MATLAB®

High Performance Numeric Computation and Visualization Software

For UNIX Workstations

Release Notes
Version 4.1

The MATH WORKS Inc.

MATLAB 4.1 Release Notes (May 1993)

Printing History

January 1993	First Printing
May 1993	Second Printing (updated)

The MathWorks, Inc.
24 Prime Park Way, Natick, Mass. 01760
Phone: (508) 653-1415 FAX: (508) 653-2997

Email addresses:

`tech@mathworks.com`	Technical support.
`suggest@mathworks.com`	Product enhancement suggestions.
`bugs@mathworks.com`	Bug reports.
`doc@mathworks.com`	Documentation error reports.
`subscribe@mathworks.com`	Subscribing user registration.
`service@mathworks.com`	Order status, license renewals, passcodes.
`info@mathworks.com`	Sales, pricing, and general information.

Contents

MATLAB 4.1

Introduction

This document provides additional information not found elsewhere in the documentation that accompanies MATLAB 4.1. More specifically, it

- Describes features that were added or enhanced after the formal documentation went to press.
- Discusses important bug fixes.
- Characterizes some of the more subtle aspects of MATLAB's behavior.

MATLAB 4.1 Documentation

MATLAB comes with an extensive set of documentation including both an on-line Help facility as well as a complete set of printed manuals. The on-line help provides readily accessible reference information on all MATLAB commands. It is accompanied by a suite of interactive demos that serve to illustrate many of MATLAB's powerful new features. These on-line resources are augmented with a full set of printed documentation consisting of the following items:

- The *MATLAB 4.1 Release Notes* (this document).
- The *Installation Guide* describes how to install MATLAB.
- The *New Features Guide* provides information useful in making the transition from MATLAB 3.5 to MATLAB 4.
- The *MATLAB User's Guide* covers platform-specific aspects of using MATLAB and includes a tutorial that introduces basic MATLAB functionality.
- The *MATLAB Reference Guide* contains an alphabetical compendium of all MATLAB commands.
- The *External Interface Guide* describes the external interfaces to MATLAB, including methods for importing and exporting data as well as facilities for dynamically linking your own C and Fortran code with MATLAB.

New Features

This section describes features that were added or enhanced since the 4.0 release of MATLAB.

New M-Files

This section provides a list of M-files that are not described in the *MATLAB Reference Guide*. You can obtain more information on any of these M-files using MATLAB's on-line Help facility. For example, typing

```
help cedit
```

at the MATLAB prompt returns information on the `cedit` command.

General MATLAB Commands

`cedit`	Set parameters for controlling command-line editing and recall facility
`ls`	List the contents of the current directory
`pwd`	Show the current working directory
`matlabroot`	Return the directory in which MATLAB was installed
`tempname`	Return a name suitable for use in creating a temporary file
`whatsnew`	Display toolbox README files
`version`	Display the MATLAB version that you are running

Language Functions

`mexdebug`	Enable in situ debugging of MEX-files
`nargchk`	Check the number of input arguments

Elementary Functions

`sec, csc, cot`	Secant, Cosecant, and Cotangent
`sech, csch, coth`	Hyperbolic Secant, Cosecant, and Cotangent
`asec, acsc, acot`	Inverse Secant, Cosecant, and Cotangent
`asech, acsch, acoth`	Inverse Hyperbolic Secant, Cosecant, and Cotangent

Special Matrices

`gallery`	A couple of small test matrices
`pascal`	Pascal matrix

Special Functions

`gcd`	Greatest common divisor
`lcm`	Least common multiple
`expint`	Exponential integral
`cart2pol`	Transform Cartesian coordinates to polar
`cart2sph`	Transform Cartesian coordinates to spherical
`pol2cart`	Transform polar coordinates to Cartesian
`sph2cart`	Transform spherical coordinates to Cartesian

Matrix Functions

`qrdelete`	Delete a column from the QR factorization
`qrinsert`	Insert a column in the QR factorization
`polyeig`	Polynomial eigenvalue solver

Data Functions

`gradient`	Approximate gradient
`subspace`	Angle between two subspaces

Polynomial Functions

`polyder`	Polynomial derivative

Function functions

`ode23p`	Solve differential equations, low order method, displaying plot

Sparse Functions

`sparsfun`	Sparse auxiliary functions and parameters

2-D Graphics

`comet`	Comet plot
`stem`	Plot discrete sequence data

3-D Graphics

`comet3`	3-D Comet plot
`slice`	Volumetric slice plot
`waterfall`	Waterfall plot

General Purpose Graphics

`imagesc`	Scale data and display as image
`ishold`	Return 1 if hold is on
`newplot`	Graphics M-file preamble to handle the `NextPlot` property
`whitebg`	Set default figure background color to white
`graymon`	Set default figure properties for gray-scale monitor
`terminal`	Set graphics terminal type
`gco`	Return handle of current object

User Interface Primitives

`rbbox`	Rubberband box for region selection
`uigetfile`	Dialog box for obtaining name of existing file
`uiputfile`	Dialog box for specifying name of new file

Color Control

`contrast`	Gray scale color map to enhance image contrast
`prism`	Color map of prism colors
`white`	All white monochrome color map

Sound Functions

`auread`	Read Sun audio file
`auwrite`	Write Sun audio file
`lin2mu`	Linear to mu-law conversion
`mu2lin`	Mu-law to linear conversion

String Functions

`blanks`	A string of blanks
`deblank`	Strip trailing blanks from end of a string
`findstr`	Finds one string within another
`isletter`	True for letters of the alphabet

Low-Level File I/O Functions

`feof`	Test for end-of-file
`fgetl`	Return the next line of the file as a string, without newline
`fgets`	Return the next line of the file as a string, with newline
`frewind`	Rewind an open file

New Language Features

This section describes additional enhancements to the MATLAB language.

Toolbox Path Cache

The names of the M-files and MEX-files that reside in the `toolbox` subdirectories are now placed in a cache when MATLAB starts up. This enhancement substantially reduces the time it takes to locate functions invoked from the command line as well as those encountered during the process of compiling M-files, particularly when running MATLAB in a networked environment.

Except for the noticeable increase in performance, this change is essentially transparent to the user. The only subtlety is that because the *toolbox* M-files are cached, MATLAB will no longer automatically detect that a new *toolbox* M-file has been added or modified. M-files that are located elsewhere on your `MATLABPATH` are *not* cached and MATLAB continues to detect changes and additions to files in these directories. Thus it is convenient to think of the M-files that reside in the toolbox directories as "read-only" entities. (You should not really have reason to modify these files anyway.)

The cache is rebuilt anytime the `matlabpath` command (or the `path` command, which calls `matlabpath`) is executed. Thus if you find it necessary to refresh the cache, you can issue the command

```
matlabpath(matlabpath)
```

If you have additional directories of M-files that you would like to have cached (e.g., a library of M-files that you use frequently) that don't reside under the standard MATLAB `toolbox` directory, you can do this by setting the environment variable `TOOLBOX` to include these additional directories. Any subdirectories at or below the specified directories will be cached. Note, however, that if you do this, you should make sure that you also include the standard MATLAB toolbox directory. For example

```
setenv TOOLBOX $MATLAB/toolbox;/home/me/M-file_lib
```

"cd" and "which" Return Values

The `cd` and `which` commands now return values when invoked as functions – that is, when called with left-hand side arguments. For instance, you can now say

```
old_dir = cd;
cd some/new/directory
cd(old_dir)       % Return to original directory
```

This behavior is based on the notion of command/function duality as described in the *New Features Guide*.

"eval" Error Trapping

The `eval` function now provides an optional mechanism for detecting and trapping error conditions that occur during the evaluation of the argument expression. To use this feature, you simply provide a second string argument to `eval` containing an expression or the name of a function to be executed in the event that an error is encountered during the evaluation of the primary argument. For example,

```
eval('cd new/dir', 'disp(''cd was unsuccessful'')');
```

will either set the current directory to `new/dir` or will display the message `cd was unsuccessful` if the `cd` command returns an error. Note the use of double quotes to delineate the string argument to `disp` within the string argument to `eval`.

"lasterr" Built-In Function

MATLAB now provides a new built-in function `lasterr` that enables you to query the most recent error that has occurred. MATLAB automatically sets this function to the corresponding error message text. You may find it useful to set `lasterr` to a known value (perhaps the empty string `''`) so that you can use it to determine when an error has occurred. This approach can be particularly useful when used in connection with `eval` error trapping as described above. Consider, for example, a simple M-file that prompts the user for the name of a new directory.

```
while (1)
      dirname = input('Enter new directory: ','s');
      lasterr('');
      eval('cd(dirname)', '')
      if isempty(lasterr)
            break
      end
      disp('That was an invalid name. Please try again')
end
```

Note that the empty string passed as the second argument to `eval` prevents it from erroring out and allows the M-file to retain control in spite of an error.

In Situ MEX-File Debugging

On most UNIX platforms it is now possible to debug MEX-files while they are running within MATLAB. Complete source code debugging is possible including setting breakpoints, examining variables, and stepping through the source code line-by-line.

To debug a MEX-file from within MATLAB, you must first compile the MEX-file with debugging information by specifying the `-g` option to `cmex` or `fmex`. For example,

```
cmex -g yprime.c
```

You also need to start MATLAB up in a debugger. Do so by specifying the name of the debugger you want to use with the `-D` option when starting MATLAB. For instance, to use `gdb`, the GNU Debugger, you would type

```
matlab -Dgdb
```

Once the debugger has loaded MATLAB into memory, you can start it by issuing a `run` command. Now, from within MATLAB, enable MEX-file debugging by typing

```
mexdebug on
```

at the MATLAB prompt. Then run the MEX-file you want to debug as you would normally (either directly or via some other function or script). Before executing the MEX-file, you will be returned to the debugger.

You may need to tell the debugger where the MEX-file was loaded, in which case MATLAB will display the appropriate command for you to issue. At this point you are ready to start debugging. You can list the source code for your MEX-file and set break points in it. It is often convenient to set one at `mexFunction` so that you stop at the beginning of your MEX-file. Then to cause execution of the MEX-file to commence, simply issue a `continue` command to the debugger.

Once you hit one of your breakpoints, you can make full use of any facilities your debugger provides to examine variables, display memory, or look at registers, etc. Please refer to the documentation provided with your debugger for additional details on it use.

If you are at the MATLAB prompt and want to return control to the debugger, you can issue the command

```
mexdebug stop
```

This facility allows you to gain access to the debugger so that you can set additional breakpoints or examine source code. To resume execution simply issue the `continue` command to the debugger.

New Mathematical Functions

Polynomial Eigenvalue - polyeig

`polyeig` is a new function that solves the polynomial eigenvalue problem of degree *p:*

$$(A_0 + \lambda A_1 + \lambda^2 A_2 + \ldots + \lambda^p A_p)\, x = 0$$

using the notation

```
[X, E] = polyeig(A0, A1, A2, ..., Ap)
```

where the input is $p + 1$ square matrices, $A_0, A_1, A_2, \ldots, A_p$, all of the same order, n. The output is an n-by-np matrix `X` whose columns are the eigenvectors, and a vector of length np, `E`, whose elements are the corresponding eigenvalues.

