# MATLAB®

The Language of Technical Computing

**Computation**

**Visualization**

**Programming**

# Graphics Reference Manual

*Version 5*

**How to Contact The MathWorks:**

| | |
|---|---|
| (508) 647-7000 | Phone |
| (508) 647-7001 | Fax |
| (508) 647-7022 | Technical Support Faxback Server |
| The MathWorks, Inc.<br>24 Prime Park Way<br>Natick, MA 01760-1500 | Mail |
| `http://www.mathworks.com` | Web |
| `ftp.mathworks.com` | Anonymous FTP server |
| `support@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 |

Printing History:

| | | |
|---|---|---|
| January 1996 | First printing | New for Alpha-2 |
| July 1996 | Second printing | Revised for Alpha-7 |
| November 1996 | Third printing | Revised FCS |

# 1 Preface

# 2 Command Summary

# Preface

The Preface gives you information about MATLAB , its documentation, and this guide.

## What Is MATLAB?

MATLAB® is a technical computing environment for high-performance numeric computation and visualization. MATLAB integrates numerical analysis, matrix computation, signal processing, and graphics in an easy-to-use environment where problems and solutions are expressed just as they are written mathematically – without traditional programming.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state of the art in software for matrix computation.

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many numerical problems in a fraction of the time it would take to write a program in a language such as Fortran, Basic, or C.

MATLAB has evolved over a period of years with input from many users. In university environments, it has become the standard instructional tool for introductory courses in applied linear algebra, as well as advanced courses in other areas. In industrial settings, MATLAB is used for research and to solve practical engineering and mathematical problems. Typical uses include general purpose numeric computation, algorithm prototyping, and special purpose problem solving with matrix formulations that arise in disciplines such as automatic control theory, statistics, and digital signal processing (time-series analysis).

MATLAB also features a family of application-specific solutions that we call *toolboxes*. Very important to most users of MATLAB, toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment in order to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems design, dynamic systems simulation, systems identification, neural networks, and others.

Probably the most important feature of MATLAB, and one that we took care to perfect, is its easy extensibility. This allows you to become a contributing author too, creating your own applications. In the years that MATLAB has been available, we have enjoyed watching many scientists, mathematicians, and engineers develop new and interesting applications, all without writing a single line of Fortran or other low-level code.

## Who Wrote MATLAB?

The original MATLAB was written in Fortran by Cleve Moler, in an evolutionary process over several years. The underlying matrix algorithms are from the many people who worked on the LINPACK and EISPACK projects.

The current MATLAB program was written in C by The MathWorks. The first release was written by Steve Bangert, who wrote the parser/interpreter, Steve Kleiman, who implemented the graphics, and John Little and Cleve Moler, who wrote the analytical routines, the user's guide, and most of the M-files. Since the first release, many other people have joined the MATLAB development team and have made substantial contributions.

## MATLAB Documentation

MATLAB comes with an extensive set of both online and printed documentation. The online MATLAB Function Reference is a compendium of all MATLAB commands functions. You can access this documentation from the MATLAB Help Desk. Users on all platforms can access the Help Desk with the MATLAB `doc` command. MS-Windows and Macintosh users can also access the Help Desk with the **Help** menu or the **?** icon on the Command Window toolbar. From the Help Desk main menu, choose “MATLAB Functions” to display the Function Reference.

The online resources are augmented with printed documentation consisting of the following titles:

- *Getting Started with MATLAB* describes MATLAB fundamentals.
- *Using MATLAB* explains how to use MATLAB as both a programming language and a command-line application.
- *Using MATLAB Graphics* describes how to use MATLAB’s graphics and visualization tools.
- The *MATLAB Application Programmer’s Interface Guide* explains how to write C or Fortran programs that interact with MATLAB.
- The *MATLAB 5 New Features Guide* provides information useful in making the transition from MATLAB 4.x to MATLAB 5.
- The *MATLAB 5 Release Notes* provide additional information about new features that are not covered in the other guides. They also include lists of problems fixed since the previous release and known documentation errors.

### How to Use the Documentation Set

If you need to install MATLAB, you should read the appropriate booklet. Once you install MATLAB, you can decide which document you prefer to use to learn the MATLAB commands.

If you are a new MATLAB user, you should start by reading *Getting Started with MATLAB. Using MATLAB* provides an extensive description of the MATLAB language.

*Using MATLAB Graphics* describes how to use MATLAB for visualizing data with both high-level functions and Handle Graphics.

## How to Use the Reference Pages

The Reference pages are organized in alphabetical order, with operators described first. Each entry contains one or more of these sections:

| | |
|---|---|
| Purpose | Provides concise descriptions. |
| Syntax | Summarizes the formats of the command or function. |
| Description | Gives overall information about the command or function and describes how each syntax behaves. |
| Remarks | Provides tangential information about the command or function. |
| Examples | Shows concrete illustrations of how the command or function can be used. |
| Limitations | Describes any unusual restrictions on how the command or function can be used. |
| Diagnostics | Tells you about error or warning messages that may appear. |
| Algorithm | Describes how the command or function is implemented or gives background information on associated procedures and routines. |
| See Also | Refers you to the reference entries of related commands. |
| References | Provides pointers to additional resources. |

### Typographical and Alphabetic Conventions

This manual uses certain typographical conventions.

| Font | Usage |
|---|---|
| `Monospace` | Commands, function names, and screen displays; for example, `conv`. |
| *`Monospace Italics`* | Names of arguments that are meant to be replaced and not typed literally; for instance: `cd` *`directory`*. |
| *Italics* | Book titles, mathematical notation, and the introduction of new terms. |
| `Color` | Command and function syntaxes. |
| **Boldface Initial Cap** | Names of keys, such as the **Return** key. |

In addition, this manual uses some alphabetic conventions.

