problem

$$Av = \lambda Bv$$

The statement

```
[U,S,V] = svds(A,k)
```

finds the k largest singular values of A and

```
[U,S,V] = svds(A,k,0)
```

finds the k smallest singular values.

For example, the statements

```
L = numgrid('L',65);
A = delsq(L);
```

set up the five-point Laplacian difference operator on a 65-by-65 grid in an L-shaped, two-dimensional domain. The statements

```
size(A)
nnz(A)
```

show that A is a matrix of order 2945 with 14,473 nonzero elements.

The statement

```
[v,d] = eigs(A,1,0);
```

computes the smallest eigenvalue and eigenvector. Finally,

```
L(L>0) = full(v(L(L>0)));
x = -1:1/32:1;
contour(x,x,L,15)
axis square
```

distributes the components of the eigenvector over the appropriate grid points and produces a contour plot of the result.

The numerical techniques used in eigs and svds are described in [6].

Selected Bibliography

[1] Amestoy, P. R., T. A. Davis, and I. S. Duff, "An Approximate Minimum Degree Ordering Algorithm," *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 4, Oct. 1996, pp. 886-905.

[2] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] Davis, T.A., Gilbert, J. R., Larimore, S.I., Ng, E., Peyton, B., "A Column Approximate Minimum Degree Ordering Algorithm," *Proc. SIAM Conference on Applied Linear Algebra*, Oct. 1997, p. 29.

[4] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1, January 1992, pp. 333-356.

[5] Larimore, S. I., *An Approximate Minimum Degree Column Ordering Algorithm*, MS Thesis, Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1998, available at `http://www.cise.ufl.edu/tech-reports/`

[6] Lehoucq, R. B., D. C. Sorensen, C. Yang, *ARPACK Users' Guide*, SIAM, Philadelphia, 1998.

[7] Saad, Yousef, *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company, 1996.

Programming and Data Types

MATLAB® is a high-level language that includes data structures, functions, control flow statements, input/output, and object-oriented capabilities. This section presents the MATLAB programming features and techniques in the following chapters:

Related Information

The following sections provide information that is also useful to MATLAB programmers. This documentation is available in MATLAB online help.

- Graphics — describes how to plot vector and matrix data in 2-D and 3-D representations, how to annotate, print, and export plots, and how to use graphics objects and their figure and axes properties.

- Calling C and Fortran Programs from MATLAB — describes how to build C and Fortran subroutines into callable MEX files.

- Calling MATLAB from C and Fortran Programs — discusses how to use the MATLAB engine library to call MATLAB from C and Fortran programs.

- Calling Java from MATLAB — describes how to use the MATLAB interface to Java classes and objects.

- Importing and Exporting Data — describes techniques for importing data to and exporting data from the MATLAB environment.

- COM and DDE Support — describes how to use Component Object Model (COM) and Dynamic Data Exchange (DDE) with MATLAB.

- Serial Port I/O — describes how to communicate with peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port.

16

M-File Programming

This chapter explains the basics of how to write script and function programs in M-files. It covers the following topics:

MATLAB Programming: A Quick Start

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of filename.m. The term you use for filename becomes the new command that MATLAB associates with the program. The file extension of .m makes this a MATLAB M-file.

M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept arguments and produce output. You create M-files using a text editor, then use them as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

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

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

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

c =

    8.2109
```

Kinds of M-Files

There are two kinds of M-files.

Script M-Files	Function M-Files
• Do not accept input arguments or return output arguments	• Can accept input arguments and return output arguments
• Operate on data in the workspace	• Internal variables are local to the function by default
• Useful for automating a series of steps you need to perform many times	• Useful for extending the MATLAB language for your application

What's in an M-File?

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

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

f = prod(1:n);        % Function body
```

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

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

Refer to "Functions" on page 16-8 for more detail on the parts of a MATLAB function.

Providing Help for Your Programs

You can provide user information for the programs you write by including a help text section at the beginning of your M-file. This section starts on the line following the function definition and ends at the first blank line. Each line of the help text must begin with a comment (%) character. MATLAB displays this information whenever you type

```
help m-file_name
```

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

```
help directory_name
```

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

Creating M-Files: Accessing Text Editors

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

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

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

```
edit foo
```

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

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

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

• List the names of the files in your current directory

```
what
```

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

```
type fact
```

• Call the `fact` function

```
fact(5)
ans =
     120
```

Note Save any M-files you create and any MathWorks-supplied M-files that you edit in a directory that is not in the $matlabroot/toolbox directory tree. If you keep your files in $matlabroot/toolbox directories, they may be overwritten when you install a new version of MATLAB. Also note that locations of files in the $matlabroot/toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to $matlabroot/toolbox directories using an external editor, or if you add or remove files from these directories using file system operations, enter the commands `clear functionname` and `rehash toolbox` before you use the files in the current session. For more information, see `rehash` or "Toolbox Path Caching" in the "Development Environment" documentation.

Scripts

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

Simple Script Example

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

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

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

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

Functions

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

This section covers the following topics regarding functions:

- "Simple Function Example"
- "Basic Parts of a Function M-File" on page 16-9
- "How Functions Work" on page 16-12
- "Checking the Number of Function Arguments" on page 16-15
- "Passing Variable Numbers of Arguments" on page 16-16

Simple Function Example

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

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

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

```
z = 1:99;

average(z)
ans =
    50
```

Basic Parts of a Function M-File

A function M-file consists of:

- "The Function Definition Line"
- "The H1 Line" on page 16-11
- "Help Text" on page 16-11
- "The Function Body" on page 16-11
- "Comments" on page 16-12

This section also covers the following topics:

- "How Functions Work" on page 16-12
- "Checking the Number of Function Arguments" on page 16-15
- "Passing Variable Numbers of Arguments" on page 16-16

The Function Definition Line

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

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

- input argument
- function name
- output argument
- keyword

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

The Function Name. MATLAB function names have the same constraints as variable names. The name must begin with a letter, which may be followed by any combination of letters, digits, and underscores. Making all letters in the name lowercase is recommended as it makes your M-files portable between platforms.

Although function 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 function name unique in the first N characters.

```
N = namelengthmax
N =
    63
```

Note Some operating systems may restrict function names to shorter lengths.

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

```
average.m
```

If the filename and the function definition line name are different, the internal (function) name is ignored. Thus, if average.m is the file that defines a function named compute_average, you would invoke the function by typing

```
average
```

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

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

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

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets

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

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

The H1 Line

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

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

This is the first line of text that appears when a user types help *function_name* at the MATLAB prompt. Further, the `lookfor` function searches on and displays only the H1 line. Because this line provides important summary information about the M-file, it is important to make it as descriptive as possible.

Help Text

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

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

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

For example, typing help `sin` results in

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

The Function Body

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

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

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

Comments

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

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

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

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

How Functions Work

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

This section provides the following information on calling MATLAB functions:

- "Function Name Resolution"
- "What Happens When You Call a Function"
- "Creating P-Code Files"
- "How MATLAB Passes Function Arguments"
- "Function Workspaces"

Function Name Resolution

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

1 Checks to see if the name is a variable.

2 Checks to see if the name is a *subfunction*, a MATLAB function that resides in the same M-file as the calling function. Subfunctions are discussed in the section, "Subfunctions" on page 16-19.

3 Checks to see if the name is a *private function*, a MATLAB function that resides in a *private directory*, a directory accessible only to M-files in the directory immediately above it. Private directories are discussed in the section, "Private Functions" on page 16-21.

4 Checks to see if the name is a function on the MATLAB search path. MATLAB uses the first file it encounters with the specified name.

If you duplicate function names, MATLAB executes the one found first using the above rules. It is also possible to overload function names. This uses additional dispatching rules and is discussed in the section, "How MATLAB Determines Which Method to Call" on page 21-66.

What Happens When You Call a Function

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

You can use `clear` in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
`clear function_name`	Remove specified function from workspace
`clear functions`	Remove all compiled M-functions
`clear all`	Remove all variables and functions

Creating P-Code Files

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

```
pcode average
```

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

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

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

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

How MATLAB Passes Function Arguments

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

Function Workspaces

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

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

Checking the Number of Function Arguments

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

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

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

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

```
function [token,remainder] = strtok(string,delimiters)
% Function requires at least one input argument
if nargin < 1
   error('Not enough input arguments.');
end
token = []; remainder = [];
len = length(string);
if len == 0
   return
end

% If one input, use white space delimiter
if (nargin == 1)
   delimiters = [9:13 32]; % White space characters
end
i = 1;
```

```
% Determine where non-delimiter characters begin
while (any(string(i) == delimiters))
   i = i + 1;
   if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))
   i = i + 1;
   if (i > len), break, end
end
finish = i   1;
token = string(start:finish);

% For two output arguments, count characters after
% first delimiter (remainder)
if (nargout == 2)
   remainder = string(finish + 1:end);
end
```

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

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

Passing Variable Numbers of Arguments

The varargin and varargout functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers:

- "Unpacking varargin Contents"
- "Packing varargout Contents"
- "varargin and varargout in Argument Lists"

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data – one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

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

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

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

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

Unpacking varargin Contents

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

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

Cell array indexing has two subscript components:

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

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

Packing varargout Contents

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

```
function [varargout] = testvar2(arrayin)
for k = 1:nargout
    varargout{k} = arrayin(k,:) % Cell array assignment
end
```

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

Here's how to call testvar2.

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

varargin and varargout in Argument Lists

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

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

Subfunctions

Function M-files can contain code for more than one function. The first function in the file is the *primary function*, the function invoked with the M-file name. Additional functions within the file are *subfunctions* that are only visible to the primary function or other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first.

```
function [avg,med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u,n);
med = median(u,n);

function a = mean(v,n)          % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v,n)        % Subfunction
% Calculate median.
w = sort(v);
if rem(n,2) == 1
    m = w((n+1)/2);
else
    m = (w(n/2)+w(n/2+1))/2;
end
```

The subfunctions mean and median calculate the average and median of the input list. The primary function newstats determines the length of the list and calls the subfunctions, passing to them the list length n. Functions within the same M-file cannot access the same variables unless you declare them as global within the pertinent functions, or pass them as arguments. In addition, the help facility can only access the primary function in an M-file.

When you call a function from within an M-file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard M-file on your search path. Because it checks for a subfunction first, you can

supersede existing M-files using subfunctions with the same name, for example, `mean` in the above code. Function names must be unique within an M-file, however.

Private Functions

Private functions are functions that reside in subdirectories with the special name `private.` They are visible only to functions in the parent directory. For example, assume the directory `newmath` is on the MATLAB search path. A subdirectory of `newmath` called `private` can contain functions that only the functions in `newmath` can call. Because private functions are invisible outside of the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a nonprivate M-file named `test.m.`

You can create your own private directories simply by creating subdirectories called `private` using the standard procedures for creating directories or folders on your computer. Do not place these private directories on your path.

Variables

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

- You do not need to type or declare variables used in M-files, (with the possible exception of designating them as global or persistent).
- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

Naming Variables

MATLAB variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so A and a are not the same variable.

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.

```
N = namelengthmax
N =
    63
```

Verifying a Variable Name

You can use the isvarname function to make sure a name is valid before you use it. isvarname returns 1 if the name is valid, and 0 otherwise.

```
isvarname 8th_column
ans =
    0                    % Not valid - begins with a number
```

Avoid Using 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, or invoke the function using `builtin`.

For example, if you enter the following command, you will not be able to use the MATLAB `disp` function until you clear the variable with `clear disp`.

```
disp = 50;
```

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

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

Local Variables

Each MATLAB function has its own local variables. These are separate from those of other functions, and from those of the base workspace. Variables defined in a function do not remain in memory from one function call to the next, unless they are defined as `global` or `persistent`.

Scripts, on the other hand, do not have a separate workspace. They 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.

Note If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

Global Variables

If several functions, and possibly the base workspace, all declare a particular name as `global`, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

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

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

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

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

Then interactively enter the statements

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

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

Creating Global Variables

Each function that uses a global variable must first declare the variable as `global`. It is usually best to put global declarations toward the beginning of the function. You would declare global variable `MAXLEN` as follows:

```
global MAXLEN
```

If the M-file contains subfunctions as well, then each subfunction requiring access to the global variable must declare it as `global`. To access the variable from the MATLAB command line, you must declare it as `global` at the command line.

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

Displaying Global Variables

To see only those variables you have declared as global, use the `who` or `whos` functions with the literal, `global`:

```
global MAXLEN MAXWID
MAXLEN = 36;  MAXWID = 78;
len = 5;  wid = 21;

whos global
  Name          Size                    Bytes  Class

  MAXLEN        1x1                         8  double array (global)
  MAXWID        1x1                         8  double array (global)

Grand total is 2 elements using 16 bytes
```

Suggestions for Using Global Variables

There is a certain amount of risk associated with using global variables and, because of this, it is recommended that you use them sparingly. You might, for example, unknowingly give a global variable in one function a name that is already used for a global variable in another function. When you run your application, one function may unintentionally overwrite the variable used by the other. This can be a difficult error to track down.

Another problem comes when you want to change the variable name. To make a change without introducing an error into the application, you must find every occurrence of that name in your code (and other people's code, if you share functions).

Alternatives to Using Global Variables. Instead of using a global variable, you may be able to

- Pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

  If this means that you have to pass a number of additional variables, you can put them into a structure or cell array and just pass it as one additional argument.

- Use a persistent variable (described in the next section), if you only need to make the variable persist in memory from one function call to the next.

Persistent Variables

Characteristics of persistent variables are

- You can only use them in functions.
- Other functions are not allowed access to them.
- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.
- If you `clear` the function or edit the M-file for that function, then MATLAB clears all persistent variables used in that function.

You must declare persistent variables as `persistent` before you can use them in a function. It is usually best to put persistent declarations toward the beginning of the function. You wouldl declare persistent variable `SUM_X` as follows:

```
persistent SUM_X
```

You can use the `mlock` function to keep an M-file from being cleared from memory, thus keeping persistent variables in the M-file from being cleared as well.

Special Values

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

Function	Return Value
ans	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in ans.
eps	Floating-point relative accuracy. This is the tolerance MATLAB uses in its calculations.
realmax	Largest floating-point number your computer can represent.
realmin	Smallest floating-point number your computer can represent.
pi	3.1415926535897...
i, j	Imaginary unit.
inf	Infinity. Calculations like n/0, where n is any nonzero real value, result in inf.
NaN	Not-a-Number, an invalid numeric value. Expressions like 0/0 and inf/inf result in a NaN, as do arithmetic operations involving a NaN. n/0, where n is complex, also returns NaN.
computer	Computer type.
version	MATLAB version string.

Here are several examples that use these values in MATLAB expressions.

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

Data Types

There are 15 fundamental data types (or classes) in MATLAB. Each of these data types is in the form of an array. This array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size. Two-dimensional versions of these arrays are called *matrices*.

All of the fundamental data types are shown in lowercase text in the diagram below. Additional data types are user-defined, object-oriented *user classes* (a subclass of structure), and *java classes*, that you can use with the MATLAB interface to Java.

Matrices of type double and logical may be either full or sparse. For matrices having a small number of nonzero elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke special methods especially tailored to solve sparse problems.

```
                              ARRAY
                        [full or sparse]
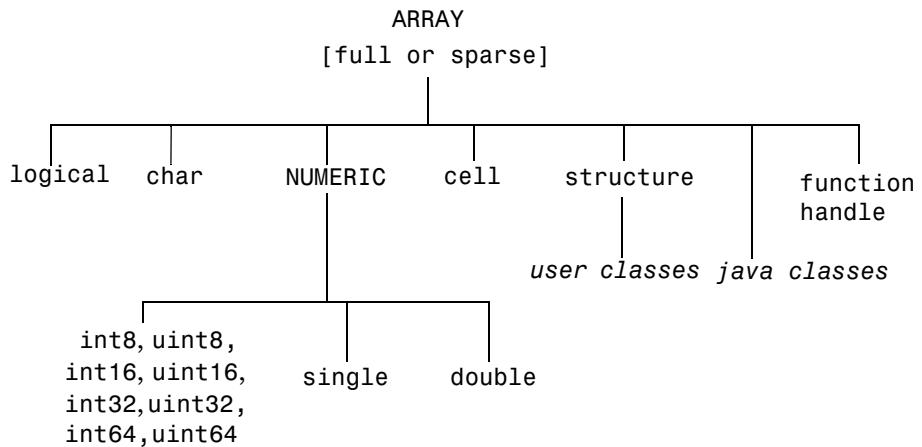

logical    char      NUMERIC     cell      structure
                                                              function
                                                               handle

                                              user classes  java classes


    int8, uint8,
    int16, uint16,
    int32, uint32,      single      double
    int64, uint64
```

The logical data type represents a logical true or false value using the numbers 1 and 0, respectively. MATLAB returns logical values from its relational (e.g., >, ~=) and logical (e.g., &&, xor) operations and functions.

The char data type holds characters. A character string is merely a 1-by-n array of characters. You can use char to hold an m-by-n array of strings as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To hold an array of strings of unequal length, use a cell array.

Numeric data types include signed and unsigned integers, and single- and double- precision floating point numbers. The following hold true for numeric data types in MATLAB:

- All data types support basic array operations, such as subscripting and reshaping.
- All MATLAB computations are done in double-precision.
- To perform mathematical operations on integer or single precision arrays, you must convert them to double precision using the `double` function.
- Integer and single precision arrays offer more memory efficient storage than double-precision.

A `cell array` provides a storage mechanism for dissimilar kinds of data. You can store arrays of different types and/or sizes within the cells of a `cell` array. For example, you can store a 1-by-50 `char` array, a 7-by-13 `double` array, and a 1-by-1 `uint32` in cells of the same `cell` array. You access data in a `cell` array using the same matrix indexing used on other MATLAB matrices and arrays.

The MATLAB `structure` data type is similar to the `cell` array in that it also stores dissimilar kinds of data. But, in this case, it stores the data in named fields rather than in cells. This enables you to attach a name to the groups of data stored within the structure. You access data in a structure using these same field names.

A `function handle` holds information to be used in referencing a function. When you create a function handle, MATLAB captures all the information about the function that it needs to locate and execute, or *evaluate*, it later on. Typically, a function handle is passed in an argument list to other functions. It is then used in conjunction with `feval` to evaluate the function to which the handle belongs.

MATLAB data types are implemented as classes. You can also create MATLAB classes of your own. These user-defined classes inherit from the MATLAB `structure` class and are shown in the previous diagram as a subset of `structure`.

MATLAB provides an interface to the Java programming language that enables you to create objects from Java classes and call Java methods on these objects. A Java class is a MATLAB data type. There are built-in and third-party classes that are already available through the MATLAB interface. You can also create your own Java class definitions and bring them into MATLAB.

The following table describes the data types in more detail.

Data Type	Example	Description
logical	magic(4) > 10	Logical array. Must contain only logical 1 (true) and logical 0 (false) elements. (Any nonzero values converted to logical become logical 1.) Logical matrices (2-D only) may be sparse.
char	'Hello'	Character array (each character is 16 bits long). This array is also referred to as a string.
int8, uint8, int16, uint16, int32, uint32, int64, uint64	uint8(magic(3))	Signed and unsigned integer arrays that are 8, 16, 32, and 64 bits in length. Enables you to manipulate integer quantities in a memory efficient manner. These data types cannot be used in mathematical operations.
single	3*10^38	Single-precision numeric array. Single precision requires less storage than double precision, but has less precision and a smaller range. This data type cannot be used in mathematical operations.
double	3*10^300 5+6i	Double-precision numeric array. This is the most common MATLAB variable type. Double matrices (2-D only) may be sparse.
cell	{17 'hello' eye(2)}	Cell array. Elements of cell arrays contain other arrays. Cell arrays collect related data and information of a dissimilar size together.
structure	a.day = 12; a.color = 'Red'; a.mat = magic(3);	Structure array. Structure arrays have field names. The fields contain other arrays. Like cell arrays, structures collect related data and information together.
function handle	@humps	Handle to a MATLAB function. A function handle can be passed in an argument list and evaluated using feval.

Data Type	Example	Description
user class	inline('sin(x)')	MATLAB class. This user-defined class is created using MATLAB functions.
java class	java.awt.Frame	Java class. You can use classes already defined in the Java API or by a third party, or create your own classes in the Java language.

Operators

The MATLAB operators fall into three categories:

- Arithmetic operators that perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power.
- Relational operators that compare operands quantitatively, using operators like "less than" and "not equal to."
- Logical operators that use the logical operators AND, OR, and NOT.

This section also discusses operator precedence.

Arithmetic Operators

MATLAB provides these arithmetic operators.

Operator	Description
+	Addition
	Subtraction
.*	Multiplication
./	Right division
.\	Left division
+	Unary plus
	Unary minus
:	Colon operator
.^	Power
.'	Transpose
'	Complex conjugate transpose
*	Matrix multiplication
/	Matrix right division

Operator	Description
\	Matrix left division
^	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand – this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

3 * A
ans =
    24     3    18
     9    15    21
    12    27     6
```

Relational Operators

MATLAB provides these relational operators.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than

Operator	Description
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Relational Operators and Arrays

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of A is equal to the corresponding element of B.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4  1 7];

A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive the value 1. Locations where the relation is false receive the value 0.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []
```

return an error if A is not 0-by-0 or 1-by-1. This behavior is consistent with that of all other binary operators, such as +, -, >, <, &, |, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operator and functions.

- Element-wise — operate on corresponding elements of logical arrays.
- Bit-wise — operate on corresponding bits of integer values or arrays.
- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform element-wise logical operations on their inputs to produce a like-sized output array. The examples shown in the following table use vector inputs A and B, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
&	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	A & B = 01001
|	Returns 1 for every element location that is true (nonzero) in either one or the other, or both, arrays and 0 for all other elements.	A | B = 11101
~	Complements each element of input array, A.	~A = 10010
xor	Returns 1 for every element location that is true (nonzero) in only one array, and 0 for all other elements.	xor(A,B)=10100

For operators and functions that take two array operands, (&, |, and xor), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not.

In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical 1, or `true`.

Operator Overloading. You can overload the &, |, and ~ operators to make their behavior dependent upon the data type on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
A & B	and(A,B)
A | B	or(A,B)
~A	not(A)

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an element-wise fashion, are `any` and `all`. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, `any` and `all` operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional, `dimension` input to operate on a specific dimension of the array.

The examples shown in the following table use array input A, where

```
A = [0   1   2;
     0  -3   8;
     0   5   0];
```

Function	Description	Example
any(A)	Returns 1 for a vector where *any* element of the vector is true (nonzero), and 0 if no elements are true.	any(A) ans = 0 1 1
all(A)	Returns 1 for a vector where *all* elements of the vector are true (nonzero), and 0 if all elements are not true.	all(A) ans = 0 1 0

Note The all and any functions ignore any NaN values in the input arrays.

Logical Expressions Using the find Function. The find function determines the indices of array elements that meet a given logical condition. The function is useful for creating masks and index matrices. In its most general form, find returns a single vector of indices. This vector can be used to index into arrays of any size or shape.

For example,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

i = find(A > 8);
A(i) = 100
A =
   100     2     3   100
     5   100   100     8
   100     7     6   100
     4   100   100     1
```

Note An alternative to using `find` in this context is to index into the matrix using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4
```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs A and B, where

```
A = 28;                  % binary 11100
B = 21;                  % binary 10101
```

Function	Description	Example
bitand	Returns the bit-wise AND of two nonnegative integer arguments.	bitand(A,B) = 20 (binary 10100)
bitor	Returns the bit-wise OR of two nonnegative integer arguments.	bitor(A,B) = 29 (binary 11101)
bitcmp	Returns the bit-wise complement as an n-bit number, where n is the second input argument to bitcmp.	bitcmp(A,5) = 3 (binary 00011)
bitxor	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	bitxor(A,B) = 9 (binary 01001)

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they only evaluate their second operand when the result is not fully determined by the first operand.

Operator	Description
&&	Returns true (1) if both inputs evaluate to true, and false (0) if they do not.
||	Returns true (1) if either input, or both, evaluate to true, and false (0) if they do not.

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

If A equals zero, then the entire expression will evaluate to false, regardless of the value of B. Under these circumstances, there is no need to evaluate B

because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is `true`. Again, regardless of the value of B, the statement will evaluate to `true`. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisifed. For example, you only want to execute an M-file function if the M-file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, myfun.m, cannot be found:

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

Similarly, this statement avoids divide-by-zero errors when b equals zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the && and || operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1 Parentheses ()

2 Transpose (.'), power (.^), complex conjugate transpose ('), matrix power (^)

3 Unary plus (+), unary minus (-), logical negation (~)

4 Multiplication (.*), right division (./), left division(.\), matrix multiplication (*), matrix right division (/), matrix left division (\)

5 Addition (+), subtraction (-)

6 Colon operator (:)

7 Less than (<), less than or equal to (<=), greater than (>), greater than or
equal to (>=), equal to (==), not equal to (~=)

8 Element-wise AND (&)

9 Element-wise OR (|)

10 Short-circuit AND (&&)

11 Short-circuit OR (||)

Precedence of AND and OR Operators

MATLAB always gives the & operator precedence over the | operator. Although
MATLAB typically evaluates expressions from left to right, the expression
a|b&c is evaluated as a|(b&c). It is a good idea to use parentheses to explicitly
specify the intended precedence of statements containing combinations of &
and |.

The same precedence rule holds true for the && and || operators.

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this
example:

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2
C =
    0.7500    9.0000    0.2000

C = (A./B).^2
C =
    2.2500   81.0000    1.0000
```

Keywords

MATLAB reserves certain words for its own use as keywords of the language.
To list the keywords, type

```
iskeyword

ans =

    'break'
    'case'
    'catch'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```

See the online function reference pages for help to learn how to use any of these
keywords.

You should not use MATLAB keywords other than for their intended purpose.
For example, a keyword should not be used as follows:

```
while = 5;
??? while = 5;
          |
Error: Expected a variable, function, or constant, found "=".
```

Flow Control

There are eight flow control statements in MATLAB:

- `if`, together with `else` and `elseif`, executes a group of statements based on some logical condition.
- `switch`, together with `case` and `otherwise`, executes different groups of statements depending on the value of some logical condition.
- `while` executes a group of statements an indefinite number of times, based on some logical condition.
- `for` executes a group of statements a fixed number of times.
- `continue` passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.
- `break` terminates execution of a `for` or `while` loop.
- `try...catch` changes flow control if an error is detected during execution.
- `return` causes execution to return to the invoking function.

All flow constructs use `end` to indicate the end of the flow control block.

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Techniques for Improving Performance" in the MATLAB "Programming and Data Types" documentation.

---

if, else, and elseif

`if` evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is true (`1`), MATLAB executes all the statements between the `if` and `end` lines. It resumes execution at the line following the `end` statement. If the condition is false (`0`), MATLAB skips all the statements

between the if and end lines, and resumes execution at the line following the
end statement.

For example,

```
if rem(a,2) == 0
    disp('a is even')
    b = a/2;
end
```

You can nest any number of if statements.

If the logical expression evaluates to a nonscalar value, all the elements of the
argument must be nonzero. For example, assume X is a matrix. Then the
statement

```
if X
    statements
end
```

is equivalent to

```
if all(X(:))
    statements
end
```

The else and elseif statements further conditionalize the if statement:

- The else statement has no logical condition. The statements associated with
  it execute if the preceding if (and possibly elseif condition) is false (0).
- The elseif statement has a logical condition that it evaluates if the
  preceding if (and possibly elseif condition) is false (0). The statements
  associated with it execute if its logical condition is true (1). You can have
  multiple elseifs within an if block.

```
if n < 0              % If n negative, display error message.
    disp('Input must be positive');
elseif rem(n,2) == 0 % If n positive and even, divide by 2.
    A = n/2;
else
    A = (n+1)/2;     % If n positive and odd, increment and divide.
end
```

if Statements and Empty Arrays

An if condition that reduces to an empty array represents a false condition. That is,

```
if A
    S1
else
    S0
end
```

will execute statement S0 when A is an empty array.

switch

switch executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)
    case value1
        statements % Executes if expression is value1
    case value2
        statements % Executes if expression is value2
    .
    .
    .
    otherwise
        statements % Executes if expression does not
                   % does not match any case
end
```

This block consists of:

- The word switch followed by an expression to evaluate.
- Any number of case groups. These groups consist of the word case followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another switch block. Execution of a case group ends when MATLAB encounters the next case statement or the otherwise statement. Only the first matching case is executed.

16-45

- An optional otherwise group. This consists of the word otherwise, followed by the statements to execute if the expression's value is not handled by any of the preceding case groups. Execution of the otherwise group ends at the end statement.
- An end statement.

switch works by comparing the input expression to each case value. For numeric expressions, a case statement is true if (value==expression). For string expressions, a case statement is true if strcmp(value,expression).

The code below shows a simple example of the switch statement. It checks the variable input_num for certain values. If input_num is 1, 0, or 1, the case statements display the value on screen as text. If input_num is none of these values, execution drops to the otherwise statement and the code displays the text 'other value'.

```
switch input_num
    case  1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
         disp('other value');
end
```

Note For C Programmers, unlike the C language switch construct, the MATLAB switch does not "fall through." That is, if the first case statement is true, other case statements do not execute. Therefore, break statements are not used.

switch can handle multiple conditions in a single case statement by enclosing the case expression in a cell array.

```
switch var
    case 1
        disp('1')
    case {2,3,4}
        disp('2 or 3 or 4')
    case 5
        disp('5')
    otherwise
        disp('something else')
end
```

while

The while loop executes a statement or group of statements repeatedly as long as the controlling expression is true (1). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be 1 for execution to continue. To reduce a matrix to a scalar value, use the all and any functions.

For example, this while loop finds the first integer n for which n! (n factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a while loop at any time using the break statement.

while Statements and Empty Arrays

A while condition that reduces to an empty array represents a false condition. That is,

```
while A, S1, end
```

never executes statement S1 when A is an empty array.

for

The for loop executes a statement or group of statements a predetermined number of times. Its syntax is:

```
for index = start:increment:end
    statements
end
```

The default increment is 1. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the *end* value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for i = 2:6
    x(i) = 2*x(i 1);
end
```

You can nest multiple for loops.

```
for i = 1:m
    for j = 1:n
        A(i,j) = 1/(i + j   1);
    end
end
```

Note You can often speed up the execution of MATLAB code by replacing for and while loops with vectorized code. See the section "Vectorizing Loops" for details.

Using Arrays as Indices

The index of a for loop can be an array. For example, consider an m-by-n array A. The statement

```
for i = A
    statements
end
```

sets i equal to the vector A(:,k). For the first loop iteration, k is equal to 1; for the second k is equal to 2, and so on until k equals n. That is, the loop iterates for a number of times equal to the number of columns in A. For each iteration, i is a vector containing one of the columns of A.

continue

The continue statement passes control to the next iteration of the for or while loop in which it appears, skipping any remaining statements in the body of the loop. In nested loops, continue passes control to the next iteration of the for or while loop enclosing it.

The example below shows a continue loop that counts the lines of code in the file, magic.m, skipping all blank lines and comments. A continue statement is used to advance to the next line in magic.m without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

break

The break statement terminates the execution of a for loop or while loop. When a break statement is encountered, execution continues with the next statement outside of the loop. In nested loops, break exits from the innermost loop only.

The example below shows a while loop that reads the contents of the file fft.m into a MATLAB character array. A break statement is used to exit the while loop when the first empty line is encountered. The resulting character array contains the M-file help for the fft program.

```
fid = fopen('fft.m','r');
s = '';
while ~feof(fid)
   line = fgetl(fid);
   if isempty(line)
      break
   end
   s = strvcat(s,line);
end
disp(s)
```

try ... catch

The general form of a `try` ... `catch` statement sequence is

```
try,
   statement,
   ...,
   statement,
catch,
   statement,
   ...,
   statement,
end
```

In this sequence the statements between `try` and `catch` are executed until an error occurs. The statements between `catch` and `end` are then executed. Use `lasterr` to see the cause of the error. If an error occurs between `catch` and `end`, MATLAB terminates execution unless another `try` ... `catch` sequence has been established.

return

`return` terminates the current sequence of commands and returns control to the invoking function or to the keyboard. `return` is also used to terminate `keyboard` mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. `return` may be inserted within the called function to force an early termination and to transfer control to the invoking function.

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The eval function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the eval syntax is

```
eval('string')
```

For example, this code uses eval on an expression to generate a Hilbert matrix of order n.

```
t = '1/(i+j 1)';
for i = 1:n
    for j = 1:n
        a(i,j) = eval(t);
    end
end
```

Here's an example that uses eval on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation

You can concatenate strings to create a complete expression for input to eval. This code shows how eval can create 10 variables named P1, P2, ...P10, and set each of them to a different value.

```
for i=1:10
    eval(['P',int2str(i),'= i.^2'])
end
```

feval

The feval function differs from eval in that it executes a function rather than a MATLAB expression. The function to be executed is specified in the first argument by either a function handle or a string containing the function name.

You can use `feval` and the `input` function to choose one of several tasks defined by M-files. This example uses function handles for the `sin`, `cos`, and `log` functions.

```
fun = [@sin; @cos; @log];
k = input('Choose function number: ');
x = input('Enter value: ');
feval(fun(k),x)
```

Note Use `feval` rather than `eval` whenever possible. M-files that use `feval` execute faster and can be compiled with the MATLAB Compiler.

Dates and Times

MATLAB provides functions for time and date handling. These functions are in a directory called timefun in the MATLAB Toolbox.

Category	Function	Description
Current date and time	clock	Current date and time as date vector
	date	Current date as date string
	now	Current date and time as serial date number
Conversion	datenum	Convert to serial date number
	datestr	Convert to string representation of date
	datevec	Date components
Utility	calendar	Calendar
	datetick	Date formatted tick labels
	eomday	End of month
	weekday	Day of the week
Timing	cputime	CPU time in seconds
	etime	Elapsed time
	tic, toc	Stopwatch timer

Date Formats

This section covers the following topics:

- "Types of Date Formats"
- "Conversions Between Date Formats"
- "Date String Formats"
- "Output Formats"

Types of Date Formats

MATLAB works with three different date formats: date strings, serial date numbers, and date vectors.

When dealing with dates you typically work with date strings (16-Sep-1996). MATLAB works internally with *serial date numbers* (729284). A serial date represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '16-Sep-1996, 6:00 pm' in MATLAB is date number 729284.75.

All functions that require dates accept either date strings or serial date numbers. If you are dealing with a few dates at the MATLAB command-line level, date strings are more convenient. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you will get better performance if you use date numbers.

Date vectors are an internal format for some MATLAB functions; you do not typically use them in calculations. A date vector contains the elements [year month day hour minute second].

MATLAB provides functions that convert date strings to serial date numbers, and vice versa. Dates can also be converted to date vectors.

Here are examples of the three date formats used by MATLAB.

Date Format	Example
Date string	02-Oct-1996
Serial date number	729300
Date vector	1996 10 2 0 0 0

Conversions Between Date Formats

Functions that convert between date formats are shown below.

Function	Description
datenum	Convert date string to serial date number
datestr	Convert serial date number to date string
datevec	Split date number or date string into individual date elements

Here are some examples of conversions from one date format to another.

```
d1 = datenum('02-Oct-1996')
d1 =
    729300

d2 = datestr(d1+10)
d2 =
   12-Oct-1996

dv1 = datevec(d1)
dv1 =
    1996       10        2        0        0        0

dv2 = datevec(d2)
dv2 =
    1996       10       12        0        0        0
```

Date String Formats

The `datenum` function is important for doing date calculations efficiently. `datenum` takes an input string in any of several formats, with `'dd-mmm-yyyy'`, `'mm/dd/yyyy'`, or `'dd-mmm-yyyy, hh:mm:ss.ss'` most common. You can form up to six fields from letters and digits separated by any other characters:

- The day field is an integer from 1 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic string with at least three characters.
- The year field is a nonnegative integer: if only two digits are specified, then a year 19yy is assumed; if the year is omitted, then the current year is used as a default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by 'AM' or 'PM'.

For example, if the current year is 1996, then these are all equivalent

```
'17-May-1996'
'17-May-96'
'17-May'
'May 17, 1996'
'5/17/96'
'5/17'
```

and both of these represent the same time

```
'17-May-1996, 18:30'
'5/17/96/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus 3/6 is March 6, not June 3.

If you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros.

Output Formats

The function `datestr(D,dateform)` converts a serial date D to one of 19 different date string output formats showing date, time, or both. The default output for dates is a day-month-year string: 01-Mar-1996. You select an alternative output format by using the optional integer argument `dateform`.

This table shows the date string formats that corespond to each `dateform` value.

dateform	Format	Description
0	01-Mar-1996 15:45:17	day-month-year hour:minute:second
1	01-Mar-1996	day-month-year
2	03/01/96	month/day/year
3	Mar	month, three letters
4	M	month, single letter
5	3	month
6	03/01	month/day
7	1	day of month
8	Wed	day of week, three letters
9	W	day of week, single letter
10	1996	year, four digits
11	96	year, two digits
12	Mar96	month year
13	15:45:17	hour:minute:second
14	03:45:17 PM	hour:minute:second AM or PM
15	15:45	hour:minute
16	03:45 PM	hour:minute AM or PM
17	Q1-96	calendar quarter-year
18	Q1	calendar quarter

Here are some examples of converting the date March 1, 1996 to various forms using the datestr function.