Exponential Integral - expint

`expint` is a new function that computes the exponential integral

$$y = \int_x^\infty \frac{e^{-t}}{t}\, dt$$

using the notation

```
y = expint(x)
```

If `x` is in the range [-38, 2], a series expansion is used; otherwise, a continued fraction representation is employed.

New Graphics Features

This section describes additional enhancements to MATLAB's graphics capabilities.

Font Support

Text objects now allow you to control their font size and typeface on a per object basis. The complete set of 35 fonts now found on most PostScript printers is supported; however, you must have access to these fonts in order to see them displayed on your screen. Since axes labels and titles are themselves text objects, this addition means that you now have much greater control over the appearance of your plots. Similarly, you can also specify the typeface and size used for axes enumerations. The “Text Properties” section later in this document provides additional information on these properties and how to use them.

24-Bit Color Support

MATLAB 4.1 includes support for 24-bit color on X11 display servers capable of rendering it. This enhancement means that you can create graphics objects having more than 256 distinct colors and you can display images containing more than 256 pixel values. Systems with 24-bit color also avoid the problems associated with sharing color maps amongst different Figure Windows as well as with other simultaneously running applications.

Note that to take advantage of this feature, you must configure your X11 display server to run in 24-bit `TrueColor` or `DirectColor` mode. For instance, on a Sun workstation with a GS graphics accelerator, you must specify the following command line option when starting your X server

```
-dev /dev/cgtwelve0 defdepth 24
```

Expanded Printer Support

MATLAB 4.1 now provides support for a greatly expanded set of printers including the complete line of HP LaserJets. The table shown below provides a complete list of newly supported devices. To use one of these printers, simply specify the device type as an optional parameter to the `print` command as shown

```
print -dlaserjet filename
```

or make it your default by editing the `dev` option in `printopt.m`.

Printer	Option
Hewlett-Packard LaserJet	`-dlaserjet`
Hewlett-Packard LaserJet+	`-dljetplus`
Hewlett-Packard LaserJet IIP	`-dljet2p`
Hewlett-Packard LaserJet III	`-dljet3`
Hewlett-Packard DeskJet 500C (1-bit color)	`-dcdeskjet`
Hewlett-Packard DeskJet 500C (24-bit color)	`-dcdjcolor`
Hewlett-Packard DeskJet 500C (B/W)	`-dcdjmono`
Hewlett-Packard DeskJet and DeskJet+	`-ddeskjet`
Hewlett-Packard PaintJet	`-dpaintjet`
Hewlett-Packard PaintJet XL	`-dpjetxl`
Cannon BubbleJet BJ10E	`-dbj10e`
DEC LN03	`-dln03`
Epson-compatible dot matrix printer (9- or 24-pin)	`-depson`
Epson-compatible (9-pin, triple resolution)	`-deps9high`
Epson LQ-2550	`-depsonc`
Fujitsu 3400/2400/1200 (LQ-2550 compatible)	`-depsonc`

This support is implemented transparently using the GhostScript post-processor, which automatically converts MATLAB-generated PostScript output into a form appropriate to the specified device.

Terminal Graphics Support

MATLAB 4.1 now contains comprehensive support for Tektronix-based graphics terminals and terminal emulators. Both Tektronix 4010/4014 and Tektronix 4100/4105 devices are supported within the context of the following limitations:

Tektronix 4010/4014

- Only black and white is supported.
- There is only one marker size for the "`o`" and the "`.`" markers.
- Polygons and rectangles cannot be filled and are rendered as outlines. For example, `surf(eye(4))` and `mesh(eye(4))` look exactly alike. As a result, hidden line/surface removal is not supported.
- There are four different font sizes derived from the value of the `FontSize` property.
- There is no support for interpolation, texture mapping, images, movies, or screen capture.
- It is not possible to set a figure's position.
- Only minimal clipping is supported. Plots do not fully clip to axes.
- Text rotation does not rotate text but displays it in a different order. For example, a text rotation of zero degrees displays text left to right, a text rotation of 90 degrees displays text top to bottom, a text rotation of 180 degrees displays text right to left (backwards), and a text rotation of 270 degrees displays text bottom to top.

Tektronix 4100/4105

- Only black and white and 3-bit color are supported. The default Tektronix color map (black, white, red, green, blue, cyan, magenta, yellow) is used for all graphics. Changing the color map of your terminal before invoking MATLAB will cause incorrect colors to appear.
- There is only one marker size for the "`o`" and the "`.`" markers.
- Filled rectangles and polygons use a Tektronix dithered fill pattern based on the RGB value in the figure's color map to which the rectangle or polygon's color corresponds.
- Only one font size is supported. The other font sizes are too large to be displayed correctly.
- There is no support for interpolation, texture mapping, movies, or screen capture. Images are supported and are dithered based on the screen depth of your terminal.
- It is not possible to set a figure's position.
- Only minimal clipping is supported. Plots do not fully clip to axes.
- Line colors use the closest entry in the Tektronix color map.

Using a Supported Graphics Terminal

To use a Tektronix terminal or emulator with MATLAB 3.5, you either used a supplied terminal personality file (TPF) or created one. With MATLAB 4.1, terminal characteristics for the supported graphics terminals are defined in a single file, called `terminal.m`, stored in the MATLAB toolbox. The table below lists the terminals whose characteristics are included in the `terminal.m` file.

Graphics Terminal	Option
C.Itoh	`citoh`
Ergo	`ergo`
Graphon	`graphon`
Hewlett-Packard 2647	`hp2647`
Human Designed Systems	`hds`
Macintosh with VersaTerm (Tektronix 4010/4014)	`versa`
Macintosh with VersaTerm (Tektronix 4100)	`versa4100`
Macintosh (Color/Grayscale) with VersaTerm (Tektronix 4105)	`versa4105`
MS-DOS Kermit 2.23	`kermit`
Retrographics card	`retro`
Selanar graphics 100	`sg100`
Selanar graphics 200	`sg200`
Tektronix 4010/4014	`tek401x`
Tektronix 4100	`tek4100`
Tektronix 4105	`tek4105`
VT240 and VT340 Tek mode	`vt240tek`
Wyse WY–99GT	`wyse`
xterm, Tektronix graphics	`xtermtek`

If you are using a graphics terminal or terminal emulator listed above, follow either step below to identify the terminal to MATLAB:

- To identify the terminal for the current MATLAB session only, specify the terminal code (from the table above) by entering the following MATLAB command:

  ```
  terminal terminal-code
  ```

  For example, to run MATLAB from a Tektronix 4014, enter:

  ```
  terminal tek401x
  ```

- To identify the terminal automatically each time MATLAB starts up, include the `terminal` command described above in the `startup` M-file, `startup.m`. The next time you run MATLAB, the terminal characteristics will be applied automatically.

If the graphics terminal or terminal emulator you want to use is not listed in the above table, you must create an M-file specifying the its characteristics as described in the section "Using an Unsupported Graphics Terminal" in the back of this document.

Keeping MATLAB from Connecting to an X Server

If you are running MATLAB remotely using a graphics terminal or terminal emulator, you must make sure that MATLAB does not automatically connect to an X Windows display server. If it does, your graphics will display on that X server and you will not be able to change the terminal settings.

The following are two ways to make sure MATLAB does not connect to an X Windows display server:

- You can specify a nonexistent display or use an invalid display name on the MATLAB command line:

  ```
  matlab -display null
  ```

- Usually when running under the X Windows environment, a shell environment variable called `DISPLAY` is set to the name of the X Windows display server on which windows are to appear. You can unset this environment variable before you run MATLAB.

  For the C shell:

  ```
  unsetenv DISPLAY
  ```

  For the Bourne shell:

  ```
  unset DISPLAY
  ```

Line Width

It is now possible to set the line width for graphics objects using the `LineWidth` property as shown below

```
set(handle, 'LineWidth', 2)
```

where the new width is specified in points (1/72 inch). This property applies to axes, lines, surfaces, and patches.

Line Style Order

It is now possible to specify which line types (e.g., solid, dashed, etc.) are used and what order they are used in when plotting multi-line data. This feature is analogous to the `ColorOrder` facility that allows you to specify the order in which line colors are used.

Texture Mapping Surfaces – The texturemap FaceColor

Texture mapping is a technique for mapping a 2-D image onto a 3-D surface by transforming color data so that it conforms to the surface. It allows you to apply a "texture," such as bumps or wood grain, to a surface without performing the geometric modeling necessary to actually create a surface with these features. The color data can also be any image, such as a scanned photograph.

MATLAB texture maps a surface by assigning the texture color data to the surface's `Cdata` property. While a surface's color is always determined by the values contained in its `Cdata` property, texture mapping differs in that the dimensions of the `Cdata` array can be different from the surface's `Zdata`. This allows you to apply an image of arbitrary size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

You must set the surface's `FaceColor` property to `texturemap` before setting `Cdata` to an arbitrarily sized array. You can do this in one of two ways:

- Use the `surface` object creation function, which allows you to specify object properties when you create the surface.
- Use `set` to change the `FaceColor` property of an existing surface and to specify new data for `Cdata`.

The following example shows how to map the mandrill image to a cylindrical surface. By default, MATLAB maps the image to the entire surface; however, this example pads the image data so that the mandrill is displayed on half of the cylinder, thus making it easier to recognize the image.

```
load mandrill
colormap(map)
[x,y,z] = cylinder;
Xhalf = [ones(480,375)*max(max(X))/2, X, ...
      ones(480,125)*max(max(X))/2];
surface(x,y,z, 'FaceColor','texturemap', ...
      'EdgeColor','none', 'Cdata',flipud(Xhalf))
view(3)
```

The size of the mandrill data is 480 by 500, so an additional 500 columns of data are added to make the image occupy half of the cylinder. The columns are added before and after the existing data so as to orient the image correctly for the default 3-D view.

The values used to pad the data are set to one-half of the maximum value of the mandrill data. This sets the color of the half cylinder that does not contain the mandrill to the color that is in the middle of the mandrill's color map (`map`).

In addition to setting the `FaceColor` to `texturemap`, the `EdgeColor` is set to `none` to remove the grid lines.

Since image data is normally displayed with `'ij'` axis numbering, the mandrill data is reversed in the vertical direction using `flipud`. (See the `axis` reference page for more information.)

To produce the same picture using high-level graphics functions, you must obtain the handle of the surface and change the relevant properties:

```
[x,y,z] = cylinder;
h = surf(x,y,z);
set(h, 'FaceColor','texturemap', 'EdgeColor','none',...
      'Cdata',flipud(Xhalf))
```

Semi-Automatic Axes Limits

You can now specify one extreme of the coordinate axes or color axis range (`XLim`, `YLim`, `ZLim`, or `CLim` properties) and allow the other extreme to auto-scale. This is done by setting the limit that you want to auto-scale to plus or minus `inf`.

For example, this statement forces the minimum limit of the *x*-axis to 0, but allows the maximum limit to auto-scale:

```
set(gca,'XLim',[0 inf])
```

To set the maximum limit to 40 and let the minimum limit auto-scale, use

```
set(gca,'XLim',[-inf 40])
```

The minimum will always be less than the maximum, regardless of which extreme is specified.