| **Data Type** | **Format** | **Examples** |
|---|---|---|
| Matrices and multidimensional arrays | Upper-case letters | `A, B, C` |
| Vectors | Lower-case letters | `u, v, w` |
| Scalars | Lower-case letters | `a, b, c` |
| Index variables | Lower-case letters | `i, j, k` |
| Sparse matrices | Upper-case letters | `S, S1, S2` |
| Parameters | Lower case if vectors, otherwise upper case | `p1, p2` |
| Strings | Lower-case letters | `str, str1` |

1

# Command Summary

## Command Summary

| | |
|---|---|
| `convhull` | Convex hull |
| `delaunay` | Delaunay triangulation |
| `dsearch` | Search Delaunay triangulation for nearest point |
| `inpolygon` | True for points inside a polygonal region |
| `polyarea` | Area of polygon |
| `tsearch` | Search for enclosing Delaunay triangle |
| `voronoi` | Voronoi diagram |

### Color Operations and Lighting

| | |
|---|---|
| `brighten` | Brighten or darken color map |
| `caxis` | Pseudocolor axis scaling |
| `colorbar` | Display color bar (color scale) |
| `colorcube` | Enhanced color-cube color map |
| `colordef` | Set up color defaults |
| `colormap` | Set the color look-up table |
| `diffuse` | Diffuse reflectance |
| `graymon` | Graphics figure defaults set for gray-scale monitor |
| `hsv2rgb` | Hue-saturation-value to red-green-blue conversion |
| `lighting` | Lighting mode |
| `material` | Material reflectance mode |
| `rgb2hsv` | RGB to HSVconversion |
| `rgbplot` | Plot color map |
| `shading` | Color shading mode |
| `specular` | Specular reflectance |
| `spinmap` | Spin the colormap |
| `surfnorm` | 3-D surface normals |
| `whitebg` | Change axes background color for plots |

### Colormaps

| | |
|---|---|
| `autumn` | Shades of red and yellow color map |
| `bone` | Gray-scale with a tinge of blue color map |
| `contrast` | Gray color map to enhance image contrast |
| `cool` | Shades of cyan and magenta color map |
| `copper` | Linear copper-tone color map |
| `flag` | Alternating red, white, blue, and black color map |
| `gray` | Linear gray-scale color map |
| `hot` | Black-red-yellow-white color map |
| `hsv` | Hue-saturation-value (HSV) color map |
| `jet` | Variant of HSV |
| `lines` | Line color colormap |
| `prism` | Colormap of prism colors |
| `spring` | Shades of magenta and yellow color map |

| | |
|---|---|
| summer | Shades of green and yellow colormap |
| winter | Shades of blue and green color map |

## Basic Plots and Graphs

| | |
|---|---|
| bar | Vertical bar chart |
| barh | Horizontal bar chart |
| hist | Plot histograms |
| hold | Hold current graph |
| loglog | Plot using log-log scales |
| pie | Pie plot |
| plot | Plot vectors or matrices. |
| polar | Polar coordinate plot |
| semilogx | Semi-log scale plot |
| semilogy | Semi-log scale plot |
| subplot | Create axes in tiled positions |

## Hardcopy/File Output

| | |
|---|---|
| hardcopy | Save figure window to file |
| orient | Hardcopy paper orientation |
| print | Print graph or save graph to file |
| printopt | Configure local printer defaults |
| savtoner | Modify graphic objects to print on a white background |

## Surface, Mesh, and Contour Plots

| | |
|---|---|
| contour | Contour (level curves) plot. |
| contourc | Contour computation |
| contourf | Filled contour plot |
| hidden | Mesh hidden line removal mode |
| meshc | Combination mesh/contourplot |
| mesh | 3-D mesh with reference plane |
| surf | 3-D shaded surface graph |
| surface | Create surface low-level objects |
| surfc | Combination surf/contourplot |
| surfl | 3-D shaded surface with lighting |
| trimesh | Triangular mesh plot |
| trisurf | Triangular surface plot |

## Domain Generation for Function Visualization

| | |
|---|---|
| griddata | Data gridding and surface fitting |
| meshgrid | Generation of X and Y arrays for 3-D plots |

## Specialized Plotting

| | |
|---|---|
| area | Area plot |

| | |
|---|---|
| `box` | Axis box for 2-D and 3-D plots |
| `comet` | Comet plot |
| `compass` | Compass plot |
| `errorbar` | Plot graph with error bars |
| `ezplot` | Easy to use function plotter |
| `feather` | Feather plot |
| `fill` | Draw filled 2-D polygons |
| `fplot` | Plot a function |
| `pareto` | Pareto chart |
| `pie3` | 3-D Pie plot |
| `plotmatrix` | Scatter plot matrix |
| `pcolor` | Pseudocolor (checkerboard) plot |
| `rose` | Plot rose or angle histogram |
| `quiver` | Quiver (or velocity) plot |
| `ribbon` | Ribbon plot |
| `stairs` | Stairstep graph |
| `stem` | Plot discrete sequence data |

## Three-Dimensional Plotting

| | |
|---|---|
| `bar3` | Vertical 3-D bar chart |
| `bar3h` | Horizontal 3-D bar chart |
| `comet3` | 3-D Comet plot |
| `cylinder` | Generate cylinder |
| `fill3` | Draw filled 3-D polygons in 3-space |
| `plot3` | Plot lines and points in 3-D space |
| `quiver3` | 3-D Quiver (or velocity) plot |
| `slice` | Volumetric slice plot |
| `sphere` | Generate sphere |
| `stem3` | Plot discrete surface data |
| `view` | 3-D graph viewpoint specification. |
| `viewmtx` | Generate view transformation matrices |
| `waterfall` | Waterfall plot |

## Plot Annotation and Grids

| | |
|---|---|
| `clabel` | Add contour labels to a contour plot |
| `datetick` | Date formatted tick labels |
| `grid` | Grid lines for 2-D and 3-D plots |
| `gtext` | Place text on a 2-D graph using a mouse |
| `legend` | Graph legend for lines and patches |
| `plotyy` | Plot graphs with Y tick labels on the left and right |
| `title` | Titles for 2-D and 3-D plots |
| `xlabel` | X-axis labels for 2-D and 3-D plots |
| `ylabel` | Y-axis labels for 2-D and 3-D plots |
| `zlabel` | Z-axis labels for 3-D plots |