```
d = '01-Mar-1999'
d =
    01-Mar-1999

datestr(d)
ans =
    01-Mar-1999

datestr(d, 2)
ans =
    03/01/99

datestr(d, 17)
ans =
    Q1-99
```

Current Date and Time

The function date returns a string for today's date.

```
date
ans =
    02-Oct-1996
```

The function now returns the serial date number for the current date and time.

```
now
ans =
    729300.71

datestr(now)
ans =
    02-Oct-1996 16:56:16

datestr(floor(now))
ans =
    02-Oct-1996
```

Calling Functions

When you call M-file or built-in functions, you often have the choice of using one of two formats for the function call. (The same applies to entering commands at the MATLAB command line.)

You can code your function calls in either a function or command syntax. This is referred to in MATLAB as *command/function duality*.

Function Syntax

Function calls written in the function syntax look essentially the same as those in many other programming languages. One difference is that, in MATLAB, functions can return more than one output value.

A function call with a single return value looks like this:

```
out = function_name(in1, in2, ..., inN)
```

If the function returns more than one value, separate the output variables with commas or spaces, and enclose them all in square brackets ([ ]).

```
[out1, out2, ..., outN] = function_name(in1, in2, ..., inN)
```

Here are two examples:

```
copyfile(srcfile, '..\mytests', 'writable')
[x1, x2, x3, x4] = deal(A{:})
```

In the function syntax, MATLAB passes arguments to the function by value. See the examples below, under "Passing Arguments" on page 16-60.

Command Syntax

A function call made in command syntax consists of the function name followed by one or more arguments separated by spaces.

```
function_name in1 in2 ... inN
```

While the command format is simpler to write, it has the restriction that you may not assign any return values the function might generate. Attempting to do so generates an error.

Two examples of command syntax are

```
save mydata.mat x y z
clear length width depth
```

In the command syntax, MATLAB treats all arguments as string literals. See the examples in the following section.

Passing Arguments

Function calls written in function syntax pass their arguments by value. That is, for those arguments expressed as variables, it is the value of the variable that gets passed to the function.

Function calls written in command syntax pass all arguments as string literals. If you pass a numeric value, such as 75 in the example below, MATLAB passes it as the string, '75'.

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

The following examples show the difference between passing arguments in the two syntaxes.

Example 1. Calling disp with the function syntax, disp(A), passes the value of variable A to the disp function.

```
A = pi;

disp(A)                     % Function syntax
    3.1416
```

Calling it with the command syntax, disp A, passes a string with the variable name, 'A'.

```
A = pi;

disp A                      % Command syntax
    A
```

Example 2. Passing two variables representing equal strings to the strcmp
function using function and command syntaxes gives different results. The
function syntax passes the values of the arguments. strcmp returns a 1, which
means they are equal.

```
str1 = 'one';        str2 = 'one';

strcmp(str1, str2)                    % Function format
ans =
   1        (equal)
```

The command syntax passes the names of the variables, 'str1' and 'str2',
which are unequal.

```
str1 = 'one';        str2 = 'one';

strcmp str1 str2                      % Command format
ans =
   0        (unequal)
```

Passing String Arguments

When using the function syntax to pass a string literal to a function, you must
enclose the string in single quotes, ('string').

For example, to create a new directory called myapptests, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in
quotes.

```
dirname = 'myapptests';
mkdir(dirname)
```

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 also can group arguments into cell arrays. The advantage over structures 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. The disadvantage is that you don't have fieldnames to describe each variable.

Obtaining User Input

There are three ways to obtain input from a user during M-file execution. You can:

- Display a prompt and obtain keyboard input.
- Pause until the user presses a key.
- Build a complete graphical user interface.

This section covers the first two topics. The third topic is discussed in online documentation under "Creating Graphical User Interfaces".

Prompting for Keyboard Input

The input function displays a prompt and waits for a user response. Its syntax is

```
n = input('prompt_string')
```

The function displays the *prompt_string*, waits for keyboard input, and then returns the value from the keyboard. If the user inputs an expression, the function evaluates it and returns its value. This function is useful for implementing menu-driven applications.

input can also return user input as a string, rather than a numeric value. To obtain string input, append 's' to the function's argument list.

```
name = input('Enter address: ','s');
```

Pausing During Execution

Some M-files benefit from pauses between execution steps. For example, the petals.m script, shown in the "Simple Script Example" section, pauses between the plots it creates, allowing the user to display a plot for as long as desired and then press a key to move to the next plot.

The pause command, with no arguments, stops execution until the user presses a key. To pause for n seconds, use

```
pause(n)
```

Subscripting and Indexing

Subscripting

This section explains how to use subscripting to access and assign to elements of a MATLAB matrix. It covers the following:

- "Accessing Single Elements of a Matrix" on page 16-64
- "Accessing Multiple Elements of a Matrix" on page 16-65
- "Expanding the Size of a Matrix" on page 16-66
- "Deleting Rows and Columns" on page 16-67
- "Concatenating Matrices" on page 16-67

Accessing Single Elements of a Matrix

The element in row i and column j of A is denoted by A(i,j). For example, suppose A = magic(4), Then A(4,2) is the number in the fourth row and second column. For our magic square, A(4,2) is 14.

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

A(4,2)
ans =
    14
```

It is also possible to refer to the elements of a matrix with a single subscript, A(k). This is the usual way of referencing row and column vectors. But it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix.

So, for our magic square, `A(8)` is another way of referring to the value 14 stored in `A(4,2)`.

```
A(8)
ans =
    14
```

Accessing Multiple Elements of a Matrix

It is possible to compute the sum of the elements in the fourth column of `A` by typing

```
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

You can reduce the size of this expression using the colon operator. Subscript expressions involving colons refer to portions of a matrix.

```
A(1:m,n)
```

refers to the elements in rows `1` through `m` of column `n` of the `A` matrix. Using this notation, you can compute the sum of the fourth column of `A` more succinctly.

```
sum(A(1:4,4))
```

The colon by itself refers to *all* the elements in a row or column of a matrix. The keyword end refers to the *last* row or column. Using the following syntax, you can compute this same column sum without having to specify row and column numbers.

```
sum(A(:,end))
ans =
    34
```

By adding an additional colon operator, you can refer to nonconsecutive elements in a matrix. The `m:3:n` in this expression means to make the assignment to every third element in the matrix.

```
B = A;

B(1:3:end) = -10
```

```
B =
   -10     2      3    -10
     5    11    -10      8
     9   -10      6     12
   -10    14     15    -10
```

You can repeatedly access an array element using the `ones` function. To create a new 2-by-6 matrix out of the the 9th element of `A`,

```
B = A(9*ones(2,6))
B =
    3     3     3     3     3     3
    3     3     3     3     3     3
```

Expanding the Size of a Matrix

If you try to access an element outside of the matrix, it is an error

```
B = A(4,5)
Index exceeds matrix dimensions
```

However, if you store a value in an element outside of the matrix, the size of the matrix increases to accommodate the new element.

```
B = A;

B(4,5) = 17
B =
   16     2     3    13     0
    5    11    10     8     0
    9     7     6    12     0
    4    14    15     1    17
```

Similarly, you can expand a matrix by assigning to a series of matrix elements.

```
B(2:5,5:6) = 5
B =
   16     2     3    13     0     0
    5    11    10     8     5     5
    9     7     6    12     5     5
    4    14    15     1     5     5
    0     0     0     0     5     5
```

Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of X, use

```
X(:,2) = []
```

This changes X to

```
X =
    16     3    13
     5    10     8
     9     6    12
     4    15     1
```

If you delete a single element from a matrix, the result isn't a matrix anymore. So expressions like

```
X(1,2) = []
```

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

```
X(2:2:10) = []
```

results in

```
X =
    16     9     3     6    13    12     1
```

Concatenating Matrices

Concatenation is the process of joining small matrices together to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [ ], is the concatenation operator. For an example, start with the 4-by-4 magic square, A, and form

```
B = [A A+32;A+48 A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =
    16     2     3    13    48    34    35    45
     5    11    10     8    37    43    42    40
     9     7     6    12    41    39    38    44
     4    14    15     1    36    46    47    33
    64    50    51    61    32    18    19    29
    53    59    58    56    21    27    26    24
    57    55    54    60    25    23    22    28
    52    62    63    49    20    30    31    17
```

This matrix is half way to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square.

```
sum(B)
ans =
   260   260   260   260   260   260   260   260
```

But, its row sums, sum(B')', are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

Advanced Indexing

MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended end to end.

For example, MATLAB stores

```
A = [2 6 9; 4 2 8; 3 0 1]
```

as

```
    2
    4
    3
    6
    2
    0
    9
    8
    1
```

Accessing A with a single subscript indexes directly into the storage column. A(3) accesses the third value in the column, the number 3. A(7) accesses the seventh value, 9, and so on.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like A has size [d1 d2], where d1 is the number of rows in the array and d2 is the number of columns. If you supply two subscripts (i,j) representing row-column indices, the offset is

```
(j 1)*d1+i
```

Given the expression A(3,2), MATLAB calculates the offset into A's storage column as (2-1)*3+3, or 6. Counting down six elements in the column accesses the value 0.

Indexing Into Multidimensional Arrays

This storage and indexing scheme also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise.

For example, consider a 5-by-4-by-3-by-2 array C.

MATLAB displays C as   MATLAB stores C as

page(1,1) =

```
1   4   3   5
2   1   7   9
5   6   3   2
0   1   5   9
3   2   7   5
```

page(2,1) =

```
6   2   4   2
7   1   4   9
0   0   1   5
9   4   4   2
1   8   2   5
```

page(3,1) =

```
2   2   8   3
2   5   1   8
5   1   5   2
0   9   0   9
9   4   5   3
```

page(1,2) =

```
9   8   2   3
0   0   3   3
6   4   9   6
1   9   2   3
0   2   8   7
```

page(2,2) =

```
7   0   1   3
2   4   8   1
7   5   8   6
6   8   8   4
9   4   1   2
```

page(3,2) =

```
1   6   6   5
2   9   1   3
7   1   1   1
8   0   1   5
3   2   7   6
```

```
1
2
5
0
3
4
1
6
1
2
3
7
3
5
7
5
9
2
9
5
6
7
0
9
1
2
1
0
4
8
4
4
1
4
2
2
9
5
2
5
2
2
5
0
9
2
5
1
9
4
⋮
```

Again, a single subscript indexes directly into this column. For example, `C(4)` produces the result

```
ans =
      0
```

If you specify two subscripts `(i,j)` indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, `C(6,2)` is invalid, because all pages of `C` have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts `(i,j,k,l)` into a four-dimensional array with size `[d1 d2 d3 d4]`. MATLAB calculates the offset into the storage column by

```
(l 1)(d3)(d2)(d1)+(k 1)(d2)(d1)+(j 1)(d1)+i
```

For example, if you index the array `C` using subscripts (3,4,2,1), MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions $[d_1 \ d_2 \ d_3 \ ... \ d_n]$ using any subscripts $(s_1 \ s_2 \ s_3 \ ... \ s_n)$ is

$$(s_n-1)(d_{n-1})(d_{n-2})...(d_1)+(s_{n-1}-1)(d_{n-2})...(d_1)+...+(s_2-1)(d_1)+s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of 1s to the subscript list because these terms become zero. For example,

```
C(3,2,1,1,1,1,1,1)
```

is equivalent to

```
C(3,2)
```

Empty Matrices

A matrix having at least one dimension equal to zero is called an *empty matrix*. The simplest empty matrix is 0-by-0 in size. Examples of more complex matrices are those of dimension 0-by-5 or 10-by-0-by-20.

To create a 0-by-0 matrix, use the square bracket operators with no value specified.

```
A = [];

whos A
  Name       Size          Bytes  Class

   A         0x0               0  double array
```

You can create empty arrays of other sizes using the zeros, ones, rand, or eye functions. To create a 0-by-5 matrix, for example, use

```
E = zeros(0,5)
```

Operating on an Empty Matrix

The basic model for empty matrices is that any operation that is defined for m-by-n matrices, and that produces a result whose dimension is some function of m and n, should still be allowed when m or n is zero. The size of the result should be that same function, evaluated at zero.

For example, horizontal concatenation

```
C = [A B]
```

requires that A and B have the same number of rows. So if A is m-by-n and B is m-by-p, then C is m-by-$(n+p)$. This is still true if m or n or p is zero.

Many operations in MATLAB produce row vectors or column vectors. It is possible for the result to be the empty row vector

```
r = zeros(1,0)
```

or the empty column vector

```
C = zeros(0,1)
```

As with all matrices in MATLAB, you must follow the rules concerning compatible dimensions. In the following example, an attempt to add a 1-by-3 matrix to a 0-by-3 empty matrix results in an error.

```
[1 2 3] + ones(0,3)

??? Error using ==> +
Matrix dimensions must agree.
```

Some MATLAB functions, like sum and max, are *reductions*. For matrix arguments, these functions produce vector results; for vector arguments they produce scalar results. Empty inputs produce the following results with these functions:

- sum([ ]) is 0
- prod([ ]) is 1
- max([ ]) is [ ]
- min([ ]) is [ ]

Using Empty Matrices with If or While

When the expression part of an if or while statement reduces to an empty matrix, MATLAB evaluates the expression as being false. The following example executes statement S0, because A is an empty array.

```
A = ones(25,0,4);

if A
    S1
else
    S0
end
```

Errors and Warnings

In many cases, it is desirable to take specific actions when different kinds of errors occur. For example, you may want to prompt the user for more input, display extended error or warning information, or repeat a calculation using default values. The error handling capabilities in MATLAB let your application check for particular error conditions and execute appropriate code depending on the situation.

This section covers the following topics:

- "Checking for Errors with try-catch"
- "Handling and Recovering from an Error" on page 16-75
- "Warnings" on page 16-79
- "Message Identifiers" on page 16-81
- "Using Message Identifiers with lasterr" on page 16-82
- "Warning Control" on page 16-84
- "Debugging Errors and Warnings" on page 16-92

Checking for Errors with try-catch

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when run under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

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 sample function that multiplies two matrices:

```
function matrix_multiply(A, B)
try
   X = A * B
catch
   disp '** Error multiplying A * B'
end
```

A try-catch block is divided into two sections. The first begins with try and the second with catch. Terminate the block with end:

- All statements in the `try` segment are executed normally, just as if they were in the regular code flow. But if any of these operations result in an error, MATLAB skips the remaining statements in the `try` and jumps to the `catch` segment of the block.

- The `catch` segment handles the error. In this example, it displays a general error message. If there are different types of errors that can occur, you will want to identify which error has been caught and respond to that specific error. You can also try to recover from an error in the `catch` section.

When you execute the above example with inputs that are incompatible for matrix multiplication (e.g., the column dimension of A is not equal to the row dimension of B), MATLAB catches the error and displays the message generated in the `catch` section of the `try-catch` block.

```
A = [1  2  3; 6  7  2; 0  1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Error multiplying A * B
```

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
```

Handling and Recovering from an Error
The `catch` segment of a `try-catch` block needs to effectively handle any errors that may be caught by the preceding `try`. Frequently, you will want to simply report the error and stop execution. This prevents erroneous data from being propagated into the remainder of the program.

16-75

16 M-File Programming

Reporting an Error

To report an error and halt program execution, use the MATLAB `error` function. You determine what the error mesage will be by specifying it as an input to the `error` function in your code. For example,

```
if n < 1
    error('n must be 1 or greater.')
end
```

displays the message shown below when `n` is equal to zero.

```
??? n must be 1 or greater.
```

Formatted Message Strings. The error message string that you specify can also contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the error string the first argument, and then add any variables used by the conversion as subsequent arguments.

```
error('formatted_errormsg', arg1, arg2, ...)
```

For example, if your program cannot find a specific file, you might report the error with

```
error('File %s not found', filename);
```

Message Identifiers. Use a message identifier argument with `error` to attach a unique tag to that error message. MATLAB uses this tag to better identify the source of an error. The first argument in this example is the message identifier.

```
error('MATLAB:noSuchFile', 'File "%s" not found', ...
        filename);
```

See "Using Message Identifiers with lasterr" on page 16-82 for more information on how to use identifiers with errors.

Formatted String Conversion. MATLAB converts special characters (like `\n` and `%d`) in the error message string only when you specify more than one input argument with `error`. In the single argument case shown below, `\n` is taken to mean `backslash-n`. It is not converted to a newline character.

```
error('In this case, the newline \n is not converted.')
??? In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This is true regardless of whether the additional argument supplies conversion values or is a message identifier.

```
error('ErrorTests:convertTest', ...
      'In this case, the newline \n is converted.')
??? In this case, the newline
 is converted.
```

Identifying an Error

Once an error has been caught, you will need to know the source of the error in order to handle it appropriately. The lasterr function returns information that enables you to identify the error that was most recently generated by MATLAB.

To return the most recent error message to the variable errormsg, type

```
errormsg = lasterr;
```

You can also change the text of the last error message with a new message or with an empty string as shown below. You might want to do this if a lower level routine detects an error that you don't want visible to the upper levels.

```
lasterr('newerrormsg');   % Replace last error with new string
lasterr('');              % Replace last error with empty string
```

Example. The matrix_multiply function shown earlier in this section could fail for various reasons. If it is called with incompatible matrices, for example, lasterr returns the following string.

```
lasterr
ans =
   Error using ==> *
   Inner matrix dimensions must agree.
```

This example uses lasterr to determine the cause of an error in matrix_multiply.

```
function matrix_multiply(A, B)
try
   A * B
```

```
catch
    errmsg = lasterr;
    if(strfind(errmsg, 'Inner matrix dimensions'))
        disp('** Wrong dimensions for matrix multiply')
    else
        if(strfind(errmsg, 'not defined for variables of class'))
            disp('** Both arguments must be double matrices')
        end
    end
end
```

When calling the function with two matrices not compatible for matrix multiplication, you get the following error message.

```
A = [1  2  3; 6  7  2; 0  1  5];
B = [9  5  6; 0  4  9];
matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

When calling the function with a cell array argument, you get a message that addresses that error.

```
C = {9 5 6; 0 4 9};
matrix_multiply(A, C)
** Both arguments must be double matrices
```

Regenerating an Error

Use the rethrow function to regenerate an error that has previously been thrown. You may want to do this from the catch part of a try-catch block, for example, after performing some required cleanup tasks following an error.

This is how you would rethrow an error in a try-catch block:

```
try
    do_something
catch
    do_cleanup
    rethrow(err)
end
```

rethrow generates an error based on the err input argument that you provide. This argument must be a MATLAB structure with at least the following two fields.

Fieldname	Description
message	Text of the error message
identifier	Message identifier of the error message

If you simply want to regenerate the last error that occurred, the lasterror function returns a structure that can then be passed directly to rethrow.

Note lasterror is not the same as lasterr. The lasterror return structure contains the message string and message identifier for the last error, and may contain more information in future versions of MATLAB. The lasterr return string arguments contain only the message string and identifier.

Warnings

Like error, the warning function alerts the user of unexpected conditions detected when running a program. However, warning does not halt the execution of the program. It displays the specified warning message and then continues.

Reporting a Warning

Use warning in your code to generate a warning message during execution. Specify the message string as the input argument to warning. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you don't want to see. You do this by invoking warning with certain control parameters. See "Warning Control" on page 16-84 for more information.

Formatted Message Strings. The warning message string that you specify can also contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the warning string the first argument, and then add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters (like \n and %d) in the warning message string only when you specify more than one input argument with `warning`. See "Formatted String Conversion" on page 16-76 for information.

Message Identifiers. Use a message identifier argument with `warning` to attach a unique tag to that warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

See "Warning Control Statements" on page 16-86 for more information on how to use identifiers with warnings.

Identifying a Warning

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. You can use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here.

```
lastwarn('newwarnmsg');   % Replace last warning with new string
lastwarn('');             % Replace last warning with empty string
```

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs. See the following topics for more information on how message identifiers are used.

- "Using Message Identifiers with lasterr" on page 16-82
- "Warning Control" on page 16-84

The message identifier is a string that specifies a *component* and a *mnemonic* label for an error or warning. A simple identifier looks like this.

```
component:mnemonic
```

Some examples of message identifiers are

```
MATLAB:divideByZero
Simulink:actionNotTaken
TechCorp:notFoundInPath
```

Both the component and mnemonic fields must adhere to the following syntax rules:

- No whitespace is allowed in the entire identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or underscore.

There is no length limitation to either field.

Component Field

The component field of a message identifier specifies a broad category under which various errors and warnings may be generated. Common components are a particular product or toolbox name, such as MATLAB or Control, or perhaps the name of your company, such as TechCorp in the example above.

You can also use this field to specify a multilevel component. The statement below has a three-level component followed by a mnemonic label.

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

One purpose of the component field is to enable you to guarantee the uniqueness of each identifier. Thus, while MATLAB uses the identifier `MATLAB:divideByZero` for its `'Divide by zero'` warning, you could reuse the `divideByZero` mnemonic by using your own unique component. For example,

```
warning('TechCorp:divideByZero', 'A sprocket value was divided by
zero.')
```

Mnemonic Field

The `mnemonic` field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a mnemonic such as `syntaxAmbiguous` might be appropriate.

```
error('MATLAB:syntaxAmbiguous', ...
      'Syntax %s could be ambiguous.\n', inputstr)
```

Using Message Identifiers with lasterr

One use of message identifiers is to enable the `lasterr` and `lasterror` functions to better identify the source of an error. These functions return a message identifier, and you can use the combination of `component` and `mnemonic` parts of that identifier to identify both a broad category of errors and a specific instance within that category, respectively.

The first step in using this feature is to determine which of your error messages need this type of identification and then tag each with an identifier. You do this by specifying a message identifier argument (`msg_id`, below) along with the error message in your calls to `error`. Either form shown below is acceptable. The latter form uses formatting conversion characters as described in "Formatted Message Strings" on page 16-80.

```
error('msg_id', 'errormsg')
error('msg_id', 'formatted_errormsg', arg1, arg2, ...)
```

Note When you specify more than one input argument with `error`, MATLAB treats the `errormsg` string as if it were a `formatted_errormsg`. This is explained in "Formatted String Conversion" on page 16-76.

The message identifier must be the first argument and must be formatted according to the rules covered in "Message Identifiers" on page 16-81.

The message identifier is not a required argument for error. If you don't need to return this type of information with lasterr, then you can omit the msg_id argument from the error function syntax shown above.

Returning a Message Identifier from lasterr

Use lasterr with one output to return just the string holding the error message from the most recently generated error.

```
errormsg = lasterr;
```

Use lasterr with two outputs to return both error message string and the message identifier for that error.

```
[errormsg, msg_id] = lasterr;
```

The following example performs an operation in the try segment of the try-catch block that results in an error. The first line of the catch segment retrieves both the error message string and message identifier for the error. The example then responds to the error in a manner that depends on the identifier returned.

```
try
   [d, x] = my_readimage(image_file);
catch
   [errmsg, msg_id] = lasterr;
   switch (lower(msg_id))
   case 'MATLAB:noSuchFile'
      error('File "%s" does not exist.', filename);
   case 'myFileIO:noAccess'
      error(['Can''t open file "%s" for reading\n', ...
             'You may not have read permission.'], filename);
   case 'myFileIO:invFormat'
      error('Unable to determine the file format.');
   end
end
```

If the last error has no message identifier tag associated with it, then MATLAB returns an empty string in the second output argument.

```
error('This error has no message identifier.');
??? This error has no message identifier.

[errstr, msgid] = lasterr
errstr =
   This error has no message identifier.
msgid =
   ''
```

Note Both `lasterr` and `lasterror` return a message identifier. Although this section discusses only `lasterr`, you can use `lasterror` in the same way.

Inputs to lasterr

In addition to returning information about the last error, `lasterr` also accepts inputs that modify the MATLAB copy of the last error. Use the command format shown below to change the error message string and message identifier returned by subsequent invocations of `lasterr`.

```
[last_errmsg, last_msgid] = lasterr('new_errmsg', 'new_msgid');
```

All `lasterr` input arguments are optional, but if you specify both an error message and message identifier input, they must appear in the order shown above.

Warning Control

MATLAB gives you the ability to control what happens when a warning is encountered during M-file program execution. Options that are available include

- Display selected warnings
- Ignore selected warnings
- Stop in the debugger when a warning is invoked
- Display an M-stack trace after a warning is invoked

Depending on how you set up your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

1 Start by determining the scope of the control you will need for the warnings generated by your code. Do you want the above control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?

2 If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to those warnings. If, on the other hand, you don't require that fine a granularity of control, then the warning statements in your code need no message identifiers.

3 When you are ready to execute your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements that you put into your M-file code must contain the string that is to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you have no need to single out certain warnings for control, then you only need to specify the message string. Use the syntax shown in the section on "Warnings" on page 16-79. Valid formats are

```
warning('warnmsg')
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement. If there are specific warnings that you want MATLAB to be able to apply control statements to, then you need to include a message identifier in the warning statement. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See "Message Identifiers" on page 16-81 for information on how to specify the msg_id argument.

Note When you specify more than one input argument with warning, MATLAB treats the warnmsg string as if it were a formatted_warnmsg. This is explained in "Formatted String Conversion" on page 16-76.

Warning Control Statements

Once you have the warning statements in your M-file code and are ready to execute the code, you can indicate how you want MATLAB to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can also return information on the state of selected warnings. This only happens if you assign the output to a variable, as shown below. See "Output from Control Statements" on page 16-89.

```
s = warning('state', 'msg_id');
```

Warning States. There are three possible values for the state argument of a warning control statement.

State	Description
on	Enable the display of selected warning message.
off	Disable the display of selected warning message.
query	Display the current state of selected warning.

Message Identifiers. In addition to the message identifiers already discussed, there are two other identifiers that you can use in control statements only.

Identifier	Description
msg_id string	Set selected warning to the specified state.
all	Set all warnings to the specified state.
last	Set only the last displayed warning to the specified state.

Note MATLAB starts up with all warnings enabled.

Example 1. Enable just the actionNotTaken warning from Simulink by first turning off all warnings and then setting just that warning to on.

```
warning off all
warning on Simulink:actionNotTaken
```

Use query to determine the current state of all warnings. It reports that you have set all warnings to off with the exception of Simulink:actionNotTaken.

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are

  State  Warning Identifier

     on  Simulink:actionNotTaken
```

Example 2. Evaluating inv on zero displays a warning message. Turn off the most recently invoked warning with warning off last.

```
inv(0)
Warning: Matrix is singular to working precision.
ans =
   Inf
```

```
warning off last

inv(0)                          % No warning is displayed this time
ans =
   Inf
```

Debug, Backtrace, and Verbose Modes

In addition to warning messages, there are three *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
debug	Stop in the debugger when a warning is invoked.	off
backtrace	Display an M-stack trace after a warning is invoked.	off
verbose	Display a message on how to suppress the warning.	on

The syntax for using this type of control statement is as follows, where state, in this case, can be only on, off, or query.

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Example 1. To enter debug mode whenever a Simulink actionNotTaken warning is invoked, first turn off all warnings and enable only this one type of warning using its message identifier. Then turn on debug mode for all enabled warnings. When you run your program, MATLAB will stop in debug mode just before this warning is executed. You will see the debug prompt (K>>) displayed.

```
warning off all
warning on Simulink:actionNotTaken
warning on debug
```

Example 2. By default, there is an extra line of information shown with each warning telling you how to suppress it.

```
(Type "warning off MATLAB:divideByZero" to suppress this
warning.)
```

Disable this extra line by turning off `verbose` mode.

```
warning off verbose
```

Output from Control Statements

When you include an output variable in a control statement, `warning` returns a MATLAB structure array containing the state of the selected warning and assigns it to this variable. You must type the command using the MATLAB function format, that is, parentheses and quotation marks are required.

```
s = warning('state', 'msg_id');
```

Note MATLAB does not display `warning` output when the terminating semicolon is left off of a control statement. You must assign the output to a variable to have output returned from `warning`.

You can use output variables with any type of control statement. If you just want to collect the information but don't want to change state, then simply perform a `query` on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but also save the former state so that you can restore it later, use the return structure array to save that state. The following example does an implicit `query`, returning state information in `s`, and then turns on all warnings.

```
s = warning('on', 'all');
```

See the section, "Saving and Restoring State" on page 16-91, for more information on restoring the former state of warnings.

Output Structure Array. Each element of the structure array returned by warning contains two fields.

Fieldname	Description
identifier	Message identifier string, 'all', or 'last'
state	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or 'last' in the command, then MATLAB returns a one-element structure array. The identifier field contains the selected message identifier and the state field holds the current state of that warning.

```
s = warning('query','last')
s =
    identifier: 'MATLAB:divideByZero'
         state: 'on'
```

If you query for the state of all warnings, using 'all' in the command, MATLAB returns a structure array having one or more elements.

- The first element of the array always represents the default state. (This is the state set by the last warning on|off all command.)

- Each other element of the array represents a warning that is in a state that is different from the default.

```
warning off all
warning on MATLAB:divideByZero
warning on MATLAB:fileNotFound

s = warning('query', 'all')
s =
   3x1 struct array with fields:
       identifier
       state
```

```
s(1)
ans =
   identifier: 'all'
        state: 'off'

s(2)
ans =
   identifier: 'MATLAB:divideByZero'
        state: 'on'

s(3)
ans =
   identifier: 'MATLAB:fileNotFound'
        state: 'on'
```

Saving and Restoring State

If you want to temporarily change the state of some warnings and then later return to your original settings, you can save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in the last section, "Output from Control Statements" on page 16-89. The following statement saves the current state of all warnings in structure array s.

```
s = warning('query', 'all');
```

To restore state from s, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1. Perform a query of all warnings to save the current state in structure array s.

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings.

```
warning(s)
```

Example 2. Turn on one particular warning, saving the previous state of this warning in s. Remember that this nonquery syntax (where state equals on or off) performs an implicit query prior to setting the new state.

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Debugging Errors and Warnings

You can direct MATLAB to temporarily stop the execution of an M-file program in the event of a run-time error or warning, at the same time opening a debug window paused at the M-file line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the dbstop function to have MATLAB stop execution and enter debug mode when any M-file you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
dbstop if all error	Stop on any error
dbstop if error	Stop on any error not detected within a try-catch block
dbstop if warning	Stop on any warning

In all three cases, the M-file you are trying to debug must be in a directory that is on the search path or in the current directory.

You cannot resume execution after an error; use dbquit to exit from the Debugger. Use dbcont or dbstep to resume execution after a warning.

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command ! to make external stand-alone programs act like new MATLAB functions. A shell escape M-function is an M-file that:

1 Saves the appropriate variables on disk.

2 Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).

3 Loads the processed file back into the workspace.

For example, look at the code for garfield.m, below. This function uses an external function, gareqn, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This M-file:

1 Saves the input arguments a, b, q, and r to a MAT-file in the workspace using the save command.

2 Uses the shell escape operator to access a C, or Fortran program called gareqn that uses the workspace variables to perform its computation. gareqn writes its results to the gardata MAT-file.

3 Loads the gardata MAT-file to obtain the results.

Using MATLAB Timers

MATLAB includes a timer object that you can use to schedule the execution of MATLAB commands. To use a timer, perform these steps:

1 Create a timer object by calling the `timer` function. See "Creating and Deleting Timer Objects" on page 16-95 for more information.

2 Set the values of timer object properties to specify which MATLAB commands you want executed and when you want them executed. (You can also set timer object properties when you create them, in Step 1.) See "Timer Object Properties" on page 16-97 for information about all the properties supported by the timer object.

3 Start the timer by calling the `start` or `startat` function. See "Starting and Stopping Timers" on page 16-99 for more information.

Note Because the timer works within the MATLAB single-threaded execution environment, there can be a variance between the specified execution time and the actual execution of timer callback functions. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue. For more information about callback processing, see The Event Queue in the MATLAB GUI programming documentation.

For example, this code sets up a timer that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, `TimerFcn` and `StartDelay`. `TimerFcn` specifies the timer callback function. This is the MATLAB command string or M-file that you want to execute. `StartDelay` specifies how much time elapses before the timer executes the callback function.

After creating the timer object, the example uses the `start` function to start the timer. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
                    'StartDelay',10);
start(t)

stat=1;
while(true)
  disp('.')
  pause(1)
end
```

When you execute this code, it produces this output:

```
.
.
.
.
.
.
.
.
.
Timer!
```

Creating and Deleting Timer Objects

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the timer function. You can create a timer object with default attributes and then use the set function to specify the values of its properties, as in this example:

```
t = timer;
set(t,'TimerFcn','disp(''Hello World!'')','StartDelay',15)
```

You can also set timer object properties when you create the timer object by specifying property name and property value pairs as arguments to the timer function:

```
t = timer('TimerFcn', 'disp(''Hello World!'')','StartDelay',15);
```

Deleting Timer Objects from Memory

When you are finished with a timer, delete it from memory using the `delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object remain. Deleted timer objects are invalid and cannot be reused. Use the `clear` command to remove workspace variables that reference deleted timer objects. To test if a timer object has been deleted, use the `isvalid` function. The `isvalid` function returns `false` for deleted timer objects:

```
isvalid(t)
ans =

    0
```

To remove all timer objects from memory, enter

```
delete(timerfind)
```

See "Finding All Timer Objects Currently in Memory" on page 16-96 for information about the `timerfind` function.

Finding All Timer Objects Currently in Memory

To find all the timers that are currently in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t = timer;
t2 = timer;
t3 = timer;
t_array = timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-3
   2       singleShot      1        ''          timer-4
   3       singleShot      1        ''          timer-5
```

MATLAB assigns a name to each timer object you create. This name has the form `'timer-i'`, where *i* is a number representing the total number of timer objects created this session. In this example, the list of timer object names start with `timer-3`. This indicates that two other timer objects were previously created and then deleted in this MATLAB session. If you issue the `clear classes` command, MATLAB resets *i* to 1.

Timer Object Properties

The timer object supports many properties that control aspects of its functioning, such as:

- What functions to execute
- When to execute it
- How many times to execute it

These properties fall into four general categories, described in the following tables. For more detailed information about these properties, see `timer`.

Informational Properties	Provide information about the current state of the timer
Behavioral Properties	Specify how the timer operates
Timer Callback Function Properties	Specify which functions the timer executes
User Specified Properties	Enable you to include application-specific information with your timers

Informational Properties

These timer object properties provide information about the current state of the timer object. These properties are all read-only properties.

Property	Description
AveragePeriod	The average time between executions of `TimerFcn` since the timer began
InstantPeriod	The time between the last two executions of `TimerFcn`

Property	Description
Running	The current state of the timer, `'on'` or `'off'`
TasksExecuted	The number of times the timer has executed `TimerFcn` since the timer was started

Behavioral Properties

These timer object properties control the basic functioning of the timer. For more information about these properties, see "Timer Object Execution Modes" on page 16-101.

Property	Description
BusyMode	Specifies how the timer object handles the queuing of the timer callback function (`TimerFcn`) when the previously queued execution of `TimerFcn` has not finished
ExecutionMode	Specifies how the timer object queues the timer callback function (`TimerFcn`) for execution
Period	The number of seconds between executions of the timer callback function (`TimerFcn`)
StartDelay	The number of seconds before queuing the first execution of the timer callback function (`TimerFcn`)
TasksToExecute	The total number of times to execute the timer callback function (`TimerFcn`)

Timer Callback Function Properties

These timer object properties specify the callback functions that the timer executes. For more information, see "Creating Timer Callback Functions" on page 16-104.

Property	Description
ErrorFcn	The function the timer executes when an error occurs
StartFcn	The function the timer executes when it starts
StopFcn	The function the timer executes when it stops executing. See "Stopping Timers" on page 16-100 for more information.
TimerFcn	The function you want the timer to execute

User-Specified Properties

These timer object properties enable users to associate application-specific data with the timer objects they create.

Property	Description
Name	A text string that identifies the timer
Tag	User-defined label for the timer object
UserData	User-defined data

Starting and Stopping Timers

After you create a timer object, you can start the timer by calling either the start or startat function.

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

The start function starts the timer immediately, changing the value of the timer object's Running property from 'off' to 'on':

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay',15);
start(t)
```

The start function returns control to the MATLAB command line immediately. You can also choose to wait until the timer callback function (TimerFcn) executes before returning control to the command line. See "Blocking the MATLAB Command Line" on page 16-100 for more information.

The startat function starts the timer immediately and sets the value of the StartDelay property to the time you specify:

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');
startat(t2,now+1/24);
```

Note The timer object can execute a callback function that you specify when it starts. See "Creating Timer Callback Functions" on page 16-104 for more information about using the StartFcn property.

Blocking the MATLAB Command Line

By default, when you start a timer, the start or startat function returns control to the command line immediately. If your application must wait until the function controlled by the timer executes, use the wait function right after calling the start function:

```
t = timer('StartDelay',15,'TimerFcn','disp(''Hello World!'')');
start(t)
wait(t)
```

Stopping Timers

The timer stops running if one of the following conditions apply:

- The timer function callback (TimerFcn) has been executed the number of times specified in the TasksToExecute property.
- The stop command is issued.
- An error occurred while executing a timer function callback (TimerFcn).