Logarithmic Scaling

MATLAB can now plot negative data on a logarithmic scale. It cannot, however, do so using negative and positive data simultaneously on the same axes. If there is any positive data, the negative data is ignored and the lower limit auto-scales so that the axes display the smallest positive data value. MATLAB creates a negative log axis only when all the data plotted to an axes is negative.

As described above, setting axes limits (the `XLim`, `YLim`, `ZLim`, or `CLim` properties) to plus or minus `inf` cause the corresponding limit to auto-scale. With logarithmic scaling, an axis limit of zero can also be used to indicate that auto-scaling is desired. The axis minimum (if the limits are `[0 +n]`) or maximum (if the limits are `[-n 0]`) will be chosen to accommodate the value of the data that is closest to zero.

For example, these statements

```
plot(rand(1:10))
set(gca,'YLim',[0 0.1])
set(gca,'YScale','log')
```

set the upper limit to 10^{-1} but allow the lower limit to auto-scale (except for data values of zero, which map to minus infinity).

If you set axis limits before plotting any data, MATLAB attempts to select appropriate limits and then revises these limits when you plot actual data.

If, while using logarithmic scaling, you specify a negative value for a minimum axis limit and a positive value for the corresponding maximum axis limit, MATLAB treats the lower limit as if it were zero and auto-scales that limit.

Window Mouse Button Functions

You can define callback functions that execute as a result of mouse button actions that occur within a Figure Window. These action are: *pressing* a mouse button, *moving* the mouse while holding a button pressed, and *releasing* a mouse button. The callback functions are defined for a particular Figure Window using the `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn` figure properties.

A typical scenario might be to have the `WindowButtonDownFcn` define the `WindowButtonMotionFcn`. This way, the `WindowButtonMotionFcn` (which may, for example, cause the pointer to drag any graphics object it touches) is only active after the mouse button is pressed.

The `WindowButtonUpFcn` can undefine the `WindowButtonMotionFcn` (by setting it to the empty string: `''`) so that pointer motion no longer affects the display (e.g., no longer drags graphics objects). In this case, the `WindowButtonMotionFcn` continues to execute until the mouse button is released, even if the pointer is moved outside the Figure Window.

The figure properties `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn` are affected by the figure `Interruptible` property described later in this document.

Object Mouse Button Down Functions

You can define a function that executes whenever you use the mouse on a Handle Graphics™ object. All Handle Graphics objects (except uimenus and the root) support a new property, `ButtonDownFcn`, that allows you to define a callback function to execute whenever you press a mouse button while the pointer is on the object.

You define the callback function as a string on which MATLAB performs an `eval('string')` when the function is invoked. Therefore, the string can be any valid MATLAB expression or the name of an M-file. The string is executed in the MATLAB workspace.

Note that the callback function defined for an object's `ButtonDownFcn` property is separate from the callback functions you can define for the uicontrol `CallBack` property and the figure `WindowButtonDownFcn` property. In fact, all callback functions can operate simultaneously. However, it is important to understand the sequence in which the callback functions execute and the criteria established for their selection. See the section "Order of Execution of Button Down Functions" later on in this document for a detailed discussion of this topic.

Sliders Now Display Arrows

Uicontrol sliders (`'style'` set to `'slider'`) now display arrows if the aspect ratio of the width to the height of the control is in the following range

aspect ratio < 1/4	[Vertical]
aspect ratio > 4	[Horizontal]

Use the `Position` property to specify aspect ratio.

When you use the mouse on an arrow, the slider position indicator (and the associated slider `Value`) moves in the indicated direction by a value of 1/100th of the total range. Clicking the mouse while the pointer is in the trough moves the slider 1/10th of the total range.

The following picture illustrates how the slider looks with an aspect ratio of 7.

Simplified Setting of Uicontrol Parent

Analogous to *uimenu* objects, *uicontrol* objects now accept a parent handle as their first argument (without identifying the value as a property). Thus a synopsis of the revised uicontrol syntax is

```
h = uicontrol('PropertyName',PropertyValue,...)
h = uicontrol(handle, 'PropertyName',PropertyValue,...)
```

where `handle` is the handle of the figure object in which the uicontrol is to appear. If no handle is specified, the uicontrol is created as a child of the current Figure Window. It is not possible to change a uicontrol object's parent once it has been created.

Uicontrol Editable Text Behavior

Editable text uicontrols (`'Style'` set to `'edit'`) now operate in two modes: *single line* and *multiline*. The values of the `Min` and `Max` properties determine which mode is used.

Single line mode is used whenever

```
(max - min) <= 1
```

In *single line* mode, typing a carriage return executes the uicontrol's callback function. Therefore, you can enter only a one line string in this mode.

Multiline mode is used whenever you specify values for `Min` and `Max` such that

```
(max - min) > 1
```

In *multiline* mode, entering a carriage return does *not* cause the callback function to execute, so you can type more than one line (containing carriage returns) into the edit box. To apply the string (i.e., execute the callback function), type **Control-Return** or move the pointer off the edit box.

Note that this represents a change in the default behavior of this uicontrol. Previously, multiline mode was the default and the values `Min` and `Max` were ignored. However, since the default value for `Min` is 0 and the default value for `Max` is 1, single line mode is now used when you do not specify values for `Min` and `Max`.

Drawnow Discard

The `drawnow` command now takes an optional argument, `discard`. With this option `drawnow` essentially performs the opposite of its normal operation – it discards all pending events, including drawing, mouse events, and keyboard events.

This option can be useful if you want to change object properties temporarily while performing an operation and then change them back without causing the Figure Window to redraw. For example, you may want to change figure properties before printing hardcopy, but do not want to see (or wait for) the window contents to redraw using these new properties, and redraw again when you reset the properties.

If you type

```
set(gcf,'Color',’r’),drawnow discard
```

the figure background color does not change. However, if you then list the value of the `Color` property it is returned as red:

```
get(gcf,'Color')

ans =
     1  0  0
```

If you resize the window, thereby generating an event that causes the window to redraw, the figure background is redrawn in red.

New Handle Graphics Object Properties

MATLAB graphics are build around an object-oriented system referred to as Handle Graphics. Associated with each graphical object are a number of properties that determine its current behavior and appearance.

Querying Handle Graphics Object Properties

You can examine the set of properties that a particular graphics object supports by entering the statement:

```
get(handle)
```

where `handle` is the object's handle. Similarly, you can type

```
set(handle)
```

to see lists of all possible values associated with each property.

Many Handle Graphics object types now have additional properties beyond those documented in the *MATLAB Reference Guide*. The following sections cover in detail the new properties for each of the basic object types.

Properties Common to all Graphics Objects

The `ButtonDownFcn` and `Interruptible` properties described below are available with all graphics objects except for the root object.

ButtonDownFcn Property

`ButtonDownFcn`   `string`

The `ButtonDownFcn` property enables you to define a function that executes whenever you press a mouse button while the pointer is over the corresponding object. You define the callback function as a string that is passed to `eval`. Therefore, the string can be any valid MATLAB expression or the name of an M-file. The string is executed in the MATLAB workspace. Note that for uimenu objects, the `CallBack` function supersedes the `ButtonDownFcn`; however, uicontrol objects support both their own `CallBack` function as well as a `ButtonDownFcn`. See the section "How Objects are Selected" later in this document for a description of how these mechanisms interact.

Interruptible Property

`Interruptible`   `yes | no`

This property controls whether or not the action defined by a `ButtonDownFcn` can be interrupted during its execution. The default is `no`, which means MATLAB does not allow other functions to execute until the currently executing function finishes.

For figure objects, this property also affects whether or not `WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn`, and `KeyPressFcn` callback functions can be interrupted during execution.

For uimenu and uicontrol objects, this property also controls whether or not a uicontrol or uimenu callback function can be interrupted during its execution. Again, the default value of `no` means MATLAB does not allow other callback functions to begin execution until the currently executing callback finishes.

It also means that the user cannot, for example, change the value of the current figure (i.e., the value returned by `gcf`) by changing the window focus. This is particularly useful in preventing impatient users from disrupting a lengthy callback function by clicking around the display with the mouse while the callback executes.

When an object's `Interruptible` property is `yes`, and another callback is selected to run, the following sequence occurs:

1. The executing callback encounters a `drawnow`, `pause`, or `getframe` command that causes MATLAB to process the event queue.
2. Execution of the interrupted callback is suspended.
3. The interrupting callback function executes to completion (unless it is interruptible and gets interrupted).
4. The interrupted callback resumes execution. The original state of MATLAB (e.g., the current figure, the current axis, workspace variables, etc.) is *not* restored.

This sequence is repeated for each level of interruption.

Clearly, it is possible for the interrupting callback function to alter conditions that affect the execution of the interrupted callback. When an object has its `Interruptible` property set to `yes`, the task of restoring (or at least monitoring) the conditions that existed when a callback is interrupted must be handled by the individual callback functions.

See "Understanding Window Events and Callback Functions" in the section "Notes on MATLAB's behavior" later in this document for a more complete discussion of this property.

Root Properties

This section describes figure properties not listed in the `root object` reference page of the *MATLAB Reference Guide*. These properties pertain to terminal characteristics used when running MATLAB via a remote terminal.

`ScreenDepth` `bits per pixel`

This property indicates the depth of the display bitmap or the number of bits per pixel. Thus, the maximum number of simultaneous colors that can be displayed on the current graphics device is 2 raised to this power. `ScreenDepth` supersedes the `BlackAndWhite` property described in the *MATLAB Reference Guide*.

If MATLAB successfully connects to an X Window display server, it automatically determines whether it is running on color or monochrome hardware and sets the value of this property to the depth of your display.

If you are not using X Windows, you should set this property. If you don't set the property or specify a value of zero, MATLAB uses a default value of 3 when the `TerminalProtocol` property is set to `tek410x`, and a value of 1 otherwise.

If you are using X Windows, setting this property to zero re-enables automatic screen depth detection.

Caution

> If `ScreenDepth` is set to an incorrect value, graphics may not display properly.

`TerminalHideGraphCommand` `string`

This property specifies the escape sequence that MATLAB issues to hide the graph window when switching from graph mode back to command mode. This property is only used by the terminal graphics driver. Consult your terminal manual for the correct escape sequence.

`TerminalOneWindow` <u>`yes`</u> | `no`

This property indicates whether or not there is only one window on your terminal. If the terminal uses only one window, MATLAB waits for you to press a key before it switches from graphics mode back to command mode. This property is only used by the terminal graphics driver.

`TerminalProtocol` <u>`none`</u> | `x` | `tek401x` | `tek410x`

This property tells MATLAB what type of terminal you are using. Specify `tek401x` for terminals that emulate Tektronix 4010/4014 terminals.

Specify `tek410x` for terminals that emulate Tektronix 4100/4105 terminals. If you are using X Windows and MATLAB can connect to your X display server, this property will automatically be set to `x`.

Once this property is set, it cannot be changed unless you quit and restart MATLAB. To find out how to prevent MATLAB from connecting to an X display server, see the section titled "Keeping MATLAB from Connecting to an X Display Server."

`TerminalShowGraphCommand` `string`

This property specifies the escape sequence that MATLAB issues to display the graph window when switching from command mode to graph mode. This property is only used by the terminal graphics driver. Consult your terminal manual for the appropriate escape sequence.