## Handle Graphics, General

| | |
|---|---|
| bwcontr | Contrasting black and/or color |
| copyobj | Make a copy of a graphics object and its children |
| findobj | Find objects with specified property values |
| gcbo | Return object whose callback is currently executing |
| gco | Return handle of current object |
| get | Get object properties |
| rotate | Rotate objects about specified origin and direction |
| ishandle | True for graphics objects |
| set | Set object properties |
| treediag | Tree diagram of objects |

## Handle Graphics, Object Creation

| | |
|---|---|
| axes | Create axis at arbitrary positions |
| figure | Create Figures (graph windows) |
| image | Display image (create image object) |
| light | Create light object |
| line | Create line low-level objects |
| patch | Create patch low-level objects |
| text | Add text to the current plot |

## Handle Graphics, Figure Windows

| | |
|---|---|
| capture | Screen capture of the current figure |
| clc | Clear figure window |
| clf | Clear Figure |
| clg | Clear Figure (graph window) |
| close | Close specified window |
| gcf | Get current figure handle |
| newplot | Graphics M-file preamble for NextPlot property |
| refresh | Refresh figure |

## Handle Graphics, Axes

| | |
|---|---|
| axis | Plot axis scaling and appearance |
| cla | Clear axis |
| gca | Get current axis handle |

## Object Manipulation

| | |
|---|---|
| propedit | Edit all properties of any selected object |
| reset | Reset axis or figure |
| rotate3d | Interactively rotate the view of a 3-D plot |
| selectmoveresize | Interactively select, move, or resize objects |
| shg | Show graph window |

## Graphical User Interface Creation

| | |
|---|---|
| `dialog` | Create a dialog box |
| `errordlg` | Create error dialog box |
| `helpdlg` | Display help dialog box |
| `inputdlg` | Create input dialog |
| `menu` | Generate a menu of choices for user input |
| `menuedit` | Menu Editor |
| `msgbox` | Create message dialog box |
| `questdlg` | Create question dialog box |
| `textwrap` | Return wrapped string matrix for given UI Control |
| `uicontrol` | Create user interface control |
| `uigetfile` | Display dialog box to retrieve name of file for reading |
| `uimenu` | Create user interface menu |
| `uiputfile` | Display dialog box to retrieve name of file for writing |
| `uiresume` | Used with `uiwait`, controls program execution |
| `uisetcolor` | Interactively set a ColorSpec via a dialog box |
| `uisetfont` | Interactively set a font by displaying a dialog box |
| `uiwait` | Used with `uiresume`, controls program execution |
| `waitbar` | Display wait bar |
| `waitforbuttonpress` | Wait for key/buttonpress over figure |
| `warndlg` | Create warning dialog box |

## Interactive User Input

| | |
|---|---|
| `ginput` | Graphical input from a mouse or cursor |
| `zoom` | Zoom in and out on a 2-D plot |

## Interface Design

| | |
|---|---|
| `algntool` | Align uicontrols and axes |
| `cbedit` | Callback Editor |
| `guide` | functions |
| `toolpal` | Initialization for Tool Palette |

## Region of Interest

| | |
|---|---|
| `dragrect` | Drag XOR rectangles with mouse |
| `drawnow` | Complete any pending drawing |
| `rbbox` | Rubberband box |

## area

**Purpose**

Area fill of a two-dimensional plot

**Syntax**

```
area(Y)
area(X,Y)
area(...,ymin)
area(...,'PropertyName',PropertyValue,...)
h = area(...)
```

**Description**

An area plot displays elements in `Y` as one or more curves and fills the area beneath each curve. When `Y` is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each *x* interval.

`area(Y)` plots the vector `Y` or the sum of each column in matrix `Y`. The *x*-axis automatically scales depending on `length(Y)` when `Y` is a vector, and on `size(Y,1)` when `Y` is a matrix.

`area(X,Y)` plots `Y` at the corresponding values of `X`. If `X` is a vector, `length(X)` must equal `length(Y)` and `X` must be monotonic. If `X` is a matrix, `size(X)` must equal `size(Y)` and each column in `X` must be monotonic. To make a vector or matrix monotonic, use `sort`.

`area(...,ymin)` specifies the lower limit in the *y* direction for the area fill. The default `ymin` is `0`.

`area(...,'`*`PropertyName`*`',PropertyValue,...)` specifies property name and property value pairs for the Patch graphics object created by `area`.

`h = area(...)` returns handles of Patch graphics objects. `area` creates one Patch object per column in `Y`.

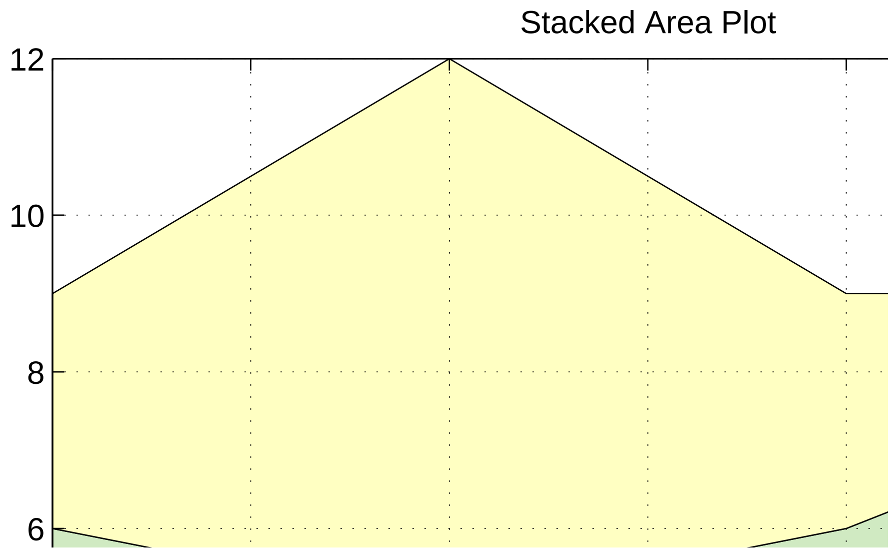
**Remarks**

`area` creates one curve from all elements in a vector or one curve per column in a matrix. The colors of the curves are selected from equally spaced intervals throughout the entire range of the colormap.