When a timer stops, the value of the Running property of the timer object is 'off'.

Note The timer object can execute a callback function that you specify when it stops. See "Creating Timer Callback Functions" on page 16-104 for more information about using the StopFcn property.

Timer Object Execution Modes

The timer object supports several execution modes that determine how the timer object schedules the timer callback function (TimerFcn) for execution. You specify the execution mode by setting the value of the ExecutionMode property. The execution modes fall into two categories:

• Modes that execute the timer function callback once
• Modes that execute the timer function callback multiple times

Executing a Timer Callback Function Once

To execute a timer callback function once, set the ExecutionMode property to 'singleShot'. This is the default execution mode.

In this mode, the timer object starts the timer and, after the time period specified in the StartDelay property elapses, adds the timer callback function (TimerFcn) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

Note Queue lag is not a timer object property. Queue lag is the indeterminate amount of time between when the timer adds the timer callback function (`TimerFcn`) to the MATLAB execution queue and when it actually gets executed. This time is dependent on what other processing MATLAB happens to be doing at the time.

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- `'fixedRate'`
- `'fixedDelay'`
- `'fixedSpacing'`

In many ways, all of these execution modes operate the same:

- The `TasksToExecute` property specifies the number of times you want the timer to execute the timer callback function (`TimerFcn`).
- The `Period` property specifies the amount of time between executions of the timer callback function.
- The `BusyMode` property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See "Handling Callback Function Queuing Conflicts" on page 16-103 for more information.

The execution modes differ only in where they start measuring the time period between executions. In `'fixedRate'` mode, the timer starts measuring this time period immediately after adding the timer callback function (`TimerFcn`) to the MATLAB execution queue. In `'fixedDelay'` mode, the timer starts measuring this time period when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue. In `'fixedSpacing'` mode, the timer starts measuring this time period, from the point when the timer callback function finishes executing.

This figure illustrates the difference between these modes.

Handling Callback Function Queuing Conflicts

At busy times, in multiple execution scenarios, the timer may need to add the timer callback function (TimerFcn) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the BusyMode property.

If you specify 'drop' as the value of the BusyMode property, the timer object skips this execution of the timer function callback.

If you specify 'queue', the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

Note In 'queue' mode, if the timer object has to wait longer than the time specified in the Period property between executions of the timer function callback, it will shorten the time period for subsequent executions to make up the time. The timer object tries to make the average time between executions equal the amount of time specified in the Period property.

If the BusyMode property is set to 'error', the timer stops and executes the timer error function (ErrorFcn).

Creating Timer Callback Functions

Four of the timer object properties specify callback functions that the timer object executes:

- ErrorFcn
- StartFcn
- StopFcn
- TimerFcn

As the value of these properties, you can specify:

- A text string containing MATLAB functions or an M-file name, e.g., 'disp(''Hello World!'')'
- A MATLAB function handle, e.g., @my_callback_fcn
- A cell array specifying an M-file and its arguments, e.g., {'my_callback_fcn',...} or {@my_callback_fcn,...}

When you set the value of these properties, you can specify arguments for the callback function using any of the following syntaxes.

Callback Function Syntax	Example
Function myfile	set(h, 'StartFcn', 'myfile')
Function myfile(obj, event)	set(h, 'StartFcn', @myfile)

Callback Function Syntax	Example
`Function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {'myfile', 5, 6})`
`Function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {@myfile, 5, 6})`

The following diagram illustrates when the timer object executes the StartFcn, StopFcn, and TimerFcn. The ErrorFcn can occur at any time.

In this figure, the timer stops after two executions of the timer callback function (TimerFcn):

```
t = timer('StartDelay',10,'TimerFcn','disp(''Hello World!'')',
'Period', 10,'TasksToExecute',2);
```

Character Arrays (Strings)

This chapter covers MATLAB support for string data. It describes the two ways that MATLAB represents strings, and also describes the operations that you can perform on string data. It covers the following topics:

Function Summary

This table shows the string functions, which are located in the directory named strfun in the MATLAB Toolbox.

Category	Function	Description
General	blanks	String of blanks
	cellstr	Create cell array of strings from character array
	char	Create character array (string)
	deblank	Remove trailing blanks
	eval	Execute string with MATLAB expression
String Operations	findstr	Find one string within another
	lower	Convert string to lowercase
	strcat	Concatenate strings
	strcmp	Compare strings
	strcmpi	Compare strings, ignoring case
	strjust	Justify string
	strmatch	Find matches for string
	strncmp	Compare first N characters of strings
	strncmpi	Compare first N characters, ignoring case
	strrep	Replace string with another
	strtok	Find token in string
	strvcat	Concatenate strings vertically
	upper	Convert string to uppercase

Category	Function	Description
String Tests	iscellstr	True for cell array of strings
	ischar	True for character array
	isletter	True for letters of alphabet.
	isspace	True for whitespace characters.
String to Number Conversion	double	Convert string to numeric codes
	int2str	Convert integer to string
	mat2str	Convert matrix to eval'able string
	num2str	Convert number to string
	sprintf	Write formatted data to string
	str2double	Convert string to double-precision value
	str2num	Convert string to number
	sscanf	Read string under format control
Base Number Conversion	base2dec	Convert base B string to decimal integer
	bin2dec	Convert binary string to decimal integer
	dec2base	Convert decimal integer to base B string
	dec2bin	Convert decimal integer to binary string
	dec2hex	Convert decimal integer to hexadecimal string
	hex2dec	Convert hexadecimal string to decimal integer
	hex2num	Convert IEEE hexadecimal to double-precision number

Character Arrays

In MATLAB, the term *string* refers to an array of characters. MATLAB represents each character internally as its corresponding numeric value. Unless you want to access these values, however, you can simply work with the characters as they display on screen.

This section covers:

- "Creating Character Arrays"
- "Creating Two-Dimensional Character Arrays"
- "Converting Characters to Numeric Values"

Creating Character Arrays

Specify character data by placing characters inside a pair of single quotes. For example, this line creates a 1-by-13 character array called name.

```
name = 'Thomas R. Lee';
```

In the workspace, the output of whos shows

```
Name        Size          Bytes Class

name        1x13             26  char array
```

You can see that a character uses two bytes of storage internally.

The class and ischar functions show name's identity as a character array.

```
class(name)
ans =
    char

ischar(name)
ans =
     1
```

You can also join two or more character arrays together to create a new character array. Use either the string concatenation function, `strcat`, or the MATLAB concatenation operator, `[]`, to do this. The latter preserves any trailing spaces found in the input arrays.

```
name = 'Thomas R. Lee';
title = ' Sr. Developer';

strcat(name,',',title)
ans =
    Thomas R. Lee, Sr. Developer
```

You can also concatenate strings vertically with `strvcat`.

Creating Two-Dimensional Character Arrays

When creating a two-dimensional character array, be sure that each row has the same length. For example, this line is legal because both input rows have exactly 13 characters.

```
name = ['Thomas R. Lee' ; 'Sr. Developer']
name =
    Thomas R. Lee
    Sr. Developer
```

When creating character arrays from strings of different lengths, you can pad the shorter strings with blanks to force rows of equal length.

```
name = ['Thomas R. Lee   '; 'Senior Developer'];
```

A simpler way to create string arrays is to use the `char` function. `char` automatically pads all strings to the length of the longest input string. In this example, `char` pads the 13-character input string `'Thomas R. Lee'` with three trailing blanks so that it will be as long as the second string.

```
name = char('Thomas R. Lee','Senior Developer')
name =
    Thomas R. Lee
    Senior Developer
```

When extracting strings from an array, use the `deblank` function to remove any trailing blanks.

```
trimname = deblank(name(1,:))
trimname =
   Thomas R. Lee

size(trimname)
ans =
    1    13
```

Converting Characters to Numeric Values

Character arrays store each character as a 16-bit numeric value. Use the `double` function to convert strings to their numeric values, and `char` to revert to character representation.

```
name = double(name)
name =
    84   104   111   109   97   115   32   82   46   32   76   101   101

name = char(name)
name =
   Thomas R. Lee
```

Use `str2num` to convert a character array to the numeric value represented by that string.

```
str = '37.294e-1';

val = str2num(str)
val =
    3.7294
```

Cell Arrays of Strings

It's often convenient to store groups of strings in cell arrays instead of standard character arrays. This prevents you from having to pad strings with blanks to create character arrays with rows of equal length. A set of functions enables you to work with cell arrays of strings:

- You can convert between standard character arrays and cell arrays of strings.
- You can apply string comparison operations to cell arrays of strings.

For details on cell arrays see the "Structures and Cell Arrays" chapter.

Converting to a Cell Array of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider the character array

```
data = ['Allison Jones';'Development  ';'Phoenix      '];
```

Each row of the matrix is padded so that all have equal length (in this case, 13 characters).

Now use `cellstr` to create a column vector of cells, each cell containing one of the strings from the `data` array.

```
celldata = cellstr(data)
celldata =
    'Allison Jones'
    'Development'
    'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix.

```
length(celldata{3})
ans =
    7
```

The iscellstr function determines if the input argument is a cell array of strings. It returns a logical true (1) in the case of celldata.

```
iscellstr(celldata)
ans =
     1
```

Use char to convert back to a standard padded character array.

```
strings = char(celldata)
strings =
   Allison Jones
   Development
   Phoenix

length(strings(3,:))
ans =
    13
```

String/Numeric Conversion

The str2double function converts a cell array of strings to the double-precision values represented by the strings.

```
c = {'37.294e-1'; '-58.375'; '13.796'};

d = str2double(c)
d =
    3.7294
  -58.3750
   13.7960

whos
  Name      Size                    Bytes  Class

  c         3x1                       224  cell array
  d         3x1                        24  double array

Grand total is 28 elements using 248 bytes
```

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.
- You can compare individual characters in two strings for equality.
- You can categorize every element within a string, determining whether each element is a character or whitespace.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings For Equality

There are four functions that determine if two input strings are identical:

- strcmp determines if two strings are identical.
- strncmp determines if the first n characters of two strings are identical.
- strcmpi and strncmpi are the same as strcmp and strncmp, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
str2 = 'help';
```

Strings str1 and str2 are not identical, so invoking strcmp returns 0 (false). For example,

```
C = strcmp(str1,str2)
C =
    0
```

Note For C programmers, this is an important difference between the MATLAB strcmp and C's strcmp(), which returns 0 if the two strings are the same.

The first three characters of str1 and str2 are identical, so invoking strncmp with any value up to 3 returns 1.

```
C = strncmp(str1,str2,2)
C =
     1
```

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings

```
A = {'pizza';'chips';'candy'};
B = {'pizza';'chocolate';'pretzels'};
```

Now apply the string comparison functions.

```
strcmp(A,B)
ans =
     1
     0
     0

strncmp(A,B,1)
ans =
     1
     1
     0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine which characters in two strings match.

```
A = 'fate';
B = 'cake';

A == B
ans =
     0   1   0   1
```

All of the relational operators (>, >=, <, <=, ==, !=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are two functions for categorizing characters inside a string:

- isletter determines if a character is a letter
- isspace determines if a character is whitespace (blank, tab, or new line)

For example, create a string named mystring.

```
mystring = 'Room 401';
```

isletter examines each character in the string, producing an output vector of the same length as mystring.

```
A = isletter(mystring)
A =
    1   1   1   1   0   0   0   0
```

The first four elements in A are 1 (true) because the first four characters of mystring are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. Consider a string named `label`.

```
label = 'Sample 1, 10/28/95';
```

The `strrep` function performs the standard search-and-replace operation. Use `strrep` to change the date from `'10/28'` to `'10/30'`.

```
newlabel = strrep(label,'28','30')
newlabel =
    Sample 1, 10/30/95
```

`findstr` returns the starting position of a substring within a longer string. To find all occurrences of the string `'amp'` inside `label`

```
position = findstr('amp',label)
position =
     2
```

The position within `label` where the only occurrence of `'amp'` begins is the second character.

The `strtok` function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of whitespace characters. You can use the `strtok` function to parse a sentence into words; for example,

```
function all_words = words(input_string)
remainder = input_string;
all_words = '';

while (any(remainder))
  [chopped,remainder] = strtok(remainder);
  all_words = strvcat(all_words,chopped);
end
```

The strmatch function looks through the rows of a character array or cell array of strings to find strings that begin with a given series of characters. It returns the indices of the rows that begin with these characters.

```
maxstrings = strvcat('max','minimax','maximum')
maxstrings =
    max
    minimax
    maximum

strmatch('max',maxstrings)
ans =
     1
     3
```

Regular Expressions

There are several MATLAB functions that support searching and replacing characters using regular expressions. These functions are

- `regexp`
- `regexpi`
- `regexprep`

The following tables list the regular expression syntax supported by MATLAB. This list is not intended to be a complete reference. For more information on regular expressions and their meanings, consult a reference on regular expressions.

Note Any character appearing in a regular expression is ordinary, unless a '\' precedes it.

This section contains the following topics:

Metacharacters

Metacharacters don't match themselves but describe something else. Generally they represent groups of characters or keyboard keys without a representative character.

Expression	Usage
`.`	Matches any single character
`[ab...]`	Matches any one of the characters, (a, b, etc.), contained within the brackets

Expression	Usage
[^ab...]	Matches any character except those contained within the brackets, (a, b, etc.).
[c_1-c_2]	Matches any characters in the range of c_1 through c_2.
\f	Form feed.
\n	New line.
\r	Carriage return.
\t	Tab.
\d	A digit. Equivalent regular expression: [0-9]
\D	A nondigit. Equivalent regular expression: [^0-9]
\s	A whitespace character. Equivalent regular expression: [\f\n\r\t]
\S	A non-whitespace character. Equivalent regular expression: [^ \f\n\r\t]
\w	A word character. Equivalent regular expression: [a-zA-Z_0-9]
\W	A nonword character. Equivalent regular expression: [^a-zA-Z_0-9]
\	If a character has special meaning in a regular expression, precede it with this character to match it literally.

Logical Operators

Logical operators do not match any specific characters. They are used to specify the context for matching an accompanying regular expression.

Expression	Usage
(...)	Groups regular expressions.
|	Matches either the expression preceding or following it.
^	Matches following expression only at the beginning of the string.
$	Matches preceding expression only at the end of the string.
\<chars	Matches the characters when they start a word.
chars\>	Matches the characters when they end a word.
\<word\>	Exact word match.

Quantifiers

Quantifiers are a type of logical operator. They are used to specify how many instances of the previous element they can match.

Expression	Usage
*	Matches the preceding element 0 or more times. Equivalent regular expression: {0,}
+	Matches the preceding element 1 or more times. Equivalent regular expression: {1,}
?	Matches the preceding element 0 times or 1 time, also minimizes. Equivalent regular expression: {0,1}
{n,m}	Must occur at least n times but no more than m times

Expression	Usage
{n,}	Must occur at least n times.
{n}	Must match exactly n times. Equivalent regular expression: {n,n}

Searching with Tokens

Within a regular expression, parentheses are used to group characters or expressions. These grouped expressions are called *tokens*. Tokens can be used in matching text. For example, the regular expression and(y|rew) matches the text andy or andrew. Tokens also remember what they matched so that you can recall and reuse the found text with a special variable for searching or replacing.

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

In this pattern there are three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern `and(y|rew)` finds the text `andrew,` token 1 is assigned the value `rew`. However, if the search pattern `(and(y|rew))` is used, token 1 is assigned the value `andrew`.

The variables that allow you to use tokens in your search pattern have the form `\1, \2,..., \n (n<17)` and are assigned left to right from parenthetical expressions in the search string.

As an example, suppose that you are searching an HTML file to find many table entries. Generally, HTML lines with table entries have the following form:

```
<TD><B>Crops<\B><\TD><TD>Wheat<\TD><TD>Corn<\TD><TD>Barley<\TD>
```

You can use search pattern tokens to search and find all table entries with the following search pattern:

```
<(\w*)>(.*?)<\\\1>
```

This search pattern finds the following:

1 The expression `<(\w*)>` finds any number of word characters enclosed by angle brackets. Because of the parentheses around `\w*`, the word characters matched are placed in token 1. In our example, `<(\w*)>` finds the string `<TD>` and places the string `TD` in token 1.

2 The expression `(.*?)` contains the expression `.*?`, which finds a minimum number of any characters. Using the expression `(.*)` instead would find the maximum number of any characters and could use up the entire file before finding the angle bracket characters sought for by the next expression.

The parentheses in the expression `(.*?)` places the matched characters in token 2. Although this is not necessary for the search, placing these characters in a token allows you to reference them in replacement text.

3 The expression `<\\\1>` uses the variable `\1` for token 1 preceded by two backslashes. Since backslashes are regular expression logical operators, two backslashes specify a single backslash as a search character. In the HTML example, since the first expression, `<(\w*)>`, finds the string `<TD>`, and places the string `TD` in token 1, the expression `<\\\1>` finds the following `<\TD>` string.

Numeric/String Conversion

The MATLAB string/numeric conversion functions change numeric values into character strings. You can store numeric values as digit-by-digit string representations, or convert a value into a hexadecimal or binary string. Consider a the scalar

```
x = 5317;
```

By default, MATLAB stores the number x as a 1-by-1 double array containing the value 5317. The int2str (integer to string) function breaks this scalar into a 1-by-4 vector containing the string '5317'.

```
y = int2str(x);
size(y)

ans =

     1     4
```

A related function, num2str, provides more control over the format of the output string. An optional second argument sets the number of digits in the output string, or specifies an actual format.

```
p = num2str(pi,9)

p =

3.14159265
```

Both int2str and num2str are handy for labeling plots. For example, the following lines use num2str to prepare automated labels for the *x*-axis of a plot.

```
function plotlabel(x,y)
plot(x,y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Another class of numeric/string conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. For example, the dec2hex function converts a decimal value into the corresponding hexadecimal string.

```
dec_num = 4035;

hex_num = dec2hex(dec_num)
hex_num =
    FC3
```

See the strfun directory for a complete listing of string conversion functions.

Array/String Conversion

The MATLAB function mat2str changes an array to a string that MATLAB can evaluate. This string is useful input for a function such as eval, which evaluates input strings just as if they were typed at the MATLAB command line.

Create a 2-by-3 array A.

```
A = [1 2 3; 4 5 6]
A =
    1    2    3
    4    5    6
```

mat2str returns a string that contains the text you would enter to create A at the command line.

```
B = mat2str(A)
B =
   [1 2 3; 4 5 6]
```

18

Multidimensional Arrays

This chapter discusses *multidimensional arrays*, MATLAB arrays with more than two dimensions. Multidimensional arrays can be numeric, logical, character, cell, or structure arrays. These arrays are broadly useful—for example, in the representation of multivariate data, or multiple pages of two-dimensional data.

Function Summary

MATLAB provides a number of functions that directly support multidimensional arrays. You can extend this support by creating M-files that work with your data architecture.

Function	Description
cat	Concatenate arrays.
circshift	Shift array circularly.
ipermute	Inverse permute array dimensions.
ndgrid	Generate arrays for N-D functions and interpolation.
ndims	Number of array dimensions.
permute	Permute array dimensions.
shiftdim	Shift dimensions.
squeeze	Remove singleton dimensions.

Multidimensional Arrays

Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.

You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index.

Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This guide uses the concept of a *page* to represent dimensions 3 and higher.

To access the element in the second row, third column of page 2, for example, you use the subscripts (2,3,2).


```
A(:,:,1) =
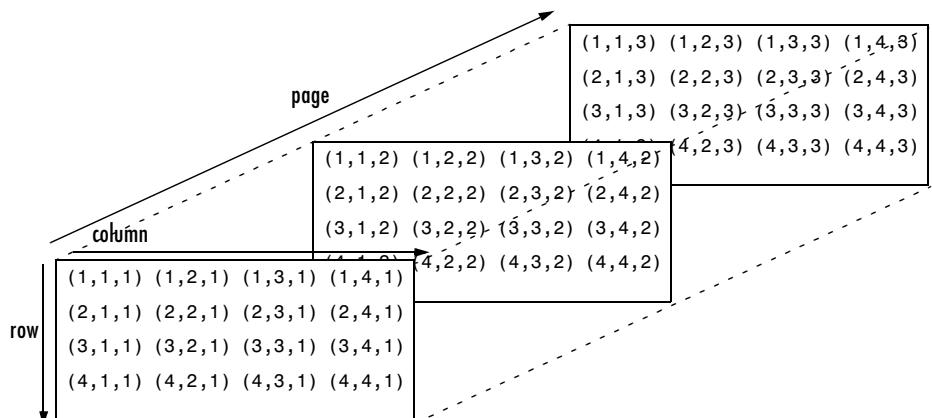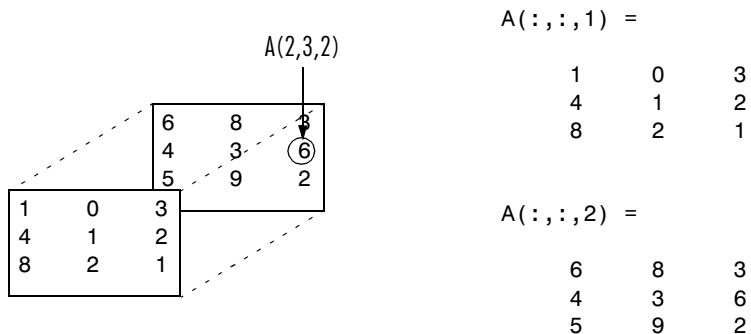
        1       0       3
        4       1       2
        8       2       1


A(:,:,2) =

        6       8       3
        4       3       6
        5       9       2
```

As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two reference a row-column pair; the second two access the third and fourth dimensions of data.

Note The general multidimensional array functions reside in the `datatypes` directory.

Creating Multidimensional Arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. In addition, MATLAB provides a special concatenation function that is useful for building multidimensional arrays.

This section discusses:

• Generating arrays using indexing

• Generating arrays using MATLAB functions

• Using the `cat` function to build multidimensional arrays

Generating Arrays Using Indexing

One way to create a multidimensional array is to create a two-dimensional array and extend it. For example, begin with a simple two-dimensional array A.

```
A = [5 7 8; 0 1 9; 4 3 6];
```

A is a 3-by-3 array, that is, its row dimension is 3 and its column dimension is 3. To add a third dimension to A,

```
A(:,:,2) = [1 0 4; 3 5 6; 9 8 7]
```

MATLAB responds with

```
A(:,:,1) =
     5     7     8
     0     1     9
     4     3     6

A(:,:,2) =
     1     0     4
     3     5     6
     9     8     7
```

You can continue to add rows, columns, or pages to the array using similar assignment statements.

Extending Multidimensional Arrays. To extend A in any dimension:

• Increment or add the appropriate subscript and assign the desired values.

• Assign the same number of elements to corresponding array dimensions. For numeric arrays, all rows must have the same number of elements, all pages must have the same number of rows and columns, and so on.

You can take advantage of the MATLAB scalar expansion capabilities, together with the colon operator, to fill an entire dimension with a single value.

```
A(:,:,3) = 5;

A(:,:,3)
ans =
     5     5     5
     5     5     5
     5     5     5
```

To turn A into a 3-by-3-by-3-by-2, four-dimensional array, enter

```
A(:,:,1,2) = [1 2 3; 4 5 6; 7 8 9];
A(:,:,2,2) = [9 8 7; 6 5 4; 3 2 1];
A(:,:,3,2) = [1 0 1; 1 1 0; 0 1 1];
```

Note that after the first two assignments MATLAB pads A with zeros, as needed, to maintain the corresponding sizes of dimensions.

Generating Arrays Using MATLAB Functions

You can use MATLAB functions such as randn, ones, and zeros to generate multidimensional arrays in the same way you use them for two-dimensional arrays. Each argument you supply represents the size of the corresponding dimension in the resulting array. For example, to create a 4-by-3-by-2 array of normally distributed random numbers.

```
B = randn(4,3,2)
```

To generate an array filled with a single constant value, use the repmat function. repmat replicates an array (in this case, a 1-by-1 array) through a vector of array dimensions.

```
B = repmat(5,[3 4 2])

B(:,:,1) =
    5      5      5      5
    5      5      5      5
    5      5      5      5

B(:,:,2) =
    5      5      5      5
    5      5      5      5
    5      5      5      5
```

Note Any dimension of an array can have size zero, making it a form of empty array. For example, 10-by-0-by-20 is a valid size for a multidimensional array.

Building Multidimensional Arrays with the cat Function

The cat function is a simple way to build multidimensional arrays; it concatenates a list of arrays along a specified dimension.

```
B = cat(dim,A1,A2...)
```

where A1, A2, and so on are the arrays to concatenate, and dim is the dimension along which to concatenate the arrays. For example, to create a new array with cat

```
B = cat(3,[2 8; 0 5],[1 3; 7 9])

B(:,:,1) =
     2     8
     0     5

B(:,:,2) =
     1     3
     7     9
```

The cat function accepts any combination of existing and new data. In addition, you can nest calls to cat. The lines below, for example, create a four-dimensional array.

```
A = cat(3,[9 2; 6 5],[7 1; 8 4])
B = cat(3,[3 5; 0 1],[5 6; 2 1])
D = cat(4,A,B,cat(3,[1 2; 3 4],[4 3;2 1]))
```

cat automatically adds subscripts of 1 between dimensions, if necessary. For example, to create a 2-by-2-by-1-by-2 array, enter

```
C = cat(4,[1 2; 4 5],[7 8; 3 2])
```

In the previous case, cat inserts as many singleton dimensions as needed to create a four-dimensional array whose last dimension is not a singleton dimension. If the dim argument had been 5, the previous statement would have produced a 2-by-2-by-1-by-1-by-2 array. This adds additional 1s to indexing

expressions for the array. To access the value 8 in the four-dimensional case, use

```
C(1,2,1,2)
```

Singleton dimension index

Accessing Multidimensional Array Properties

You can use the following MATLAB functions to get information about multidimensional arrays you have created.

Array Property	Function	Example

Size	size	size(C)
		ans =
		    2 2 1 2
		  rows columns dim3 dim4

Dimensions	ndims	ndims(C) ans = 4
Storage and format	whos	whos Name Size Bytes Class A 2x2x2 64 double array B 2x2x2 64 double array C 4 D 64 double array D 4 D 192 double array Grand total is 48 elements using 384 bytes

Indexing

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well.

To access a single element of a multidimensional array, use integer subscripts. Each subscript indexes a dimension–the first indexes the row dimension, the second indexes the column dimension, the third indexes the first page dimension, and so on.

Consider a 10-by-5-by-3 array nddata of random integers:

```
nddata = fix(8*randn(10,5,3));
```

To access element (3,2) on page 2 of nddata, for example, use nddata(3,2,2).

You can use vectors as array subscripts. In this case, each vector element must be a valid subscript, that is, within the bounds defined by the dimensions of the array. To access elements (2,1), (2,3), and (2,4) on page 3 of nddata, use

```
nddata(2,[1 3 4],3);
```

The Colon and Multidimensional Array Indexing

The MATLAB colon indexing extends to multidimensional arrays. For example, to access the entire third column on page 2 of nddata, use nddata(:,3,2).

The colon operator is also useful for accessing other subsets of data. For example, nddata(2:3,2:3,1) results in a 2-by-2 array, a subset of the data on page 1 of nddata. This matrix consists of the data in rows 2 and 3, columns 2 and 3, on the first page of the array.

The colon operator can appear as an array subscript on both sides of an assignment statement. For example, to create a 4-by-4 array of zeros

```
C = zeros(4,4)
```

Now assign a 2-by-2 subset of array nddata to the four elements in the center of C.

```
C(2:3,2:3) = nddata(2:3,1:2,2)
```

Avoiding Ambiguity in Multidimensional Indexing

Some assignment statements, such as

```
A(:,:,2) = 1:10
```

are ambiguous because they do not provide enough information about the shape of the dimension to receive the data. In the case above, the statement tries to assign a one-dimensional vector to a two-dimensional destination. MATLAB produces an error for such cases. To resolve the ambiguity, be sure you provide enough information about the destination for the assigned data, and that both data and destination have the same shape. For example,

```
A(1,:,2) = 1:10;
```

Reshaping

Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by respecifying the array's row, column, or page dimensions while retaining the same elements. The reshape function performs the latter operation. For multidimensional arrays, its form is

```
B = reshape(A,[s1 s2 s3 ...])
```

s1, s2, and so on represent the desired size for each dimension of the reshaped matrix. Note that a reshaped array must have the same number of elements as the original array (that is, the product of the dimension sizes is constant).

M	**reshape(M, [6 5])**

The reshape function operates in a columnwise manner. It creates the reshaped matrix by taking consecutive elements down each column of the original data construct.

C	reshape(C, [6 2])

Here are several new arrays from reshaping nddata.

```
B = reshape(nddata,[6 25])
C = reshape(nddata,[5 3 10])
D = reshape(nddata,[5 3 2 5])
```

Removing Singleton Dimensions

MATLAB creates singleton dimensions if you explicitly specify them when you create or reshape an array, or if you perform a calculation that results in an array dimension of one.

```
B = repmat(5,[2 3 1 4]);

size(B)
ans =
     2     3     1     4
```

The squeeze function removes singleton dimensions from an array.

```
C = squeeze(B);

size(C)
ans =
     2     3     4
```

The squeeze function does not affect two-dimensional arrays; row vectors remain rows.

Permuting Array Dimensions

The permute function reorders the dimensions of an array.

```
B = permute(A,dims);
```

dims is a vector specifying the new order for the dimensions of A, where 1 corresponds to the first dimension (rows), 2 corresponds to the second dimension (columns), 3 corresponds to pages, and so on.

A	B= permute(A,[2 1 3])		C = permute(A,[3 2 1])	

A(:,:,1) =
 	
    1   2   3
    4   5   6
    7   8   9
 	
A(:,:,2) =
 	
    0   5   4
    2   7   6
    9   3   1
	B(:,:,1) =
	
	    1 4 7
	    2 5 8
	    3 6 9
	
	B(:,:,2) =
	
	    0 2 9
	    5 7 3
	    4 6 1
		Row and column subscripts are reversed (page-by-page transposition).	C(:,:,1) =
			
			    1 2 3
			    0 5 4
			
			C(:,:,2) =
			
			    4 5 6
			    2 7 6
			
			C(:,:,3) =
			
			    7 8 9
			    9 3 1
				Row and page subscripts are reversed.

For a more detailed look at the permute function, consider a four-dimensional array A of size 5-by-4-by-3-by-2. Rearrange the dimensions, placing the column dimension first, followed by the second page dimension, the first page dimension, then the row dimension. The result is a 4 by 2 by 3 by 5 array.

```
B = permute(A,[2 4 3 1])
```

Move dimension 2 of A to first subscript position of B, dimension 4 to second subscript position, and so on.

Input array A

Dimension	1	2	3	4
Size	5	4	3	2

Output array B

Dimension	1	2	3	4
Size	4	2	3	5

The order of dimensions in permute's argument list determines the size and shape of the output array. In this example, the second dimension moves to the first position. Because the second dimension of the original array had size four, the output array's first dimension also has size four.

You can think of permute's operation as an extension of the transpose function, which switches the row and column dimensions of a matrix. For permute, the order of the input dimension list determines the reordering of the subscripts. In the example above, element (4,2,1,2) of A becomes element (2,2,1,4) of B, element (5,4,3,2) of A becomes element (4,2,3,5) of B, and so on.

Inverse Permutation

The ipermute function is the inverse of permute. Given an input array A and a vector of dimensions v, ipermute produces an array B such that permute(B,v) returns A.

For example, these statements create an array E that is equal to the input array C.

```
D = ipermute(C,[1 4 2 3]);
E = permute(D,[1 4 2 3])
```

You can obtain the original array after permuting it by calling ipermute with the same vector of dimensions.

Computing with Multidimensional Arrays

Many of the MATLAB computational and mathematical functions accept multidimensional arrays as arguments. These functions operate on specific dimensions of multidimensional arrays; that is, they operate on individual elements, on vectors, or on matrices.

Operating on Vectors

Functions that operate on vectors, like sum, mean, and so on, by default typically work on the first nonsingleton dimension of a multidimensional array. Most of these functions optionally let you specify a particular dimension on which to operate. There are exceptions, however. For example, the cross function, which finds the cross product of two vectors, works on the first nonsingleton dimension having length three.

Note In many cases, these functions have other restrictions on the input arguments—for example, some functions that accept multiple arrays require that the arrays be the same size. Refer to the online help for details on function arguments.

Operating Element-by-Element

MATLAB functions that operate element-by-element on two-dimensional arrays, like the trigonometric and exponential functions in the elfun directory, work in exactly the same way for multidimensional cases. For example, the sin function returns an array the same size as the function's input argument. Each element of the output array is the sine of the corresponding element of the input array.

Similarly, the arithmetic, logical, and relational operators all work with corresponding elements of multidimensional arrays that are the same size in every dimension. If one operand is a scalar and one an array, the operator applies the scalar to each element of the array.

Operating on Planes and Matrices

Functions that operate on planes or matrices, such as the linear algebra and matrix functions in the matfun directory, do not accept multidimensional arrays as arguments. That is, you cannot use the functions in the matfun directory, or the array operators *, ^, \, or /, with multidimensional arguments. Supplying multidimensional arguments or operands in these cases results in an error.

You can use indexing to apply a matrix function or operator to matrices within a multidimensional array. For example, create a three-dimensional array A

```
A = cat(3,[1 2 3;9 8 7;4 6 5],[0 3 2;8 8 4;5 3 5],[6 4 7;6 8 5;...
5 4 3]);
```

Applying the eig function to the entire multidimensional array results in an error.

```
eig(A)
??? Error using ==> eig
Input arguments must be 2-D.
```

You can, however, apply eig to planes within the array. For example, use colon notation to index just one page (in this case, the second) of the array.

```
eig(A(:,:,2))
ans =
    2.6260
   12.9129
    2.7131
```

Note In the first case, subscripts are not colons, you must use squeeze to avoid an error. For example, eig(A(2,:,:)) results in an error because the size of the input is [1 3 3]. The expression eig(squeeze(A(2,:,:))), however, passes a valid two-dimensional matrix to eig.

Organizing Data in Multidimensional Arrays

You can use multidimensional arrays to represent data in two ways:

- As planes or pages of two-dimensional data. You can then treat these pages as matrices.
- As multivariate or multidimensional data. For example, you might have a four-dimensional array where each element corresponds to either a temperature or air pressure measurement taken at one of a set of equally spaced points in a room.

For example, consider an RGB image. For a single image, a multidimensional array is probably the easiest way to store and access data.

Array RGB

Page 3 —
blue
intensity

```
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
            0.421 0.398 ...
            0.912 0.713 ...
            0.219 0.328 ...
            0.128 0.133 ...
```

Page 2 —
green
intensity
values

```
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
        0.204 0.175 ...
        0.760 0.531 ...
        0.997 0.910 ...
        0.995 0.726 ...
```

Page 1—
red
intensity

```
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
                .
                .
                .
```

To access an entire plane of the image, use

```
red_plane = RGB(:,:,1);
```

To access a subimage, use

```
subimage = RGB(20:40,50:85,:);
```

The RGB image is a good example of data that needs to be accessed in planes for operations like display or filtering. In other instances, however, the data itself might be multidimensional. For example, consider a set of temperature measurements taken at equally spaced points in a room. Here the location of each value is an integral part of the data set – the physical placement in three-space of each element is an aspect of the information. Such data also lends itself to representation as a multidimensional array.

Array TEMP

```
67.9°  68.0°  67.9°
67.8°  67.8°  67.9°
67.9°  68.0°  68.0°  67.7°
67.7°  67.8°  67.7°
68.0°  68.0°  67.8°  67.5°
67.9°  67.8°  67.6°
67.8°  67.6°  67.6°
```

Now to find the average of all the measurements, use

```
mean(mean(mean(TEMP)));
```

To obtain a vector of the "middle" values (element (2,2)) in the room on each page, use

```
B = TEMP(2,2,:);
```

Multidimensional Cell Arrays

Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the cat function to build multidimensional cell arrays, just as you use it for numeric arrays.