Figure Properties

This section describes figure properties not listed in the `figure` reference page of the *MATLAB Reference Guide*.

BackingStore Property

`BackingStore` <u>`on`</u> | `off`

When `BackingStore` is on, MATLAB stores a copy of each Figure Window in an off screen pixel buffer. When an obscured Figure Window is exposed, its contents are copied from this buffer rather than being regenerated, thereby increasing the speed at which the screen is redrawn.

While this is generally a desirable situation, these buffers do consume system memory. If memory limitations occur, you can set `BackingStore` to `off` to disable this feature and release the memory used by these buffers.

Not all machines that MATLAB runs on support `BackingStore`. If your machine does not support it, setting the `BackingStore` property will result in a warning message and otherwise has no effect.

FixedColors Property

`FixedColors` `n by 3 matrix (read only)`

This property lists all fixed colors defined for the figure. Fixed colors are independent of the figure color map. They are directly defined colors that MATLAB uses when you explicitly specify the color of an object.

For example, if you enter the following statement

```
line('Color',[.2 .4 .6])
```

and then get the value of the `FixedColors` property

```
fc = get(gcf,'FixedColors')
```

`fc` is assigned the values

```
fc =
     0 0 0
     1.0000 1.0000 1.0000
     0.2000 0.4000 0.6000
```

Note that black (`[0 0 0]`) and white (`[1 1 1]`) are created because the figure has a black background and white text by default. If you change the figure `Color` property to green, for example, the black entry is replaced by `[0 1 0]`.

KeyPressFcn Property

`KeyPressFcn` `string`

The `KeyPressFcn` property is analogous to the `ButtonDownFcn` property in that it enables you to specify a callback function that is to be invoked any time a key is pressed when the corresponding window has focus. The callback routine can query the figure `CurrentCharacter` property to determine what character was typed thereby causing the callback to be executed.

MenuBar Property

`MenuBar` `none | figure`

This property enables you to display or hide the menu bar that sits at the top of Figure Windows. Note that Figure Window menu bars are not supported on all systems; however, for those that do, the default is to display them.

ShareColors Property

`ShareColors` `yes | no` (default: yes, underlined)

The `ShareColors` property affects the use of slots in the system color table. On systems having eight or fewer bits per pixel, color table slots are typically a precious resource and should be conserved in order to allow the maximum number of windows to render their contents with reasonable looking colors. When this property is set to `yes`, MATLAB is very careful about reusing existing color table slots whenever possible.

Occasionally, however, this behavior is undesirable as, for instance, when dynamically adjusting the color map for a specific window. In this case, the time required to readjust the color table assignments for any other windows is prohibitively expensive. Thus, under these circumstances, it is desirable that the window whose color map is being adjusted not share any of its color table slots with other windows, and hence the user should set this property to `no`.

WindowButtonDownFcn Property

`WindowButtonDownFcn` `string`

This property allows you to define a function for the particular Figure Window that MATLAB executes whenever a *button down* event occurs in that window (i.e., whenever a mouse button is *pressed* while the pointer is in the window).

MATLAB performs an `eval(string)` on the specified string. This means the string can be any valid MATLAB expression or the name of an M-file.

WindowButtonMotionFcn Property

`WindowButtonMotionFcn` `string`

This property allows you to define a function for the particular Figure Window that MATLAB executes whenever a *motion event* occurs in that window (i.e., whenever the pointer is *moved* within the Figure Window).

MATLAB performs an `eval(string)` on the specified string. This means the string can be any valid MATLAB expression or the name of an M-file.

WindowButtonUpFcn Property

`WindowButtonUpFcn` `string`

This property allows you to define a function for the particular Figure Window that MATLAB executes whenever a *button up* event occurs for that window (i.e., whenever a mouse button is *released*).

The button up event is associated with the window in which the preceding button down event occurred. Therefore, the pointer need not be in the Figure Window when the button is released to generate the button up event.

MATLAB performs an `eval(string)` on the specified string. This means the string can be any valid MATLAB expression or the name of an M-file.

Axes Properties

This section describes axes properties that are not listed in the `axes` reference page of the *MATLAB Reference Guide*

CurrentPoint Property

`CurrentPoint` 2 by 3 matrix

The axes `CurrentPoint` property contains the coordinates of two points that are defined by the location of the pointer. MATLAB updates this property continually. The axes `CurrentPoint` is derived from the figure `CurrentPoint` by translating it to axes coordinates.

Pointers exist in the 2-D space of the computer screen whereas MATLAB graphics objects exist in 3-D data space. To accommodate this difference, MATLAB returns the line perpendicular to the plane of the screen and passing through the pointer. It does so by providing the 3-D coordinates of the points on this line where it intersects the front and back surfaces of the axes volume. The axes volume is defined by its *x*, *y*, and *z* limits.

The returned matrix is of the form:

$$\begin{bmatrix} x_{back} & y_{back} & z_{back} \\ x_{front} & y_{front} & z_{front} \end{bmatrix}$$

The coordinates are returned in the data space of the current axes (i.e., the same units as the data plotted on the axes). The pointer does not have to be within the axes, or even the Figure Window; the coordinates are returned with respect to the requested axes regardless of the location.

The following example allows you to see the nature the data returned by the `CurrentPoint` property. It is instructive to try this example.

First create a 2-D plot of a sine wave (any 2-D plot will do):

```
t = 0:pi/20:2*pi;
plot(sin(t))
```

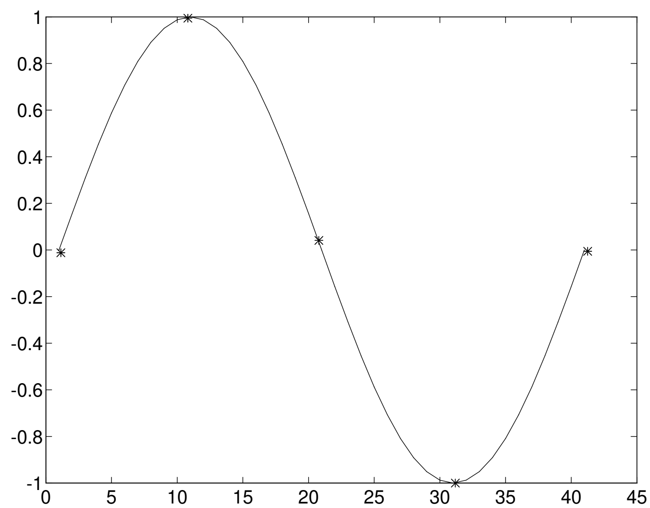
Next, set `hold` to `on` so that you can plot additional data in the same axes and use the `axis` command to freeze the scaling at the current limits:

```
hold on
axis(axis)
```

Now define a window button down function that retrieves and plots the data returned by the `CurrentPoint`. (Note that there is no carriage return after the second line; it is wrapped to fit on the page.)

```
set(gcf,'WindowButtonDownfcn',...
'p=get(gca,''CurrentPoint'');plot3(p(:,1),p(:,2),p(:,3),
''*'');plot3(p(:,1),p(:,2),p(:,3),'':'')')
```

You can now press a mouse button anywhere on the plot and invoke the window *button down* function. When you click a mouse button, a * marker appears on the plot at the location of the `CurrentPoint`. (Actually, what you see is the front end point.)



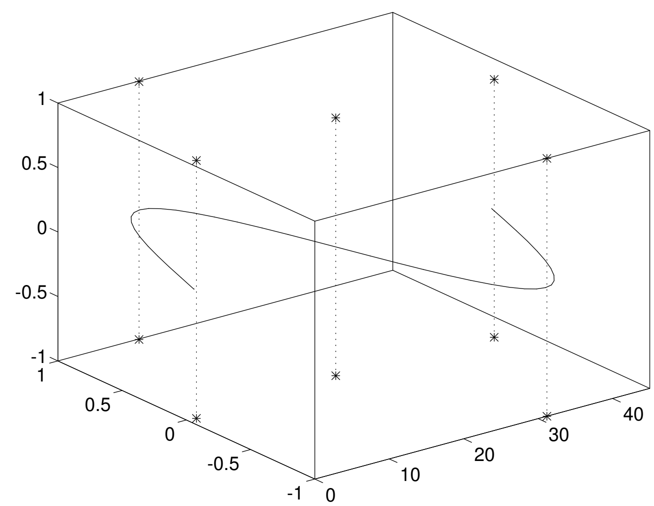
Now change to a 3-D view:

```
view(3)
```

From another point of view, you can see that there are two end points plotted (connected with a dotted line to make it easier to associate together the correct points).

In this example, the two end points lie on the z = 1 and the z = –1 planes. In the more general case, the points returned by `CurrentPoint` do not necessarily lie along an axis; they can be at an arbitrary orientation.

To illustrate this, click on the plot while the view is still set for 3-D. Once again the `CurrentPoint` location is displayed as single marker because you are looking along the line it defines. This time, however, the line is not parallel to either the *x*-, *y*-, or *z*-axis. If you again change the view, you can see the lines defined by the two end points.



Allowing the `CurrentPoint` to define a line segment enables you to implement a 3-D picking scheme. It is particularly useful when the view is set at some arbitrary orientation and you need to determine which object is first intersected by the line. You can do this by comparing the x, y, and z data of all the objects in the axes to see if and where they intersect the line defined by the end points.

Font Properties

The font characteristics used in rendering axes tick mark labels can be specified using the same font property values as those of the text object described later in this chapter. Specifically, you can specify the `FontName`, `FontSize`, `FontWeight`, and `FontAngle` properties.

LineStyleOrder Property

`LineStyleOrder        column-array of text strings`

This property enables you to specify which line types (e.g., solid, dashed, etc.) are used and what order they are used in when plotting multi-line data. For example to use solid, dashed, and dotted lines in that order you would say:

```
set(gca, 'LineStyleOrder', ['- ', '--', ': '])
```

or

```
set(gca, 'LineStyleOrder', '-|--|:')
```

Note that when using the first form shown, you must pad the single-character line type specifiers with spaces if you are also using two character specifiers so that all strings in the parameter matrix have the same length.

The default value is `'-'` indicating that all data is to be plotted as solid lines. Colors rather than line styles are used to differentiate them.

Title Property

`Title` `text handle`

This property holds the handle of the text object that is displayed as the figure title. You can use this handle to change the properties of the title text object or to create a title for a figure.

For example, the following statement changes the color of the current axes' title to red:

```
set(get(gca,'Title'),'Color','r')
```

To create a title, set `Title` to the handle of the text you want to use as the title:

```
set(gca,'Title',text('String','Profound Data'))
```

However, it is generally simpler to use the `title` command to initially create the title.

Properties Common to all Axes Children

EraseMode Property

`EraseMode` `normal` | `none` | `xor` | `background`

All children of axes except images now allow you to specify an erase mode that determines how they are erased and redrawn. This property applies to line, patch, surface, and text objects. Graphics objects are erased when you change their coordinate data (i.e., their `XData`, `YData`, and `ZData` properties)

This property is useful when creating animated sequences, where control of how individual objects redraw is necessary for improving performance and obtaining special effects.