## Examples

Plot the values in Y as a stacked area plot:

```
Y = [  1, 5, 3;
       3, 2, 7;
       1, 5, 3;
       2, 6, 1];
area(Y)
set(gca,'Layer','top')
title 'Stacked Area Plot'
```



## See Also

plot

**Purpose**

Create Axes graphics object

**Syntax**

```
axes
axes('PropertyName',PropertyValue,...)
axes(h)
h = axes(...)
```

**Description**

`axes` is the low-level function for creating Axes graphics objects.

`axes` creates an Axes graphics object in the current Figure using default property values.

`axes('`*`PropertyName`*`',PropertyValue,...)` creates an Axes object having the specified property values. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`h = axes(...)` returns the handle of the created Axes object.

`axes(h)` makes existing axes `h` the current Axes. It also makes `h` the first Axes listed in the Figure's `Children` property and set the Figure's `CurrentAxes` property to `h`. The current Axes is the target for functions that draw Image, Line, Patch, Surface, and Text graphics objects.

**Remarks**

MATLAB automatically creates an Axes, if one does not already exist, when you issue a command that draws Image, Light, Line, Patch, Surface, or Text graphics objects.

The `axes` function accepts property name/property value pairs, structure arrays, and cell arrays as input arguments (see the `set` and `get` reference pages for examples of how to specify these data types). These properties, which control various aspects of the Axes object, are described in the "Axes Properties" section.

Use the `set` function to modify the properties of an existing Axes or the `get` function to query the current values of Axes properties. Use the `gca` command to obtain the handle of the current Axes.

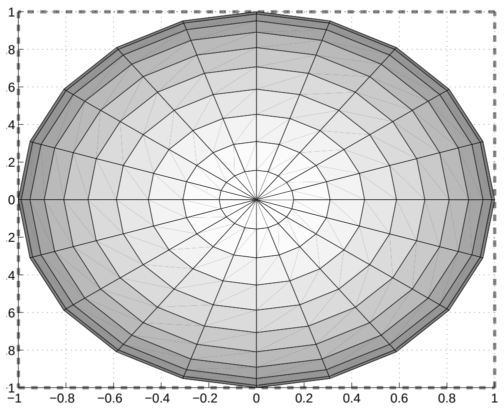
The `axis` (not `axes`) function provides simplified access to commonly used properties that control the scaling and appearance of Axes.

While the basic purpose of an Axes object is to provide a coordinate system for plotted data, Axes properties provide considerable control over the way MATLAB displays data.

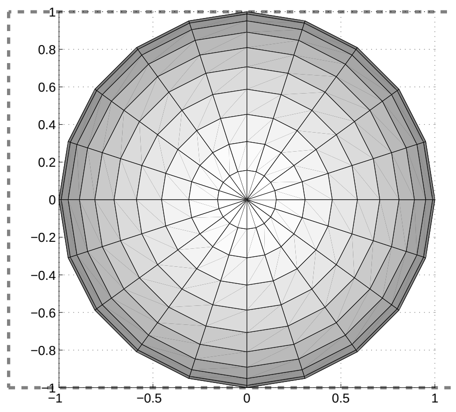
### Stretch-to-fill

By default, MATLAB stretches the Axes to fill the Axes position rectangle (the rectangle defined by the last two elements in the `Position` property). This results in graphs that use the available space in the rectangle. However, some 3-D graphs (such as a sphere) appear distorted because of this stretching, and are better viewed with some specific three dimensional aspect ratio. Stretch-to-fill is active when the `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto` (the default). However, stretch-to-fill is turned off when `DataAspectRatio`, `PlotBoxAspectRatio`, or `CameraViewAngle` are user-specified, or when one or more of the corresponding modes is set to `manual` (which happens automatically when you set the corresponding property value).

This picture shows the same sphere displayed both with and without the Stretch-to-fill . The dotted lines show the Axes `Position` rectangle.



Stretch-to-fill active



Stretch-to-fill disabled

When Stretch-to-fill is disabled, MATLAB sets the size of the Axes to be as large as possible within the constraints imposed by the `Position` rectangle without introducing distortion. In the picture above, the height of the rectangle constrains the Axes size.

## Examples

### Zooming

Zoom in using aspect ratio and limits:

```
sphere
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],'ZLim',[-0.6 0.6])
```

Zoom in and out using the `CameraViewAngle`:

```
sphere
set(gca,'CamerViewAngle',get(gca,'CameraViewAngle')-5)
set(gca,'CamerViewAngle',get(gca,'CameraViewAngle')+5)
```

Note that both examples disable MATLAB's stretch-to-fill behavior.

### Positioning the Axes

The Axes `Position` property enable you to define the location of the Axes within the Figure window. For example,

```
h = axes('Position',position_rectangle)
```

creates an Axes object at the specified position within the current Figure and returns a handle to it. Specify the location and size of the Axes with a rectangle defined by a four-element vector,