For example, create a simple three-dimensional cell array C.

```
A{1,1} = [1 2;4 5];
A{1,2} = 'Name';
A{2,1} = 2 4i;
A{2,2} = 7;
B{1,1} = 'Name2';
B{1,2} = 3;
B{2,1} = 0:1:3;
B{2,2} = [4 5]';
C = cat(3,A,B);
```

The subscripts for the cells of C look like

Multidimensional Structure Arrays

Multidimensional structure arrays are extensions of rectangular structure arrays. Like other types of multidimensional arrays, you can build them using direct assignment or the cat function.

```
patient(1,1,1).name = 'John Doe';patient(1,1,1).billing = 127.00;
patient(1,1,1).test = [79 75 73; 180 178 177.5; 220 210 205];
patient(1,2,1).name = 'Ann Lane';patient(1,2,1).billing = 28.50;
patient(1,2,1).test = [68 70 68; 118 118 119; 172 170 169];
patient(1,1,2).name = 'Al Smith';patient(1,1,2).billing = 504.70;
patient(1,1,2).test = [80 80 80; 153 153 154; 181 190 182];
patient(1,2,2).name = 'Dora Jones';patient(1,2,2).billing =
1173.90;
patient(1,2,2).test = [73 73 75; 103 103 102; 201 198 200];
```

Applying Functions to Multidimensional Structure Arrays

To apply functions to multidimensional structure arrays, operate on fields and field elements using indexing. For example, find the sum of the columns of the test array in patient(1,1,2).

```
sum((patient(1,1,2).test));
```

Similarly, add all the billing fields in the patient array.

```
total = sum([patient.billing]);
```

19

Structures and Cell Arrays

Structures are collections of different kinds of data organized by named fields. Cell arrays are a special class of MATLAB array whose elements consist of cells that themselves contain MATLAB arrays. Both structures and cell arrays provide a hierarchical storage mechanism for dissimilar kinds of data. They differ from each other primarily in the way they organize data. You access data in structures using named fields, while in cell arrays, data is accessed through matrix indexing operations.

Function Summary

This table describes the MATLAB functions for working with structures and cell arrays..

Category	Function	Description
Structure functions	`deal`	Deal inputs to outputs
	`fieldnames`	Get structure field names.
	`isfield`	True if field is in structure array.
	`isstruct`	True for structures.
	`rmfield`	Remove structure field.
	`struct`	Create or convert to structure array.
	`struct2cell`	Convert structure array into cell array.
Cell array functions	`cell`	Create cell array.
	`cell2struct`	Convert cell array into structure array.
	`celldisp`	Display cell array contents.
	`cellfun`	Apply a cell function to a cell array.
	`cellplot`	Display graphical depiction of cell array.
	`deal`	Deal inputs to outputs.
	`iscell`	True for cell array.
	`num2cell`	Convert numeric array into cell array.

Category	Function	Description
Cell array of strings functions	deblank	Remove trailing blanks from a string
	intersect	Set the intersection of two vectors
	ismember	Detect members of a set
	setdiff	Return the set difference of two vectors
	setxor	Set the exclusive-or of two vectors
	sort	Sort elements in ascending order
	strcat	Concatenate strings
	strcmp	Compare strings
	strmatch	Find possible matches for a string
	union	Set the union of two vectors
	unique	Set the unique elements of a vector

Structures

Structures are MATLAB arrays with named "data containers" called *fields*. The fields of a structure can contain any kind of data. For example, one field might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on.

patient

```
├─.name ──────────────── 'John Doe'
├─.billing ──────────── 127.00
└─.test ─────────────
```

79	75	73
180	178	177.5
220	210	205

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional structure arrays. For examples of higher-dimension structure arrays, see Chapter 18, "Multidimensional Arrays"

The following list summarizes the contents of this section:

- "Building Structure Arrays"
- "Accessing Data in Structure Arrays"
- "Finding the size of Structure Arrays"
- "Adding Fields to Structures"
- "Deleting Fields from Structures"
- "Applying Functions and Operators"
- "Writing Functions to Operate on Structures"

- "Accessing Data Using Dynamic Field Names"
- "Organizing Data in Structure Arrays"
- "Nesting Structures"

Building Structure Arrays

You can build structures in two ways:

- Using assignment statements
- Using the `struct` function

Building Structure Arrays Using Assignment Statements

You can build a simple 1-by-1 structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the 1-by-1 `patient` structure array shown at the beginning of this section.

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering

```
patient
```

at the command line results in

```
name: 'John Doe'
billing: 127
test: [3x3 double]
```

`patient` is an array containing a structure with three fields. To expand the structure array, add subscripts after the structure name.

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
```

The `patient` structure array now has size [1 2]. Note that once a structure array contains more than a single element, MATLAB does not display individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains.

19-5

```
patient
patient =
1x2 struct array with fields:
    name
    billing
    test
```

You can also use the `fieldnames` function to obtain this information. `fieldnames` returns a cell array of strings containing field names.

As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that:

• All structures in the array have the same number of fields.

• All fields have the same field names.

For example, entering `patient(3).name = 'Alan Johnson'` expands the `patient` array to size `[1 3]`. Now both `patient(3).billing` and `patient(3).test` contain empty matrices.

Note Field sizes do not have to conform for every element in an array. In the `patient` example, the `name` fields can have different lengths, the `test` fields can be arrays of different sizes, and so on.

Building Structure Arrays Using the struct Function

You can preallocate an array of structures with the `struct` function. Its basic form is

```
str_array = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their corresponding values. A field value can be a single value, represented by any MATLAB data construct, or a cell array of values. All field values in the argument list must be of the same scale (single value or cell array).

You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an example, consider the allocation of a 1-by-3 structure array, `weather`, with the

structure fields temp and rainfall. Three different methods for allocating such an array are shown in this table.

Method	Syntax	Initialization
struct	weather(3) = struct('temp',72,'rainfall',0.0);	weather(3) is initialized with the field values shown. The fields for the other structures in the array, weather(1) and weather(2), are initialized to the empty matrix.
struct with repmat	weather = repmat(struct('temp',72, 'rainfall',0.0),1,3);	All structures in the weather array are initialized using one set of field values.
struct with cell array syntax	weather = struct('temp',{68,80,72}, 'rainfall',{0.2,0.4,0.0});	The structures in the weather array are initialized with distinct field values specified with cell arrays.

Accessing Data in Structure Arrays

Using structure array indexing, you can access the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. For the examples in this section, consider this structure array.

You can access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-2 structure array.

```
mypatients = patient(1:2)
1x2 struct array with fields:
    name
    billing
    test
```

The first structure in the `mypatients` array is the same as the first structure in the `patient` array.

```
mypatients(1)
ans =
        name: 'John Doe'
     billing: 127
        test: [3x3 double]
```

To access a field of a particular structure, include a period (.) after the structure name followed by the field name.

```
str = patient(2).name
str =
    Ann Lane
```

To access elements within fields, append the appropriate indexing mechanism to the field name. That is, if the field contains an array, use array subscripting; if the field contains a cell array, use cell array subscripting, and so on.

```
test2b = patient(3).test(2,2)
test2b =
    153
```

Use the same notations to assign values to structure fields, for example,

```
patient(3).test(2,2) = 7;
```

You can extract field values for multiple structures at a time. For example, the line below creates a 1-by-3 vector containing all of the `billing` fields.

```
bills = [patient.billing]
bills =
  127.0000   28.5000  504.7000
```

Similarly, you can create a cell array containing the `test` data for the first two structures.

```
tests = {patient(1:2).test}
tests =
    [3x3 double]    [3x3 double]
```

Accessing Data Using Dynamic Field Names

All of the structures discussed up to this point have elements that can only be referenced using fixed field names. Another means of accessing structures is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes *expression* a dynamic field name.

```
struct_name.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate *expression* into a field name and obtain the values of that field at columns 1 through 25 of row 7, use

```
struct_name.(expression)(7,1:25)
```

Example. The function `avgscore` computes an average test score, retrieving information from the `testscores` structure using dynamic field names.

```
function avg = avgscore(testscores, student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field, `student`.

```
avgscore(testscores, 'Ann_Lane', 1, 20)
ans =
   83.5000

avgscore(testscores, 'William_King', 1, 20)
ans =
   92.1000
```

Finding the size of Structure Arrays

Use the `size` function to obtain the size of a structure array, or of any structure field. Given a structure array name as an argument, `size` returns a vector of array dimensions. Given an argument in the form `array(n).field`, the `size` function returns a vector containing the size of the field contents.

For example, for the 1-by-3 structure array `patient`, `size(patient)` returns the vector `[1 3]`. The statement `size(patient(1,2).name)` returns the length of the `name` string for element `(1,2)` of `patient`.

Adding Fields to Structures

You can add a field to every structure in an array by adding the field to a single structure. For example, to add a social security number field to the `patient` array, use an assignment like

```
patient(2).ssn = '000 00 0000';
```

Now `patient(2).ssn` has the assigned value. Every other structure in the array also has the `ssn` field, but these fields contain the empty matrix until you explicitly assign a value to them.

Deleting Fields from Structures

You can remove a given field from every structure within a structure array using the `rmfield` function. Its most basic form is

```
struc2 = rmfield(array, 'field')
```

where `array` is a structure array and `'field'` is the name of a field to remove from it. To remove the `name` field from the `patient` array, for example, enter:

```
patient = rmfield(patient,'name');
```

Applying Functions and Operators

Operate on fields and field elements the same way you operate on any other MATLAB array. Use indexing to access the data on which to operate. For example, this statement finds the mean across the rows of the `test` array in `patient(2)`.

```
mean((patient(2).test)');
```

There are sometimes multiple ways to apply functions or operators across fields in a structure array. One way to add all the `billing` fields in the `patient` array is

```
total = 0;
for k = 1:length(patient)
        total = total + patient(k).billing;
end
```

To simplify operations like this, MATLAB enables you to operate on all like-named fields in a structure array. Simply enclose the `array.field` expression in square brackets within the function call. For example, you can sum all the `billing` fields in the `patient` array using

```
total = sum ([patient.billing]);
```

This is equivalent to using the comma-separated list.

```
total = sum ([patient(1).billing, patient(2).billing...]);
```

This syntax is most useful in cases where the operand field is a scalar field.

Writing Functions to Operate on Structures

You can write functions that work on structures with specific field architectures. Such functions can access structure fields and elements for processing.

Note When writing M-file functions to operate on structures, you must perform your own error checking. That is, you must ensure that the code checks for the expected fields.

As an example, consider a collection of data that describes measurements, at different times, of the levels of various toxins in a water source. The data consists of fifteen separate observations, where each observation contains three separate measurements.

You can organize this data into an array of 15 structures, where each structure has three fields, one for each of the three measurements taken.

The function concen, shown below, operates on an array of structures with specific characteristics. Its arguments must contain the fields lead, mercury, and chromium.

```
function [r1,r2] = concen(toxtest);
% Create two vectors. r1 contains the ratio of mercury to lead
% at each observation. r2 contains the ratio of lead to chromium.
r1 = [toxtest.mercury]./[toxtest.lead];
r2 = [toxtest.lead]./[toxtest.chromium];
% Plot the concentrations of lead, mercury, and chromium
% on the same plot, using different colors for each.
lead = [toxtest.lead];
mercury = [toxtest.mercury];
chromium = [toxtest.chromium];
plot(lead,'r'); hold on
plot(mercury,'b')
plot(chromium,'y'); hold off
```

Try this function with a sample structure array like test.

```
test(1).lead = .007; test(2).lead = .031; test(3).lead = .019;
test(1).mercury = .0021; test(2).mercury = .0009;
test(3).mercury = .0013;
test(1).chromium = .025; test(2).chromium = .017;
test(3).chromium = .10;
```

Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; RED, GREEN, and BLUE.

Blue intensity values

```
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
          0.421 0.398 ...
          0.912 0.713 ...
          0.219 0.328 ...
          0.128 0.133 ...
```

Green intensity values

```
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
          0.204 0.175 ...
          0.760 0.531 ...
          0.997 0.910 ...
          0.995 0.726 ...
```

Red intensity values

```
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
              .
              .
              .
```

There are at least two ways you can organize such data into a structure array.

Plane organization   Element-by-element organization

1-by-1 structure array where each field is a 128-by-128 array

128-by-128 structure array where each field is a single data element

Plane Organization

In case 1 above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
red_plane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as A(2), A(3), and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately.

```
red_sub = A.r(2:12,13:30);
grn_sub = A.g(2:12,13:30);
blue_sub = A.b(2:12,13:30);
```

Element-by-Element Organization

Case 2 has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement.

```
Bsub = B(1:10,1:10);
```

To access an entire plane of the image using the element-by-element method, however, requires a loop.

```
red_plane = zeros(128,128);
for k = 1:(128*128)
    red_plane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications; however, it can be the best for other applications wherein you will routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example - A Simple Database

Consider organizing a simple database.


```
A.name = strvcat('Ann Jones','Dan Smith',...
A.address = strvcat('80 Park St.','5 Lake
Ave.',...);
A.amount = [12.5;81.29;30; ...];
```

```
B(1).name = 'Ann Jones';
B(1).address = '80 Park St.';
B(1).amount = 12.5;
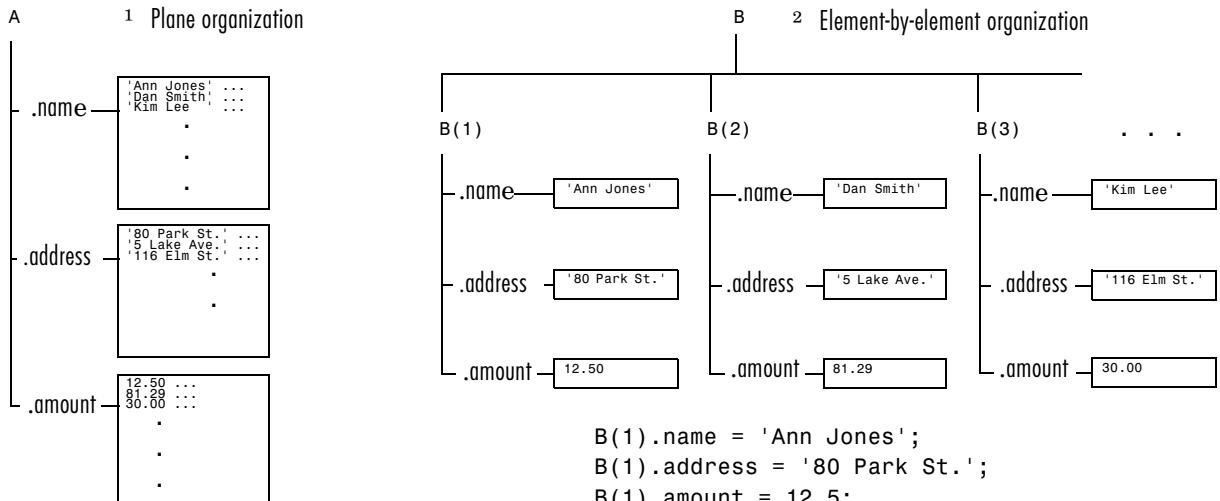
B(2).name = 'Dan Smith';
B(2).address = '5 Lake Ave.';
B(2).amount = 81.29;
    .
    .
```

Each of the possible organizations has advantages depending on how you want to access the data:

- Plane organization makes it easier to operate on all field values at once. For example, to find the average of all the values in the amount field,
  - Using plane organization

    ```
    avg = mean(A.amount);
    ```
  - Using element-by-element organization

    ```
    avg = mean([B.amount]);
    ```

- Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, `client.m`, which displays the name and address of a given client on screen.

  Using plane organization, pass individual fields.

  ```
  function client(name,address)
  disp(name)
  disp(address)
  ```

  To call the `client` function,

  ```
  client(A.name(2,:),A.address(2,:))
  ```

  Using element-by-element organization, pass an entire structure.

  ```
  function client(B)
  disp(B)
  ```

  To call the `client` function,

  ```
  client(B(2))
  ```

- Element-by-element organization makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you may need to frequently recreate the `name` or `address` field to accommodate longer strings.

Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Nesting Structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the `struct` function or direct assignment statements to nest structures within existing structure fields.

Building Nested Structures with the struct Function

To build nested structures, you can nest calls to the `struct` function. For example, create a 1-by-1 structure array.

```
A = struct('data',[3 4 7; 8 0 1],'nest',...
    struct('testnum','Test 1', 'xdata',[4 2 8],...
    'ydata',[7 1 6]));
```

You can build nested structure arrays using direct assignment statements. These statements add a second element to the array.

```
A(2).data = [9 3 2; 7 6 5];
A(2).nest.testnum = 'Test 2';
A(2).nest.xdata = [3 4 2];
A(2).nest.ydata = [5 0 9];
```


Indexing Nested Structures

To index nested structures, append nested field names using dot notation. The first text string in the indexing expression identifies the structure array, and subsequent expressions access field names that contain other structures.

For example, the array A created earlier has three levels of nesting:

- To access the nested structure inside A(1), use A(1).nest.
- To access the xdata field in the nested structure in A(2), use A(2).nest.xdata.
- To access element 2 of the ydata field in A(1), use A(1).nest.ydata(2).

Cell Arrays

A cell array is a MATLAB array for which the elements are *cells*, containers that can hold other MATLAB arrays. For example, one cell of a cell array might contain a real matrix, another an array of text strings, and another a vector of complex values.

cell 1,1	cell 1,2	cell 1,3
3　4　2 9　7　6 8　5　1	'Anne Smith' '9/12/94　　' 'Class II　' 'Obs. 1　　' 'Obs. 2　　'	.25+3i　8 16i 34+5i 7+.92i
cell 2,1	**cell 2,2**	**cell 2,3**
[1.43 2.98 5.67]	7　2　14 8　3　45 52　16　3	'text'　　4　2 　　　　　1　5 [4 2 7]　.02 + 8i

You can build cell arrays of any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see Chapter 18, "Multidimensional Arrays".

The following list summarizes the contents of this section:

- "Creating Cell Arrays"
- "Obtaining Data from Cell Arrays"
- "Deleting Cells"

Creating Cell Arrays

You can create cell arrays by:

- Using assignment statements
- Preallocating the array using the `cell` function, then assigning data to cells

Using Assignment Statements

You can build a cell array by assigning data to individual cells, one cell at a time. MATLAB automatically builds the array as you go along. There are two ways to assign data to cells:

- Cell indexing

  Enclose the cell subscripts in parentheses using standard array notation. Enclose the cell contents on the right side of the assignment statement in curly braces, "{}." For example, create a 2-by-2 cell array A.

  ```
  A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
  A(1,2) = {'Anne Smith'};
  A(2,1) = {3+7i};
  A(2,2) = { pi:pi/10:pi};
  ```

Note The notation "{}" denotes the empty cell array, just as "[ ]" denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

- Content indexing

  Enclose the cell subscripts in curly braces using standard array notation.
  Specify the cell contents on the right side of the assignment statement.

  ```
  A{1,1} = [1 4 3; 0 5 8; 7 2 9];
  A{1,2} = 'Anne Smith';
  A{2,1} = 3+7i;
  A{2,2} =  pi:pi/10:pi;
  ```

The various examples in this guide do not use one syntax throughout, but
attempt to show representative usage of cell and content addressing. You can
use the two forms interchangeably.

Note If you already have a numeric array of a given name, don't try to create
a cell array of the same name by assignment without first clearing the
numeric array. If you do not clear the numeric array, MATLAB assumes that
you are trying to "mix" cell and numeric syntaxes, and generates an error.
Similarly, MATLAB does not clear a cell array when you make a single
assignment to it. If any of the examples in this section give unexpected
results, clear the cell array from the workspace and try again.

MATLAB displays the cell array A in a condensed form.

```
A =
         [3x3 double]    'Anne Smith'
    [3.0000+ 7.0000i]    [1x21 double]
```

To display the full cell contents, use the `celldisp` function. For a high-level
graphical display of cell architecture, use `cellplot`.

If you assign data to a cell that is outside the dimensions of the current array,
MATLAB automatically expands the array to include the subscripts you
specify. It fills any intervening cells with empty matrices. For example, the
assignment below turns the 2-by-2 cell array A into a 3-by-3 cell array.

```
A(3,3) = {5};
```

cell 1,1	cell 1,2	cell 1,3
1 4 3 0 5 8 7 2 9	'Anne Smith'	[ ]
cell 2,1	cell 2,2	cell 2,3
3+7i	[3.14...3.14]	[ ]
cell 3,1	cell 3,2	cell 3,3
[ ]	[ ]	5

Cell Array Syntax: Using Braces

The curly braces, "{}", are cell array constructors, just as square brackets are numeric array constructors. Curly braces behave similarly to square brackets, except that you can nest curly braces to denote nesting of cells (see "Nesting Cell Arrays" for details).

Curly braces use commas or spaces to indicate column breaks and semicolons to indicate row breaks between cells. For example,

```
C = {[1 2], [3 4]; [5 6], [7 8]};
```

results in

cell 1,1	cell 1,2
[1 2]	[3 4]
cell 2,1	cell 2,2
[5 6]	[7 8]

Use square brackets to concatenate cell arrays, just as you do for numeric arrays.

Preallocating Cell Arrays with the cell Function

The `cell` function allows you to preallocate empty cell arrays of the specified size. For example, this statement creates an empty 2-by-3 cell array.

```
B = cell(2,3);
```

Use assignment statements to fill the cells of B.

```
B(1,3) = {1:3};
```

Obtaining Data from Cell Arrays

You can obtain data from cell arrays and store the result as either a standard array or a new cell array. This section discusses:

- Accessing cell contents using content indexing
- Accessing a subset of cells using cell indexing

Accessing Cell Contents Using Content Indexing

You can use content indexing on the right side of an assignment to access some or all of the data in a single cell. Specify the variable to receive the cell contents on the left side of the assignment. Enclose the cell index expression on the right side of the assignment in curly braces. This indicates that you are assigning cell contents, not the cells themselves.

Consider the 2-by-2 cell array N.

```
N{1,1} = [1 2; 4 5];
N{1,2} = 'Name';
N{2,1} = 2 4i;
N{2,2} = 7;
```

You can obtain the string in N{1,2} using

```
c = N{1,2}
c =
   Name
```

> **Note** In assignments, you can use content indexing to access only a single cell, not a subset of cells. For example, the statements A{1,:} = *value* and B = A{1,:} are both invalid. However, you can use a subset of cells any place you would normally use a comma-separated list of variables (for example, as function inputs or when building an array). See the "Replacing Lists of Variables with Cell Arrays" section for details.

To obtain subsets of a cell's contents, concatenate indexing expressions. For example, to obtain element (2,2) of the array in cell N{1,1}, use:

```
d = N{1,1}(2,2)
d =
     5
```

Accessing a Subset of Cells Using Cell Indexing

Use cell indexing to assign any set of cells to another variable, creating a new cell array. Use the colon operator to access subsets of cells within a cell array.

Deleting Cells

You can delete an entire dimension of cells using a single statement. Like standard array deletion, use vector subscripting when deleting a row or column of cells and assign the empty matrix to the dimension.

```
A(cell_subscripts) = []
```

When deleting cells, curly braces do not appear in the assignment statement at all.

Reshaping Cell Arrays

Like other arrays, you can reshape cell arrays using the reshape function. The number of cells must remain the same after reshaping; you cannot use reshape to add or remove cells.

```
A = cell(3,4);

size(A)
ans =
    3    4

B = reshape(A,6,2);

size(B)
ans =
    6    2
```

Replacing Lists of Variables with Cell Arrays

Cell arrays can replace comma-separated lists of MATLAB variables in:

- Function input lists
- Function output lists
- Display operations
- Array constructions (square brackets and curly braces)

If you use the colon to index multiple cells in conjunction with the curly brace notation, MATLAB treats the contents of each cell as a separate variable. For example, assume you have a cell array T where each cell contains a separate vector. The expression T{1:5} is equivalent to a comma-separated list of the vectors in the first five cells of T.

Consider the cell array C.

```
C(1) = {[1 2 3]};
C(2) = {[1 0 1]};
C(3) = {1:10};
C(4) = {[9 8 7]};
C(5) = {3};
```

To convolve the vectors in `C(1)` and `C(2)` using `conv`,

```
d = conv(C{1:2})
d =
    1    2    4    2    3
```

Display vectors two, three, and four with

```
C{2:4}
ans =
    1    0    1

ans =
    1    2    3    4    5    6    7    8    9    10

ans =
    9    8    7
```

Similarly, you can create a new numeric array using the statement

```
B = [C{1}; C{2}; C{4}]
B =
    1    2    3
    1    0    1
    9    8    7
```

You can also use content indexing on the left side of an assignment to create a new cell array where each cell represents a separate output argument.

```
[D{1:2}] = eig(B)
D =
    [3x3 double]    [3x3 double]
```

You can display the actual eigenvalues and eigenvectors using `D{1}` and `D{2}`.

Note The `varargin` and `varargout` arguments allow you to specify variable numbers of input and output arguments for MATLAB functions that you create. Both `varargin` and `varargout` are cell arrays, allowing them to hold various sizes and kinds of MATLAB data. See "Passing Variable Numbers of Arguments" on page 16-16 for details.

Applying Functions and Operators

Use indexing to apply functions and operators to the contents of cells. For example, use content indexing to call a function with the contents of a single cell as an argument.

```
A{1,1} = [1 2; 3 4];
A{1,2} = randn(3,3);
A{1,3} = 1:5;

B = sum(A{1,1})
B =
     4     6
```

To apply a function to several cells of a non-nested cell array, use a loop.

```
for k = 1:length(A)
    M{k} = sum(A{1,k});
end
```

Organizing Data in Cell Arrays

Cell arrays are useful for organizing data that consists of different sizes or kinds of data. Cell arrays are better than structures for applications where:

- You need to access multiple fields of data with one statement.
- You want to access subsets of the data as comma-separated variable lists.
- You don't have a fixed set of field names.
- You routinely remove fields from the structure.

As an example of accessing multiple fields with one statement, assume that your data consists of:

- A 3-by-4 array consisting of measurements taken for an experiment.
- A 15-character string containing a technician's name.
- A 3-by-4-by-5 array containing a record of measurements taken for the past five experiments.

For many applications, the best data construct for this data is a structure. However, if you routinely access only the first two fields of information, then a cell array might be more convenient for indexing purposes.

This example shows how to access the first and second elements of the cell array TEST.

```
[newdata,name] = deal(TEST{1:2})
```

This example shows how to access the first and second elements of the structure TEST.

```
newdata = TEST.measure
name = TEST.name
```

The varargin and varargout arguments are examples of the utility of cell arrays as substitutes for comma-separated lists. Create a 3-by-3 numeric array A.

```
A = [0 1 2;4 0 7;3 1 2];
```

Now apply the normest (2-norm estimate) function to A, and assign the function output to individual cells of B.

```
[B{1:2}] = normest(A)
B =
    [8.8826]    [4]
```

All of the output values from the function are stored in separate cells of B. B(1) contains the norm estimate; B(2) contains the iteration count.

Nesting Cell Arrays

A cell can contain another cell array, or even an array of cell arrays. (Cells that contain noncell data are called *leaf cells*.) You can use nested curly braces, the cell function, or direct assignment statements to create nested cell arrays. You can then access and manipulate individual cells, subarrays of cells, or cell elements.

Building Nested Arrays with Nested Curly Braces

You can nest pairs of curly braces to create a nested cell array. For example,

```
clear A
A(1,1) = {magic(5)};

A(1,2) = {{[5 2 8; 7 3 0; 6 7 3] 'Test 1'; [2 4i 5+7i] {17 []}}}
A =
    [5x5 double]    {2x2 cell}
```

Note that the right side of the assignment is enclosed in two sets of curly braces. The first set represents cell (1,2) of cell array A. The second "packages" the 2-by-2 cell array inside the outer cell.

Building Nested Arrays with the cell Function

To nest cell arrays with the cell function, assign the output of cell to an existing cell:

1 Create an empty 1-by-2 cell array.

```
A = cell(1,2);
```

2 Create a 2-by-2 cell array inside A(1,2).

```
A(1,2) = {cell(2,2)};
```

3 Fill A, including the nested array, using assignments.

```
A(1,1) = {magic(5)};
A{1,2}(1,1) = {[5 2 8; 7 3 0; 6 7 3]};
A{1,2}(1,2) = {'Test 1'};
A{1,2}(2,1) = {[2 4i 5+7i]};
A{1,2}(2,2) = {cell(1,2)}
A{1,2}{2,2}(1) = {17};
```

Note the use of curly braces until the final level of nested subscripts. This is required because you need to access cell contents to access cells within cells.

You can also build nested cell arrays with direct assignments using the statements shown in step 3 above.

Indexing Nested Cell Arrays

To index nested cells, concatenate indexing expressions. The first set of subscripts accesses the top layer of cells, and subsequent sets of parentheses access successively deeper layers.

For example, array A has three levels of nesting:

```
┌─────────────────────────────┬─────────────────────────────────┐
│ cell 1,1                     │ cell 1,2                         │
│                             │                                 │
│   ┌─────────────────┐       │  ┌──────────┐                   │
│   │17 24  1  8 15    │       │  │5  2  8   │                   │
│   │23  5  7 14 16    │       │  │7  3  0   │    'Test 1'       │
│   │ 4  6 13 20 22    │       │  │6  7  3   │                   │
│   │10 12 19 21  3    │       │  └──────────┘                   │
│   │11 18 25  2  9    │       │                                 │
│   └─────────────────┘       │  ┌───────────┐  ┌──────────┐    │
│                             │  │[2 4i 5+7i]│  │ 17 │      │    │
│                             │  └───────────┘  └──────────┘    │
└─────────────────────────────┴─────────────────────────────────┘
```

- To access the 5-by-5 array in cell (1,1), use A{1,1}.

- To access the 3-by-3 array in position (1,1) of cell (1,2), use A{1,2}{1,1}.

- To access the 2-by-2 cell array in cell (1,2), use A{1,2}.

- To access the empty cell in position (2,2) of cell (1,2), use A{1,2}{2,2}{1,2}.

Converting Between Cell and Numeric Arrays

Use for loops to convert between cell and numeric formats. For example, create a cell array F.

```
F{1,1} = [1 2; 3 4];
F{1,2} = [ 1 0; 0 1];
F{2,1} = [7 8; 4 1];
F{2,2} = [4i 3+2i; 1 8i 5];
```

Now use three `for` loops to copy the contents of `F` into a numeric array `NUM`.

```
for k = 1:4
   for m = 1:2
      for n = 1:2
         NUM(m,n,k) = F{k}(m,n);
      end
   end
end
```

Similarly, you must use `for` loops to assign each value of a numeric array to a single cell of a cell array.

```
G = cell(1,16);
for m = 1:16
   G{m} = NUM(m);
end
```

Cell Arrays of Structures

Use cell arrays to store groups of structures with different field architectures.

```
c_str = cell(1,2);
c_str{1}.label = '12/2/94   12/5/94';
c_str{1}.obs = [47 52 55 48; 17 22 35 11];
c_str{2}.xdata = [ 0.03 0.41 1.98 2.12 17.11];
c_str{2}.ydata = [ 3 5 18 0 9];
c_str{2}.zdata = [0.6 0.8 1 2.2 3.4];
```


Cell 1 of the `c_str` array contains a structure with two fields, one a string and the other a vector. Cell 2 contains a structure with three vector fields.

When building cell arrays of structures, you must use content indexing. Similarly, you must use content indexing to obtain the contents of structures within cells. The syntax for content indexing is:

```
cell_array{index}.field
```

For example, to access the `label` field of the structure in cell 1, use `c_str{1}.label`.

20

Function Handles

This chapter covers the following topics on function handles:

Overview

A *function handle* is a MATLAB data type that contains information used in referencing a function. When you create a function handle, MATLAB stores in the handle all the information about the function that it needs to execute, or *evaluate*, it later on. Typically, a function handle is passed in an argument list to other functions. It is then used in conjunction with `feval` to evaluate the function to which the handle belongs.

A MATLAB function handle is more than just a reference to a function. It often represents a collection of function methods, overloaded to handle different argument types. When you create a handle to a function, MATLAB takes a snapshot of all built-in and M-file methods of that name that are on the MATLAB path and in scope at that time, and stores access information for all of those methods in the handle.

When you evaluate a function handle, MATLAB considers only those functions that were stored within the handle when it was created. Other functions that might now be on the path or in scope are not considered. It is the combination of which function methods are mapped to by the handle and what arguments the handle is evaluated with that determines which is the actual function that MATLAB dispatches to.

Benefits of Using Function Handles

Function handles enable you to do all of the following:

- Pass function access information to other functions
- Capture all methods of an overloaded function
- Allow wider access to subfunctions and private functions
- Ensure reliability when evaluating functions
- Reduce the number of files that define your functions
- Improve performance in repeated operations
- Manipulate handles in arrays, structures, and cell arrays

This section also includes an example of using a simple function handle. See "A Simple Function Handle" on page 20-5.

Pass Function Access Information to Other Functions

You can pass a function handle as an argument in a call to another function. The handle contains access information that enables the receiving function to call the function for which the handle was constructed.

You can evaluate a function handle from within another function even if the handle's function is not in the scope of the evaluating function. This is because the function performing the evaluation has all the information it needs within the function handle.

For the same reason, you can also evaluate a function handle even when the handle's function is no longer on the MATLAB search path.

You must use the MATLAB `feval` command to evaluate the function in a function handle. When you pass a function handle as an argument into another function, then the function receiving the handle uses `feval` to evaluate the function handle.

Capture All Methods of An Overloaded Function

Because many MATLAB functions are overloaded, a function handle often maps to a number of code sources (e.g., built-in code, M-files), that implement the function. A function handle stores the access to all of the overloaded sources, or *methods*, that are on the MATLAB path at the time the handle is created.

When you evaluate an overloaded function handle, MATLAB follows the usual rules of selecting which method to evaluate, basing the selection on the argument types passed in the function call. See "How MATLAB Determines Which Method to Call" on page 21-66, for more information on how MATLAB selects overloaded functions.

For example, there are three built-in functions and one M-file function that define the abs function on the standard MATLAB path. A function handle created for the abs function contains access information on all four of these function sources. If you evaluate the function with an argument of the double type, then the built-in function that takes a double argument is executed.

Allow Wider Access to Subfunctions and Private Functions

By definition, all MATLAB functions have a certain scope. They are visible to other MATLAB entities within that scope, but not visible outside of it. You can invoke a function directly from another function that is within its scope, but not from a function outside that scope.

Subfunctions and private functions are, by design, limited in their visibility to other MATLAB functions. You can invoke a subfunction only by another function that is defined within the same M-file. You can invoke a private function only from a function in the directory immediately above the \private subdirectory.

When you create a handle to a function that has limited scope, the function handle stores all the information MATLAB needs to evaluate the function from any location in the MATLAB environment. If you create a handle to a subfunction while the subfunction is in scope, (that is, you create it from within the M-file that defines the subfunction), you can then pass the handle to code that resides outside of that M-file and evaluate the subfunction from beyond its usual scope. The similar case holds true for private functions.

Ensure Reliability When Evaluating Functions

Function handles allow you more control over what methods get executed when a function is evaluated. If you create a function handle for a function with overloaded methods, making sure that only the intended methods are within scope when the handle is created gives you control over which methods are executed when MATLAB evaluates the handle. This can isolate you from methods that might be in scope at the time of evaluation that you didn't know about.

Reduce the Number of Files That Define Your Functions

You can use function handles to help reduce the number of M-files required to define your functions. The problem with grouping a number of functions in one M-file is that this defines them as subfunctions, and thus reduces their scope in MATLAB. Using function handles to access these subfunctions removes this limitation. This enables you to group functions as you want and reduce the number of files you have to manage.

Improve Performance in Repeated Operations

MATLAB performs a lookup on a function at the time you create a function handle and then stores this access information in the handle itself. Once defined, you can use this handle in repeated evaluations without incurring the performance delay associated with function lookup each time.

Manipulate Handles in Arrays, Structures, and Cell Arrays

As a standard MATLAB data type, a function handle can be manipulated and operated on in the same manner as other MATLAB data types. You can create arrays, structures, or cell arrays of function handles. Access individual function handles within these data structures in the same way that you access elements of a numeric array or structure.

Create n-dimensional arrays of handles using either of the concatenation methods used to form other types of MATLAB arrays, `[ ]` or `cat`. All operations involving matrix manipulation are supported for function handles.

A Simple Function Handle

The `repmat` function is an elementary matrix function, which is defined in MATLAB with the single M-file, `repmat.m`. If you create a function handle to the `repmat` function, MATLAB stores in the handle the information it will need later to evaluate the function. Included in this information is the `repmat.m` source file path, `toolbox\matlab\elmat\repmat.m`.

Once you create a function handle, it is not affected by certain changes you might make in your MATLAB environment. For example, if you construct a handle to the `repmat` function and, later, write additional `repmat.m` methods to overload the function, the handle still *sees* only the original function. Also, if you remove the path to `repmat.m` from the search path, MATLAB is still able to locate and evaluate the function using the handle that you created prior to the path change.

Since repmat is not an overloaded function in this case, evaluation of the function through its handle is fairly simple. You call feval on the function handle, also passing any arguments the function should act upon. MATLAB executes the one function whose access information is stored in the handle.

Constructing a Function Handle

Construct a function handle in MATLAB using the *at* sign, @, before the function name. The following example creates a function handle for the humps function and assigns it to the variable, fhandle.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to fminbnd, which then minimizes over the interval [0.3, 1].

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The fminbnd function evaluates the @humps function handle using feval. A small portion of the fminbnd M-file is shown below. In line 1, the funfcn input parameter receives the function handle, @humps, that was passed in. The feval statement, in line 113, evaluates the handle.

```
1     function [xf,fval,exitflag,output] = ...
         fminbnd(funfcn,ax,bx,options,varargin)
             .
             .
             .
113  fx = feval(funfcn,x,varargin{:});
```

Note When creating a function handle, you may only use the function *name* after the @ sign. This must not include any path information. The following syntax is invalid: fhandle = @\home\user4\humps.

Maximum Length of a Function Name

Function names used in handles are unique up to `N` characters, where `N` is the number returned by the function `namelengthmax`. If the function name exceeds that length, MATLAB truncates the latter part of the name.