`normal` mode redraws affected regions of the display, performing the three dimensional analysis necessary to ensure that all objects are rendered correctly. While this mode produces the most accurate picture, it also takes the most time. The other modes do not perform a complete redraw and therefore are considerably faster, but can produce a less accurate picture.

When the erase mode is `none`, the object is not erased when it is moved or destroyed.

`xor` mode erases the object by xor-ing its color with the color of the screen beneath it. When the object is erased, it does not damage the objects beneath it. However, when objects are drawn in `xor` mode, their colors are dependent on the color of the screen beneath them. Therefore, the object is colored correctly only when rendered over the figure background color.

`background` mode produces a properly colored object. However, the object is erased by drawing it in the figure's background color. This damages objects that are behind the erased object.

LineWidth Property

All children of axes except text and images now allow you to specify the thickness of the lines used when they are rendered. This property applies to line, patch, and surface objects.

```
LineWidth      width
```

The new width is specified in points (1/72 inch). The default value is 0.5 points.

Surface Properties

This section describes figure properties not listed in the `surface` reference page of the *MATLAB Reference Guide*.

MarkerSize Property

```
MarkerSize     point size
```

As with line objects, you can now specify the size of markers used with surfaces. This property is only used when the `LineStyle` property is set to one of the marker types (point, plus, star, circle or x-mark). The default value is 6 points.

Text Properties

This section describes figure properties not listed in the `text` reference page of the *MATLAB Reference Guide*.

Text objects now support font properties that allow you to specify font characteristics.

FontName Property

`FontName` `font family`

This property specifies the font family (e.g., Helvetica).

FontSize Property

`FontSize` `point size`

This property specifies the font size in points (one point = 1/72 inch).

FontWeight Property

`FontWeight` `light | normal | demi | bold`

This property specifies the character weight.

FontAngle Property

`FontAngle` `normal | italic | oblique`

This property specifies the character slant.

A font is defined by a number of characteristics in addition to its name. Not all combinations of these properties are allowed. MATLAB currently supports the eleven font families typically found on most PostScript printers. These include the basic four fonts Times, Helvetica, Courier, and Symbol as well as the now common Avant Garde, Bookman, Helvetica Narrow, New Century Schoolbook, Palatino, Zapf Chancery, and Zapf Dingbats.

For example, to specify 10-point Helvetica-BoldOblique, set the font properties to:

```
FontName    Helvetica
FontSize    10
FontWeight  bold
FontAngle   oblique
```

When the set of currently specified parameters does not correspond to an available font, MATLAB uses the following rules for selecting the current font:

1. MATLAB accepts `oblique` in place of `italic` and vice versa.
2. If a match is still not found, MATLAB ignores the `FontAngle`.
3. If a match is still not found, MATLAB ignores the `FontWeight`.
4. If a match is still not found, MATLAB ignores the `FontSize`.
5. If a match is still not found, MATLAB does not change the font.

When MATLAB generates hardcopy output, it does not attempt to determine what fonts are available on the hardcopy device before it sends output to the device.

The default font for text objects as well as for axes enumerations is 12-point Helvetica. When using TrueType fonts, and Times and Helvetica are unavailable, Times will be replaced with New Times Roman, Helvetica will be replaced with Arial, and Courier will be replaced with New Courier.

Uimenu Properties

Uimenu objects support several additional properties beyond those discussed in the `uimenu` reference page of the *MATLAB Reference Guide*.

BackgroundColor Property

`BackgroundColor` `ColorSpec`

This property specifies the color used to fill the rectangle defined by the menu. Specify this color using a vector of RGB values or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information on specifying color. The default color is a light gray which is defined by the RGB triple

```
[0.7020 0.7020 0.7020]
```

ForegroundColor Property

`ForegroundColor` `ColorSpec`

This property specifies the color of the text displayed on the uimenu object. Specify the color using a vector of RGB values or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information on specifying color. The default color is black.

Checked Property

`Checked` `on | off`

Setting this property to `on` causes a check mark to be placed next to the corresponding menu item. This feature can be used to create menus that list features that can be turned on or off at the discretion of the user. Note that there is no formal mechanism for indicating that a particular menu item can or cannot be checked.

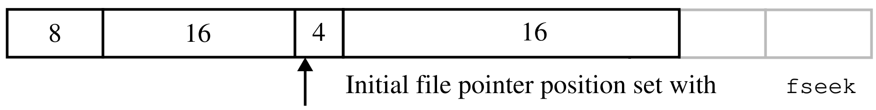
New Features for File I/O Functions

This section describes new features supported by the file I/O functions as well as additional file I/O functions not described in the *MATLAB Reference Manual*.

fread

```
[a,count] = fread(fid,size,'precision',skip)
```

The `skip` argument optionally specifies the number of bytes to skip *after* each read. It is used to extract the data in noncontiguous fields from fixed length records. For example, consider a file containing a series of records, each with fields of length 8, 16, and 4 bytes.

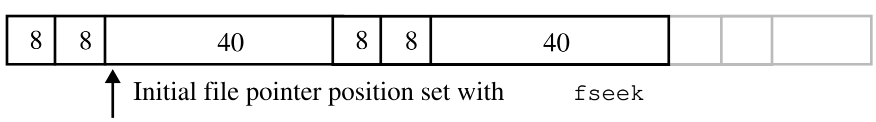


Suppose you want to read the data from the 4-byte fields into a matrix. Rather than repeatedly calling `fseek` to reposition the file pointer and `fread` to read the single field of data, you can instruct `fread` to skip 24 bytes between each read. Reading continues until the end of the file or until the specified `size` is reached. The following statements illustrate this procedure:

```
fid = fopen(‘filename’); % open the file for reading
status = fseek(fid,24,'bof'); % set file pointer
A = fread(fid,'float',24); %read to end of file
```

These statements open the file for reading and obtain the file identifier. `fseek` moves the file pointer 24 bytes from the beginning of the file to position it correctly for the first read. `fread` reads the 4-byte field (float precision) and then skips 24 bytes before again reading the next 4 bytes. Since no `size` argument is specified, this process continues until the end of the file is reached.

`fread` also supports a repetition factor that is useful for reading multi-element fields. You specify the repetition factor as a multiplier applied to the `precision` argument. For example, consider the following data structure:



The file is composed of a repeating sequence of records, each with fields of lengths 8, 8, and 40 bytes. The 40-byte field contains 40 `chars`. The following statements read 10, 40-byte fields into the columns of a 40 by 10 matrix:

```
fid = fopen('filename'); % open the file for reading
status = fseek(fid,16,'bof'); % set file pointer
A = fread(fid,[40,10]'40*char',16);
A = setstr(A')
```

Transposing the matrix `A` arranges the data as 10 lines of characters, each 40 columns wide.

Note that you must specify a nonzero `skip` argument in order to use a repetition factor.

fwrite

```
count = fwrite(fid,A,'precision',skip)
```

The `skip` argument optionally specifies the number of bytes to skip *before* each write. `fwrite` writes the elements of matrix `A` into the specified file, skipping the specified number of bytes before each write.

You do not need to call `fseek` first (unless you want to skip other parts of the file, such as a header) as you do when reading data.

For example, to write data to the first data structure discussed in the "fread" section, open the file for writing and specify a `skip` of 24 bytes:

```
fid = fopen('filename','w');
count = fwrite(fid,A,'float',24);
```

You can also specify a repetition factor with `fwrite`. This is useful for writing to multi-element fields within a data record. For example, to write data to the second structure discussed in the "fread" section, open the file for writing and specify a repetition factor of 40 and a `skip` of 16 bytes:

```
fid = fopen('filename','r+');
count = fwrite(fid,A,'40*char',16);
```

These statements open the file for updating and write the elements of `A` separated by 16 byte intervals. If the end of the file is reached before writing all elements in `A`, `fwrite` appends to the file by continuing to skip 16 bytes and writing 40 `chars` until all elements are written.

Note that you must specify a nonzero `skip` argument in order use a repetition factor.

feof

```
result = feof(fid)
```

`feof` tests whether the EOF (end of file) indicator has been set for the specified file. It returns 1 if the EOF indicator is set and 0 if it is not set.

The EOF indicator is set when `fread` attempts to read past the last character in the file. Moving the file position indicator to the end of the file using `fseek(fid,0,'eof')` does *not* set the EOF indicator. For example,

```
fseek(fid,0,'eof')
result = feof(fid,'eof');
result =
      0
```

fprintf

```
count = fprintf(fid,'format',A,...)
```

The `format` argument is a string containing C language conversion specifications. Format conversion specifications involve the character `%`, optional flags, optional width and precision fields, optional subtype specifiers, and conversion characters `d`, `i`, `o`, `u`, `x`, `X`, `f`, `e`, `E`, `g`, `G`, `c`, and `s`. See an ANSI C manual for complete details.

Complete ANSI C support for these conversion characters is provided consistent with "expected" MATLAB behavior. If a MATLAB matrix element (type '`double`') maps without loss of significance to the underlying C data type associated with the conversion specifier, then the result is the same as that of ANSI C. Otherwise, `e` format is used. For example, using the `d` conversion specifier to print a matrix containing a combination of types produces the following output:

```
A = [2 3 4;pi 2*pi 5.235;Nan Inf NaN];
fprintf(1,'%d\n',A)
2
3.141593e+00
NaN
3
6.283185e+00
Inf
4
5.234000e+00
NaN
```

You must explicitly convert non-integral MATLAB values to integral MATLAB values (using `floor`, `ceil`, `round`, or `fix`) before they print with the expected ANSI C behavior. Note that `NaN`s and `Inf`s are unaffected by the conversion specifier.

MATLAB supports the following nonstandard subtype specifiers for conversion characters `o`, `u`, `x`, and `X`.

- `t` – The underlying C data type is a *float* rather than an unsigned integer.
- `b` – The underlying C data type is a *double* rather than an unsigned integer.

For example, to print a double value in hex use a format such as `'%bx'`

fscanf

```
[A,count] = fscanf(fid,'format',size)
```

The `size` argument specifies the number of "objects" to scan. There is a one to one correspondence between objects and format specifiers. Now, `%s` corresponds to one object. Since MATLAB stores one character per matrix element the resulting returned matrix may be larger than the size originally specified. MATLAB increases the number of columns as required to accommodate any additional elements.

The `format` argument is a string containing C language conversion specifications. Format conversion specifications involve the character `%`, optional assignment-suppressing asterisk and width field, and conversion characters `d`, `i`, `o`, `u`, `x`, `e`, `f`, `g`, `s`, `c`, and `[. . .]` (scanset). For a complete conversion character specification, see an ANSI C manual.

Complete ANSI C support for these conversion characters is provided consistent with "expected" MATLAB behavior. For example, an important difference occurs when using `e`, `f`, and `g` conversions. `%e/%f/%g` map directly to a double (*not* to a float) as if you specified `%le/%lf/%lg`. (All MATLAB matrix elements are doubles.) Use the nonstandard construct `%he/%hf/%hg` to map directly to a float.

Note that subtype specifiers in Standard C like `h` and `l`, though not mentioned specifically above, are supported in the following way: `fscanf` scans the incoming data and converts it to the specified data type before converting it to a double.

fseek

`fseek` does not let you move the file position indicator passed the last byte written. See `feof` for more information.

fgetl, fgets

These functions return the next line of a text file as a string. `fgetl` returns the string *without* a newline, whereas `fgets` returns the string *with* a newline.

frewind

```
frewind(fid)
```

`frewind` sets the file position indicator of the file identified by `fid` to the beginning of the file.