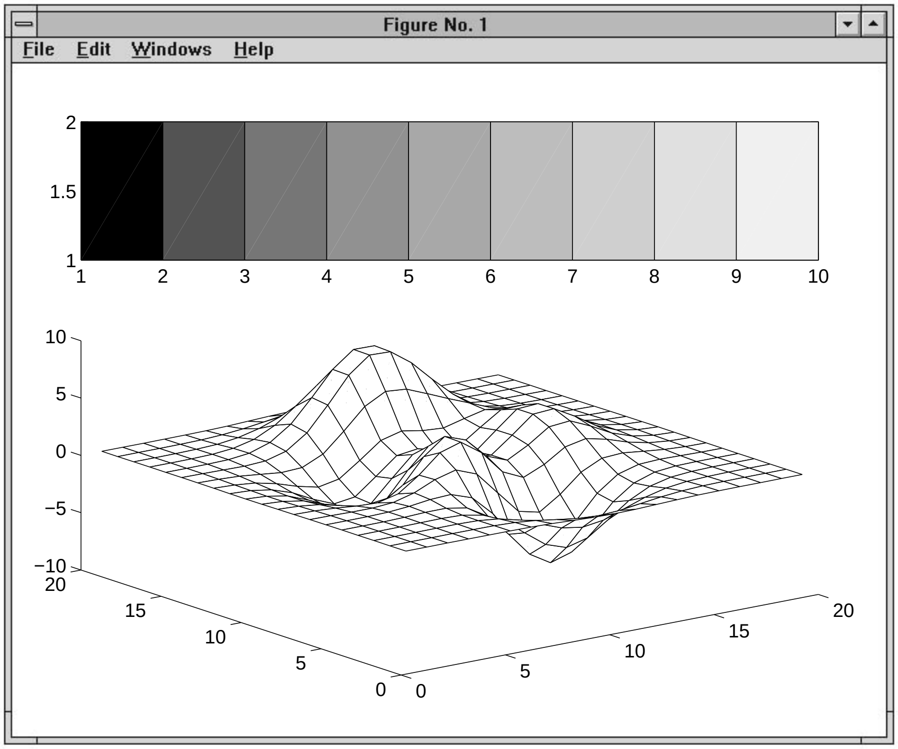
```
position_rectangle = [left, bottom, width, height];
```

The `left` and `bottom` elements of this vector define the distance from the lower-left corner of the Figure to the lower-left corner of the rectangle. The `width` and `height` elements define the dimensions of the rectangle. You specify these values in units determined by the `Units` property. By default, MATLAB uses normalized units where (0,0) is the lower-left corner and (1.0,1.0) is the upper-right corner of the Figure window.

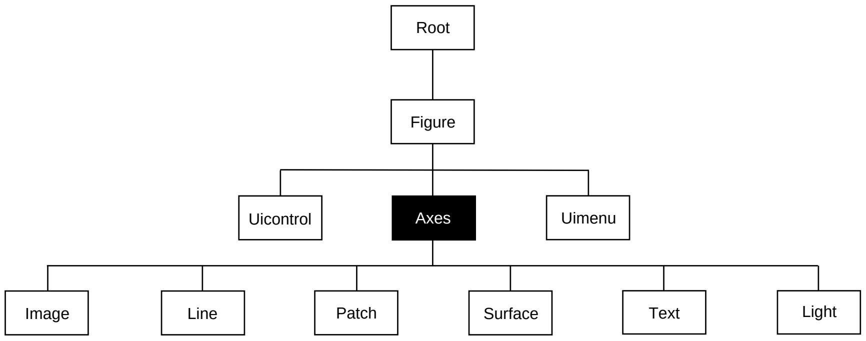
You can define multiple Axes in a single Figure window:

```
axes('position',[.1  .1  .8  .6])
mesh(peaks(20));
axes('position',[.1  .7  .8  .2])
pcolor([1:10;1:10]);
```

In this example, the first plot occupies the bottom two-thirds of the Figure, and the second occupies the top third.



## Object Hierarchy



### Setting Property Defaults

You can set default Axes properties on the Figure and Root levels:

```
set(0,'DefaultAxesPropertyName',PropertyValue,...)
set(gcf,'DefaultAxesPropertyName',PropertyValue,...)
```

Where *PropertyName* is the name of the Axes property and `PropertyValue` is the value you are specifying.

## Axes Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`AmbientLightColor`** `ColorSpec`

*The background light in a scene*. Ambient light is a directionless light that shines uniformly on all objects in the Axes. However, if there are no visible Light objects in the Axes, MATLAB does not use `AmbientLightColor`. If there are Light objects in the Axes, the `AmbientLightColor` is added to the other light sources.

**`AspectRatio`** (Obsolete)

This property produces a warning message when queried or changed. It has been superseded by the `DataAspectRatio[Mode]` and `PlotBoxAspectRatio[Mode]` properties.

**`Box`** `on | {off}`

*Axes box mode*. This property specifies whether to enclose the Axes extent in a box for 2-D views or a cube for 3-D views. The default is to not display the box.

**BusyAction** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn** string

*Button press callback routine*. A callback routine that executes whenever you press a mouse button while the pointer is within the Axes, but not over another graphics object displayed in the Axes. For 3-D views, the active area is defined by a rectangle that encloses the Axes.

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**CameraPosition** `[x, y, z] Axes coordinates`

*The location of the camera*. This property defines the position from which the camera views the scene. Specify the point in Axes coordinates.

If you fix `CameraViewAngle`, you can zoom in and out on the scene by changing the `CameraPosition`, moving the camera closer to the `CameraTarget` to zoom in and farther away from the `CameraTarget` to zoom out. As you change the `CameraPosition`, the amount of perspective also changes, if `Projection` is `perspective`. You can also zoom by changing the `CameraViewAngle`, however, this does not change the amount of perspective in the scene.

**CameraPositionMode** `{auto} | manual`

*Auto or manual CameraPosition*. When set to `auto`, MATLAB automatically calculates the `CameraPosition` such that the camera lies a fixed distance from the `CameraTarget` along the Azimuth and Elevation specified in the `View`. Setting a value for `CameraPosition` sets this property to `manual`.

**`CameraTarget`** `[x, y, z]` Axes coordinates

*Camera aiming point*. This property specifies the location in the Axes that the camera points to. The `CameraTarget` and the `CameraPosition` define the vector along which the camera looks.

**`CameraTargetMode`** `{auto} | manual`

*Auto or manual CameraTarget placement*. When this property is `auto`, MATLAB automatically positions the `CameraTarget` at the centroid of the Axes plotbox. Specifying a value for `CameraTarget` sets this property to `manual`.

**`CameraUpVector`** `[x, y, z]` Axes coordinates

*Camera rotation*. This property specifies the rotation of the camera around the viewing axis defined by the `CameraTarget` and the `CameraPosition` properties. Specify `CameraUpVector` as a three-element array containing the $x$, $y$, and $z$ components of the vector. For example, `[0 1 0]` specifies the positive $y$-axis as the up direction.