```
N = namelengthmax
N =
    63
```

For function handles created for Java constructors, the length of any segment of the package name or class name must not exceed `namelengthmax` characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, there are three segments in java.lang.String.) There is no limit to the overall length of the string specifying the package and class.

Evaluating a Function Through Its Handle

Execute the target function of a function handle using the MATLAB `feval` command. The syntax for using this command with a function handle is

```
feval(fhandle, arg1, arg2, ..., argn)
```

This acts similarly to a direct call to the function represented by `fhandle`, passing arguments `arg1` through `argn`. The principal differences are:

- A function handle can be evaluated from within any function that you pass it to.
- The code source that MATLAB selects for evaluation depends upon which overloaded methods of the function were on the MATLAB path and in scope at the time the handle was constructed. (Argument types also affect method selection.) Path and scope are not considered at the time of evaluation.
- MATLAB does the work of initial function lookup at the time the function handle is constructed. This does not need to be done each time MATLAB evaluates the handle.

Note The `feval` command does not operate on nonscalar function handles. Passing a nonscalar function handle to `feval` results in an error.

Function Evaluation and Overloading

To understand the relationship between function handles and overloading, it is helpful to review, briefly, the nature of MATLAB function calls. Because of overloading, it is useful to think of a single MATLAB function as comprising a number of code sources (for example, built-in code, M-files). When you call a MATLAB function without `feval`, the choice of which source is called depends upon two factors:

- The methods that are visible on the path at the time of the call
- The classes of the arguments to the function

MATLAB evaluates function handles in a similar manner. In most cases, a function handle represents a collection of methods that overload the function.

When you evaluate a function handle using `feval`, the choice of the particular method called depends on:

- The methods that were visible on the path at the time the handle was constructed
- The classes of the arguments passed with the handle to the `feval` command

Examples of Function Handle Evaluation

This section provides two examples of how function handles are used and evaluated.

Example 1 - A Simple Function Handle

The following example defines a function, called `plot_fhandle`, that receives a function handle and data, and then performs an evaluation of the function handle on that data.

```
function x = plot_fhandle(fhandle, data)
plot(data, feval(fhandle, data))
```

When you call `plot_fhandle` with a handle to the `sin` function and the argument shown below, the resulting evaluation produces the following plot.

```
plot_fhandle(@sin, -pi:0.01:pi)
```

Example 2 - Function Handles and Subfunctions

The M-file in this example defines a primary function, `fitcurvedemo`, and a subfunction called `expfun`. The subfunction, by definition, is visible only within the scope of its own M-file. This, of course, means that it is available for use only by other functions within that M-file.

The author of this code would like to use `expfun` outside the confines of this one M-file. This example creates a function handle to the `expfun` subfunction, storing access information for the subfunction so that it can be called from anywhere in the MATLAB environment. The function handle is passed to `fminsearch`, which successfully evaluates the subfunction outside of its usual scope.

The code shown below defines `fitcurvedemo` and subfunction, `expfun`. Line 6 constructs a function handle to `expfun` and assigns it to the variable, `fun`. In line 16, a call to `fminsearch` passes the function handle outside the normal scope of a subfunction. The `fminsearch` function uses `feval` to evaluate the subfunction through its handle.

```
1    function Estimates = fitcurvedemo
2    % FITCURVEDEMO
3    % Fit curve to data where user chooses equation to fit.
4
5    % Define function and starting point of fitting routine.
6    fun = @expfun;
7    Starting = rand(1, 2);
8
9    % First, we create the data.
10   t = 0:.1:10;    t=t(:);   % to make 't' a column vector
11   Data = 40 * exp(-.5 * t) + randn(size(t));
12   m = [t Data];
13
14   % Now, we can call FMINSEARCH:
15   options = optimset('fminsearch');   % Use FMINSEARCH defaults
16   Estimates = fminsearch(fun, Starting, options, t, Data);
17
18   % To check the fit
19   plot(t, Data, '*')
20   hold on
21   plot(t, Estimates(1) * exp(-Estimates(2) * t), 'r')
22   xlabel('t')
```

```
23  ylabel('f(t)')
24  title(['Fitting to function ', func2str(fun)]);
25  legend('data', ['fit using ', func2str(fun)])
26  hold off
27
28  % ----------------------------------------------------------
29
30  function sse = expfun(params, t, Data)
31  % Accepts curve parameters as inputs, and outputs fitting the
32  % error for the equation y = A * exp(-lambda * t);
33  A = params(1);
34  lambda = params(2);
35
36  Fitted_Curve = A .* exp(-lambda * t);
37  Error_Vector = Fitted_Curve - Data;
38
39  % When curve fitting, a typical quantity to minimize is the sum
40  % of squares error
41  sse = sum(Error_Vector .^ 2);
```

Displaying Function Handle Information

The `functions` command returns information about a function handle that might be useful for debugging. Calling `functions` on a function handle returns the function name, type, filename, and all of the methods for the function that were in scope at the time the function handle was created.

The information returned from `functions` is in the form of a MATLAB structure. The fields of this structure are listed in the following table.

Field Name	Field Description
function	Function name.
type	Function type. See the table in "Function Type" on page 20-15.
file	The file to be executed when the function handle is evaluated with a nonoverloaded data type. For built-in functions, it reads 'MATLAB built-in function.'
methods	All overloaded methods of the function that are bound to the function handle. This field exists only for functions of type, overloaded.

For example, to obtain information on a function handle for the `display` function,

```
f = functions(@display)
ans =
    function: 'display'
        type: 'overloaded'
        file: 'MATLAB built-in function'
     methods: [1x1 struct]
```

Individual fields of the structure are accessible using the dot selection notation used to access MATLAB structure fields.

```
f.type
ans =
    'overloaded'
```

The methods field is a separate structure containing one fieldname for each class that overloads the function. The value of each field is the path and name of the file that defines the function.

```
f.methods
ans =
     polynom: '\home\user4\@polynom\display.m'
      inline: 'matlabroot\toolbox\matlab\funfun\@inline\display.m'
      serial: 'matlabroot\toolbox\matlab\iofun\@serial\display.m'
     avifile: 'matlabroot\toolbox\matlab\iofun\@avifile\display.m'

f.methods.polynom
ans =
     polynom: '\home\user4\@polynom\display.m'
```

Note The functions command does not operate on nonscalar function handles. Passing a nonscalar function handle to functions results in an error.

Fields Returned by the Functions Command

The functions command returns a MATLAB structure with the fields function, type, file, and, for overloaded functions, methods. This section describes each of those fields.

Function Name

The function field is a character array that holds the name of the function corresponding to the function handle.

Function Type

The type field is a character array that holds one of five possible strings listed in the following table.

Function Type	Type Description
simple	Nonoverloaded MATLAB built-in or M-file, or any function for which the type cannot be determined until it is evaluated
subfunction	MATLAB subfunction
private	MATLAB private function
constructor	Constructor to a MATLAB class
overloaded	Overloaded built-in, M-file function, or constructor

The contents of the next two fields, file and methods, depend upon the function type.

Function File

The file field is a character array that holds one of the following:

- The string, 'MATLAB built-in function', for built-in functions
- The path and name of the file that implements the default function, for nonbuilt-in functions

The *default* function is the one function implementation that is not specialized to operate on any particular data type. Unless the arguments in the function call specify a class that has a specialized version of the function defined, it is the default function that gets called.

The example below operates on a function handle for the deblank function. The function has a default implementation in the strfun directory. This is shown in f.file. It also has an overloaded method in the @cell directory. This is shown in f.methods.cell.

```
f = functions(@deblank)
f =
    function: 'deblank'
        type: 'overloaded'
        file: 'matlabroot\toolbox\matlab\strfun\deblank.m'
     methods: [1x1 struct]

f.methods
ans =
    cell: 'matlabroot\toolbox\matlab\strfun\@cell\deblank.m'
```

If you evaluate the @deblank function handle with a cell argument, MATLAB calls the deblank method in the @cell directory. But, for any other argument types, MATLAB calls the default M-file shown in the file field.

Function Methods

The methods field exists only for functions of type, overloaded. This field is a separate MATLAB structure that identifies all overloaded methods that are bound to the function handle.

The structure contains one field for each method of the function handle. The field names are the classes that overload the function. Each field value is a character array holding the path and name of the source file that defines the method.

For example, a function handle for the display function may be bound to the M-files shown below. The functions command returns a methods structure having a field for each class overloading the function: polynom, inline, serial, and avifile. For each class, it shows the path and name of the method source file.

```
f = functions(@display);

f.methods
ans =
     polynom: '\home\user4\@polynom\display.m'
      inline: 'matlabroot\toolbox\matlab\funfun\@inline\display.m'
      serial: 'matlabroot\toolbox\matlab\iofun\@serial\display.m'
     avifile: 'matlabroot\toolbox\matlab\iofun\@avifile\display.m'
```

Note The set of methods returned by functions is determined at the time a function handle is created. This depends on the state of the MATLAB path and also which functions are in scope when the handle is created.

Types of Function Handles

The information returned by functions varies depending on the type of function represented by the function handle. This section explains what is returned for each type of function. The categories of function handles are:

- Simple function handles
- Overloaded function handles
- Constructor function handles
- Subfunction handles
- Private function handles

Simple Function Handles

These are handles to nonoverloaded MATLAB built-in or M-file functions. Any function handles for which the function type has not yet been determined (e.g., Java methods, nonexistent functions), also fall into this category.

Structure fields:

```
function: function name
    type: 'simple'
    file: 'MATLAB built-in function'          for built-ins
          path and name of the default M-file   for nonbuilt-ins
(there is no methods field)
```

Examples:

Using functions on a function handle to a built-in function

```
functions(@ones)
ans =
    function: 'ones'
        type: 'simple'
        file: 'MATLAB built-in function'
```

Using `functions` on a function handle to a nonbuilt-in function

```
functions(@fzero)
ans =
    function: 'fzero'
        type: 'simple'
        file: 'matlabroot\toolbox\matlab\funfun\fzero.m'
```

Overloaded Function Handles

These are handles to MATLAB built-in or M-file functions that are overloaded implementations for different classes.

Structure fields:

```
function: function name
    type: 'overloaded'
    file: 'MATLAB built-in function'          for built-ins
          path and name of the default M-file   for nonbuilt-ins
 methods: [1x1 struct]
```

Examples:

Using `functions` on a function handle to a built-in function

```
functions(@display)
ans =
    function: 'display'
        type: 'overloaded'
        file: 'MATLAB built-in function'
     methods: [1x1 struct]
```

Using `functions` on a function handle to a nonbuilt-in function

```
functions(@deblank)
ans =
    function: 'deblank'
        type: 'overloaded'
        file: 'matlabroot\toolbox\matlab\strfun\deblank.m'
     methods: [1x1 struct]
```

Constructor Function Handles

These are handles to functions that construct objects of MATLAB classes.

Structure fields:

```
function: function name
    type: 'constructor'
    file: path and name of the constructor M-file
(there is no methods field)
```

Example:

Using `functions` on a function handle to a constructor function

```
functions(@inline)
ans =
    function: 'inline'
        type: 'constructor'
        file: 'matlabroot\toolbox\matlab\funfun\@inline\inline.m'
```

Subfunction Handles

These are handles to MATLAB subfunctions, which are functions defined within an M-file that are only visible to the primary function of that M-file. When you use `functions` on a subfunction handle, the `file` field of the return structure contains the path and name of the M-file in which the subfunction is defined.

Structure fields:

```
function: function name
    type: 'subfunction'
    file: path and name of the M-file defining the subfunction
(there is no methods field)
```

Example:

The `getLocalHandle` M-file, shown below, defines a primary function and a subfunction, named `subfunc`.

```
% -- File GETLOCALHANDLE.M --
function subhandle = getLocalHandle()
subhandle = @subfunc;          % return handle to subfunction
```

```
function subfunc()
disp 'running subfunc'
```

A call to getLocalHandle returns a function handle to the subfunction. When you pass that handle to functions, it returns the following information.

```
fhandle = getLocalHandle;

functions(fhandle)
ans =
    function: 'subfunc'
        type: 'subfunction'
        file: '\home\user4\getLocalHandle.m'
```

Private Function Handles

These are handles to MATLAB private functions, which are functions defined in a private subdirectory that are only visible to functions in the parent directory. When you use functions on a private function handle, the file field of the return structure contains the path and name of the M-file in the private subdirectory that defines the function.

Structure fields:

```
function: function name
    type: 'private'
    file: path and name of the M-file in \private
(there is no methods field)
```

Example:

The getPrivateHandle function, shown below, returns a handle to a private function named privatefunc.

```
% -- File GETPRIVATEHANDLE.M --
function privhandle = getPrivateHandle()
privhandle = @privatefunc;     % return handle to private function
```

The following function, privatefunc, resides in the \private subdirectory.

```
% -- File \PRIVATE\PRIVATEFUNC.M --
function privatefunc()
disp 'running privatefunc'
```

A call to getPrivateHandle returns a handle to the function, privatefunc, defined in \private. When you pass that handle to functions, it returns the following information.

```
fhandle = getPrivateHandle;

functions(fhandle)
ans =
   function: 'privatefunc'
       type: 'private'
       file: '\home\user4\private\privatefunc.m'
```

Function Handle Operations

MATLAB provides two functions that enable you to convert between a function handle and a function name string. It also provides functions for testing to see if a variable holds a function handle, and for comparing function handles.

Converting Function Handles to Function Names

If you need to perform string operations, such as string comparison or display, on a function handle, you can use func2str to obtain the function name in string format. To convert a sin function handle to a string

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Note The func2str command does not operate on nonscalar function handles. Passing a nonscalar function handle to func2str results in an error.

Example - Displaying the Function Name in an Error Message

The catcherr function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, catcherr uses sprintf to display an error message giving the name of the failing function. The function name must be a string for sprintf to display it. The code derives the function name from the function handle using func2str.

```
function catcherr(func, data)
try
    ans = feval(func, data);
    disp('Answer is:');
    ans
catch
    sprintf('Error executing function ''%s''\n', func2str(func))
end
```

The first call to catcherr, shown below, passes a handle to the round function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, round fails, causing catcherr to display an error message that includes the failing function name.

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

Converting Function Names to Function Handles

Using the str2func function, you can construct a function handle from a string containing the name of a MATLAB function. To convert the string, 'sin', into a handle for that function

```
fh = str2func('sin')
fh =
    @sin
```

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using str2func. The example below passes the variable, funcname, to function makeHandle, which then creates a function handle.

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
% -- end of makeHandle.m file --

makeHandle('sin')
ans =
    @sin
```

You can also perform the str2func operation on a cell array of function name strings. In this case, str2func returns an array of function handles.

```
fh_array = str2func({'sin' 'cos' 'tan'})
fh_array =
    @sin    @cos    @tan
```

Example - More Flexible Parameter Checking

In the following example, the myminbnd function expects to receive either a function handle or string in the first argument. If you pass a string, myminbnd constructs a function handle from it using str2func, and then uses that handle in a call to fminbnd.

```
function myminbnd(fhandle, lower, upper)
if ischar(fhandle)
    disp 'converting function string to function handle ...'
    fhandle = str2func(fhandle);
end
fminbnd(fhandle, lower, upper)
```

Whether you call myminbnd with a function handle or function name string, it is able to handle the argument appropriately.

```
myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
    0.6370
```

Testing for Data Type

The isa function identifies the data type or class of a MATLAB variable or object. You can see if a variable holds a function handle by using isa with the function_handle tag. The following function tests an argument passed in to see if it is a function handle before attempting to evaluate it.

```
function evaluate_handle(arg1, arg2)
if isa(arg1, 'function_handle')
    feval(arg1, arg2);
else
    disp 'You need to pass a function handle';
end
```

Testing for Equality

You can use the isequal function to compare two function handles for equality. For example, you want to execute one particular method you have written for an overloaded function. You create a function handle within a confined scope so that it provides access to that method alone. The function shown below,

test_myfun, receives this function handle in the first argument, arg1, and evaluates it.

Before evaluating the handle in arg1, test_myfun checks it against another function handle that has a broader definition. This other handle, @myfun, provides access to all methods for the function. If the function handle in arg1 represents more than the one intended method, an error message is displayed and the function is not evaluated.

```
function test_myfun(arg1, arg2)
if isequal(arg1, @myfun)
    disp 'Function handle holds unexpected context'
else
    feval(arg1, arg2);
end
```

Saving and Loading Function Handles

You can use the MATLAB `save` and `load` functions to save function handles to MAT files, and then load them back into your MATLAB workspace later on. This example shows an array of function handles saved to the file, `savefile`, and then restored.

```
fh_array = [@sin @cos @tan];
save savefile fh_array;
clear

load savefile
whos
  Name              Size          Bytes  Class
   fh_array        1x3              48   function_handle array
Grand total is 3 elements using 48 bytes
```

Possible Effects of Changes Made Between Save and Load

If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the M-files that define the function have been moved, and thus no longer exist on the path stored in the handle.

- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either doesn't exist or is located in a different directory than on the system on which the handle was saved.

- You have overloaded the function with additional methods since the `save` was done. The function handle you just loaded doesn't know about the new methods.

In the first two cases, the function handle is now invalid, since it no longer maps to any existing source code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. An attempt to evaluate the handle however results in an error.

Handling Error Conditions

The following are error conditions associated with the use of function handles.

Handles to Nonexistent Functions

If you create a handle to a function that does not exist, MATLAB catches the error when the handle is evaluated by feval. MATLAB allows you to assign an invalid handle and use it in such operations as func2str, but will catch and report an error when you attempt to use it in a runtime operation. For example,

```
fhandle = @no_such_function;

func2str(fhandle)
ans =
no_such_function

feval(fhandle)
??? Error using ==> feval
Undefined function 'no_such_function'.
```

Including Path In the Function Handle Constructor

You construct a function handle using the *at* sign, @, or the str2func function. In either case, you specify the function using only the simple function name. The function name cannot include path information. Either of the following successfully creates a handle to the deblank function.

```
fhandle = @deblank;
fhandle = str2func('deblank');
```

The next example includes the path to deblank.m, and thus returns an error.

```
fhandle = str2func(which('deblank'))
??? Error using ==> str2func
Invalid function name
    'matlabroot\toolbox\matlab\strfun\deblank.m'.
```

Evaluating a Nonscalar Function Handle

The feval function evaluates function handles only if they are scalar. Calling feval with a nonscalar function handle results in an error.

```
feval([@sin @cos], 5)
??? Error using ==> feval
Function_handle argument must be scalar.
```

Historical Note - Evaluating Function Names

Evaluating a function by means of a function handle replaces the former MATLAB mechanism of evaluating a function through a string containing the function name. For example, of the following two lines of code that evaluate the humps function, the second supersedes the first and is considered to be the preferable mechanism to use.

```
feval('humps', 0.5674);      % uses a function name string

feval(@humps, 0.5674);       % uses a function handle
```

To support backward compatibility, feval still accepts a function name string as a first argument and evaluates the function named in the string. However, function handles offer you the additional performance, reliability, and source file control benefits listed in "Benefits of Using Function Handles" on page 20-3.

21

MATLAB Classes and Objects

This chapter describes how to define your own classes in MATLAB. Classes and objects enable you to add new data types and new operations to MATLAB. The *class* of a variable describes the structure of the variable and indicates the kinds of operations and functions that can apply to the variable. An *object* is an instance of a particular class. The phrase *object-oriented programming* describes an approach to writing programs that emphasizes the use of classes and objects.

Classes and Objects: An Overview

You can view classes as new data types having specific behaviors defined for the class. For example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as *overloading* the operator.

Features of Object-Oriented Programming

When using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend. Programming with classes and objects differs from ordinary structured programming in these important ways:

- **Function and operator overloading**. You can create methods that override existing MATLAB functions. When you call a function with a user-defined object as an argument, MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal MATLAB function.

- **Encapsulation of data and methods**. Object properties are not visible from the command line; you can access them only with class methods. This protects the object properties from operations that are not intended for the object's class.

- **Inheritance**. You can create class hierarchies of parent and child classes in which the child class inherits data fields and methods from the parent. A child class can inherit from one parent (*single inheritance*) or many parents (*multiple inheritance*). Inheritance can span one or more generations. Inheritance enables sharing common parent functions and enforcing common behavior amongst all child classes.

- **Aggregation**. You can create classes using *aggregation*, in which an object contains other objects. This is appropriate when an object type is part of another object type. For example, a savings account object might be a part of a financial portfolio object.

MATLAB Data Class Hierarchy

All MATLAB data types are designed to function as classes in object-oriented programming. The diagram below shows the fifteen fundamental data types (or classes) defined in MATLAB. You can add new data types to MATLAB by extending the class hierarchy.

```
                          ARRAY
                     [full or sparse]



logical   char      NUMERIC     cell     structure      function
                                                          handle

                                              user classes  java classes


   int8, uint8,
  int16, uint16,
  int32,uint32,     single    double
  int64,uint64
```

The diagram shows a *user class* that inherits from the structure class. All classes that you create are structure based since this is the point in the class hierarchy where you can insert your own classes. (For more information about MATLAB data types, see the section on "Data Types.")

Creating Objects

You create an object by calling the class constructor and passing it the appropriate input arguments. In MATLAB, constructors have the same name as the class name. For example, the statement,

```
p = polynom([1 0 –2 –5]);
```

creates an object named p belonging to the class polynom. Once you have created a polynom object, you can operate on the object using methods that are defined for the polynom class. See "Example — A Polynomial Class" on page 21-23 for a description of the polynom class.

Invoking Methods on Objects

Class methods are M-file functions that take an object as one of the input arguments. The methods for a specific class must be placed in the class directory for that class (the @*class_name* directory). This is the first place that MATLAB looks to find a class method.

The syntax for invoking a method on an object is similar to a function call. Generally, it looks like

```
[out1,out2,...] = method_name(object,arg1,arg2, ...);
```

For example, suppose a user-defined class called polynom has a char method defined for the class. This method converts a polynom object to a character string and returns the string. This statement calls the char method on the polynom object p.

```
s = char(p);
```

Using the class function, you can confirm that the returned value s is a character string.

```
class(s)
ans =
    char

s
s =
    x^3–2*x–5
```

You can use the `methods` command to produce a list of all of the methods that are defined for a class.

Private Methods

Private methods can be called only by other methods of their class. You define private methods by placing the associated M-files in a `private` subdirectory of the @*class_name* directory. In the example,

```
@class_name/private/update_obj.m
```

the method `update_obj` has scope only within the *class_name* class. This means that `update_obj` can be called by any method that is defined in the @*class_name* directory, but it cannot be called from the MATLAB command line or by methods outside of the class directory, including parent methods.

Private methods and private functions differ in that private methods (in fact all methods) have an object as one of their input arguments and private functions do not. You can use private functions as helper functions, such as described in the next section.

Helper Functions

In designing a class, you may discover the need for functions that perform support tasks for the class, but do not directly operate on an object. These functions are called *helper functions*. A helper function can be a subfunction in a class method file or a private function. When determining which version of a particular function to call, MATLAB looks for these functions in the order listed above. For more information about the order in which MATLAB calls functions and methods, see "How MATLAB Determines Which Method to Call" on page 21-66.

Debugging Class Methods

You can use the MATLAB debugging commands with object methods in the same way that you use them with other M-files. The only difference is that you need to include the class directory name before the method name in the command call, as shown in this example using `dbstop`.

```
dbstop @polynom/char
```

While debugging a class method, you have access to all methods defined for the class, including inherited methods, private methods, and private functions.

Changing Class Definition

If you change the class definition, such as the number or names of fields in a class, you must issue a

```
clear classes
```

command to propagate the changes to your MATLAB session. This command also clears all objects from the workspace. See the `clear` command help entry for more information.

Setting Up Class Directories

The M-files defining the methods for a class are collected together in a directory referred to as the class directory. The directory name is formed with the class name preceded by the character @. For example, one of the examples used in this chapter is a class involving polynomials in a single variable. The name of the class, and the name of the class constructor, is `polynom`. The M-files defining a polynomial class would be located in directory with the name `@polynom`.

The class directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For instance, the new `@polynom` directory could be a subdirectory of the MATLAB working directory or your own personal directory that has been added to the search path.

Adding the Class Directory to the MATLAB Path

After creating the class directory, you need to update the MATLAB path so that MATLAB can locate the class source files. The class directory should not be directly on the MATLAB path. Instead, you should add the parent directory to the MATLAB path. For example, if the `@polynom` class directory is located at

```
c:\my_classes\@polynom
```

you add the class directory to the MATLAB path with the `addpath` command

```
addpath c:\my_classes;
```

If you create a class directory with the same name as another class, MATLAB treats the two class directories as a single directory when locating class methods. For more information, see "How MATLAB Determines Which Method to Call" on page 21-66.

Data Structure

One of the first steps in the design of a new class is the choice of the data structure to be used by the class. Objects are stored in MATLAB structures. The fields of the structure, and the details of operations on the fields, are visible only within the methods for the class. The design of the appropriate data structure can affect the performance of the code.

Tips for C++ and Java Programmers

If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

- In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.

- In MATLAB, there is no equivalent to a destructor method. To remove an object from the workspace, use the `clear` function.

- Construction of MATLAB data types occurs at runtime rather than compile time. You register an object as belonging to a class by calling the `class` function.

- When using inheritance in MATLAB, the inheritance relationship is established in the child class by creating the parent object, and then calling the `class` function. For more information on writing constructors for inheritance relationships, see "Building on Other Classes" on page 21-34.

- When using inheritance in MATLAB, the child object contains a parent object in a property with the name of the parent class.

- In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For instance, this call to the `set` method updates the `name` field of the object `A` and returns the updated object.

  ```
  A = set(A,'name','John Smith');
  ```

- In MATLAB, there is no equivalent to an abstract class.
- In MATLAB, there is no equivalent to the C++ scoping operator.
- In MATLAB, there is no virtual inheritance or virtual base classes.
- In MATLAB, there is no equivalent to C++ templates.

Designing User Classes in MATLAB

This section discusses how to approach the design of a class and describes the basic set of methods that should be included in a class.

The MATLAB Canonical Class

When you design a MATLAB class, you should include a standard set of methods that enable the class to behave in a consistent and logical way within the MATLAB environment. Depending on the nature of the class you are defining, you may not need to include all of these methods and you may include a number of other methods to realize the class's design goals.

This table lists the basic methods included in MATLAB classes.

Class Method	Description
class constructor	Creates an object of the class
display	Called whenever MATLAB displays the contents of an object (e.g., when an expression is entered without terminating with a semicolon)
set and get	Accesses class properties
subsref and subsasgn	Enables indexed reference and assignment for user objects
end	Supports end syntax in indexing expressions using an object; e.g., A(1:end)
subsindex	Supports using an object in indexing expressions
converters like double and char	Methods that convert an object to a MATLAB data type

The following sections discuss the implementation of each type of method, as well as providing references to examples used in this chapter.

The Class Constructor Method

The @ directory for a particular class must contain an M-file known as the *constructor* for that class. The name of the constructor is the same as the name of the directory (excluding the @ prefix and .m extension) that defines the name of the class. The constructor creates the object by initializing the data structure and instantiating an object of the class.

Guidelines for Writing a Constructor

Class constructors must perform certain functions so that objects behave correctly in the MATLAB environment. In general, a class constructor must handle three possible combinations of input arguments:

- No input arguments
- An object of the same class as an input argument
- The input arguments used to create an object of the class (typically data of some kind)

No Input Arguments. If there are no input arguments, the constructor should create a default object. Since there are no inputs, you have no data from which to create the object, so you simply initialize the object's data structures with empty or default values, call the `class` function to instantiate the object, and return the object as the output argument. Support for this syntax is required for two reasons:

- When loading objects into the workspace, the `load` function calls the class constructor with no arguments.
- When creating arrays of objects, MATLAB calls the class constructor to add objects to the array.

Object Input Argument. If the first input argument in the argument list is an object of the same class, the constructor should simply return the object. Use the `isa` function to determine if an argument is a member of a class. See "Overloading the + Operator" on page 21-28 for an example of a method that uses this constructor syntax.

Data Input Arguments. If the input arguments exist and are not objects of the same class, then the constructor creates the object using the input data. Of course, as in any function, you should perform proper argument checking in your constructor function. A typical approach is to use a `varargin` input

argument and a `switch` statement to control program flow. This provides an easy way to accommodate the three cases: no inputs, object input, or the data inputs used to create an object.

It is in this part of the constructor that you assign values to the object's data structure, call the `class` function to instantiate the object, and return the object as the output argument. If necessary, place the object in an object hierarchy using the `superiorto` and `inferiorto` functions.

Using the class Function in Constructors

Within a constructor method, you use the `class` function to associate an object structure with a particular class. This is done using an internal class tag that is only accessible using the `class` and `isa` functions. For example, this call to the `class` function identifies the object p to be of type `polynom`.

```
p = class(p,'polynom');
```

Examples of Constructor Methods

See the following sections for examples of constructor methods:

- "The Polynom Constructor Method" on page 21-23
- "The Asset Constructor Method" on page 21-38
- "The Stock Constructor Method" on page 21-45
- "The Portfolio Constructor Method" on page 21-54

Identifying Objects Outside the Class Directory

The `class` and `isa` functions used in constructor methods can also be used outside of the class directory. The expression

```
isa(a,'class_name');
```

checks whether a is an object of the specified class. For example, if p is a polynom object, each of the following expressions is true.

```
isa(pi,'double');
isa('hello','char');
isa(p,'polynom');
```

Outside of the class directory, the `class` function takes only one argument (it is only within the constructor that `class` can have more than one argument).

The expression

```
class(a)
```

returns a string containing the class name of a. For example,

```
class(pi),
class('hello'),
class(p)
```

return

```
'double',
'char',
'polynom'
```

Use the `whos` function to see what objects are in the MATLAB workspace.

```
whos

  Name      Size           Bytes  Class
  p         1x1              156  polynom object
```

The display Method

MATLAB calls a method named `display` whenever an object is the result of a statement that is not terminated by a semicolon. For example, creating the variable a, which is a double, calls the MATLAB `display` method for doubles.

```
a = 5
a =
    5
```

You should define a `display` method so MATLAB can display values on the command line when referencing objects from your class. In many classes, `display` can simply print the variable name, and then use the `char` converter method to print the contents or value of the variable, since MATLAB displays output as strings. You must define the `char` method to convert the object's data to a character string.

Examples of display Methods

See the following sections for examples of `display` methods:

Accessing Object Data

You need to write methods for your class that provide access to an object's data. Accessor methods can use a variety of approaches, but all methods that change object data always accept an object as an input argument and return a new object with the data changed. This is necessary because MATLAB does not support passing arguments by reference (i.e., pointers). Functions can change only their private, temporary copy of an object. Therefore, to change an existing object, you must create a new one, and then replace the old one.

The following sections provide more detail about implementation techniques for the `set`, `get`, `subsasgn`, and `subsref` methods.

The set and get Methods

The `set` and `get` methods provide a convenient way to access object data in certain cases. For example, suppose you have created a class that defines an arrow object that MATLAB can display on graphs (perhaps composed of existing MATLAB line and patch objects).

To produce a consistent interface, you could define `set` and `get` methods that operate on arrow objects the way the MATLAB `set` and `get` functions operate on built-in graphics objects. The `set` and `get` verbs convey what operations they perform, but insulate the user from the internals of the object.

Examples of set and get Methods

See the following sections for examples of `set` and `get` methods:

Property Name Methods

As an alternative to a general `set` method, you can write a method to handle the assignment of an individual property. The method should have the same name as the property name.

For example, if you defined a class that creates objects representing employee data, you might have a field in an employee object called `salary`. You could then define a method called `salary.m` that takes an employee object and a value as input arguments and returns the object with the specified value set.

Indexed Reference Using subsref and subsasgn

User classes implement new data types in MATLAB. It is useful to be able to access object data via an indexed reference, as is possible with the MATLAB built-in data types. For example, if `A` is an array of class `double`, `A(i)` returns the ith element of `A`.

As the class designer, you can decide what an index reference to an object means. For example, suppose you define a class that creates polynomial objects and these objects contain the coefficients of the polynomial.

An indexed reference to a polynomial object,

```
p(3)
```

could return the value of the coefficient of x^3, the value of the polynomial at $x = 3$, or something different depending on the intended design.

You define the behavior of indexing for a particular class by creating two class methods – `subsref` and `subsasgn`. MATLAB calls these methods whenever a subscripted reference or assignment is made on an object from the class. If you do not define these methods for a class, indexing is undefined for objects of this class.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. For details, see "Structures" on page 19-4.

Handling Subscripted Reference

The use of a subscript or field designator with an object on the right-hand side of an assignment statement is known as a *subscripted reference*. MATLAB calls a method named `subsref` in these situations.

Object subscripted references can be of three forms – an array index, a cell array index, and a structure field name:

```
A(I)
A{I}
A.field
```

Each of these results in a call by MATLAB to the subsref method in the class directory. MATLAB passes two arguments to subsref.

```
B = subsref(A,S)
```

The first argument is the object being referenced. The second argument, S, is a structure array with two fields:

- S.type is a string containing '()', '{}', or '.' specifying the subscript type. The parentheses represent a numeric array; the curly braces, a cell array; and the dot, a structure array.
- S.subs is a cell array or string containing the actual subscripts. A colon used as a subscript is passed as the string ':'.

For instance, the expression

```
A(1:2,:)
```

causes MATLAB to call subsref(A,S), where S is a 1-by-1 structure with

```
S.type = '()'
S.subs = {1:2,':'}
```

Similarly, the expression

```
A{1:2}
```

uses

```
S.type ='{}'
S.subs = {1:2}
```

The expression

```
A.field
```

calls `subsref(A,S)` where

```
S.type = '.'
S.subs = 'field'
```

These simple calls are combined for more complicated subscripting expressions. In such cases, `length(S)` is the number of subscripting levels. For example,

```
A(1,2).name(3:4)
```

calls `subsref(A,S)`, where S is a 3-by-1 structure array with the values:

```
S(1).type = '()'    S(2).type = '.'      S(3).type = '()'
S(1).subs = '{1,2}' S(2).subs = 'name'   S(3).subs = '{3:4}'
```

How to Write subsref

The `subsref` method must interpret the subscripting expressions passed in by MATLAB. A typical approach is to use the `switch` statement to determine the type of indexing used and to obtain the actual indices. The following three code fragments illustrate how to interpret the input arguments. In each case, the function must return the value B.

For an array index:

```
switch S.type
case '()'
   B = A(S.subs{:});
end
```

For a cell array:

```
switch S.type
case '{}'
   B = A(S.subs{:}); % A is a cell array
end
```

For a structure array:

```
switch S.type
case '.'
   switch S.subs
   case 'field1'
       B = A.field1;
```

```
        case 'field2'
            B = A.field2;
        end
    end
end
```

Examples of the subsref Method

See the following sections for examples of the subsref method:

Handling Subscripted Assignment

The use of a subscript or field designator with an object on the left-hand side of an assignment statement is known as a *subscripted assignment*. MATLAB calls a method named subsasgn in these situations. Object subscripted assignment can be of three forms – an array index, a cell array index, and a structure field name.

```
A(I) = B
A{I} = B
A.field = B
```

Each of these results in a call to subsasgn of the form

```
A = subsasgn(A,S,B)
```

The first argument, A, is the object being referenced. The second argument, S, has the same fields as those used with subsref. The third argument, B, is the new value.

Examples of the subsasgn Method

See the following sections for examples of the subsasgn method:

Object Indexing Within Methods

If a subscripted reference is made within a class method, MATLAB uses its built-in subsref function to access data within the method's own class. If the method accesses data from another class, MATLAB calls the overloaded subsref function in that class. The same holds true for subscripted assignment and subsasgn.

The following example shows a method, testref, that is defined in the class, employee. This method makes a reference to a field, address, in an object of its own class. For this, MATLAB uses the built-in subsref function. It also references the same field in another class, this time using the overloaded subsref of that class.

```
% ---- EMPLOYEE class method: testref.m ----
function testref(myclass,otherclass)

myclass.address              % use built-in subsref
otherclass.address           % use overloaded subsref
```

The example creates an employee object and a company object.

```
empl = employee('Johnson','Chicago');
comp = company('The MathWorks','Natick');
```

The employee class method, testref, is called. MATLAB uses an overloaded subsref only to access data outside of the method's own class.

```
testref(empl,comp)
ans =                        % built-in subsref was called
   Chicago

ans =                        % @company\subsref was called
Executing @company\subsref ...
   Natick
```

Defining end Indexing for an Object

When you use end in an object indexing expression, MATLAB calls the object's end class method. If you want to be able to use end in indexing expressions involving objects of your class, you must define an end method for your class.

The end method has the calling sequence

```
end(a,k,n)
```

where a is the user object, k is the index in the expression where the end syntax is used, and n is the total number of indices in the expression.