Reading and Writing Strings – Error Messages

`sprintf` and `sscanf` now support an additional output argument to return error messages that occur during their execution. You must use this optional argument to obtain error messages since no file descriptor is available to pass to `ferror`.

sprintf

The new syntax for the `sprintf` function includes an error output argument:

```
[s,errmsg] = sprintf('format',a,...)
```

`errmsg` is an optional output argument that returns an error message string if an error occurs or an empty matrix if an error does not occur. The format changes discussed under `fprintf` apply identically to `sprintf`.

New fopen Permissions

By default, files are now opened in binary mode. To open a text file, add 't' to the permission string, for example 'rt' and 'wt+'. (On UNIX systems, text and binary files are the same so this has no effect. But on PC, Macintosh, and VMS systems this is critical.)

The new `W` (write) and `A` (append) permission options have the same meaning as their lower-case counterparts except they prevent `fwrite` and `fprintf` from flushing the current output buffer. This is useful for writing to tape devices. For example, a command such as

```
fid = fopen('/dev/rst0','W')
```

typically opens a 1/4" cartridge tape for writing with no flushing on a Sun4.

sscanf

The `sscanf` function returns two new arguments: `errmsg`, and `nextindex`:

```
[a,count,errmsg,nextindex] = sscanf(s,'format',size)
```

`errmsg` is an optional output argument that returns an error message string if an error occurs or an empty matrix if an error does not occur. The format changes discussed under `fscanf` apply identically to `sscanf`.

`nextindex` contains a value equal to one greater than the index of the last character scanned from the string specified in the `s` input argument. The third output argument, `errmsg`, no longer returns the message `'At end-of-string'` when the end-of-string is encountered during the scanning process.

You can test for end-of-string by checking `errmsg` to see if it is empty and checking `nextindex` to see if its value is greater than the length of the scanned string. For example, you can use an `if` statement to evaluate the returned arguments:

```
s = ‘2.7183 3.1416’;
[a,count,errmsg,nextindex] = sscanf(s,'%f');
if nextindex > size(s,2) & errmsg == []
```

Improvements and Bug Fixes

This section describes improvements made to MATLAB for the 4.1 release. It also discusses the most important software defects that are fixed in MATLAB 4.1.

Matrix Decomposition Functions

The following enhancements have been made to MATLAB's matrix decomposition functions:

- `null` and `orth` now use `svd` rather than `qr`. This change makes the rank calculation more reliable and provides consistency with the `rank` function.
- The statement

  ```
  [Q, R] = qr(A, 0)
  ```

  now produces an "economy size" decomposition. If `A` has more rows than columns, the resulting `Q` will be the same size as `A` rather than a full, square matrix.

Polynomial Functions

The polynomial functions `polyfit` and `polyval` have been enhanced to optionally generate error estimates for fitted data using an additional parameter `S`. If a polynomial is fit to a set of data using

```
[p, S] = polyfit(x, y, n)
```

and then evaluated using

```
[y, delta] = polyval(p, x, S)
```

the band `y` ± `delta` will contain at least 50 percent of the original data if the data errors were independent and normally distributed with constant variance.

2-D Signal Processing Functions

The 2-D signal processing functions `filter2` and `conv2` have been enhanced to accept an optional third argument that specifies the size of the resulting matrix.

```
Y = filter2(B, X, 'shape')

C = conv2(A, B, 'shape')
```

For `filter2`, this “shape” parameter can have the following values:

`'same'`	Returns the central part of the convolution that is the same size as `X`. This is the default.
`'valid'`	Returns only those parts of the convolution that are computed without the zero-padded edges, `size(Y) < size(X)`.
`'full'`	Returns the full 2-D convolution, `size(Y) > size(X)`.

For `conv2`, this “shape” parameter can have the following values:

`'full'`	Returns the full 2-D convolution. This is the default.
`'same'`	Returns the central part of the convolution that is the same size as `A`.
`'valid'`	Returns only those parts of the convolution that are computed without the zero-padded edges, `size(C) = [ma-mb+1,na-nb+1]` when `size(A) > size(B)`.

`conv2` is fastest when `size(A) > size(B)`.

Interpolation Functions

The suite of M-files for interpolating data has been modified and extended for MATLAB 4.1. The underlying algorithms have been substantially improved so as to be more memory efficient.

`interp1`	1-D data interpolation (table lookup)
`interp2`	2-D data interpolation (table lookup)
`interp3`	2-D biharmonic data interpolation and gridding
`interp4`	2-D bilinear data interpolation
`interp5`	2-D bicubic data interpolation

Bessel Functions

The suite of M-files for implementing Bessel functions has been modified and extended for MATLAB 4.1. The underlying algorithms have been substantially improved. There are now four primary functions for real `x`.

`besselj(alpha,x)`	Bessel functions of the first kind
`bessely(alpha,x)`	Bessel functions of the second kind
`besseli(alpha,x)`	Modified Bessel functions of the first kind
`besselk(alpha,x)`	Modified Bessel functions of the second kind

An old function, `bessela(alpha,x)`, accepts complex `x`, but may produce inaccurate results for large `alpha` or large `x`.

The `bessel(alpha,x)` M-file, calls `besselj(alpha,x)` if `x` is real, `besseli(alpha,x)` if `x` is imaginary, and `bessela(alpha,x)` if `x` is complex. The auxiliary routine `besseln` is no longer used. `besselh` has been superseded by `bessely`.

The Reciprocal Condition Estimator–rcond

This release of MATLAB fixes a bug in the `rcond` function. Previously, `rcond` returned a larger than expected estimate for some matrices. It did, however, return a result on the order of `eps` for matrices that are singular to working precision.

`rcond` now returns an estimate that matches the value returned by the Fortran LINPACK library.

Surface Object CData

When flat shaded, a surface object now accepts a `CData` matrix whose size is one less (in each dimension) than the size of the `ZData` matrix. This is because the extra row and column are not needed to flat shade the object. Of course, `CData` and `ZData` can be the same size and must be for interpolated shading.

When the surface `FaceColor` property is set to a `ColorSpec` (a single color), `none`, or `texturemap`, `CData` can be any size. Previously, only a `FaceColor` of `texturemap` allowed the `CData` and the `ZData` matrices to be of different sizes.

Save Handles Global Variables Correctly

In MATLAB 4.0, issuing `save` from within an M-file function caused both the function's local variables as well as *all* global variables to be saved in the resulting MAT-file. Thus, global variables that were not visible from inside the M-file function were being saved. In MATLAB 4.1, issuing a `save` command from within an M-file function saves the function's local variables and only those global variables that have been explicitly declared `global` within the function.

Domain Name Server

MATLAB no longer requires a special version to support the Domain Name Server on the Sun4.

License Manager

There is a new interface for starting the license manager at boot time called `lmboot`. See the *MATLAB Installation Guide* for more information on using `lmboot`. You should continue to use `lmstart` to start the license manager during an interactive session.

The default license logfile pathname has changed. The new pathname is `/usr/adm/license.log4`. The old name was `/usr/adm/license.log`.

matlab/etc Directory Is Now Self-Contained

You can run the license manager from a machine that is separate from the one on which you install the MATLAB root directory. The following steps describe how to do this. Note that `$MATLAB` refers to the root directory where you install MATLAB. For example, this might be `/usr/local/matlab`. `$SERVER` refers to the root directory where you are going to place the license manager portion of MATLAB. For example, you might use `/usr/local/lm`.

1. Install MATLAB according to the instructions in the *MATLAB Installation Guide*.
2. Copy the `$MATLAB/etc` directory in its entirety to the `$SERVER` directory on each license server.
3. Edit the `license.dat` file to specify the correct pathnames for the `MLM` daemon script and the `local.options` file. These pathnames must refer to the new license manager location. See the "Understanding the `license.dat` File" section of the *MATLAB Installation Guide* for more information.
4. Use the `ln` command with the `-s` option to create a symbolic link on the license manager server from `/etc/lmboot` to `$SERVER/etc/lmboot` as shown below

   ```
   ln -s $SERVER/etc/lmboot /etc/lmboot
   ```

5. Insert the Bourne Shell code fragment in the appropriate boot script for your platform as described in the "Installing MATLAB from Tape" section of the *MATLAB Installation Guide*. You must perform this step on the license manager server (even though you already performed it on the machine when you installed MATLAB).

This procedure allows you to run the license manager from a machine that does not have access to the MATLAB root directory.

Changes to the print Command

The `print` command supports a new option, `-f`, that allows you to specify the Figure Window or the SIMULINK window that you want to print. Previously, `print` assumed the current figure as the window to print and did not allow you to specify another window except by making it the current figure (i.e., the value returned by `gcf`).

The syntax is

```
print -ffigurehandle
print -fwindowtitle
```

For example, specifying a value of 2 for the `-f` option

```
print -f2
```

prints the Figure Window whose handle is 2, regardless of which figure is the current figure.

If you are running SIMULINK you can print the block diagram displayed in a SIMULINK window using the window's title. For example,

```
print -ff14
```

prints the SIMULINK window entitled `f14`.

Creating EPS Files with print

In release 4.0, the `print` command terminated with an error if you specified an Encapsulated PostScript (EPS) device option, but did not specify a filename. This error occurred because you cannot print a standalone EPS file. (Specifying the `print` command without a filename automatically sends the output to the printer.)

With version 4.1, `print` automatically creates an EPS file if you do not specify a filename when selecting one of the EPS device options (`-deps`, `-depsc`, `-deps2`, or `-depsc2`). You can now enter a command such as the following without creating an error condition:

```
print -f2 -deps
```

In this case, MATLAB creates a file named after the Figure Window used to create the file (in this case,. Figure 2) and issues the following message:

```
Encapsulated PostScript files cannot be sent to the
printer.
File saved to disk under name 'figure2.eps'
```

Notes On MATLAB's Behavior

This section describes aspects of MATLAB's current behavior that are not presented in the documentation. Note that there may be changes to this behavior in future releases.

Variable Names in Data Files

You cannot retrieve variables from data files (i.e., load from MAT-files) if the variable names begin with a number. To avoid problems, you should ensure all workspace variables that you want to save have names beginning with a character.

Logical Operators

The logical operators `&`, `|`, `~`, `xor`, `any`, and `all` treat any nonzero number as one. This includes `i` ($\sqrt{-1}$), `inf`, `NaN`, and complex numbers whose real part is zero, but whose imaginary part is nonzero.

On the other hand, `if` and `while` treat a number as nonzero when the real part of the number is nonzero. This means that `i` (represented as `0 + 1.0000i` in MATLAB) evaluates to zero (i.e., false). `NaN`s return an error when used with `if` and `while`.