The default `CameraUpVector` is `[0 0 1]`, which defines the positive $z$-axis as the up direction.

**`CameraUpVectorMode`** `{auto} | manual`

*Default or user-specified up vector*. When `CameraUpVectorMode` is `auto`, MATLAB uses a value of `[0 0 1]` (positive $z$-direction is up) for 3-D views and `[0 1 0]` (positive $y$-direction is up) for 2-D views. Setting a value for `CamerаUpVector` sets this property to `manual`.

**`CameraViewAngle`** scalar between 0 and 180 (angle in degrees)

*The field of view*. This property determines the camera field of view. Changing this value affects the size of graphics objects displayed in the Axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view, and the smaller objects appear in the scene.

**`CameraViewAngleMode`** `{auto} | manual`

*Auto or manual CameraViewAngle*. When in `auto` mode, MATLAB sets `CameraViewAngle` to the minimum angle that captures the entire scene (up to 180˚).

The following table summarizes MATLAB's automatic camera behavior.

| CameraView Angle | Camera Target | Camera Position | Behavior |
|---|---|---|---|
| `auto` | `auto` | `auto` | `CameraTarget` is set to plot box centroid, `CameraViewAngle` is set to capture entire scene, `CameraPosition` is set along the view axis. |
| `auto` | `auto` | `manual` | `CameraTarget` is set to plot box centroid, `CameraViewAngle` is set to capture entire scene. |
| `auto` | `manual` | `auto` | `CameraViewAngle` is set to capture entire scene, `CameraPosition` is set along the view axis. |
| `auto` | `manual` | `manual` | `CameraViewAngle` is set to capture entire scene. |
| `manual` | `auto` | `auto` | `CameraTarget` is set to plot box centroid, `CameraPosition` is set along the view axis. |
| `manual` | `auto` | `manual` | `CameraTarget` is set to plot box centroid |
| `manual` | `manual` | `auto` | `CameraPosition` is set along the view axis. |
| `manual` | `manual` | `manual` | All Camera properties are user-specified. |

**`Children`** vector of graphics object handles

*Children of the Axes*. A vector containing the handles of all graphics objects rendered within the Axes (whether visible or not). The graphics objects that can be children of Axes are Images, Lights, Lines, Patches, Surfaces, and Text.

The Text objects used to label the *x*-, *y*-, and *z*-axes are also children of Axes, but their `HandleVisibility` properties are set to `callback`. This means their handles do not show up in the Axes `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

**`CLim`** `[cmin, cmax]`

*Color axis limits*. A two-element vector that determines how MATLAB maps the `CData` values of Surface and Patch objects to the Figure's colormap. `cmin` is the value of the data mapped to the first color in the colormap, and `cmax` is the value of the data mapped to the last color in the colormap. Data values in between are linearly interpolated across the colormap, while data values outside are clamped to either the first or last colormap color, whichever is closest.

When `CLimMode` is `auto` (the default), MATLAB assigns `cmin` the minimum data value and `cmax` the maximum data value in the graphics object's `CData`. This maps `CData` elements with the minimum data value to the first colormap entry and with the maximum data value to the last colormap entry.

If the Axes contains multiple graphics objects, MATLAB sets `CLim` to span the range of all objects' `CData`.

**`CLimMode`** `{auto} | manual`

*Color axis limits mode*. In `auto` mode, MATLAB sets the `CLim` property to span the `CData` limits of the graphics objects displayed in the Axes. If `CLimMode` is `manual`, MATLAB does not change the value of `CLim` when the `CData` limits of axes children change. Setting the `CLim` property sets this property to `manual`.

**`Clipping`** `{on} | off`

This property has no effect on Axes.

**`Color`** `{none} | ColorSpec`

*Color of the Axes back planes*. Setting this property to `none` means the Axes is transparent and the Figure color shows through. A `ColorSpec` is a three-element RGB vector or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information on specifying color. Note that while the default value is `none`, the `matlabrc.m` file may set the Axes `color` to a specific color.

**`ColorOrder`** m-by-3 matrix of RGB values

*Colors to use for multiline plots*. An *m*-by-3 matrix of RGB values that define the colors used by the `plot` and `plot3` functions to color each line plotted. If you do not specify a line color with `plot` and `plot3`, these functions cycle through the `ColorOrder` to obtain the color for each line plotted. To obtain the current , `ColorOrder`, which may be set during startup, get the property:

```
get(gca,'ColorOrder')
```

Note that if the Axes `NextPlot` property is set to `replace` (the default), high-level functions like `plot` reset the `ColorOrder` property before determining the colors to use. If you want MATLAB to use a `ColorOrder` that is different than the default, set `NextPlot` to `replacedata`. You can also specify your own default `ColorOrder`.

**`CreateFcn`** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates an Axes object. You must define this property as a default value for Axes. For example, the statement,

```
set(0,'DefaultAxesCreateFcn','set(gca,''Color'',''b'')')
```

defines a default value on the Root level that sets the current Axes' background color to blue whenever you (or MATLAB) create an Axes. MATLAB executes this routine after setting all properties for the Axes. Setting this property on an existing Axes object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`CurrentPoint`** 2-by-3 matrix

*Location of last button click, in Axes data units*. A 2-by-3 matrix containing the coordinates of two points defined by the location of the pointer. These two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. The 3-D coordinates are the points, in the axes coordinate system, where this line intersects the front and back surfaces of the Axes volume (which is defined by the Axes x, y, and z limits).

The returned matrix is of the form:

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

MATLAB updates the `CurrentPoint` property whenever a button-click event occurs. The pointer does not have to be within the Axes, or even the Figure window; MATLAB returns the coordinates with respect to the requested Axes regardless of the pointer location.

**`DataAspectRatio`** `[dx dy dz]`

*Relative scaling of data units*. A three-element vector controlling the relative scaling of data units in the *x*, *y*, and *z* directions. For example, setting this property t o `[1 2 1]` causes the length of one unit of data in the *x* direction to be the same length as two units of data in the *y* direction and one unit of data in the *z* direction.