For example, consider the expression

```
A(end-1,:)
```

MATLAB calls the end method defined for the object A using the arguments

```
end(A,1,2)
```

That is, the end statement occurs in the first index element and there are two index elements. The class method for end must then return the index value for the last element of the first dimension. When you implement the end method for your class, you must ensure it returns a value appropriate for the object.

Indexing an Object with Another Object

When MATLAB encounters an object as an index, it calls the subsindex method defined for the object. For example, suppose you have an object a and you want to use this object to index into another object b.

```
c = b(a);
```

A subsindex method might do something as simple as convert the object to double format to be used as an index, as shown in this sample code.

```
function d = subsindex(a)
%SUBSINDEX
% convert the object a to double format to be used
% as an index in an indexing expression
d = double(a);
```

subsindex values are 0-based, not 1-based.

Converter Methods

A converter method is a class method that has the same name as another class, such as char or double. Converter methods accept an object of one class as input and return an object of another class. Converters enable you to:

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly

A converter function call is of the form

```
b = class_name(a)
```

where a is an object of a class other than *class_name*. In this case, MATLAB looks for a method called *class_name* in the class directory for object a. If the input object is already of type *class_name*, then MATLAB calls the constructor, which just returns the input argument.

Examples of Converter Methods

See the following sections for examples of converter methods:

- "The Polynom to Double Converter" on page 21-24
- "The Polynom to Char Converter" on page 21-25

Overloading Operators and Functions

In many cases, you may want to change the behavior of the MATLAB operators and functions for cases when the arguments are objects. You can accomplish this by *overloading* the relevant functions. Overloading enables a function to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest-precedence object. See "Object Precedence" on page 21-64 for more information on object precedence.

Overloading Operators

Each built-in MATLAB operator has an associated function name (e.g., the + operator has an associated `plus.m` function). You can overload any operator by creating an M-file with the appropriate name in the class directory. For example, if either p or q is an object of type *class_name*, the expression

```
p + q
```

generates a call to a function `@class_name/plus.m`, if it exists. If p and q are both objects of different classes, then MATLAB applies the rules of precedence to determine which method to use.

Examples of Overloaded Operators

See the following sections for examples of overloaded operators:

- "Overloading the + Operator" on page 21-28
- "Overloading the – Operator" on page 21-29
- "Overloading the ∗ Operator" on page 21-29

The following table lists the function names for most of the MATLAB operators.

Operation	M-File	Description
a + b	plus(a,b)	Binary addition
a - b	minus(a,b)	Binary subtraction
-a	uminus(a)	Unary minus
+a	uplus(a)	Unary plus
a.*b	times(a,b)	Element-wise multiplication
a*b	mtimes(a,b)	Matrix multiplication
a./b	rdivide(a,b)	Right element-wise division
a.\b	ldivide(a,b)	Left element-wise division
a/b	mrdivide(a,b)	Matrix right division
a\b	mldivide(a,b)	Matrix left division
a.^b	power(a,b)	Element-wise power
a^b	mpower(a,b)	Matrix power
a < b	lt(a,b)	Less than
a > b	gt(a,b)	Greater than
a <= b	le(a,b)	Less than or equal to
a >= b	ge(a,b)	Greater than or equal to
a ~= b	ne(a,b)	Not equal to
a == b	eq(a,b)	Equality
a & b	and(a,b)	Logical AND
a | b	or(a,b)	Logical OR
~a	not(a)	Logical NOT

Operation	M-File	Description
`a:d:b` `a:b`	`colon(a,d,b)` `colon(a,b)`	Colon operator
`a'`	`ctranspose(a)`	Complex conjugate transpose
`a.'`	`transpose(a)`	Matrix transpose
`command window output`	`display(a)`	Display method
`[a b]`	`horz-cat(a,b,...)`	Horizontal concatenation
`[a; b]`	`vert-cat(a,b,...)`	Vertical concatenation
`a(s1,s2,...sn)`	`subsref(a,s)`	Subscripted reference
`a(s1,...,sn) = b`	`subsasgn(a,s,b)`	Subscripted assignment
`b(a)`	`subsindex(a)`	Subscript index

Overloading Functions

You can overload any function by creating a function of the same name in the class directory. When a function is invoked on an object, MATLAB always looks in the class directory before any other location on the search path. To overload the plot function for a class of objects, for example, simply place your version of plot.m in the appropriate class directory.

Examples of Overloaded Functions

See the following sections for examples of overloaded functions:

- "Overloading Functions for the Polynom Class" on page 21-30
- "The Portfolio pie3 Method" on page 21-56

Example — A Polynomial Class

This example implements a MATLAB data type for polynomials by defining a new class called polynom. The class definition specifies a structure for data storage and defines a directory (@polynom) of methods that operate on polynom objects.

Polynom Data Structure

The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in decreasing order. Therefore, a polynom object p is a structure with a single field, p.c, containing the coefficients. This field is accessible only within the methods in the @polynom directory.

Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method polynom.m
- A polynom to double converter
- A polynom to char converter
- A display method
- A subsref method
- Overloaded +, −, and * operators
- Overloaded roots, polyval, plot, and diff functions

The Polynom Constructor Method

Here is the polynom class constructor, @polynom/polynom.m.

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
%   p = POLYNOM(v) creates a polynomial object from the vector v,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
```

```
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

Constructor Calling Syntax

You can call the polynom constructor method with one of three different arguments:

- No Input Argument – If you call the constructor function with no arguments, it returns a polynom object with empty fields.
- Input Argument is an Object – If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The isa function (pronounced "is a") checks for this situation.
- Input Argument is a coefficient vector – If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the .c field of the object's structure. The class function creates the polynom object, which is then returned by the constructor.

An example use of the polynom constructor is the statement

```
p = polynom([1 0 -2 -5])
```

This creates a polynomial with the specified coefficients.

Converter Methods for the Polynom Class

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are double and char. Conversion to double produces the MATLAB traditional matrix, although this may not be appropriate for some classes. Conversion to char is useful for producing printed output.

The Polynom to Double Converter

The double converter method for the polynom class is a very simple M-file, @polynom/double.m, which merely retrieves the coefficient vector.

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

On the object p,

```
p = polynom([1 0 -2 -5])
```

the statement

```
double(p)
```

returns

```
ans =
    1    0   -2   -5
```

The Polynom to Char Converter

The converter to char is a key method because it produces a character string involving the powers of an independent variable, x. Therefore, once you have specified x, the string returned is a syntactically correct MATLAB expression, which you can then evaluate.

Here is @polynom/char.m.

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
    s = '0';
else
    d = length(p.c) - 1;
    s = [];
    for a = p.c;
        if a ~= 0;
            if ~isempty(s)
                if a > 0
                    s = [s ' + '];
                else
                    s = [s ' - '];
                    a = -a;
```

21-25

```
                    end
                end
                if a ~= 1 | d == 0
                    s = [s num2str(a)];
                    if d > 0
                        s = [s '*'];
                    end
                end
                if d >= 2
                    s = [s 'x^' int2str(d)];
                elseif d == 1
                    s = [s 'x'];
                end
            end
            d = d - 1;
        end
    end
```

Evaluating the Output

If you create the polynom object p

```
p = polynom([1 0 -2 -5]);
```

and then call the char method on p

```
char(p)
```

MATLAB produces the result

```
ans =
    x^3 - 2*x - 5
```

The value returned by char is a string that you can pass to eval once you have defined a scalar value for x. For example,

```
x = 3;

eval(char(p))
ans =
    16
```

See "The Polynom subsref Method" on page 21-27 for a better method to evaluate the polynomial.

The Polynom display Method

Here is @polynom/display.m. This method relies on the char method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp(['   ' char(p)])
disp(' ');
```

The statement

```
p = polynom([1 0 -2 -5])
```

creates a polynom object. Since the statement is not terminated with a semicolon, the resulting output is

```
p =
   x^3 - 2*x - 5
```

The Polynom subsref Method

Suppose the design of the polynom class specifies that a subscripted reference to a polynom object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object p,

```
p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at $x = 3$ and $x = 4$.

```
p([3 4])
ans =
   16   51
```

subsref Implementation Details

This implementation takes advantage of the char method already defined in the polynom class to produce an expression that can then be evaluated.

```
function b = subsref(a,s)
% SUBSREF
switch s.type
case '()'
   ind = s.subs{:};
   for k = 1:length(ind)
       b(k) = eval(strrep(char(a),'x',num2str(ind(k))));
   end
otherwise
   error('Specify value for x as p(x)')
end
```

Once the polynomial expression has been generated by the char method, the strrep function is used to swap the passed in value for the character x. The eval function then evaluates the expression and returns the value in the output argument.

Overloading Arithmetic Operators for polynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, the plus, minus, and mtimes methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

Overloading the + Operator

If either p or q is a polynom, the expression

```
p + q
```

generates a call to a function @polynom/plus.m, if it exists (unless p or q is an object of a higher precedence, as described in "Object Precedence" on page 21-64).

The following M-file redefines the + operator for the polynom class.

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

```
p + 1
```

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the polynom constructor a third time to create the properly typed result.

Overloading the − Operator

You can implement the overloaded minus operator (-) using the same approach as the plus (+) operator. MATLAB calls @polynom/minus.m to compute p–q.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

Overloading the ∗ Operator

MATLAB calls the method @polynom/mtimes.m to compute the product p*q. The letter m at the beginning of the function name comes from the fact that it is overloading the MATLAB *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Using the Overloaded Operators

Given the polynom object

```
p = polynom([1 0 -2 -5])
```

MATLAB calls these two functions @polynom/plus.m and @polynom/mtimes.m
when you issue the statements

```
q = p+1
r = p*q
```

to produce

```
q =
   x^3 - 2*x - 4

r =
   x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading Functions for the Polynom Class

MATLAB already has several functions for working with polynomials
represented by coefficient vectors. They should be overloaded to also work with
the new polynom object. In many cases, the overloading methods can simply
apply the original function to the coefficient field.

Overloading roots for the Polynom Class

The method @polynom/roots.m finds the roots of polynom objects.

```
function r = roots(p)
% POLYNOM/ROOTS.  ROOTS(p) is a vector containing the roots of p.
r = roots(p.c);
```

The statement

```
roots(p)
```

results in

```
ans =
    2.0946
   -1.0473 + 1.1359i
   -1.0473 - 1.1359i
```

Overloading polyval for the Polynom Class

The function `polyval` evaluates a polynomial at a given set of points. `@polynom/polyval.m` uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
    y = y.*x + a;
end
```

Overloading plot for the Polynom Class

The overloaded `plot` function uses both `root` and `polyval`. The function selects the domain of the independent variable to be slightly larger than an interval containing all real roots. Then `polyval` is used to evaluate the polynomial at a few hundred points in the domain.

```
function plot(p)
% POLYNOM/PLOT  PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Overloading diff for the Polynom Class

The method `@polynom/diff.m` differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Listing Class Methods

The function call

```
methods('class_name')
```

or its command form

```
methods class_name
```

shows all the methods available for a particular class. For the `polynom` example, the output is

```
methods polynom
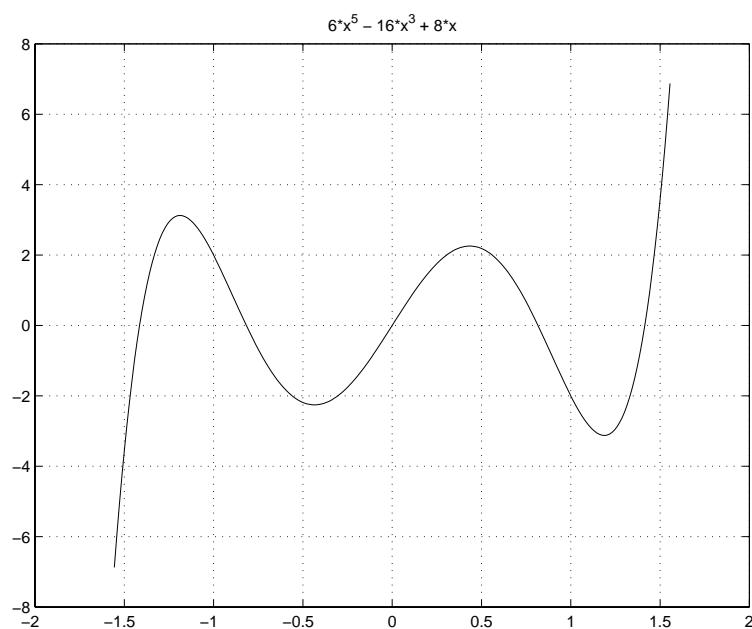
Methods for class polynom:

char    display   minus     plot      polynom   roots

diff    double    mtimes    plus      polyval   subsref
```

Plotting the two polynom objects x and p calls most of these methods.

```
x = polynom([1 0]);
p = polynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```

$6*x^5 - 16*x^3 + 8*x$

Building on Other Classes

A MATLAB object can *inherit* properties and behavior from another MATLAB object. When one object (the child) inherits from another (the parent), the child object includes all the fields of the parent object and can call the parent's methods. The parent methods can access those fields that a child object inherited from the parent class, but not fields new to the child class.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing child objects to take advantage of code that exists for parent objects. Inheritance enables a child object to behave exactly like a parent object, which facilitates the development of related classes that behave similarly, but are implemented differently.

There are two kinds of inheritance:

- Simple inheritance, in which a child object inherits characteristics from one parent class.
- Multiple inheritance, in which a child object inherits characteristics from more than one parent class.

This section also discusses a related topic, *aggregation*. Aggregation allows one object to contain another object as one of its fields.

Simple Inheritance

A class that inherits attributes from a single parent class, and adds new attributes of its own, uses simple inheritance. Inheritance implies that objects belonging to the child class have the same fields as the parent class, as well as additional fields. Therefore, methods associated with the parent class can operate on objects belonging to the child class. The methods associated with the child class, however, cannot operate on objects belonging to the parent class. You cannot access the parent's fields directly from the child class; you must use access methods defined for the parent.

The constructor function for a class that inherits the behavior of another has two special characteristics:

- It calls the constructor function for the parent class to create the inherited fields.
- The calling syntax for the `class` function is slightly different, reflecting both the child class and the parent class.

The general syntax for establishing a simple inheritance relationship using the `class` function is

```
child_obj = class(child_obj,'child_class',parent_obj);
```

Simple inheritance can span more than one generation. If a parent class is itself an inherited class, the child object will automatically inherit from the grandparent class.

Visibility of Class Properties and Methods

The parent class does not have knowledge of the child properties or methods. The child class cannot access the parent properties directly, but must use parent access methods (e.g., `get` or `subsref` method) to access the parent properties. From the child class methods, this access is accomplished via the parent field in the child structure. For example, when a constructor creates a child object c,

```
c = class(c,'child_class_name',parent_object);
```

MATLAB automatically creates a field, c.*parent_class_name*, in the object's structure that contains the parent object. You could then have a statement in the child's display method that calls the parent's display method.

```
display(c.parent_class_name)
```

See "Designing the Stock Class" on page 21-44 for examples that use simple inheritance.

Multiple Inheritance

In the multiple inheritance case, a class of objects inherits attributes from more than one parent class. The child object gets fields from all the parent classes, as well as fields of its own.

Multiple inheritance can encompass more than one generation. For example, each of the parent objects could have inherited fields from multiple grandparent objects, and so on. Multiple inheritance is implemented in the constructors by calling `class` with more than three arguments.

```
obj = class(structure,'class_name',parent1,parent2,...)
```

You can append as many parent arguments as desired to the class input list.

Multiple parent classes can have associated methods of the same name. In this case, MATLAB calls the method associated with the parent that appears first in the `class` function call in the constructor function. There is no way to access subsequent parent function of this name.

Aggregation

In addition to standard inheritance, MATLAB objects support *containment* or *aggregation*. That is, one object can contain (embed) another object as one of its fields. For example, a rational object might use two polynom objects, one for the numerator and one for the denominator.

You can call a method for the contained object only from within a method for the outer object. When determining which version of a function to call, MATLAB considers only the outermost containing class of the objects passed as arguments; the classes of any contained objects are ignored.

See "Example — The Portfolio Container" on page 21-53 for an example of aggregation.

Example - Assets and Asset Subclasses

As an example of simple inheritance, consider a general asset class that can be used to represent any item that has monetary value. Some examples of an asset are: stocks, bonds, savings accounts, and any other piece of property. In designing this collection of classes, the asset class holds the data that is common to all of the specialized asset subclasses. The individual asset subclasses, such as the stock class, inherit the asset properties and contribute additional properties. The subclasses are "kinds of" assets.

Inheritance Model for the Asset Class

An example of a simple inheritance relationship using an asset parent class is shown in this diagram.

As shown in the diagram, the stock, bond, and savings classes inherit structure fields from the asset class. In this example, the asset class is used to provide storage for data common to all subclasses and to share asset methods with these subclasses. This example shows how to implement the

asset and stock classes. The bond and savings classes can be implemented in a way that is very similar to the stock class, as would other types of asset subclasses.

Asset Class Design

The asset class provides storage and access for information common to all asset children. It is not intended to be instantiated directly, so it does not require an extensive set of methods. To serve its purpose, the class needs to contain the following methods:

- Constructor
- get and set
- subsref and subsasgn
- display

Other Asset Methods

The asset class provides inherited data storage for its child classes, but is not instanced directly. The set, get, and display methods provide access to the stored data. It is not necessary to implement the full complement of methods for asset objects (such as converters, end, and subsindex) since only the child classes access the data.

The Asset Constructor Method

The asset class is based on a structure array with four fields:

- descriptor – Identifier of the particular asset (e.g., stock name, savings account number, etc.)
- date – The date the object was created (calculated by the date command)
- type – The type of asset (e.g., savings, bond, stock)
- current_value – The current value of the asset (calculated from subclass data)

This information is common to asset child objects (stock, bond, and savings), so it is handled from the parent object to avoid having to define the same fields in each child class. This is particularly helpful as the number of child classes increases.

```
function a = asset(varargin)
% ASSET Constructor function for asset object
% a = asset(descriptor, current_value)
switch nargin
case 0
% if no input arguments, create a default object
    a.descriptor = 'none';
    a.date = date;
    a.type = 'none';
    a.current_value = 0;
    a = class(a,'asset');
case 1
% if single argument of class asset, return it
    if (isa(varargin{1},'asset'))
        a = varargin{1};
    else
        error('Wrong argument type')
    end
case 2
% create object using specified values
    a.descriptor = varargin{1};
    a.date = date;
    a.type = varargin{2};
    a.current_value = varargin{3};
    a = class(a,'asset');
otherwise
    error('Wrong number of input arguments')
end
```

The function uses a switch statement to accommodate three possible scenarios:

- Called with no arguments, the constructor returns a default asset object.
- Called with one argument that is an asset object, the object is simply returned.
- Called with two arguments (subclass descriptor, and current value), the constructor returns a new asset object.

The asset constructor method is not intended to be called directly; it is called from the child constructors since its purpose is to provide storage for common data.

The Asset get Method

The asset class needs methods to access the data contained in asset objects. The following function implements a get method for the class. It uses capitalized property names rather than literal field names to provide an interface similar to other MATLAB objects.

```
function val = get(a,prop_name)
% GET Get asset properties from the specified object
% and return the value
switch prop_name
case 'Descriptor'
   val = a.descriptor;
case 'Date'
   val = a.date;
case 'CurrentValue'
   val = a.current_value;
otherwise
   error([prop_name,' Is not a valid asset property'])
end
```

This function accepts an object and a property name and uses a switch statement to determine which field to access. This method is called by the subclass get methods when accessing the data in the inherited properties. See "The Stock get Method" on page 21-47 for an example.

The Asset set Method

The asset class set method is called by subclass set methods. This method accepts an asset object and variable length argument list of property name/property value pairs and returns the modified object.

```
function a = set(a,varargin)
% SET Set asset properties and return the updated object
property_argin = varargin;
while length(property_argin) >= 2,
   prop = property_argin{1};
   val = property_argin{2};
```

```
            property_argin = property_argin(3:end);
            switch prop
            case 'Descriptor'
                a.descriptor = val;
            case 'Date'
                a.date = val;
            case 'CurrentValue'
                a.current_value = val;
            otherwise
                error('Asset properties: Descriptor, Date, CurrentValue')
            end
    end
```

Subclass set methods call the asset set method and require the capability to return the modified object since MATLAB does not support passing arguments by reference. See "The Stock set Method" on page 21-48 for an example.

The Asset subref Method

The subsref method provides access to the data contained in an asset object using one-based numeric indexing and structure field name indexing. The outer switch statement determines if the index is a numeric or field name syntax. The inner switch statements map the index to the appropriate value.

MATLAB calls subsref whenever you make a subscripted reference to an object (e.g., A(i), A{i}, or A.*fieldname*).

```
function b = subsref(a,index)
%SUBSREF Define field name indexing for asset objects
switch index.type
case '()'
    switch index.subs{:}
    case 1
        b = a.descriptor;
    case 2
        b = a.date;
    case 3
        b = a.current_value;
    otherwise
        error('Index out of range')
    end
```

```
case '.'
   switch index.subs
   case 'descriptor'
       b = a.descriptor;
   case 'date'
       b = a.date;
   case 'current_value'
       b = a.current_value;
   otherwise
       error('Invalid field name')
   end
case '{}'
   error('Cell array indexing not supported by asset objects')
end
```

See the "The Stock subsref Method" on page 21-49 for an example of how the child subsref method calls the parent subsref method.

The Asset subsasgn Method

The subsasgn method is the assignment equivalent of the subsref method. This version enables you to change the data contained in an object using one-based numeric indexing and structure field name indexing. The outer switch statement determines if the index is a numeric or field name syntax. The inner switch statements map the index value to the appropriate value in the stock structure.

MATLAB calls subsasgn whenever you execute an assignment statement (e.g., A(i) = val, A{i} = val, or A.*fieldname* = val).

```
function a = subsasgn(a,index,val)
% SUBSASGN Define index assignment for asset objects
switch index.type
case '()'
   switch index.subs{:}
   case 1
       a.descriptor = val;
   case 2
       a.date = val;
   case 3
       a.current_value = val;
```

```
            otherwise
                error('Index out of range')
            end
        case '.'
            switch index.subs
            case 'descriptor'
                a.descriptor = val;
            case 'date'
                a.date = val;
            case 'current_value'
                a.current_value = val;
            otherwise
                error('Invalid field name')
            end
        end
```

The subsasgn method enables you to assign values to the asset object data structure using two techniques. For example, suppose you have a child stock object s.

```
    s = stock('XYZ',100,25);
```

Within stock class methods, you could change the descriptor field with either of the following statements

```
    s.asset(1) = 'ABC';
```

or

```
    s.asset.descriptor = 'ABC';
```

See the "The Stock subsasgn Method" on page 21-50 for an example of how the child subsasgn method calls the parent subsasgn method.

The Asset display Method

The asset display method is designed to be called from child-class display methods. Its purpose is to display the data it stores for the child object. The method simply formats the data for display in a way that is consistent with the formatting of the child's display method.

```
function display(a)
% DISPLAY(a) Display an asset object
stg = sprintf(...
    'Descriptor: %s\nDate: %s\nType: %s\nCurrent Value:%9.2f',...
    a.descriptor,a.date,a.current_value);
disp(stg)
```

The stock class display method can now call this method to display the data stored in the parent class. This approach isolates the stock display method from changes to the asset class. See "The Stock display Method" on page 21-52 for an example of how this method is called.

The Asset fieldcount Method

The asset fieldcount method returns the number of fields in the asset object data structure. fieldcount enables asset child methods to determine the number of fields in the asset object during execution, rather than requiring the child methods to have knowledge of the asset class. This allows you to make changes to the number of fields in the asset class data structure without having to change child-class methods.

```
function num_fields = fieldcount(asset_obj)
% Determines the number of fields in an asset object
% Used by asset child class methods
num_fields = length(fieldnames(asset_obj));
```

The struct function converts an object to its equivalent data structure, enabling access to the structure's contents.

Designing the Stock Class

A stock object is designed to represent one particular asset in a person's investment portfolio. This object contains two properties of its own and inherits three properties from its parent asset object.

Stock properties:

- NumberShares – The number of shares for the particular stock object.
- SharePrice – The value of each share.

Asset properties:

- `Descriptor` – The identifier of the particular asset (e.g., stock name, savings account number, etc.).
- `Date` – The date the object was created (calculated by the `date` command).
- `CurrentValue` – The current value of the asset.

Note that the property names are not actually the same as the field names of the structure array used internally by stock and asset objects. The property name interface is controlled by the stock and asset `set` and `get` methods and is designed to resemble the interface of other MATLAB object properties.

The `asset` field in the stock object structure contains the parent asset object and is used to access the inherited fields in the parent structure.

Stock Class Methods

The stock class implements the following methods:

- Constructor
- `get` and `set`
- `subsref` and `subsasgn`
- `display`

The Stock Constructor Method

The stock constructor creates a stock object from three input arguments:

- The stock name
- The number of shares
- The share price

The constructor must create an asset object from within the stock constructor to be able to specify it as a parent to the stock object. The stock constructor must, therefore, call the asset constructor. The `class` function, which is called to create the stock object, defines the asset object as the parent.

Keep in mind that the asset object is created in the temporary workspace of the stock constructor function and is stored as a field (`.asset`) in the stock structure. The stock object inherits the asset fields, but the asset object is not returned to the base workspace.

21-45

```
function s = stock(varargin)
% STOCK Stock class constructor.
% s = stock(descriptor, num_shares, share_price)
switch nargin
case 0
% if no input arguments, create a default object
   s.num_shares = 0;
   s.share_price = 0;
   a = asset('none',0);
   s = class(s, 'stock',a);
case 1
% if single argument of class stock, return it
   if (isa(varargin{1},'stock'))
       s = varargin{1};
   else
       error('Input argument is not a stock object')
   end
case 3
% create object using specified values
   s.num_shares = varargin{2};
   s.share_price = varargin{3};
   a = asset(varargin{1},'stock',varargin{2} * varargin{3});
   s = class(s,'stock',a);
otherwise
   error('Wrong number of input arguments')
end
```

Constructor Calling Syntax

The stock constructor method can be called in one of three ways:

- No Input Argument – If called with no arguments, the constructor returns a default object with empty fields.
- Input Argument is a Stock Object – If called with a single input argument that is a stock object, the constructor returns the input argument. A single argument that is not a stock object generates an error.
- Three Input Arguments – If there are three input arguments, the constructor uses them to define the stock object.

Otherwise, if none of the above three conditions are met, return an error.

For example, this statement creates a stock object to record the ownership of 100 shares of XYZ corporation stocks with a price per share of 25 dollars.

```
XYZ_stock = stock('XYZ',100,25);
```

The Stock get Method

The get method provides a way to access the data in the stock object using a "property name" style interface, similar to Handle Graphics. While in this example the property names are similar to the structure field name, they can be quite different. You could also choose to exclude certain fields from access via the get method or return the data from the same field for a variety of property names, if such behavior suits your design.

```
function val = get(s,prop_name)
% GET Get stock property from the specified object
% and return the value. Property names are: NumberShares
% SharePrice, Descriptor, Date, CurrentValue
switch prop_name
case 'NumberShares'
   val = s.num_shares;
case 'SharePrice'
   val = s.share_price;
case 'Descriptor'
   val = get(s.asset,'Descriptor'); % call asset get method
case 'Date'
   val = get(s.asset,'Date');
case 'CurrentValue'
   val = get(s.asset,'CurrentValue');
otherwise
   error([prop_name ,'Is not a valid stock property'])
end
```

Note that the asset object is accessed via the stock object's asset field (s.asset). MATLAB automatically creates this field when the class function is called with the parent argument.

21-47

The Stock set Method

The set method provides a "property name" interface like the get method. It is designed to update the number of shares, the share value, and the descriptor. The current value and the date are automatically updated.

```
function s = set(s,varargin)
% SET Set stock properties to the specified values
% and return the updated object
property_argin = varargin;
while length(property_argin) >= 2,
   prop = property_argin{1};
   val = property_argin{2};
   property_argin = property_argin(3:end);
   switch prop
   case 'NumberShares'
       s.num_shares = val;
   case 'SharePrice'
       s.share_price = val;
   case 'Descriptor'
       s.asset = set(s.asset,'Descriptor',val);
   otherwise
       error('Invalid property')
   end
end
s.asset = set(s.asset,'CurrentValue',...
            s.num_shares * s.share_price,'Date',date);
```

Note that this function creates and returns a new stock object with the new values, which you then copy over the old value. For example, given the stock object,

```
s = stock('XYZ',100,25);
```

the following set command updates the share price.

```
s = set(s,'SharePrice',36);
```

It is necessary to copy over the original stock object (i.e., assign the output to s) because MATLAB does not support passing arguments by reference. Hence the set method actually operates on a copy of the object.

The Stock subsref Method

The subsref method defines subscripted indexing for the stock class. In this example, subsref is implemented to enable numeric and structure field name indexing of stock objects.

```
function b = subsref(s,index)
% SUBSREF Define field name indexing for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
    if (index.subs{:} <= fc)
        b = subsref(s.asset,index);
    else
        switch index.subs{:} − fc
        case 1
            b = s.num_shares;
        case 2
            b = s.share_price;
        otherwise
        error(['Index must be in the range 1 to ',num2str(fc + 2)])
        end
    end
case '.'
    switch index.subs
    case 'num_shares'
        b = s.num_shares;
    case 'share_price'
        b = s.share_price;
    otherwise
        b = subsref(s.asset,index);
    end
end
```

The outer switch statement determines if the index is a numeric or field name syntax.

The fieldcount asset method determines how many fields there are in the asset structure, and the if statement calls the asset subsref method for indices 1 to fieldcount. See "The Asset fieldcount Method" on page 21-44 and "The Asset subsref Method" on page 21-41 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner `switch` statement, which maps the index value to the appropriate field in the stock structure.

Field-name indexing assumes field names other than `num_shares` and `share_price` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset `subsref` method performs field-name error checking.

See the `subsref` help entry for general information on implementing this method.

The Stock subsasgn Method

The `subsasgn` method enables you to change the data contained in a stock object using numeric indexing and structure field name indexing. MATLAB calls `subsasgn` whenever you execute an assignment statement (e.g., `A(i) = val`, `A{i} = val`, or `A.fieldname = val`).

```
function s = subsasgn(s,index,val)
% SUBSASGN Define index assignment for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
    if (index.subs{:} <= fc)
        s.asset = subsasgn(s.asset,index,val);
    else
        switch index.subs{:}-fc
        case 1
            s.num_shares = val;
        case 2
            s.share_price = val;
        otherwise
        error(['Index must be in the range 1 to ',num2str(fc + 2)])
        end
    end
case '.'
    switch index.subs
    case 'num_shares'
        s.num_shares = val;
```

```
        case 'share_price'
            s.share_price = val;
        otherwise
            s.asset = subsasgn(s.asset,index,val);
        end
    end
```

The outer `switch` statement determines if the index is a numeric or field name syntax.

The `fieldcount` asset method determines how many fields there are in the asset structure and the `if` statement calls the asset `subsasgn` method for indices 1 to `fieldcount`. See "The Asset fieldcount Method" on page 21-44 and "The Asset subsasgn Method" on page 21-42 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner `switch` statement, which maps the index value to the appropriate field in the stock structure.

Field-name indexing assumes field names other than `num_shares` and `share_price` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset `subsasgn` method performs field-name error checking.

The `subsasgn` method enables you to assign values to stock object data structure using two techniques. For example, suppose you have a stock object

```
s = stock('XYZ',100,25)
```

You could change the `descriptor` field with either of the following statements

```
s(1) = 'ABC';
```

or

```
s.descriptor = 'ABC';
```

See the `subsasgn` help entry for general information on assignment statements in MATLAB.

The Stock display Method

When you issue the statement (without terminating with a semicolon)

```
XYZStock = stock('XYZ',100,25)
```

MATLAB looks for a method in the @stock directory called display. The display method for the stock class produces this output.

```
Descriptor: XYZ
Date: 17-Nov-1998
Type: stock
Current Value:  2500.00
Number of shares: 100
Share price: 25.00
```

Here is the stock display method.

```
function display(s)
% DISPLAY(s) Display a stock object
display(s.asset)
stg = sprintf('Number of shares: %g\nShare price: %3.2f\n',...
    s.num_shares,s.share_price);
disp(stg)
```

First, the parent asset object is passed to the asset display method to display its fields (MATLAB calls the asset display method because the input argument is an asset object). The stock object's fields are displayed in a similar way using a formatted text string.

Note that if you did not implement a stock class display method, MATLAB would call the asset display method. This would work, but would display only the descriptor, date, type, and current value.

Example — The Portfolio Container

Aggregation is the containment of one class by another class. The basic relationship is: each contained class "is a part of" the container class.

For example, consider a financial portfolio class as a container for a set of assets (stocks, bonds, savings, etc.). Once the individual assets are grouped, they can be analyzed, and useful information can be returned. The contained objects are not accessible directly, but only via the portfolio class methods.

See "Example - Assets and Asset Subclasses" on page 21-37 for information about the assets collected by this portfolio class.

Designing the Portfolio Class

The portfolio class is designed to contain the various assets owned by a given individual and provide information about the status of his or her investment portfolio. This example implements a somewhat over-simplified portfolio class that:

- Contains an individual's assets
- Displays information about the portfolio contents
- Displays a 3-D pie chart showing the relative mix of asset types in the portfolio

Required Portfolio Methods

The portfolio class implements only three methods:

- `portfolio` – The portfolio constructor.
- `display` – Displays information about the portfolio contents.
- `pie3` – Overloaded version of `pie3` function designed to take a single portfolio object as an argument.

Since a portfolio object contains other objects, the portfolio class methods can use the methods of the contained objects. For example, the portfolio `display` method calls the stock class `display` method, and so on.

The Portfolio Constructor Method

The portfolio constructor method takes as input arguments a client's name and a variable length list of asset subclass objects (stock, bond, and savings objects in this example). The portfolio object uses a structure array with the following fields:

- name – The client's name.
- ind_assets – The array of asset subclass objects (stock, bond, savings).
- total_value – The total value of all assets. The constructor calculates this value from the objects passed in as arguments.
- account_number – The account number. This field is assigned a value only when you save a portfolio object (see "Saving and Loading Objects" on page 21-59).

```
function p = portfolio(name,varargin)
% PORTFOLIO Create a portfolio object containing the
% client's name and a list of assets
switch nargin
case 0
   % if no input arguments, create a default object
   p.name = 'none';
   p.total_value = 0;
   p.ind_assets = {};
   p.account_number = '';
   p = class(p,'portfolio');
case 1
   % if single argument of class portfolio, return it
   if isa(name,'portfolio')
       p = name;
   else
       disp([inputname(1) ' is not a portfolio object'])
       return
   end
otherwise
   % create object using specified arguments
   p.name = name;
   p.total_value = 0;
```

```
        for k = 1:length(varargin)
            p.ind_assets(k) = {varargin{k}};
            asset_value = get(p.ind_assets{k},'CurrentValue');
            p.total_value = p.total_value + asset_value;
        end
        p.account_number = '';
        p = class(p,'portfolio');
    end
```

Constructor Calling Syntax

The portfolio constructor method can be called in one of three different ways:

- No input arguments – If called with no arguments, it returns an object with empty fields.

- Input argument is an object – If the input argument is already a portfolio object, MATLAB returns the input argument. The isa function checks for this case.

- More than two input arguments – If there are more than two input arguments, the constructor assumes the first is the client's name and the rest are asset subclass objects. A more thorough implementation would perform more careful input argument checking, for example, using the isa function to determine if the arguments are the correct class of objects.

The Portfolio display Method

The portfolio display method lists the contents of each contained object by calling the object's display method. It then lists the client name and total asset value.

```
function display(p)
% DISPLAY Display a portfolio object
for k = 1:length(p.ind_assets)
    display(p.ind_assets{k})
end
stg = sprintf('\nAssets for Client: %s\nTotal Value: %9.2f\n',...
p.name,p.total_value);
disp(stg)
```

The Portfolio pie3 Method

The portfolio class overloads the MATLAB `pie3` function to accept a portfolio object and display a 3-D pie chart illustrating the relative asset mix of the client's portfolio. MATLAB calls the `@portfolio/pie3.m` version of `pie3` whenever the input argument is a single portfolio object.