Relational operators produce some unexpected results when used with `i`, as the following table illustrates. Results using `NaN`s, however, are as expected.

i	Evaluates To	NaN	Evaluates To
`i == 0`	`false`	`NaN == 0`	`false`
`i < 0`	`false`	`NaN < 0`	`false`
`i <= 0`	`true`	`NaN <= 0`	`false`
`i > 0`	`false`	`NaN > 0`	`false`
`i >= 0`	`true`	`NaN >= 0`	`false`
`i ~= 0`	`true`	`NaN ~= 0`	`true`

Graphics Issues

Color Error Tolerance

When you specify a color directly (i.e., with its RGB values), MATLAB either defines that color using a slot in the system color table or selects a color that is close enough, as defined by the color error tolerance. By default, colors that are closer than an average of 1 part in 257 for each red, green, and blue are mapped to the same color. (Generally, your eyes cannot distinguish values closer than this anyway.)

If you set the figure `MinColormap` property to 256, and create an image using

```
colormap(gray(256))
image(1:256)
```

MATLAB allocates 256 separate colors on the system because no two colors are closer together than their error tolerance.

However, if you set the figure `MinColormap` property to 512 (assuming your system can handle that many colors), and create an image using

```
colormap(gray(512))
image(1:512)
```

MATLAB allocates only a little more than 256 separate colors and most colors are the same as one of their neighbors.

Running Movies

`movie` now displays each frame as it loads the data into memory. It then runs the movie the specified number of times at the specified speed. This eliminates long delays with a blank screen when a memory-intensive movie is loaded. Note that the first time the movie is displayed it runs slowly, but this run is not counted as one of the movie repetitions.

Execution Speed of Callback Functions

MATLAB allows you to define callback functions for uicontrols, uimenus, and various mouse button events. The ways in which you define these functions can affect the speed with which they execute. Basically, there are three possibilities. You can specify the callback function as a string argument that is

- The name of a function M-file
- A string that you pass as an argument
- The name of a script M-file

In each case, MATLAB passes the string (which can be the name of a file) to the `eval` function.

If the string is the name of a function M-file, MATLAB compiles the file once and can repeatedly execute it without recompiling. This provides the fastest callback execution, if it is to be called more than once.

If the string is passed directly as an argument (not the name of a file), MATLAB interprets the commands contained in the string each time it executes the callback. This is generally slower than defining the callback as a function M-file.

If the string is the name of a script M-file, MATLAB loads the file into memory and interprets each line each time the callback is executed. This produces the slowest callback execution.

Using Non-Normal Erase Mode

When a graphics object's `Erasemode` property is set to `background`, `xor`, or `none`, the order in which you specify property/value pairs for that object becomes important. In such cases both the *first* and the *last* property/value pairs, specified in a single call to `set`, must be properties that change the appearance of the object in order for MATLAB to initiate a redraw.

This means you cannot specify `UserData` or `Interruptible` as the *first* or *last* property in a single call to `set`. For example, if you define a surface whose `EraseMode` is `background`:

```
h = surface(peaks,'EraseMode','background')
```

and then attempt to change some of the surface's properties

```
set(h,'ZData',peaks.*(-1),'UserData',1)
```

the surface is erased but not redrawn. To avoid this problem, issue two separate statements:

```
set(h,'ZData', -peaks)
set(h,'UserData',1)
```

If you specify an odd number of properties, a single call to `set` works if you can arrange the properties so that the *first* and *last* property/value pairs are not setting `UserData` or `Interruptible`. For example,

```
set(h,'ZData',-peaks,'UserData',1,'EdgeColor','none')
```

Order of Execution of Button Down Functions

When you press a mouse button in a Figure Window, the following sequence occurs (unless there is a noninterruptible callback function already executing):

1. MATLAB determines on what child of the figure the button press occurs and sets the `CurrentObject` property to that object. If the button press did not occur on a child object, then `CurrentObject` is set to the figure itself.

   MATLAB also updates the figure `CurrentPoint` and `SelectionType` properties, the root `CurrentFigure` property, and the axes `CurrentPoint` property.

   The current object is restacked to the top (i.e., its handle becomes the first listed as the children of its parent). The importance of stacking order is discussed in the "How Objects Are Selected" section.

2. If a figure `WindowButtonDownFcn` defined, MATLAB executes it.
3. If there is a `ButtonDownFcn` defined for the `CurrentObject`, MATLAB executes it. (Notice that the figure's `WindowButtonDownFcn` executes *before* the `CurrentObject`'s `ButtonDownFcn`.)

All of these actions occur as a result of a single button down event.

If, at this point in time, you move the mouse thereby generating a mouse motion event, the follow sequence occurs:

1. If there is a figure `WindowButtonMotionFcn` defined *and* there is no noninterruptible callback function already executing, then MATLAB updates both the figure and axes `CurrentPoint` properties.
2. MATLAB then executes the `WindowButtonMotionFcn`.

When you release the mouse button, a similar sequence occurs:

1. If there is a figure `WindowButtonUpFcn` defined *and* there is not a noninterruptible callback function already executing, then MATLAB updates both the figure and axes `CurrentPoint` properties.
2. MATLAB then executes the `WindowButtonUpFcn`.

The uicontrol object is a special case in that it executes the function defined by either its `CallBack` or `ButtonDownFcn` property, depending on where the pointer is when you press the mouse button. This is described in the "How Objects Are Selected" section. However, when the uicontrol's `CallBack` function is invoked, the figure's `WindowButtonDownFcn` and `WindowButtonUpFcn` callbacks are *not* invoked.

The figure's `Interruptible` property affects `ButtonDownFcn` callback functions. See the "Understanding Window Events and Callback Functions" section for information on how this property affects the way callback functions execute.

How Objects Are Selected

Since graphics objects can overlap each other on the display, MATLAB establishes criteria that determine which one is selected by a given mouse button press (and thereby has its `ButtonDownFcn` executed). Basically, two criteria are considered: the stacking order of the objects and the region of selectability defined around the object.

The stacking order is determined by the order in which you issue the object creation commands. The most recently specified object is placed highest on the stack. Pressing a mouse button while the pointer is on an object moves that object to the top of the stacking order. (Note that stacking order is not the arrangement of the objects in 3-D space. Objects are drawn in the correct 3-D order regardless of their stacking order.)

Stacking order is a factor in selection when objects overlap. When overlap occurs, the object selected by a button press on the region of overlap is the one highest in the stacking order.

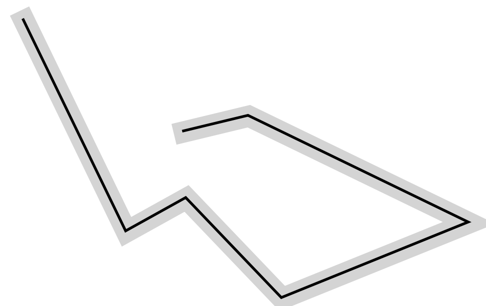
The region of selectability of an object is greater than the area actually occupied by the object. The shape of this region varies with object type.

Axes

The axes region of selectability is determined by the axes box, but extends to include to the area outside the axes where labels and titles appear.

Lines

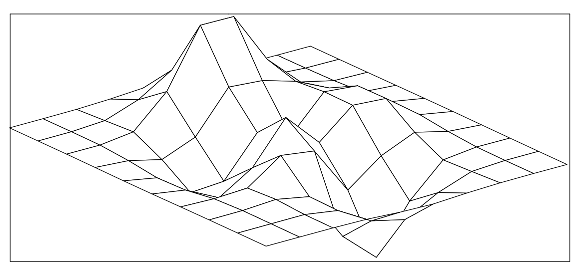
Select line objects by clicking in a region five pixels wide surrounding the line, as illustrated in the following picture:



This simplifies the selection of lines since most are too thin to be easily clicked on with the mouse.

Surfaces, Patches, and Text

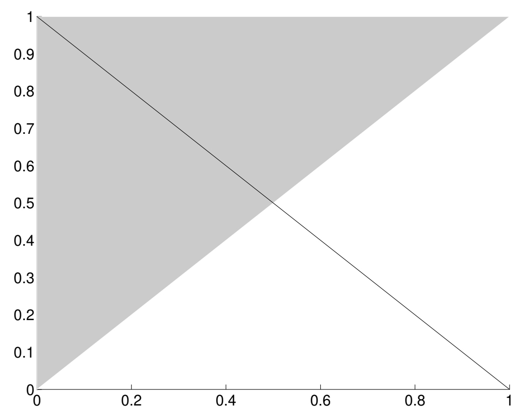
Surface, patch, and text objects can be selected anywhere within a rectangle that encloses the object. If the view is 3-D, the rectangle encloses the object after it is transformed it to the 2-D screen. This is illustrated in the following picture.



Depending on the shape of the object, the enclosing rectangle can produce unexpected results. For example, suppose you create a patch and a line object:

```
patch('ButtonDownFcn','disp(''patch'')')
line([1,0],[0,1],'ButtonDownFcn','disp(''line'')')
```

The default patch object is a triangle, but its region of selectability is a rectangle in which the line object lies when the default 2-D view is used:



Since the line object is the last object specified, it is stacked above the patch object. You can therefore select the line object even though it lies within the patch's region of selectability. However, if you click on the patch, it is moved to the top of the stacking order, after which you can no longer select the line.

Since the patch object's region of selectability lies in the x-y plane, you can change to a 3-D view and once again select the line.

Uicontrols

Uicontrol objects are designed to execute a callback function via their `CallBack` property. They can also execute a callback function defined with their `ButtonDown` property. Since both callbacks are invoked by mouse button down events, MATLAB defines two different regions to discriminate between the two.

If you press a mouse button when the pointer is *on* the uicontrol, its `CallBack` property is evaluated. When you use the mouse on a region *around* the uicontrol, the `ButtonDownFcn` property is evaluated. This region extends on all sides of the uicontrol for a width of approximately five pixels. The following picture indicates the uicontrol's region of selectability with the hash lines:

When you invoke a `ButtonDownFcn`, you also invoke the figure's `WindowButtonDownFcn`, if one is defined. This is not the case with the uicontrol's `CallBack` function.

While the `ButtonDownFcn` property does not contain the functionality necessary to create a robust 3-D picking scheme, it does provide a means for implementing features like object dragging. See the `sigdemo1.m` and `sigdemo2.m` M-files in the `demos` subdirectory for examples.

Understanding Window Events and Callback Functions

MATLAB allows you to define callback functions for uicontrols, uimenus, and mouse actions that occur in a Figure Window. Callback functions execute in response to window events (e.g., pressing a mouse button while the pointer is on a pushbutton, selecting a menu, etc.). Problems can occur, however, when users generate additional window events while a callback function is still executing.

As a mechanism to control the way in which callback functions execute, MATLAB provides the `Interruptible` property. This property determines whether or not an executing callback function can be interrupted to allow another callback to execute.

In order for MATLAB to know that an action capable of invoking a callback function occurred, it must look at the event queue (a list of window events in sequence maintained by the X server). MATLAB does this only at specific times:

- When returning to the MATLAB prompt
- When encountering a `pause` statement
- When executing a `drawnow` command
- When executing a `getframe` command

An interruption of a callback function can occur only when MATLAB checks the event queue and then only if the object's `Interruptible` property is `yes`. The best way to understand the process is to consider some likely scenarios.