Note that the `DataAspectRatio` property interacts with the `PlotBoxAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control how MATLAB scales the *x*-, *y*-, and *z*-axis. Setting the `DataAspectRatio` will disable the Stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` were previously all `auto`. The following table describes the interaction between properties when the Stretch-to-fill behavior is disabled.

| **X-, Y-, Z-Limits** | **DataAspect Ratio** | **PlotBox AspectRatio** | **Behavior** |
|---|---|---|---|
| `auto` | `auto` | `auto` | Limits chosen to span data range in all dimensions. |
| `auto` | `auto` | `manual` | Limits chosen to span data range in all dimensions. `DataAspectRatio` is modified to achieve the requested `PlotBoxAspectRatio` within the limits selected by MATLAB. |
| `auto` | `manual` | `auto` | Limits chosen to span data range in all dimensions. `PlotBoxAspectRatio` is modified to achieve the requested `DataAspectRatio` within the limits selected by MATLAB. |
| `auto` | `manual` | `manual` | Limits chosen to completely fit and center the plot within the requested `PlotBoxAspectRatio` given the requested `DataAspectRatio` (this may produce empty space around 2 of the 3 dimensions). |
| `manual` | `auto` | `auto` | Limits are honored. The `DataAspectRatio` and `PlotBoxAspectRatio` are modified as necessary. |
| `manual` | `auto` | `manual` | Limits and `PlotBoxAspectRatio` are honored. The `DataAspectRatio` is modified as necessary. |
| `manual` | `manual` | `auto` | Limits and `DataAspectRatio` are honored. The `PlotBoxAspectRatio` is modified as necessary. |

| X-, Y-, Z-Limits | DataAspect Ratio | PlotBox AspectRatio | Behavior |
|---|---|---|---|
| 1 manual<br>2 auto | manual | manual | The 2 automatic limits are selected to honor the specified aspect ratios and limit. See “Examples” |
| 2 or 3 manual | manual | manual | Limits and DataAspectRatio are honored; the PlotBoxAspectRatio is ignored. |

**DataAspectRatioMode** {auto} | manual

*User or MATLAB controlled data scaling*. This property controls whether the values of the DataAspectRatio property are user defined or selected automatically by MATLAB. Setting values for the DataAspectRatio property automatically sets this property to manual. Changing DataAspectRatioMode to manual will disable the Stretch-to-fill behavior, if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode were previously all auto

**DeleteFcn** string

*Delete Axes callback routine*. A callback routine that executes when the Axes object is deleted (e.g., when you issue a delete or a close command). MATLAB executes the routine before destroying the object’s properties so the callback routine can query these values.

The handle of the object whose DeleteFcn is being executed is accessible only through the Root CallbackObject property, which can be queried using gcbo.

**DrawMode** {normal} | fast

*Rendering method*. This property controls the method MATLAB uses to render graphics objects displayed in the Axes, when the Figure Renderer is painters.

- normal mode draws objects in back to front ordering based on the current view, in order to handle hidden surface elimination and object intersections.
- fast mode draws objects in the order in which you specify the drawing commands, without considering the relationships of the objects in three dimensions. This results in faster rendering because it requires no sorting of objects according to location in the view, but may produce undesirable results because it bypasses the hidden surface elimination and object interstection handling provided by normal DrawMode.

When the Figure `Renderer` is `zbuffer`, `DrawMode` is ignored, and hidden surface elimination and object intersection handling are always provided.

**`FontAngle`** `{normal} | italic | oblique`

*Select italic or normal font*. This property selects the character slant for Axes text. `normal` specifies a nonitalic font. `italic` and `oblique` specify italic font.

**`FontName`** The default is Helvetica on many systems

*Font family name*. The font family name specifying the font to use for Axes labels. To display and print properly, `FontName` must be a font that your system supports. Note that the *x*-, *y*-, and *z*-axis labels do not display in a new font until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

**`FontSize`** Font size specified in `FontUnits`

*Font size*. An integer specifying the font size to use for Axes labels and titles, in units determined by the `FontUnits` property. The default point size is 12. The *x*-, *y*-, and *z*-axis text labels do not display in a new font size until you manually reset them (by setting the `XLabel`, `YLabel`, or `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

**`FontUnits`** `{points} | normalized | inches | centimeters | pixels`

*Units used to interpret the* `FontSize` *property*. When set to `normalized`, MATLAB interprets the value of `FontSize` as a fraction of the height of the Axes. For example, a `normalized` `FontSize` of 0.1 sets the text characters to a font whose height is one tenth of the Axes' height. The default units (`points`), are equal to 1/72 of an inch.

**`FontWeight`** `{normal} | bold | light | demi`

*Select bold or normal font*. The character weight for Axes text. The *x*-, *y*-, and *z*-axis text labels do not display in bold until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` commands). Tick mark labels change immediately.

**`GridLineStyle`** `– | ––| {:} | –. | none`

*Line style used to draw grid lines*. The line style is a string consisting of a character, in quotes, specifying solid lines (–), dashed lines (– –), dotted lines(`:`), or dash-dot lines (–.). The default grid line style is dotted. To turn on grid lines, use the `grid` command.

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to on to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`Interruptible`** `{on} | off`

*Callback routine interruption mode.* The `Interruptible` property controls whether an Axes callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `EventQueue` property for related information.

Setting `Interruptible` to `on` allows any graphics object's callback routine to interrupt callback routines originating from an Axes property. Note that MATLAB does not save the state of variables or the display (e.g., the handle returned by the `gca` or `gcf` command) when an interruption occurs.

**`Layer`** `{bottom} | top`

*Draw axis lines below or above graphics objects*. This property determines if axis lines and tick marks draw on top or below Axes children objects when the view is `[0 90]` and the Axes `DrawMode` is `fast` (or when there are no Axes `Children` with nonzero `ZData`). This enables you to place grid lines and tick marks on top of Images.