```
function pie3(p)
% PIE3 Create a 3-D pie chart of a portfolio
stock_amt = 0; bond_amt = 0; savings_amt = 0;
for k = 1:length(p.ind_assets)
   if isa(p.ind_assets{k},'stock')
      stock_amt = stock_amt + ...
         get(p.ind_assets{k},'CurrentValue');
   elseif isa(p.ind_assets{k},'bond')
      bond_amt = bond_amt + ...
         get(p.ind_assets{k},'CurrentValue');
   elseif isa(p.ind_assets{k},'savings')
      savings_amt = savings_amt + ...
         get(p.ind_assets{k},'CurrentValue');
   end
end
i = 1;
if stock_amt ~= 0
   label(i) = {'Stocks'};
   pie_vector(i) = stock_amt;
   i = i +1;
end
if bond_amt ~= 0
   label(i) = {'Bonds'};
   pie_vector(i) = bond_amt;
   i = i +1;
end
if savings_amt ~= 0
   label(i) = {'Savings'};
   pie_vector(i) = savings_amt;
end
pie3(pie_vector,label)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'Type','Text'),'FontSize',14)
cm = gray(64);
```

```
colormap(cm(48:end,:))
stg(1) = {['Portfolio Composition for ',p.name]};
stg(2) = {['Total Value of Assets: $',num2str(p.total_value)]};
title(stg,'FontSize',12)
```

There are three parts in the overloaded `pie3` method.

- The first uses the asset subclass `get` methods to access the `CurrentValue` property of each contained object. The total value of each class is summed.
- The second part creates the pie chart labels and builds a vector of graph data, depending on which objects are present.
- The third part calls the MATLAB `pie3` function, makes some font and colormap adjustments, and adds a title.

Creating a Portfolio

Suppose you have implemented a collection of asset subclasses in a manner similar to the stock class. You can then use a portfolio object to present the individual's financial portfolio. For example, given the following assets

```
XYZStock = stock('XYZ',200,12);
SaveAccount = savings('Acc # 1234',2000,3.2);
Bonds = bond('U.S. Treasury',1600,12);
```

Create a portfolio object.

```
p = portfolio('Gilbert Bates',XYZStock,SaveAccount,Bonds)
```

The portfolio `display` method summarizes the portfolio contents (because this statement is not terminated by a semicolon).

```
Descriptor: XYZ
Date: 24-Nov-1998
Current Value:   2400.00
Type: stock
Number of shares: 200
Share price: 12.00

Descriptor: Acc # 1234
Date: 24-Nov-1998
Current Value:   2000.00
Type: savings
Interest Rate: 3.2%
```

```
Descriptor: U.S. Treasury
Date: 24-Nov-1998
Current Value:   1600.00
Type: bond
Interest Rate: 12%

Assets for Client: Gilbert Bates
Total Value:   6000.00
```

The portfolio `pie3` method displays the relative mix of assets using a pie chart.

```
pie3(p)
```


Portfolio Composition for Gilbert Bates
Total Value of Assets: $6000

Saving and Loading Objects

You can use the MATLAB `save` and `load` commands to save and retrieve user-defined objects to and from `.mat` files, just like any other variables.

When you load objects, MATLAB calls the object's class constructor to register the object in the workspace. The constructor function for the object class you are loading must be able to be called with no input arguments and return a default object. See "Guidelines for Writing a Constructor" on page 21-9 for more information.

Modifying Objects During Save or Load

When you issue a `save` or `load` command on objects, MATLAB looks for class methods called `saveobj` and `loadobj` in the class directory. You can overload these methods to modify the object before the save or load operation. For example, you could define a `saveobj` method that saves related data along with the object or you could write a `loadobj` method that updates objects to a newer version when this type of object is loaded into the MATLAB workspace.

Example — Defining saveobj and loadobj for Portfolio

In the section "Example — The Portfolio Container" on page 21-53, portfolio objects are used to collect information about a client's investment portfolio. Now suppose you decide to add an account number to each portfolio object that is saved. You can define a portfolio saveobj method to carry out this task automatically during the save operation.

Suppose further that you have already saved a number of portfolio objects without the account number. You want to update these objects during the load operation so that they are still valid portfolio objects. You can do this by defining a loadobj method for the portfolio class.

Summary of Code Changes

To implement the account number scenario, you need to add or change the following functions:

- portfolio – The portfolio constructor method needs to be modified to create a new field, account_number, which is initialized to the empty string when an object is created.
- saveobj – A new portfolio method designed to add an account number to a portfolio object during the save operation, only if the object does not already have one.
- loadobj – A new portfolio method designed to update older versions of portfolio objects that were saved before the account number structure field was added.
- subsref – A new portfolio method that enables subscripted reference to portfolio objects outside of a portfolio method.
- getAccountNumber – a MATLAB function that returns an account number that consists of the first three letters of the client's name.

New Portfolio Class Behavior

With the additions and changes made in this example, the portfolio class now:

- Includes a field for an account number
- Adds the account number when a portfolio object is saved for the first time

- Automatically updates the older version of portfolio objects when you load them into the MATLAB workspace

The saveobj Method

MATLAB looks for the portfolio saveobj method whenever the save command is passed a portfolio object. If @portfolio/saveobj exists, MATLAB passes the portfolio object to saveobj, which must then return the modified object as an output argument. The following implementation of saveobj determines if the object has already been assigned an account number from a previous save operation. If not, saveobj calls getAccountNumber to obtain the number and assigns it to the account_number field.

```
function b = saveobj(a)
if isempty(a.account_number)
    a.account_number = getAccountNumber(a);
end
b = a;
```

The loadobj Method

MATLAB looks for the portfolio loadobj method whenever the load command detects portfolio objects in the .mat file being loaded. If loadobj exists, MATLAB passes the portfolio object to loadobj, which must then return the modified object as an output argument. The output argument is then loaded into the workspace.

If the input object does not match the current definition as specified by the constructor function, then MATLAB converts it to a structure containing the same fields and the object's structure with all the values intact (that is, you now have a structure, not an object).

The following implementation of loadobj first uses isa to determine whether the input argument is a portfolio object or a structure. If the input is an object, it is simply returned since no modifications are necessary. If the input argument has been converted to a structure by MATLAB, then the new account_number field is added to the structure and is used to create an updated portfolio object.

```
function b = loadobj(a)
% loadobj for portfolio class
if isa(a,'portfolio')
```

```
    b = a;
else % a is an old version
    a.account_number = getAccountNumber(a);
    b = class(a,'portfolio');
end
```

Changing the Portfolio Constructor

The portfolio structure array needs an additional field to accommodate the account number. To create this field, add the line

```
p.account_number = '';
```

to @portfolio/portfolio.m in both the zero argument and variable argument sections.

The getAccountNumber Function

In this example, getAccountNumber is a MATLAB function that returns an account number composed of the first three letters of the client name prepended to a series of digits. To illustrate implementation techniques, getAccountNumber is not a portfolio method so it cannot access the portfolio object data directly. Therefore, it is necessary to define a portfolio subsref method that enables access to the name field in a portfolio object's structure.

For this example, getAccountNumber simply generates a random number, which is formatted and concatenated with elements 1 to 3 from the portfolio name field.

```
function n = getAccountNumber(p)
% provides a account number for object p
n = [upper(p.name(1:3)) strcat(num2str(round(rand(1,7)*10)))')'];
```

Note that the portfolio object is indexed by field name, and then by numerical subscript to extract the first three letters. The subsref method must be written to support this form of subscripted reference.

The Portfolio subsref Method

When MATLAB encounters a subscripted reference, such as that made in the getAccountNumber function

```
p.name(1:3)
```

MATLAB calls the portfolio subsref method to interpret the reference. If you do not define a subsref method, the above statement is undefined for portfolio objects (recall that here p is an object, not just a structure).

The portfolio subsref method must support field-name and numeric indexing for the getAccountNumber function to access the portfolio name field.

```
function b = subsref(p,index)
% SUBSREF Define field name indexing for portfolio objects
switch index(1).type
case '.'
    switch index(1).subs
    case 'name'
        if length(index)== 1
            b = p.name;
        else
            switch index(2).type
            case '()'
                b = p.name(index(2).subs{:});
            end
        end
    end
end
```

Note that the portfolio implementation of subsref is designed to provide access to specific elements of the name field; it is not a general implementation that provides access to all structure data, such as the stock class implementation of subsref.

See the subsref help entry for more information about indexing and objects.

Object Precedence

Object precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

```
objectA + objectB
```

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the `inferiorto` and `superiorto` functions.

For example, in the section "Example — A Polynomial Class" on page 21-23 the polynom class defines a `plus` method that enables addition of polynom objects. Given the polynom object p

```
p = polynom([1 0 –2 –5])
p =
    x^3–2*x–5
```

The expression,

```
1 + p
ans =
    x^3–2*x–4
```

calls the polynom `plus` method (which converts the double, 1, to a polynom object, and then adds it to p). The user-defined polynom class has precedence over the MATLAB double class.

Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferiorto` or `superiorto` function in the class constructor.

The `inferiorto` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferiorto` function is

```
inferiorto('class1','class2',...)
```

You can specify multiple classes in the argument list, placing the class below many other classes in the hierarchy.

Similarly, the `superiorto` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superiorto` function is

```
superiorto('class1','class2',...)
```

Location in the Hierarchy

If *objectA* is above *objectB* in the precedence hierarchy, then the expression

```
objectA + objectB
```

calls @*classA*/plus.m. Conversely, if *objectB* is above *objectA* in the precedence hierarchy, then MATLAB calls @*classB*/plus.m.

See "How MATLAB Determines Which Method to Call" on page 21-66 for related information.

How MATLAB Determines Which Method to Call

In MATLAB, functions exist in directories in the computer's file system. A directory may contain many functions (M-files). Function names are unique only within a single directory (e.g., more than one directory may contain a function called pie3). When you type a function name on the command line, MATLAB must search all the directories it is aware of to determine which function to call. This list of directories is called the *MATLAB path*.

When looking for a function, MATLAB searches the directories in the order they are listed in the path, and calls the first function whose name matches the name of the specified function.

If you write an M-file called pie3.m and put it in a directory that is searched before the specgraph directory that contains the MATLAB pie3 function, then MATLAB uses your pie3 function instead (note that this is not true for built-in functions like plot, which are always found first).

Object-oriented programming allows you to have many methods (MATLAB functions located in class directories) with the same name and enables MATLAB to determine which method to use based on the type or class of the variables passed to the function. For example, if p is a portfolio object, then

```
pie3(p)
```

calls @portfolio/pie3.m because the argument is a portfolio object.

Selecting a Method

When you call a method for which there are multiple versions with the same name, MATLAB determines the method to call by:

- Looking at the classes of the objects in the argument list to determine which argument has the highest object precedence; the class of this object controls the method selection and is called the *dispatch type*.
- Applying the *function precedence order* to determine which of possibly several implementations of a method to call. This order is determined by the location and type of function.

Determining the Dispatch Type

MATLAB first determines which argument controls the method selection. The class type of this argument then determines the class in which MATLAB searches for the method. The controlling argument is either:

- The argument with the highest precedence, or
- The leftmost of arguments having equal precedence

User-defined objects take precedence over the MATLAB built-in classes such as double or char. You can set the relative precedence of user-defined objects with the inferiorto and superiorto functions, as described in "Object Precedence" on page 21-64.

MATLAB searches for functions by name. When you call a function, MATLAB knows the name, number of arguments, and the type of each argument. MATLAB uses the dispatch type to choose among multiple functions of the same name, but does not consider the number of arguments.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. From the perspective of method selection, MATLAB contains two types of functions: those built into MATLAB, and those written as M-files. MATLAB treats these types differently when determining the function precedence order.

MATLAB selects the correct function for a given context by applying the following function precedence rules, in the order given.

For built-in functions:

1 Overloaded Methods

   If there is a method in the class directory of the dispatching argument that has the same name as a MATLAB built-in function, then this method is called instead of the built-in function.

2 Nonoverloaded MATLAB Functions

   If there is no overloaded method, then the MATLAB built-in function is called.

MATLAB built-in functions take precedence over both subfunctions and private functions. Therefore, subfunctions or private functions with the same name as MATLAB built-in functions can never be called.

For nonbuilt-in functions:

1 Subfunctions

Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

2 Private Functions

Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

3 Class Constructor Functions

Constructor functions (functions having names that are the same as the @ directory, for example @polynom/polynom.m) take precedence over other MATLAB functions. Therefore, if you create an M-file called polynom.m and put it on your path before the constructor @polynom/polynom.m version, MATLAB will always call the constructor version.

4 Overloaded Methods

MATLAB calls an overloaded method if it is not masked by a subfunction or private function.

5 Current Directory

A function in the current working directory is selected before one elsewhere on the path.

6 Elsewhere On Path

Finally, a function anywhere else on the path is selected.

Selecting Methods from Multiple Directories

There may be a number of directories on the path that contain methods with the same name. MATLAB stops searching when it finds the first implementation of the method on the path, regardless of the implementation type (MEX-file, P-code, M-file).

Selecting Methods from Multiple Implementation Types

There are four file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is:

1 MEX-files

2 MDL-file (Simulink model)

3 P-code

4 M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Method MATLAB Will Call

You can determine which method MATLAB will call using the `which` command. For example,

```
which pie3
your_matlab_path/toolbox/matlab/specgraph/pie3.m
```

However, if p is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `–all` option. See the `which` reference page for more information on this command.

21-69

22

Maximizing MATLAB Performance

This chapter describes techniques that often improve the execution speed and memory management of MATLAB code. It covers the following topics:

Techniques for Improving Performance

This section covers the following topics:

- "Analyzing Your Program's Performance"
- "Vectorizing Loops" on page 22-3
- "Preallocating Arrays" on page 22-6
- "Other Ways to Speed Up Performance" on page 22-7

Analyzing Your Program's Performance

The M-file Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement.

The M-File Profiler

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

MATLAB provides the M-file Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type profile viewer or select **View -> Profiler** in the MATLAB Command Window. See "Measuring Performance" on page 22-29 for more information.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, tic and toc. Invoking tic starts the timer, and the first subsequent toc stops it and reports the time elapsed between the two.

Use tic and toc as shown here.

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

Measuring Smaller Programs. Shorter programs 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
```

Vectorizing Loops

MATLAB is a matrix language, which means it is designed for vector and matrix operations. You can often speed up your M-file code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Note Before taking the time to vectorize your code, read the section on "Performance Acceleration" on page 22-8. You may be able to speed up your program by just as much using the MATLAB JIT Accelerator instead of vectorizing.

A Simple Example

Here is one way to compute the sine of 1001 values ranging from 0 to 10.

```
i = 0;
for t = 0:.01:10
    i = i+1;
    y(i) = sin(t);
end
```

A vectorized version of the same code is:

```
t = 0:.01:10;
y = sin(t);
```

The second example executes much faster than the first and is the way
MATLAB is meant to be used. Test this on your system by creating M-file
scripts that contain the code shown, then using the `tic` and `toc` functions to
time the M-files.

An Advanced Example

`repmat` is an example of a function that takes advantage of vectorization. It
accepts three input arguments: an array `A`, a row dimension `M`, and a column
dimension `N`.

`repmat` creates an output array that contains the elements of array `A`,
replicated and "tiled" in an `M`-by-`N` arrangement.

```
A = [1 2 3; 4 5 6];

B = repmat(A,2,3);
B =
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
```

`repmat` uses vectorization to create the indices that place elements in the
output array.

```
function B = repmat(A,M,N)
if nargin < 2
   error('Requires at least 2 inputs.')
elseif nargin == 2
   N = M;
end

% Step 1 Get row and column sizes
[m,n] = size(A);

% Step 2 Generate vectors of indices from 1 to row/column size
mind = (1:m)';
nind = (1:n)';
```

```
% Step 3 Creates index matrices from vectors above
mind = mind(:,ones(1,M));
nind = nind(:,ones(1,N));

% Step 4 Create output array
B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. mind contains the integers from 1 through the row size of A. The nind variable contains the integers from 1 through the column size of A.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,n_cols))
```

where n_cols is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, mind, is paired with each element of the column index array, nind, using the following procedure:

1 The first element of mind, the row index, is paired with each element of nind. MATLAB moves through the nind matrix in a columnwise fashion, so mind(1,1) goes with nind(1,1), then nind(2,1), and so on. The result fills the first row of the output array.

2 Moving columnwise through mind, each element is paired with the elements of nind as above. Each complete pass through the nind matrix fills one row of the output array.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are

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

Preallocating Arrays

You can often improve code execution time by preallocating the arrays that store output results. Preallocation makes it unnecessary for MATLAB to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type	Function	Examples
Numeric array	zeros	y = zeros(1,100);
Cell array	cell	B = cell(2,3); B{1,3} = 1:3; B{2,2} = 'string';
Structure array	struct, repmat	data = repmat(struct('x',[1 3],... 'y',[5 6]), 1, 3);

Preallocation also helps reduce memory fragmentation if you work with large matrices. In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. This can result in plenty of free memory, but not enough contiguous space to hold a large variable. Preallocation helps prevent this by allowing MATLAB to "grab" sufficient space for large data constructs at the beginning of a computation.

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);
```

Use repmat When You Need to Enlarge Arrays

In cases where you cannot preallocate, see if you can increase the size of your array using the repmat function. repmat tries to get you a contiguous block of memory for your expanding array.

Other Ways to Speed Up Performance

Here are several other suggestions that may help you to get increased performance from your M-file programs. Also be sure to read "Performance Acceleration" on page 22-8.

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.

Functions Are Faster Than Scripts

Your code will execute 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.

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. The load and save functions are optimized to run faster and reduce memory fragmentation.

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.

Performance Acceleration

The MathWorks is committed to making MATLAB as fast as 3GL programming languages, like C and Fortran. Speeding up the execution of programs written in MATLAB is an ongoing endeavor that will be delivered over a number of product releases.

This section discusses MATLAB performance acceleration, a capability introduced in MATLAB 6.5 that is aimed at speeding up the processing of M-file functions and scripts. As acceleration is a work in progress, its effect is greater on some components of the MATLAB language than on others at this time. For some M-file programs, you will see a marked improvement, and on others, a lesser, if any, effect.

In MATLAB 6.5, the most significant performance improvement is in functions and scripts that spend most of their time in self-contained loops, particularly loops that make no calls to M-file functions. The accelerated loops often execute at least as fast as a vectorized version of the same loop. See "Sample Accelerated Programs" on page 22-15 for examples of such programs and comparative performance measurements.

The following sections explain how to write programs in MATLAB to maximize the benefit of performance acceleration. The first two sections, in particular, discuss which programming constructs to use and which to avoid to achieve optimal performance:

- "What MATLAB Accelerates" on page 22-9
- "What MATLAB Does Not Accelerate" on page 22-10
- "How Vectorizing and Preallocation Fit In" on page 22-12
- "What to Avoid When Running MATLAB" on page 22-13
- "Operating System Considerations" on page 22-14

Note The MATLAB Profiler has been enhanced to help you measure and optimize the performance benefits covered here. See "Measuring Performance" on page 22-29 for more information.

What MATLAB Accelerates

MATLAB performance enhancements are supported on a subset of the entire MATLAB language capability. This section defines the supported subset of data types, array shapes, and operations that execute faster.

Your programs will see the greatest speed improvement when large, contiguous portions of your code are restricted to only those elements of MATLAB that support performance acceleration. Whenever MATLAB encounters an unsupported element, (for example, one of the unaccelerated data types in the diagram below), it interrupts accelerated processing to handle the instruction through the non-accelerated interpreter. The more MATLAB has to do this, the greater price you pay in execution time.

Data Types

MATLAB accelerates code that uses the data types that are shaded in the class hierarchy diagram below, and does not accelerate those that are unshaded. Both real and complex doubles are accelerated. Only full arrays, not sparse, are accelerated.

Array Shapes

Performance acceleration applies to all MATLAB array shapes except for arrays having more than three dimensions.

for Loops

Loops controlled by a `for` statement execute faster in MATLAB as long as:

- Indices of the `for` loop are set to a range of scalar values.
- Code in the `for` loop uses only the supported data types and array shapes.
- If the code in the `for` loop calls any functions, they are built-in functions.

Loop performance is optimal when every line of code in the loop meets the requirements for acceleration. When this is the case, MATLAB speeds up execution of the entire loop, including the `for` and `end` statements. If this is not true, then acceleration of the loop is temporarily interrupted on each iteration of the loop to process the disqualifying lines, as well as the `for` and `end` statements.

Read the section on "What MATLAB Does Not Accelerate" on page 22-10 to see what practices you should avoid using in your `for` loop code.

Conditional Statements

`if`, `elseif`, `while`, and `switch` statements execute faster as long as the conditional expression in such statements evaluates to a scalar value.

Array Size

There is a certain amount of overhead work that MATLAB performs when manipulating arrays. For large arrays, this overhead represents only a small percentage of the time required to handle the data. For small matrices, however, the overhead is a greater percentage of the time spent and thus is more significant.

Performance acceleration reduces the amount of overhead required to handle arrays. You will see more of a speed up in programs that use smaller arrays.

What MATLAB Does Not Accelerate

The following programming elements and practices can slow MATLAB down and should be avoided when speed is the primary consideration.

Data Types and Array Shapes

Data types not supported for acceleration are shown as unshaded in the figure in the previous section. The only unaccelerated array shapes are arrays having more than three dimensions.

Function Calls

Calls to other functions (M-file or MEX) or subfunctions are costly in terms of performance and are not accelerated in MATLAB. The code that makes up the function being called may have improved performance, but the mechanism of making the call may reduce this benefit.

Functions That Overload MATLAB Built-Ins. Since the execution of an overloaded function involves making a function call to an M-file or MEX-file, overloading a MATLAB built-in function slows down its execution. For example, if you overload the eq built-in function for type char, then comparing characters for equality will run slower than when using the MATLAB built-in eq, which is accelerated for characters.

More than One Operation on a Line

Under certain circumstances, including more than one operation on a line in your program might slow it down. In the example shown here, the first operation involves a structure array, which is not supported by the performance accelerator.

```
x = a.name;    for k=1:10000, sin(A(k)), end;
```

Since MATLAB processes the code sequentially, an entire line at a time, any instructions that follow the disqualifying operation on the same line are not accelerated. This means that the for loop in this example would not get the performance enhancement it would have gotten had it been on a separate line.

Changing Data Types or Shapes of Variables

If your program changes the data type or array shape of an existing variable, MATLAB must temporarily stop its code acceleration process and handle the change. For example, this code changes the type for X from double to char, and thus slows the acceleration process.

```
X = 23;
   .
-accelerated code-
   .
X = 'A';                 % This line breaks accelerated region
   .
-more accelerated code-
```

Using Complex Constants in Scripts

In scripts, by default MATLAB interprets the terms i and j to be variables rather than the MATLAB complex constants i and j. If you use i and j as complex constants in scripts, you need to make it clear in the script that they are to be interpreted as such. If it is not clear, then MATLAB will treat them as undefined variables and, as a result, will not accelerate the statements in which they appear.

For example, MATLAB interprets i in the following statement to be a variable of undefined data type and array size. As such, MATLAB must temporarily stop the acceleration process to interpret the statement.

```
x = 7 + 2 * i
```

This next statement expresses the same equation in a way that cannot be misinterpreted. The phrase 2i only makes sense if i represents the imaginary constant. Expressed in this way, the statement does accelerate.

```
x = 7 + 2i
```

The same holds true for a simple statement such as x = i. If you express this as x = 1i, MATLAB accelerates the statement.

How Vectorizing and Preallocation Fit In

With many programs being automatically accelerated by MATLAB, how important is it to use other optimizing techniques discussed in this chapter, such as vectorizing and preallocation?

Vectorizing Loops

When processing code that contains loops, the MATLAB performance accelerator does the equivalent of implementing the loops with vectorized code. If you have already vectorized your programs, you will not see a significant performance improvement in these programs when run with the accelerator.

However, greater speed in executing loops means that you now have a choice of whether to vectorize your code or not. Either way, you will get about the same performance from MATLAB. Choose the form that is the most understandable or that best fits the application.

Preallocating Arrays

Preallocating arrays keeps MATLAB from having to repeatedly find and allocate contiguous storage as arrays expand in size during program execution. Therefore, it continues to be just as important with performance acceleration as in earlier versions without this feature.

Note If you have a loop in your code that should accelerate but does not show much improvement, check to see that you are using preallocation rather than growing your arrays on each iteration of the loop.

What to Avoid When Running MATLAB

Performance acceleration is enabled only when you run your M-Files outside of debug mode and with command echoing turned off.

M-File Debug

The following debug commands, entered at the command prompt or within an M-File, disable MATLAB from accelerating any subsequent M-File code:

- `dbstop if warning`
- `dbstop if naninf` (or `infnan`)
- `dbstop in mfile ...`

`dbstop if error` does not disable the performance accelerator.

The `dbstop in mfile` command only affects the M-file (or M-files) in which `dbstop` is set.

To turn off debug mode, and thus enable code acceleration, use the appropriate form of the `dbclear` function.

Command Echoing

Enabling echoing (with the `echo` function) for an M-file script or function disables performance acceleration for that script or function. Enabling echoing for all scripts or functions disables acceleration for all M-file scripts and functions.

Operating System Considerations

Depending on the degree to which your program code follows the acceleration guidelines described earlier in this section, your M-File programs should see a marked improvement in performance on all platforms. On Intel X86-based Linux and Windows systems, you are likely to see more of a performance gain than on other platforms.

Sample Accelerated Programs

The programs in this section should provide some insight into how to make the best use of the MATLAB performance enhancements described in the previous section on "Performance Acceleration" on page 22-8. Measurements taken on each program show a sizeable performance improvement over earlier versions of MATLAB.

The programs are:

- "Program 1 — Bayes' Rule" on page 22-16
- "Program 2 — Relaxation Algorithm" on page 22-19
- "Program 3a — Vector Comparison, with Loop" on page 22-22
- "Program 3b — Vector Comparison, Vectorized" on page 22-25
- "Program 4 — Tic-Tac-Toe" on page 22-26

The following table shows hardware specifications for the systems used in measuring performance on the program examples in this section.

Operating System	CPU Type	Processor Speed	Main Memory
Windows	Intel x86	1 GHz	256 MB
Linux	Pentium III	550 MHz	256 MB
Solaris	Sparc	270 MHz	128 MB

Program 1 — Bayes' Rule

This example uses a program that implements Bayes' rule for computing probability based on prior probability and updated information. The program contains nested `for` loops that, unless they were vectorized, would be quite costly in execution time without performance acceleration.

The table below compares the performance of MATLAB on this program in MATLAB 6.1 (with no acceleration) and in MATLAB 6.5 (with acceleration). The increase in performance is shown in the far right column for the three operating systems tested.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	16 min., 0.5 sec.	2.7 sec.	x 355.7
Linux	38 min., 15.8 sec.	5.9 sec.	x 389.1
Solaris	1 hr., 47 min., 32.7 sec.	57.9 sec.	x 111.4

Here is the `Bayes.m` program. Refer to the section "What Makes It Faster" on page 22-18 to see how this program was written to run at optimal speed.

```
function score = Bayes(Seq, Matrix, priorProbability)
% BAYES  Use Bayes' rule to determine signal probabilities.
%
%   score = BAYES(Seq, Matrix, priorProbability) is the
%   probability that the signal whose probability is
%   expressed in Matrix occurs at each position in Seq,
%   where priorProbability is the probability of seeing
%   the signal at any position.

% Using Bayes' rule:
% P(a|b) = P(b|a) * P(a) / P(b), where Pa is the probability
% that whatever signal we're looking for occurs at a given
% position, Pb|a is the probability that we would see this
% specific nucleotide if the signal were here, and Pb is the
% unconditional probability of seeing this nucleotide here.
```

```
% Initially, we'll assume each nt is equally likely.
Pb = 1/4;

% Initialize storage for the result.
score = zeros(1, length(Seq));

lm = length(Matrix);
ls = length(Seq) - length(Matrix);

for m = 1:ls
   Pa = priorProbability;
   k = m - 1;

   for n = 1:lm
      nt = Seq(k + n);
      if (nt > 0) && (nt < 5)
         PbGa = Matrix(nt, n);
         Pb = Pa * PbGa + (1 - Pa) * 0.25;
         Pa = PbGa * Pa / Pb;
      end
   end

   score(m) = Pa;
end
```

The times shown in the table above were recorded by running the Bayes program on data stored in a double matrix and a large 8-bit integer vector, with a priorProbability of 0.0001.

```
whos
  Name       Size                    Bytes  Class

  Matrix     4x20                      640  double array
  Seq        1x912211               912211  int8 array
```

What Makes It Faster

The program spends most of its time in a nested `for` loop that calculates the `P(b|a)`, `P(b)`, and `P(a)` values. When run with the data shown above, the inner loop executes over 18 million times. Because all of the code in the program uses elements of MATLAB that support acceleration, the entire program runs much faster.

Some of the reasons MATLAB accelerates this code are as follows.

Supported Data Types and Array Shapes. All of the statements within the inner and outer loops use `double` and 8-bit integer data in scalar, vector, and two-dimensional matrix form. These data types and array shapes support performance acceleration.

Scalar loop indices. Both `for` loops operate on a range of scalar values. For example, `ls` used in this statement is scalar.

```
for m = 1:ls
```

Function Calls and Overloading. Calling functions that are MATLAB built-ins costs very little time in execution, but calling functions implemented in M-files can use up considerable time. This program calls only built-in functions, like `zeros` and `length`, with no function calls at all in the nested loops. In addition to direct function calls, there are also no overloaded operations that implicitly call M-file functions.

Conditional Statements. The one conditional statement, residing in the inner loop, evaluates scalar terms. This statement uses the scalar, `nt`, as shown here.

```
if (nt > 0) && (nt < 5)
```

Small Array Size. The second argument, `Matrix`, is a 4-by-20 array. Arrays that are small in size execute more quickly with acceleration. The overhead of array handling, more noticeable with smaller arrays, is insignificant in accelerated versions of MATLAB.

The third argument, `Seq`, is quite large and does not speed up much due to its size alone.

Program 2 — Relaxation Algorithm

This program starts by creating a sharply contrasted graphics display in a figure window. When you press any key after starting the program, it runs a relaxation algorithm on the figure, gradually blurring the color boundaries.

When run on a MATLAB version that doesn't support performance acceleration, you can see the algorithm taking effect in distinct steps that are spaced over time. When run with acceleration, the display changes smoothly over a much shorter time period.

This table shows comparative execution times for 300 iterations of the program. (The version of this program used in these measurements is slightly different from what is shown below. The final image handling functions (set and drawnow) were moved to the outside of the three nested for loops, and thus refresh the image only once, at the very end.)

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	1 min., 25.9 sec.	1.1 sec.	x 78.1
Linux	3 min., 13.8 sec.	1.7 sec.	x 114.0
Solaris	8 min., 51.0 sec.	23.5 sec.	x 22.6

Here is the program.

```
function relax(iterations)

sz = 102;

plate = magic(sz) * 64 / (sz * sz);
newPlate = plate;

im = image(plate);
axis off
set(gcf, 'DoubleBuffer', 'on')

% Wait for user to press a key to kick off the image processing
pause
```

22-19

```
for i = 1:iterations
   for j = 2:(sz-1)
      jm1 = j - 1;
      jp1 = j + 1;
      for k = 2:(sz-1)
         km1 = k - 1;
         kp1 = k + 1;
         newPlate(j,k) = (plate(jm1,km1)/2 + plate(jm1,k) + ...
           plate(jm1,kp1)/2 + plate(j,km1) + plate(j,kp1) + ...
          plate(jp1,km1)/2 + plate(jp1,k) + plate(jp1,kp1)/2)/6;
      end
   end
   plate = newPlate;

% Refresh the image once every 5 times through the loop.
   if (0 == rem(i,5))
      set(im, 'cdata', plate)
      drawnow
   end

end
```

You can see the visual effect of the faster execution by running the program yourself. Put the code into an M-file named relax.m, and run it for 300 iterations by typing

```
relax(300);
```

A new window is created showing the initial image. Once you see this, reselect the MATLAB Command Window, and then press any key to start processing the image.

What Makes It Faster

The program spends most of its time in a nested for loop that modifies the image data. The newPlate calculation in the inner loop executes 3 million times when iterations is set to 300, as it is in the above test.

MATLAB accelerates the two inner for loops for the reasons explained below. The outer loop does not accelerate (see "The Outer for Loop" on page 22-21), yet that has little effect on the overall execution time, as nearly all of the time spent is in the inner loops.

Scalar loop indices. The `for` loops operate on a range of scalar values. For example,

```
for j = 2:(sz-1)
```

Supported Data Types and Array Shapes. All of the statements within the inner loops use a `double` data type with either a scalar value or two-dimensional matrix. These are among the data types and array shapes that MATLAB accelerates.

Higher Complexity Operations. MATLAB usually shows a noticeable performance gain for statements containing multiple operators and/or functions. The `plate` computation is an example of this.

```
newPlate(j,k) = (plate(jm1,km1)/2 + plate(jm1,k) + ...
   plate(jm1,kp1)/2 + plate(j,km1) + plate(j,kp1) + ...
   plate(jp1,km1)/2 + plate(jp1,k) + plate(jp1,kp1)/2)/6;
```

Function Calls and Overloading. One factor that enables the acceleration of the two inner loops is that the only function calls made in this code are to built-in functions. The program performs a number of mathematical operations, but as long as none of the math operators used (+, -, and /) is overloaded for the data type being operated on (`double`), these math operations execute quickly, not having to make M-file calls.

The Outer for Loop

MATLAB does not accelerate the outer `for` loop. One reason is that the leading `for` statement relies on an ambiguous data type for the maximum index value. The value for `iterations` is passed into the program and thus may not necessarily be one of the data types or array shapes supported for acceleration.

```
for i = 1:iterations
```

If you include the final `set` and `drawnow` calls in the loop, MATLAB does not accelerate these either, because they operate on Handle Graphics® objects, which are not among the data types supported for performance acceleration.

```
        set(im,'cdata',plate)
        drawnow
```

The fact that the outer loop does not accelerate is not that important in this case, as nearly all of the time spent is in the inner loops.

Program 3a — Vector Comparison, with Loop

This program scans two sorted, input vectors and finds the elements that are common to both. It returns the indices of these common elements.

There are two versions of this program. This version, "Program 3a", processes the vectors using a `while` loop. The version shown afterward, as "Program 3b", replaces the loop with vectorized code. When run on MATLAB without acceleration, there is a big difference in performance between the two. When run with acceleration, there is no significant difference at all.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	10.6 sec.	0.1 sec.	x 106.0
Linux	24.0 sec.	0.1 sec.	x 240.0
Solaris	1 min., 4.7 sec.	1.5 sec.	x 43.1

```
function [aIndex, bIndex] = vfind_scalar(avec, bvec)

avecLen = length(avec);
bvecLen = length(bvec);

% Size aIndex and bIndex to be large enough
outlen = min(avecLen, bvecLen);
aIndex = zeros(outlen,1);
bIndex = zeros(outlen,1);

n  = 0;
ai = 1;
bi = 1;

while (ai <= avecLen || bi <= bvecLen)
   % Get vector elements at indices ai and bi
   A = avec(ai);
   B = bvec(bi);
```

```
      % If equal, record indices where elements match
      if A == B
         n = n + 1;
         aIndex(n) = ai;
         bIndex(n) = bi;
      end

      % Advance index of avec, when appropriate
      if A <= B
         if ai < avecLen
            ai = ai + 1;
         else
            break;
         end
      end

      % Advance index of bvec, when appropriate
      if A >= B
         if bi < bvecLen
            bi = bi + 1;
         else
            break;
         end
      end
   end

   % Snip aIndex and bIndex to correct size
   aIndex = aIndex(1:n);
   bIndex = bIndex(1:n);
```

In preparation for running the program, you'll need to create two sorted vectors having some number of common elements. The following statements create vectors a and b, append the elements from a third vector, c, to both, and then sort. This gives vectors a and b at least 20,000 common elements.

```
a = rand(200000,1);
b = rand(260000,1);
c = rand(20000,1);
a = sort([a;c]);
b = sort([b;c]);
```

Now pass a and b into the function shown above. Use the `tic` and `toc` functions to track how much time it takes to execute.