Scenario 1: Non-interruptible Callback Function

Suppose a non-interruptible callback function is executing. Contained in this function is a `drawnow` command. While this callback function executes, the user presses the mouse button in a Figure Window for which `WindowButtonDownFcn` is defined. The following sequence occurs:

1. When `drawnow` executes, MATLAB checks the event queue and processes all pending events, thus discovering that a button press occurred in a Figure Window with a `WindowButtonDownFcn` defined.
2. MATLAB checks the executing callback's `Interruptible` property to see if it can stop executing the callback function and begin executing the action defined by the `WindowButtonDownFcn`.
3. MATLAB determines that it cannot interrupt the executing callback and flushes all events from the queue. Thus the fact that a mouse button was pressed in a Figure Window having a defined `WindowButtonDownFcn` is lost.
4. The original callback continues to execute.

Note that if the original callback did not contain a `drawnow` command (or a `pause` or `getframe`), the window button down event would not be processed until MATLAB returned to the prompt. In this case, the event is not lost – it is just delayed.

Scenario 2: Interruptible Callback Function

Suppose an interruptible callback function is executing. Contained in this function is a `drawnow` command. While this callback function executes, the user presses the mouse button in a Figure Window for which a `WindowButtonDownFcn` is defined. The following sequence occurs:

1. When `drawnow` executes, MATLAB checks the event queue and processes all pending events, thus discovering that a button press occurred in a Figure Window with a `WindowButtonDownFcn` defined.
2. MATLAB checks the executing callback's `Interruptible` property to see if it can stop executing the callback function and begin executing the action defined by the `WindowButtonDownFcn`.
3. Since the callback function is interruptible, MATLAB suspends its execution and executes the `WindowButtonDownFcn` function.
4. When the `WindowButtonDownFcn` ends, execution of the original callback function resumes where it left off. However, MATLAB's state (such as the current figure or current axes) at the time of interruption is not restored.

Note that if the original callback did not execute a `drawnow`, the interrupting event would not be processed until MATLAB returned to the prompt (i.e., no interruption would occur). Note also that interrupting callbacks can themselves be interrupted.

You can use the `drawnow` command's `discard` option to prevent MATLAB from changing the display while executing interrupting callbacks. See the "Drawnow Discard" section for more information.

Using Error Trapping with Call Back Functions

Error trapping is useful when implementing Handle Graphics callback functions, as it provides a mechanism for ensuring that callback functions maintain control in the event of an error. Since the `CallBack` property accepts only a single string argument, you must explicitly specify `eval` with two arguments as the callback string in order to use error trapping. For instance, without error trapping you could say

```
set(h, 'CallBack', 'disp(''Button Was Pushed'')');
```

whereas to use error trapping you would say

```
set(h, 'CallBack', ...
      'eval(''disp(''''Button Was Pushed'''')'', ...
            ''disp(''''Error During CallBack'''')'')';
```

Note that each time quotes are nested deeper, two more must be used.

Using an Unsupported Graphics Terminal

If your graphics terminal or terminal emulator is not one of those supported in the `terminal.m` file, you must create an M-file that specifies the terminal characteristics by following these steps:

1. Determine the type of Tektronix terminal your terminal or terminal program emulates.
2. Look in your terminal or terminal program user's manual to determine what values to use for the terminal characteristics described below. The Tektronix 4105 definition in `terminal.m` is a good example of how to set these characteristics.
3. Create an M-file that sets these root object properties and displays an appropriate initialization string for your terminal. Again, see the Tektronix 4105 definition in `terminal.m` for a good example. This step is described in more detail in the next section.
4. Either execute your M-file directly or use `terminal.m` to execute it. To run `terminal.m`, enter the following command:

   ```
   terminal m-filename
   ```

   where `m-filename` is the name of the M-file that describes the terminal. `terminal.m` automatically checks to see if the terminal you specify exists as an M-file and, if so, executes it.

Defining Terminal Characteristics

To properly identify your terminal, you must define its characteristics by specifying the root object properties described in the next subsections.

Defining the Terminal Type

You define the terminal type with the `TerminalProtocol` property. This property indicates whether your terminal emulates a Tektronix 4010/4014, a 4100, or a 4105.

Specify `tek401x` for terminals that emulate Tektronix 4010/4014 terminals. Specify `tek410x` for terminals that emulate Tektronix 4100/4105 terminals. If you are using X Windows and MATLAB can connect to your X display server, this property will automatically be set to `x`.

Defining the Number of Colors that Can Be Displayed

You indicate the number of colors your terminal can display by specifying the `ScreenDepth` property.

If you are using a black and white terminal, set this property to 1. If you are using an eight-color terminal, set this property to 3. If you are using a 256-color terminal, set this property to 8.

If you are using X Windows and MATLAB can connect to an X display server, if you omit this property or set its value to zero, MATLAB will automatically determine whether it is running on color or monochrome hardware and set the value of this property to the depth of your display.

If you are not using X Windows and do not specify this property, MATLAB uses a default value of 3 when the `TerminalProtocol` property is set to `tek410x`, and a value of 1 otherwise.

Specifying this property is useful if you are using X Windows on color hardware but displaying results on a monochrome monitor. If the property is not specified, MATLAB might determine erroneously that the monitor is a color device.

Changing from Graphics Mode to Text Mode

To tell MATLAB how to change from graphics mode (the graph window) to text mode (the MATLAB command window), specify the escape sequences with the `TerminalHideGraphCommand` property. Consult your terminal manual for the correct escape sequences.

Changing from Text Mode to Graphics Mode

To tell MATLAB how to change from text mode (the MATLAB command window) to graphics mode (the graph window), specify the escape sequences with the `TerminalShowGraphCommand` property. Consult your terminal manual for the correct escape sequences.

Using Separate Graphics and Text Windows

To indicate whether the terminal can maintain separate graphics and text screens in different windows, specify the `TerminalOneWindow` property. For example, for an xterm emulating a Tektronix 4010 or 4014, set this property to `no` because the graph window and command window are two separate windows that can be displayed at the same time.

The default value for this property is `yes`.

Very Large Variables on IBM Systems

MATLAB running on IBM RS6000 systems can encounter problems allocating very large variables. Attempting to create a variable that exceeds a certain size limit (which depends on system memory), can kill your MATLAB process. This problem is due to the very non-standard manner in which IBM has implemented memory management in the operating system. In essence, the OS will indicate that it has successfully allocated space for a variable, but does not, in fact, guarantee that the memory is really available when you actually try to access it, thus resulting in an unpredictable, yet fatal, error.

Platform-Specific File I/O Behavior

Reading Data Using fscanf and sscanf

The MATLAB functions `fscanf` and `sscanf` do not behave in a consistent manner on all platforms when reading `NaN`s and `inf`s. This is because MATLAB relies on the respective vendors' C library routines to implement these functions.The following table summarizes the behavior on each platform.

Platform	`fscanf` Read `NaN`	`fscanf` Read `Inf`	`sscanf` Read `NaN`	`sscanf` Read `Inf`
MS Windows	No	No	No	No
SunOS 4.1.X	Yes	Yes	Yes	Yes
Solaris 2.X	Yes	Yes	Yes	Yes
HP 700	No	No	No	No
SGI	No	No	No	No
IBM RS/6000	Yes	Yes	Yes	Yes
DECstation	No	No	No	No

In addition, each platform varies in the way it handles error conditions. Consider the following example:

```
fid = fopen('error.file','w+')
n = fprintf(fid,'%s\n%s\n%s\n','6.25e-038','.45375e+003',...
   '3e8');
```

These statements create a file containing the following data:

```
6.25e-038
.45375+e003
3e8
```

Next call `fopen` again to ensure the buffer is flushed before reading the file with `fscanf`:

```
fid = fopen('error.file')
[a,b] = fscanf(fid,'%6e')
ferror(fid)
fclose(fid)
```

The following table lists the values returned on each platform for the `fscanf` statement:

Platform	`a`	`b`

MS Windows	`6.25`
	`38.00`
	`0.4538`
		`3`

SunOS 4.1.X	`6.25`	`1`

Solaris 2.X	`6.25`
	`-38.00`
	`0.4537`
		`3`

HP700	`6.25`
	`38.00`
	`0.45375`
		`3`

SGI	`6.25`
	`38.00`
	`0.45375`
		`3`

IBM RS/6000	`[ ]`	`0`

DECstation	`6.25`
	`38.00`
	`0.4537`
		`3`

An error condition exists because `fscanf` is instructed by the `%6e` formatting argument to read up to a maximum of six characters to obtain a floating point number. Reading six characters gives

```
6.25e-
```

which is not a valid floating point number.

It is interesting to note that the first four platforms in the table return incorrect data and in three cases return three values. This illustrates the different ways in which each platform deals with error conditions.

While it is important to realize how functions behave on your platform, the real lesson to be learned in this example is the importance of calling `ferror` whenever you read data from a file. In this case, all platforms terminated the read with the following error message:

```
Sorry. No help in figuring out the problem....
```

MATLAB is not able to provide specific information about the nature of the error. However, receiving an error should make you question the validity of any of the returned data.

MATLAB's `sscanf` function does not directly call the C library's `sscanf`. Therefore, it has been implemented to always return an empty string when it encounters an error. For example, the following statement

```
[a,b,c,d] = sscanf('6.25e-038','%6e')
```

again produces an error condition. However, `sscanf` returns the following values on all platforms:

```
a = [ ]
b = 0
c = 'Matching failure in format'
d = 1
```

Reaching EOF with fread and fscanf

Whenever the `fread` or `fscanf` function reads to the end of a file (EOF), the `ferror` function always returns the string `'At end-of-file'`. This is because these functions rely on the Standard C `fread` and `fscanf`, which are implemented to work this way.

This is potentially problematic only in the case where an error occurs on the last element read in the file. Any error that occurs at that point is masked by the fact that `ferror` *always* returns the end of file message when it reaches EOF.

For example, suppose `file` contains `3.65e-` as its last element. Clearly this constitutes bad data which leads to an error condition when you read the file:

```
fid = fopen('file')
[a,b] = fscanf(fid,'%g')
if feof(fid)
     ferror(fid)
end
```

However, `ferror` returns `'At end-of-file'` without indicating the occurrence of an error. In cases where the integrity of the data file is unclear, it is advisable to attempt to determine the validity of the data.

Inconsistencies in fprintf and sprintf Output

In extreme cases, such as the following example which deals with a number on the limit of what double precision can represent, the output of `fprintf` and `sprintf` can vary across different platforms. Consider the following MATLAB code fragment.

```
ber = 3141592653589793.238462643383279;
fprintf (1,'%.0f',ber)
format hex
ber
```

The output on different platforms is

MS Windows:	`3141592653589794`	`43265286144ada43`
SunOS 4.1.X:	`3141592653589794`	`43265286144ada43`
Solaris 2.X:	`3141592653589794`	`43265286144ada43`
HP 700:	`3141592653589794`	`43265286144ada43`
IBM RS/6000:	`3141592653589794`	`43265286144ada43`
SGI:	`3141592653589793`	`43265286144ada43`
DECstation:	`3141592653589793`	`43265286144ada43`

The internal representations in hex are the same. Notice, however, that the output from `fprintf` is not consistent across all platforms.

MATLAB `fprintf` and `sprintf` functions use the C library `fprintf` and `sprintf` functions. Since Standard C does not specify how the conversion between number systems should behave, the actual output depends on whether the function performs rounding or truncation.