```
tic; [ia, ib] = vfind_scalar(a, b);  toc
```

What Makes It Faster

MATLAB accelerates every line in this program. The code in the `while` loop is what matters most since this is what takes up nearly all of the program execution time. Consider the following factors.

Supported Data Types and Array Shapes. All of the code in the program operates on vectors of type `double`. This is one of the data types and array shapes that supports acceleration.

Conditional Expression Evaluates to Scalar. The expressions in the `while` and `if` statements all evaluate to scalar values. For example,

```
while (ai <= avecLen || bi <= bvecLen)
```

No Disqualifying Statements in Loop. Every line of code in the `while` loop qualifies for acceleration. This means that every iteration of the loop can execute at a higher speed without being interrupted to separately process any disqualifying lines.

Function Calls and Overloading. The only functions called are MATLAB built-ins. No M-file or MEX-file functions are called and no operations are overloaded for the data types being used.

Program 3b — Vector Comparison, Vectorized

Here is the vectorized version of the same program. Note that there is little difference in performance between the accelerated `while` loop, shown in "Program 3a", and the vectorized code shown below. This means that, with accelerated MATLAB, you have the freedom to choose the style of coding that you prefer for each MATLAB application without affecting performance.

The table below also shows little difference in the times measured for running the vectorized code on unaccelerated and accelerated versions of MATLAB. You will not see a significant performance improvement in vectorized programs when run with the accelerator.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	0.7 sec.	0.6 sec.	x 1.2
Linux	1.0 sec.	0.9 sec.	x 1.1
Solaris	1.2 sec.	1.2 sec.	x 1.0

Here is the vectorized version of the `vfind` program:

```
function [aIndex, bIndex] = vfind_vector(avec, bvec)

avecLen = length(avec);
bvecLen = length(bvec);

avec = reshape(avec, avecLen, 1);
bvec = reshape(bvec, bvecLen, 1);

[c, pc] = sort([avec; bvec]);
cIndex = find(diff(c) == 0);
aIndex = pc(cIndex);
bIndex = pc(cIndex + 1) - avecLen;
```

Program 4 — Tic-Tac-Toe

This program plays a game of tic-tac-toe. It plays the number of rounds indicated by the `tries` input argument and returns a vector with the results. The first element of the return vector shows the number of wins for player `X`, the second element shows the wins for player `O`, and the third element is the number of games that resulted in a tied score.

Operating System	MATLAB 6.1	MATLAB 6.5	Performance Gain
Windows	7.7 sec.	0.9 sec.	x 8.6
Linux	17.9 sec.	1.8 sec.	x 9.9
Solaris	43.9 sec.	7.4 sec.	x 5.9

```
% FILE:    tictactoe.m
% PURPOSE: if random legal moves are tried, how often do each of
%    X and O win?

function tictactoe(tries)
X   = 1;
O   = 2;
NOWIN = 3;

result = [0 0 0];                  % [X_wins, O_wins, No_Winner]

for k = 1:tries
   board = zeros(3);
   player = X;
   winner = NOWIN;
   for moves = 1:9
      illegal = 1;
      while illegal              % Find a random, empty space.
         m = ceil(rand*3);
         n = ceil(rand*3);
         illegal = board(m,n);   % Is this space already taken?
      end
      board(m,n) = player;       % No, place the X or O here.
```

```
    % Starting with the 5th alternating move [X O X O X],
    % if the current player occupies 3 squares across, or
    % 3 down, or 3 on \ diagonal, or 3 on / diagonal, ...
    if moves > 4 && ...
       (board(m,1) == player && board(m,2) == player && ...
          board(m,3) == player) || ...
       (board(1,n) == player && board(2,n) == player && ...
          board(3,n) == player) || ...
       (board(1,1) == player && board(2,2) == player && ...
          board(3,3) == player) || ...
       (board(1,3) == player && board(2,2) == player && ...
          board(3,1) == player)

       % ... then that player wins this round.
       winner = player;
       break;
    end

    % Give the other player a turn.
    if player == O, player = X;  else player = O;  end
  end

  % Keep score for each round.
  result(winner) = result(winner) + 1;
end

result
```

Try running the program for 10,000 iterations, recording the elapsed time with
tic and toc.

```
  tic;  tictactoe(10000);  toc
```

What Makes It Faster

Like the previous sample programs, the tictactoe program spends nearly all
of its time in the nested for and while loop. It uses only scalar and matrix data
types that support MATLAB performance acceleration. It also makes no
function calls and does not use function overloading.

In addition, the following factors also speed up program execution.

Logical Operators. One of the more interesting things about this program is the extended `if` statement in the middle of the `for` loop.

```
if moves > 4 && ...
    (board(m,1) == player && board(m,2) == player && ...
        board(m,3) == player) || ...
```

This statement has the following characteristics, each of which is a factor in qualifying for performance acceleration:

- Each expression of the extended `if` statement evaluates to a logical scalar value. The expression `board(m,1) == player` is an example of this.
- The `if` statement performs logical AND and OR using the `&&` and `||` operators rather than the slower `&` and `|`. You should use the `&` and `|` operators only for element-wise logical operations on arrays.

Short-Circuiting. One more thing to note about this extended `if` statement, although not associated with performance acceleration, is that it saves time on some iterations by short-circuiting. Short-circuiting means that the conditional statement may at times be resolved without having to evaluate every part of the expression. For example, if `moves` is not greater than 4, then the conditions comparing board spaces to player positions do not need to be evaluated, thus saving processing time.

```
if moves > 4 && ...
    (board(m,1) == player && board(m,2) == player && ...
        board(m,3) == player) || ...
```

Measuring Performance

One way to improve the performance of your M-files is measuring performance using profiling tools. MATLAB provides the M-file Profiler, a graphical user interface that is based on the results returned by the `profile` function. Use the Profiler to help you determine where you can modify your code to make performance improvements. This section covers the following topics:

- What Is Profiling? (p. 22-29)—Profiling assesses where time is being spent in your M-code.
- The Profiler (p. 22-30)—A graphical user interface for viewing where the time is being spent in your M-code. The Profiler allows you to best take advantage of MATLAB tools described in "Performance Acceleration" on page 22-8.
- The profile Function (p. 22-48)—The function on which the Profiler is based, `profile`, offers a few features not included in the Profiler, including saving results of the `profile` function to a file.

What Is Profiling?

Profiling is a way to measure where a program spends its time. You can deal with obvious speed issues at design time and can then discover unanticipated effects through measurement. One key to effective coding is to create a first implementation that is as simple as possible, and then to use a profiler to identify bottlenecks if speed is an issue. Premature optimization often increases code complexity unnecessarily without providing a real gain in performance.

Use the MATLAB Profiler to identify statements that are not taking advantage of the MATLAB tools described in "Performance Acceleration" on page 22-8. You may then be able to modify your statements to take advantage of these performance acceleration tools.

Also use the MATLAB Profiler to identify functions that consume the most time, then determine why you are calling them and look for ways to minimize their use. It is often helpful to decide whether the number of times a particular function is called is reasonable. Because programs often have several layers, your code may not explicitly call the most time-consuming functions. Rather, functions within your code might be calling other time-consuming functions

that can be several layers down in the code. In this case it is important to determine which of your functions are responsible for such calls.

The Profiler often helps to uncover performance problems that you can solve by

- Avoiding unnecessary computation, which can arise from oversight
- Changing your algorithm to avoid costly functions
- Avoiding recomputation by storing results for future use

When you reach the point where most of the time is spent on calls to a small number of built-in functions, you have probably optimized the code as much as you can expect.

The Profiler

The Profiler is a tool that shows you where an M-file is spending its time. This section covers

- "Opening the Profiler" on page 22-30
- "Running the Profiler" on page 22-31
- "Profile Summary Report" on page 22-35
- "Profile Detail Report" on page 22-37
- "File Listing" on page 22-39
- "Using Information in the Profiler Reports" on page 22-44
- "Changing Fonts for the Profiler" on page 22-46

Opening the Profiler

To open the Profiler, select **View -> Profiler** from the MATLAB desktop, or type `profile viewer` in the Command Window. The MATLAB Profiler opens.

Running the Profiler

This is a quick summary. Details follow.

1 Type profile viewer to open the Profiler.

2 Enter statement to run.

3 Click **Start Profiling.**

To profile an M-file or a line of code, follow these steps:

1 In the **Run this code** field in the Profiler, enter the statement you want to run.

Run this code: [t,y] = ode23('lotka',[0 2],[20;20])

You can run this example

```
[t,y] = ode23('lotka',[0 2],[20;20])
```

as the code is provided with MATLAB demos. It runs the Lotka-Volterra predator-prey population model. For more information about this model, type lotkademo, which runs demonstration.

If you are running a statement you previously profiled in the current MATLAB session, select the statement from the list box and skip to step 3.

Run this code: [t,y] = ode23('lotka',[0 2],[20;20])
[t,y] = ode23('lotka',[0 2],[20;20])
collatz(100)
collatzplot(100)

2 Click **Start Profiling** (or press **Enter** after entering the statement)**.**

Start Profiling

While the Profiler is running, the **Profile time** indicator (at the top right of the Profiler window) is green and the number of seconds it reports increases.

⚪ Profile time: 4 sec

When the profiling is finished, the **Profile time** indicator becomes black and shows the length of time the Profiler ran.

⚫ Profile time: 6 sec

This is not the actual time that your statements took to run, but is the time elapsed from when you clicked **Start Profiling** until you clicked **Stop Profiling**. If the time reported is much different from what you expected (for example hundreds of seconds for a simple statement), you might have had profiling on longer than you realized.

3 When profiling is complete, the **Profile Summary** report appears in the Profiler window.

Profiling a Graphical User Interface. You can run the Profiler for a graphical user interface, such as the Filter Design and Analysis tool included with the Signal Processing Toolbox. You can also run the Profiler for an interface you created, such as one built using GUIDE.

To profile a graphical user interface, do the following.

1 In the Profiler, click **Start Profiling**.

2 Start the graphical user interface. If you do not want to include its startup process in the profile, do not click **Start Profiling** until after you have started the graphical interface.

3 Use the graphical interface. When you are finished, click **Stop Profiling** in the **Profiler**.

The **Profile Summary** report appears in the Profiler.

Profiling Statements from the Command Window. To profile more than one statement, do the following:

1 In the Profiler, clear the **Run this code** field and click the **Start Profiling** button.

2 In the Command Window, enter and run the statements you want to profile.

The status bar in the desktop reports `Profiler on` when MATLAB is not busy and the Profiler is running.

3 After running all the statements, click **Stop Profiling** in the Profiler.

The **Profile Summary** report appears in the Profiler.

Profile Summary Report

The **Profile Summary** report provides time and calling frequency for the files that ran while the Profiler was on. Note that total time can be zero for files whose running time was inconsequential.

From this summary, identify the functions whose speed you want to improve, either because the time or the number of calls is high. For those functions, click the name in the **Filename** list. A detailed report appears for the function.

Click a filename to display a detailed profile report for it.

This example shows the detail report for the `odearguments` file.

Profile Detail Report

The profile detail report shows profiling results for a selected file that was called during profiling. There are three basic sections you can use:

- "Header—Location, Link to File, and Copy Report"
- "Parents, Children, and Lines with Most Time"
- "File Listing"

To return to the Profile Summary report, click the home button 🏠 in the toolbar.

Header—Location, Link to File, and Copy Report. The first section provides the filename, a link to the file, and a link that copies the report to a separate window. After you copy the report, you can make changes to the file, run the Profiler for the updated file, and compare the Profiler detail reports for the two runs. Do not make changes to M-files provided with MathWorks products, that is, files in `$matlabroot/toolbox` directories.

Copy this detail report to a new window.

Open the file in your default editor.

E:\...fun\private\odearguments (1 call, 1.322 sec)

Generated 08-Apr-2002 18:54:20
M-function: E:\toolbox\matlab\funfun\private\odearguments.m
[Copy to new window for comparing multiple runs]

Parents, Children, and Lines with Most Time. The second section provides information about the parent function, a summary of lines where the most time was spent, and the children called. Subfunctions in the file are considered children, so you view the Profiler details separately.

Change the view using the links:

- **table**
- **list**, to conserves space but with less information
- **hide**, to temporarily not show the information

Click a line number to go to that line in the third section of the detail report.

Change the view to see results as a table or list, or hide the results.

Here, **Parents** and **Lines** use a table view.
Children uses a list view.

Click a line number to go directly
to it in the file listing.

Parents (calling functions) [table | list | hide]

Filename	File type	Calls
ode23	M-function	1

Lines where the most time was spent [table | list | hide]

Line number	Code	Calls	Total time	% time	Time plot
114	rtol = odeget(options,'Re...	1	0.451 s	34.1%	████
76	if (nargin(ode) == 2) ...	1	0.321 s	24.3%	███
144	hmax = min(abs(tfinal-t0)...	1	0.200 s	15.1%	██
104	f0 = feval(ode,t0,y0,args...	1	0.160 s	12.1%	██
100	if any(tdir*diff(tspan) ...	1	0.070 s	5.3%	█
All other lines			0.120 s	9.1%	█
Totals			1.322 s	100%	

Children (called functions) [table | list | hide]
odeget (30.3%), lotka (0.8%)

File Listing

The third section lists the file and allows you to highlight lines where the most time was spent, among other options.

Information you can glean from the File listing report includes

- "Line Colors"
- "Color Highlighting"
- "Time, Calls, and Acceleration Columns"

Line Colors. Lines are colored as follows. (This is not apparent in the printed documentation.)

Line Color	Example	Description
Green	`5 %ODEARGUMENTS  Helper`	Comment
Black	`13 if FcnHandlesUsed`	Line was executed
Gray	`14    msg = ['When`	Line was not executed

Use the line colors to see which lines of code actually ran. This is useful not only to profile an M-file, but to debug or to see how an M-file works. Color highlighting by coverage, described next, is another way to view this information.

Color Highlighting. In **Color highlight code according to**, click the type of highlight to change to that view. **Time** is selected in this example. (Note that **Time** is shaded pink to indicate it is selected, but that is not apparent in the printed documentation.)

File listing
Color highlight code according to [Time | Number of calls | Coverage | Acceleration | No color]

Lines shaded pink used the most time. Among them, the darkest pink used the most time, while the lightest pink used the least time.

Click a line number to open the M-file in your default editor at that line.
Click a function to view its profile detail report.

```
 0.000   1 \ .     99 tdir = sign(tfinal - t0);
 0.070   1 \ .    100 if any( tdir*diff(tspan) <= 0 )
                  101    error('The entries in tspan must strictly increase or d
                  102 end
                  103
 0.160   1  x     104 f0 = feval(ode,t0,y0,args{:});
 0.020   1  x     105 [m,n] = size(f0);
 0.010   1  .     106 if n > 1
                  107    error([funstring(ode) ' must return a column vector.'])
 0.010   1  .     108 elseif m ~= neq
                  109    msg = sprintf('an initial condition vector of length %d
                  110    error(['Solving ' funstring(ode) ' requires ' msg]);
                  111 end
                  112
                  113 % Get the error control options, and set defaults.
 0.451   1  x     114 rtol = odeget(options,'RelTol',1e-3,'fast');
 0.000   1  .     115 if (length(rtol) ~= 1) | (rtol <= 0)
```

These types of highlighting provide you with another view of the information to help you improve your code. They allow you to scan the report and hone in on areas for improvement at a quick glance. You can then view more details for those lines using the "Time, Calls, and Acceleration Columns" on page 22-43.

Type of Highlight	Examples	Description

Time	`100 if any( tdir*diff`
	
	 `104 f0 = feval(ode,t0,`
	
	 `114 rtol = odeget(opti`
	
	 `13 if FcnHandlesUsed`
		Lines shaded pink used the most time. Among the shaded lines, the darkest pink used the most time, while the lightest pink used the least time. A line that is not shaded but has black text (rather than gray) ran, but was not among the lines using the most time.

Number of calls	From ode23
	
	 `283 done = false;`
	 `284 while ~done`
		Lines shaded blue were called the most. Among the shaded lines, the darkest blue were called most frequently, while the lightest blue were called least frequently. A line that is not shaded but has black text (rather than gray) ran, but was not among the lines called most often.

Coverage	`115 if (length(rtol) ~=`
	 `116   error('RelTol must`
	 `117 end`
	 `118 if rtol < 100 * eps`
		Lines shaded blue were called. Lines that are not shaded were not called.
		
		 All lines that are shaded use black text. Unshaded lines use gray text.

Type of Highlight	Examples	Description (Continued)
Acceleration	`33` else % ode-file used `34` if ~(isempty(options)	Lines shaded pink were accelerated using MATLAB performance acceleration tools. Lines that are not shaded were not accelerated.
No color	`33` else % ode-file used `34` if ~(isempty(options)	No lines are highlighted.

Time, Calls, and Acceleration Columns. For each line of the M-file, there are five columns of information.

Column	Value or Example time calls acc line	Description
time	0.130 1 x **114** rtol	Total time spent on that line, in seconds. If the line was called but no time is listed, the time was so insignificant that it was not captured. In this example, line 114 ran for .13 seconds.
calls	0.130 1 x **114** rtol	Number of calls to that line. In this example, line 114 was called once.
acc	. (dot) 0.000 1 . **115**	A . (dot) indicates the line was accelerated by MATLAB performance acceleration tools.

Column	Value or Example	Description (Continued)
	time calls acc line	
	x 0.130 1 x 114 rtol	An **x** indicates the line was not accelerated by MATLAB performance acceleration tools. Click the **x** to see information about why the line was not accelerated. For details on acceleration, see "Performance Acceleration" on page 22-8.
Line Number	0.130 1 x 114 rtol	The line number in the M-file. This example shows line 114. Click the line number to open your default editor to that line number. While this is a convenient way to go to a line and modify it in your own M-files, do not modify functions provided with MathWorks products, that is, functions stored in $matlabroot/toolbox directories.
Code	114 rtol = odeget(options,	The line of code. This example shows part of the line rtol =. Scroll to see more of the line. Click a function to go to its Profiler detail report, for example odeget.

Using Information in the Profiler Reports

These are some guidelines for using the information provided by the Profiler reports.

- "Process for Improving Performance"
- "Using the Profiler for Debugging"
- "Using the Profiler for Understanding an M-File"

Process for Improving Performance. Here is a general process you can follow to use the Profiler to improve performance in your M-files.

1 In the Profile Summary report, look for functions that used a significant amount of time or were called most frequently.

2 View the detail report for those functions and look for the lines that use the most time or are called most often.

You might want to copy the detail report to its own window using the **Copy to new window** link near the top of the detail report. The copy will be a reference to compare with after you make changes and profile again. You can also print a report by clicking the print button 🖨.

3 Determine whether there are changes you can make to the lines most called or the most time-consuming lines to improve performance.

For example, if you have a load statement within a loop, load is called every time the loop is called. You might be able to save time by moving the load statement so it is before the loop and therefore is only called once.

4 If lines are not accelerated (there is an **x** in the **Acc** column of the detail report), particularly those that are called frequently or use significant time, click the **x** for information about why they were not accelerated. This is an example.

Determine whether there are changes you can make to the lines so they can be accelerated. For details, see "Performance Acceleration" on page 22-8. Most code includes lines that are not accelerated. As long as your code runs acceptably fast, you do not need to have every statement accelerate.

5 Click the links to the files and make the changes you identified for potential performance improvement. Save the files and run `clear all`. Run the Profiler again and compare the results to the original report. Note that there are inherent time fluctuations that are not dependent on your code. If you profile the exact same code twice, you can get slightly different results each time.

6 Repeat this process to continue improving the performance.

Using the Profiler for Debugging. The Profiler is a useful tool for isolating problems in your M-files.

For example, if a particular part of the file did not run, you can look at the detail reports to see what lines did run, which might point you to the problem.

You can also view the lines that did not run to help you develop test cases that exercise that code.

If you get an error in the M-file when profiling, the Profiler provides partial results in the reports. You can see what ran and what did not to help you isolate the problem. Similarly, you can do this if you stop the execution using **Ctrl+C**, which might be useful when a file is taking much more time to run than expected.

Using the Profiler for Understanding an M-File. For lengthy M-files that you did not create or that you have not used for awhile and are unfamiliar with, you can use the Profiler to see how the M-file actually worked. Use the Profiler detail reports to see the lines actually called.

If there is an existing GUI tool (or M-file) similar to one that you want to create, start profiling, use the tool, then stop profiling. Look through the Profiler detail reports to see what functions and lines ran. This helps you determine the lines of code in the file that are most like the code you want to create.

Changing Fonts for the Profiler

To change the fonts used in the Profiler, follow these steps:

1 Select **File -> Preferences -> Help -> Fonts** to open the **Preferences** dialog box.

2 In the **Preferences** dialog box, set the **HTML browser font** and click **Apply** or **OK**.

3 In the Profiler, click the Reload button to update the display.

The font change also applies to the Help browser display pane.

The profile Function

The Profiler is based on the results returned by the profile function. There are some features available with the profile function that are not part of the Profiler.

To use the profile function:

1 Type profile on in the Command Window. Specify any options you want to use.

2 Execute your M-file.

The profile function counts how many seconds each line in the M-files uses. It works cumulatively, that is, adding to the count for each M-file you execute until you clear the statistics.

3 Use profile report to display the statistics gathered in an HTML-formatted report in your system's default Web browser. This report contains some information different from what is available in the Profiler reports.

Here is a summary of the main forms of profile. For details about these and other options, type doc profile.

Syntax	Option	Description
profile **on**		Starts profile, clearing previously recorded statistics.
	-**detail** *level*	Specifies the level of function to be profiled.
	-**history**	Specifies that the exact sequence of function calls is to be recorded.
profile **off**		Suspends profile.

Syntax	Option	Description (Continued)
profile **report**		Suspends profile, generates a profile report in HTML format, and displays the report in your system's default Web browser. This report contains some information different from what is available in the Profiler reports.
	basename	Saves the report in the file basename in the current directory.
profile **plot**		Suspends profile and displays in a figure window a bar graph of the functions using the most execution time.
profile **resume**		Restarts profile without clearing previously recorded statistics.
profile **clear**		Clears the statistics recorded by profile.
profile **viewer**		Opens the Profiler, a graphical user interface. This provides information gathered using the profile function, but presents the information in a different format from the profile function reports.
s = profile('**status**')		Displays a structure containing the current profile status.
stats = profile('**info**')		Suspends profile and displays a structure containing profile results.

Some people use profile simply to see the child functions; see also depfun for that purpose.

An Example Using profile

This example demonstrates how to run profile:

1 To start profile, type in the Command Window

```
profile on -detail builtin -history
```

The `-detail builtin` option instructs `profile` to gather statistics for built-in functions, in addition to the default M-functions, M-subfunctions, and MEX-functions.

The `-history` option instructs `profile` to track the exact sequence of entry and exit calls.

2 Execute an M-file. This example runs the Lotka-Volterra predator-prey population model. For more information about this model, type `lotkademo`, which runs a demonstration.

```
[t,y] = ode23('lotka',[0 2],[20;20]);
```

3 Generate the profile report and save the results to the file `lotkaprof`.

```
profile report lotkaprof
```

This suspends `profile`, displays the profile report in your system's default Web browser, and saves the results. See "Viewing profile Results" on page 22-51 for more information.

4 Restart `profile`, without clearing the existing statistics.

```
profile resume
```

The `profile` function is now ready to continue gathering statistics for any more M-files you run. It will add these new statistics to those generated in the previous steps.

5 Stop `profile` when you finish gathering statistics.

```
profile off
```

Viewing profile Results

There are two main ways to view `profile` results:

- "Viewing profile Reports" on page 22-51
- "Profile Plot" on page 22-54

To save results, see "Saving Profile Reports" on page 22-55.

Viewing profile Reports. To display `profile` results, type

```
profile report
```

This suspends `profile` and produces three reports:

- "Summary profile Report" on page 22-51
- "Function Details profile Report" on page 22-53
- "Function Call History Profile Report" on page 22-54

The summary report appears in your system's default Web browser. Use the links at the top of the report page to see the other reports.

Summary profile Report. The summary report presents statistics about the overall execution and provides summary statistics for each function called. Values reported include

- **Number of functions**—The numbers of built-in functions, M-functions, and M-subfunctions are reported.
- **Clock precision**—The precision of `profile` time measurement. When Time for a function is 0, it is actually a positive value, but smaller than `profile` can detect given the clock precision.
- **Time** columns—The total time spent in a function, including all child functions called. Because the time for a function includes time spent on child functions, the times do not add up to the **Total recorded time** and the percentages add up to more than 100%.
- **Self time** columns —The total time spent in a function, *not* including time for any child functions called. Adding the **Self time** values for all functions listed equals the **Total recorded time**. The **Self time** percentages for all functions add up to approximately 100%.

Note that `profile` itself uses some time, which is included in the results.

Following is the summary report for the Lotka-Volterra model described in "An Example Using profile" on page 22-49.

View summary report (shown in this illustration).

View details for all functions.

View sequence of function calls.

Total time profile was recording.

Follow link to view details for each function.

Report includes built-in functions because -detail builtin option was used.

MATLAB Profiler Report - Netscape

File   Edit   View   Go   Communicator   Help

Summary | Function Details | Function Call History

MATLAB Profile Report: Summary

Report generated 04-Aug-2000 12:06:28

Total recorded time:	0.11 s
Number of Builtin-functions:	32
Number of M-functions:	9
Number of M-subfunctions:	1
Clock precision:	0.010 s

Function List

Name	Time		Calls	Time/call	Self time		Location
ode23	0.11	100.0%	1	0.1100	0.03	27.3%	D:\matlab
odearguments	0.08	72.7%	1	0.0800	0.06	54.5%	D:\matlab
lotka	0.01	9.1%	34	0.0003	0.01	9.1%	D:\matlab
feval	0.01	9.1%	34	0.0003	0.00	0.0%	*Builtin-functio*
isfield	0.01	9.1%	11	0.0009	0.01	9.1%	D:\matlab

Document: Done

Time per function includes time spent in child functions.

Self time does NOT include time spent in child functions.

Function Details profile Report. The function details report provides statistics for the parent and child functions of a function, and reports the line numbers on which the most time was spent. Following is the detail report for the lotka function, which is one of the functions called in "An Example Using profile" on page 22-49.

Time in seconds

Percentage of the function's time spent on

Line number

Function Call History Profile Report. The function call history displays the exact sequence of functions called. To view this report, you must have started profile using the -history option.

```
profile on -history
```

The profile function records up to 10,000 function entry and exit events. For more than 10,000 events, profile continues to record other profile statistics, but not the sequence of calls. Following is the history report generated from "An Example Using profile" on page 22-49.

Exact sequence of calls

Profile Plot. To view a bar graph for the functions using the most execution time, type

```
profile plot
```

This suspends the profile. The bar graph appears in a figure window. Following is the bar graph generated from "An Example Using profile" on page 22-49.

Saving Profile Reports. When you generate the profile report, use the basename option to save it. For example,

```
profile report basename
```

saves the profile report to the file basename in the current directory. Later you can view the saved results using a Web browser.

Another way to save results is with the `info = profile` function, which displays a structure containing `profile` results. Save this structure so that later you can generate and view the profile report using `profreport(info)`.

Example Using Structure of Profiler Results. The `profile` results are stored in a structure that you can view or access. This example illustrates how you can view the results:

1 Run `profile` for code that computes the Lotka-Volterra predator-prey population model.

```
profile on -detail builtin -history
[t,y] = ode23('lotka',[0 2],[20;20]);
```

2 To view the structure containing `profile` results, type

```
stats = profile('info')
```

MATLAB returns

```
stats =
       FunctionTable: [41x1 struct]
     FunctionHistory: [2x826 double]
      ClockPrecision: 0.0100
                Name: 'MATLAB'
```

3 You can view and access the contents of the structure. For example, type

```
stats.FunctionTable
```

MATLAB displays the `FunctionTable` structure.

```
ans =
41x1 struct array with fields:
    FunctionName
    FileName
    Type
    NumCalls
    TotalTime
    TotalRecursiveTime
    Children
    Parents
    ExecutedLines
```

4 To view the contents of an element in the `FunctionTable` structure, type, for example:

```
stats.FunctionTable(2)
```

MATLAB returns the second element in the structure.

```
ans =
          FunctionName: 'horzcat'
              FileName: ''
                  Type: 'Builtin-function'
               NumCalls: 43
              TotalTime: 0.0100
     TotalRecursiveTime: 0.0100
               Children: [0x1 struct]
                Parents: [2x1 struct]
           ExecutedLines: [0x3 double]
```

5 Save the results:

```
save profstats
```

6 In a later session, to generate the profile report using the saved results, type

```
load profstats
profreport(stats)
```

MATLAB displays the profile report.

Making Efficient Use of Memory

This section discusses how to conserve memory and improve memory use. Topics include

- "Memory Management Functions"
- "Ways to Conserve Memory" on page 22-59
- ""Out of Memory" Errors" on page 22-61
- "Platform-Specific Memory Topics" on page 22-63

For more information on memory management, see Technical Note 1106: "Memory Management Guide", at the following URL:

```
http://www.mathworks.com/support/tech-notes/1100/1106.shtml
```

Memory Management Functions

The following functions can help you to manage memory use in MATLAB:

- `whos` shows how much memory has been allocated for variables in the workspace.
- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.

  See "Compressing Data in Memory" on page 22-59.
- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically `clear` variables from memory that you no longer need.
- `save` selectively stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then `clear` the saved data from memory.
- `load` reloads a data file saved with the `save` function.
- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on UNIX systems as UNIX does not free up memory allocated to an application (e.g., MATLAB) until the application exits.

  See "Freeing Cleared Memory on UNIX" on page 22-63.

Note `save` and `load` are faster than the MATLAB low-level file I/O routines. `save` and `load` have been optimized to run faster and reduce memory fragmentation.

Ways to Conserve Memory

This section discusses various ways that may help you to use less memory and avoid `Out of Memory` errors in MATLAB.

Working with Variables

To conserve memory when creating variables,

- Avoid creating large temporary variables, and clear temporary variables when they are no longer needed.
- When working with arrays of fixed size, preallocate them rather than having MATLAB resize the array each time you enlarge it.
- Set variables equal to the empty matrix `[]` to free memory, or clear the variables using the `clear` function.
- Reuse variables as much as possible.

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.

Note Because of time considerations, you should not use `pack` within loops or M-file functions.

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 are no longer using from memory. Use `clear` to do this.

Working 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.

Converting Full Matrices into Sparse

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.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. As shown below, approximately half as much memory is required for the sparse matrix.

```
whos
  Name       Size                   Bytes  Class

  X        1000x1000              8000000  double array
  Y        1000x1000              4004000  double array (sparse)
```

Memory Requirements for Cell Arrays

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.

Comparing a Structure of Arrays with an 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.

Nested Function Calls

The amount of memory used by nested functions is the same as the amount used by calling them on consecutive lines. These two examples require the same amount of memory.

```
result = function2(function1(input99));
```

```
result = function1(input99);
result = function2(result);
```

"Out of Memory" Errors

Typically, MATLAB generates an Out of Memory message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see this message, use any of the techniques discussed earlier in this section to help optimize the available memory.

If the Out of Memory message still appears, you can try any of the following:

• Increase the size of the swap file. We recommend that your machine be configured with twice as much swap space as you have RAM.

See "Increasing the System Swap Space", below.

• Allocate larger matrices earlier in the MATLAB session

• If possible, break large matrices into several smaller matrices so that less memory is used at any one time.

• Reduce the size of your data.

• Make sure that there are no external constraints on the memory accessible to MATLAB. (On UNIX systems, use the limit command to check).

• On UNIX systems, you can run memory intensive tasks with more available address space by starting MATLAB with the -nojvm switch.

See "Running MATLAB Without the Java Virtual Machine" on page 22-62.

• Add more memory to the system.

Increasing the System Swap Space

How you set the swap space for your computer depends on what operating system you are running on.

UNIX. Information about swap space can be procured by typing `pstat -s` at the UNIX command prompt. For detailed information on changing swap space, ask your system administrator.

Linux. Swap space can be changed by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man command_name` at the Linux prompt.

Windows 98, Windows NT, and Windows ME. Follow the steps shown here:

1 Right-click on the **My Computer** icon, and select **Properties**.

2 Select the **Performance** tab and click the **Change** button to change the amount of virtual memory.

Windows 2000. Follow the steps shown here:

1 Right-click on the **My Computer** icon, and select **Properties**.

2 Select the **Advanced** tab and choose **Performance Options**.

3 Click on the **Change** button to change the amount of virtual memory.

Running MATLAB Without the Java Virtual Machine

You are more likely to encounter `Out of Memory` errors when using MATLAB Version 6.0 or greater due to the use of the Java Virtual Machine (JVM) within MATLAB. The JVM and MATLAB use the same virtual address space, leaving less memory available for your MATLAB applications to use.

MATLAB uses the JVM to implement its GUI-based development environment. You are encouraged to use this environment for your application development work. However, if you get `Out of Memory` errors while running memory intensive applications, and you are running MATLAB on a UNIX platform, you may want to run these without the JVM.

On UNIX systems, it is possible to run MATLAB without the JVM by starting MATLAB with a `-nojvm` switch. Type the following at the UNIX prompt:

```
matlab -nojvm
```

When you use this option, you cannot use the desktop or any of the MATLAB tools that require Java.

Platform-Specific Memory Topics

This section presents information specific to the Windows and UNIX platforms that may be helpful in conserving memory.

Windows Topics

The following topics apply to how the Microsoft Windows operating system handles memory.

Freeing Up System Resources. There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows uses system resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UIcontrols. One way to free up system resources is to close all inactive windows. Iconified windows still use resources.

UNIX Topics

The following topics apply to how the UNIX operating system handles memory.

Freeing Cleared 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.

Additional Memory Used to Execute External Commands. 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 example, if your MATLAB session occupies 5 Mbytes of memory, if you shell out, the operating system must allocate another 5 Mbytes.

Reusing Heap Memory. MATLAB requests memory from the operating system when there is not enough memory available in the MATLAB heap to store the

current variables. It reuses memory in the heap as long as the size of the memory segment required is available in the MATLAB heap.

For example, on one machine these statements use approximately 15.4 MB of RAM:

```
a = rand(1e6,1);
b = rand(1e6,1);
```

This statement uses approximately 16.4 MB of RAM:

```
c = rand(2.1e6,1);
```

These statements use approximately 32.4 MB of RAM. This is because MATLAB is not able to fit a 2.1 MB array in the space previously occupied by two 1-MB arrays.

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.1e6,1);
```

The simplest way to prevent overallocation of memory, is to allocate the largest vectors first. These statements use only about 16.4 MB of RAM

```
c = rand(2.1e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

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 cases 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, O, '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 23-50, but apply to optimizing for speed as well:

- "Conserving Memory with Large Amounts of Data" on page 23-51
- "Matrix Manipulation with Sparse Matrices" on page 23-51
- "Structure of Arrays Rather Than Array of Structures" on page 23-52
- "Preallocating Is Better Than Growing an Array" on page 23-52
- "Preallocating a Nondouble Matrix" on page 23-52

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
```

External Interfaces and the MATLAB API

Finding the Documentation in Online Help (p. A-2)

A summary of what information is available on the MATLAB external interfaces

Reference Documentation (p. A-5)

A summary of the MATLAB functions that support external interfaces

Finding the Documentation in Online Help

MATLAB provides interface capabilities that allow you to communicate between MATLAB and the following programs and devices:

• External C and Fortran programs
• Object-oriented technologies like Java and COM
• Hardware devices on your computer's serial port

You can also import and export data to and from MATLAB.

These interfaces, also referred to as the MATLAB Application Program Interface (API), are documented in full in the online help. Use the following path in the left pane of the MATLAB help browser to locate the help sections listed below:

```
MATLAB -> External Interfaces/API
```

Calling C and Fortran Programs from MATLAB

MATLAB provides an interface to external programs written in the C and Fortran languages that enables you to interact with data and programs external to the MATLAB environment. This section explains how to call your own C or Fortran subroutines from MATLAB as if they were built-in functions.

Creating C Language MEX-Files

MATLAB callable C and Fortran programs are referred to as MEX-files. This section explains how to create and work with C MEX-files.

Creating Fortran MEX-Files

This section explains how to create and work with Fortran MEX-files.

Calling MATLAB from C and Fortran Programs

You can employ MATLAB as a computational engine that responds to calls from your C and Fortran programs. This section describes the MATLAB functions that allow you to

• Start and end a MATLAB process
• Send commands to and exchange data with MATLAB
• Compile and link MATLAB engine programs

Calling Java from MATLAB

This section describes how to use the MATLAB interface to Java classes and objects. This MATLAB capability enables you to

- Bring Java classes into the MATLAB environment
- Construct objects from those classes
- Work with Java arrays in MATLAB
- Call methods on Java objects, passing MATLAB or Java data types

Importing and Exporting Data

This section describes how to use MAT-files to import data to and export data from the MATLAB environment. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms in a highly portable manner. In addition, they provide a means to import and export your data to other stand-alone MATLAB applications.

COM and DDE Support

MATLAB has interfaces that allow you to interact with Component Object Model (COM) and Dynamic Data Exchange (DDE). This section explains how to

- Integrate COM control components into a COM control container such as a MATLAB figure window
- Use COM Automation components to control or be controlled by MATLAB
- Enable access between MATLAB and other Windows applications using DDE

Serial Port I/O

This section describes the MATLAB serial port interface which provides direct access to peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port. This interface is established through a serial port object that allows you to

- Configure serial port communications
- Use serial port control pins
- Write data to and read data from the device

- Execute an action when a particular event occurs
- Create a record of your serial port session

Reference Documentation

The online Help Reference section provides a detailed description of each of the MATLAB functions available in the MATLAB external interfaces. Use the following path in the left pane of the MATLAB help browser to locate the help sections listed below:

```
MATLAB -> External Interfaces/API Reference
```

C Engine Functions

Functions that allow you to call MATLAB from your own C programs.

C MAT-File Functions

Functions that allow you to incorporate and use MATLAB data in your own C programs.

C MEX-Functions

Functions that you use in your C files to perform operations in the MATLAB environment.

C MX-Functions

Array access and creation functions that you use in your C files to manipulate MATLAB arrays.

Fortran Engine Functions

Functions that allow you to call MATLAB from your own Fortran programs.

Fortran MAT-File Functions

Functions that allow you to incorporate and use MATLAB data in your own Fortran programs.

Fortran MEX-Functions

Functions that you use in your Fortran files to perform operations in the MATLAB environment.

Fortran MX-Functions

Array access and creation functions that you use in your Fortran files to manipulate MATLAB arrays.

Java Interface Functions

Functions that enable you to create and interact with Java classes and objects from MATLAB.

COM Functions

Functions that create Component Object Model objects and manipulate their interfaces.

DDE Functions

Dynamic Data Exchange functions that enable MATLAB to access other Windows applications and vice versa.

Serial Port I/O Functions

Functions that enable you to interact with devices connected to your computer's serial port.

Index

O