**`LineStyleOrder`** `LineSpec`

*Order of line styles and markers used in a plot*. This property specifies which line styles and markers to use and in what order when creating multiple-line plots. For example,

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

sets `LineStyleOrder` to solid line with asterisk marker, dotted line, and hollow circle marker. The default is (–), which specifies a solid line for all data plotted. Alternatively, you can create a cell array of character strings to define the line styles:

```
set(gca,'LineStyleOrder',{'-*',':','o'})
```

MATLAB supports four line styles, which you can specify any number of times in any order. MATLAB cycles through the line styles only after using all colors defined by the `ColorOrder` property. For example, the first eight lines plotted use the different colors defined by `ColorOrder` with the first line style. MATLAB then cycles through the colors again, using the second line style specified, and so on.

You can also specify line style and color directly with the `plot` and `plot3` functions or by altering the properties of the Line objects.

Note that, if the Axes `NextPlot` property is set to `replace` (the default), high-level functions like `plot` reset the `LineStyleOrder` property before determining the line style to use. If you want MATLAB to use a `LineStyleOrder` that is different than the default, set `NextPlot` to `replacedata`. You can also specify your own default `LineStyleOrder`.

**`LineWidth`** linewidth in points

*Width of axis lines*. This property specifies the width, in points, of the *x*-, *y*-, and *z*-axis lines. The default line width is 0.5 points (1 point = $1/72$ inch).

**`NextPlot`** `add | {replace} | replacechildren`

*Where to draw the next plot*. This property determines how high-level plotting functions draw into an existing Axes.

- `add` — use the existing Axes to draw graphics objects.
- `replace` — reset all Axes properties, except `Position`, to their defaults and delete all Axes children before displaying graphics (equivalent to `cla reset`).
- `replacechildren` — remove all child objects, but do not reset Axes properties (equivalent to `cla`).

The `newplot` function simplifies the use of the `NextPlot` property and is used by M-file functions that draw graphs using only low-level object creation routines. See the M-file `pcolor.m` for an example. Note that Figure graphics objects also have a `NextPlot` property.

**`Parent`** Figure handle

*Axes parent*. The handle of the Axes' parent object. The parent of an Axes object is the Figure in which it is displayed. The utility function `gcf` returns the handle of the current Axes' `Parent`. You can reparent Axes to other Figure objects.

**`PlotBoxAspectRatio`** `[px py pz]`

*Relative scaling of Axes plotbox*. A three-element vector controlling the relative scaling of the plot box in the *x*-, *y*-, and *z*-directions. The plot box is a box enclosing the Axes data region as defined by the *x*-, *y*-, and *z*-axis limits.

Note that the `PlotBoxAspectRatio` property interacts with the `DataAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control the way graphics objects are displayed in the Axes. Setting the `PlotBoxAspectRatio` will disable the Stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` were previously all `auto`. .

**`PlotBoxAspectRatioMode`** `{auto} | manual`

*User or MATLAB controlled axis scaling*. This property controls whether the values of the `PlotBoxAspectRatio` property are user defined or selected automatically by MATLAB. Setting values for the `PlotBoxAspectRatio` property automatically sets this property to `manual`. Changing the `PlotBoxAspectRatioMode` to `manual` will disable the Stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` were previously all `auto`.

**`Position`** `4-element vector`

*Position of Axes*. A four-element vector specifying a rectangle that locates the Axes within the Figure window. The vector is of the form:

```
[left bottom width height]
```

where `left` and `bottom` define the distance from the lower-left corner of the Figure window to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

When Axes Stretch-to-fill behavior is enabled (when `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, `CameraViewAngleMode` are all `auto`), the axes are stretched to fill the `Position` rectangle. When Stretch-to-fill is disabled, the Axes are made as big as possible while obeying all other properties, without extending outside the `Position` rectangle

**`Projection`** `{orthographic} | perspective`

*Type of projection*. This property selects between two projection types:

- `orthographic` – This projection maintains the correct relative dimensions of the graphics objects with regard to the distance a given point is from the viewer. Parallel lines in the data are drawn parallel on the screen.
- `perspective` – This projection incorporates foreshortening, which allows you to perceive depth in a 2-D representation of 3-D objects. Objects appear to become smaller as they are moved further from the viewer, and parallel lines in the data may not appear parallel on screen.

**Selected** on | off

*Is object selected*. When this property is on. MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects highlight when selected*. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

**Tag** string

*User-specified object label*. The Tag property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you want to direct all graphics output from an M-file to a particular Axes, regardless of user actions that may have changed the current Axes. To do this, identify the Axes with a Tag:

```
axes('Tag','Special Axes')
```

Then make that Axes the current Axes before drawing by searching for the Tag with findobj:

```
axes(findobj('Tag','Special Axes'))
```

**TickDir** in | out

*Direction of tick marks*. For 2-D views, the default is to direct tick marks inward from the axis lines; 3-D views direct tick marks outward from the axis line.

**TickDirMode** {auto} | manual

*Automatic tick direction control*. In auto mode, MATLAB directs tick marks inward for 2-D views and outward for 3-D views. When you specify a setting for TickDir, MATLAB sets TickDirMode to manual. In manual mode, MATLAB does not change the specified tick direction.

**`TickLength`** `[2DLength 3DLength]`

*Length of tick marks*. A two-element vector specifying the length of Axes tick marks. The first element is the length of tick marks used for 2-D views and the second element is the length of tick marks used for 3-D views. Specify tick mark lengths in units normalized relative to the longest of the visible X-, Y-, or Z-axis annotation lines.

**`Title`** handle of text object

*Axes title*. The handle of the Text object that is used for the Axes title. You can use this handle to change the properties of the title Text or you can set `Title` to the handle of an existing Text object. For example, the following statement changes the color of the current title to red:

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

To create a new title, set this property to the handle of the Text object you want to use:

```
set(gca,'Title',text('String','New Title','Color','r'))
```

However, it is generally simpler to use the `title` command to create or replace an Axes title:

```
title('New Title','Color','r')
```

**`Type`** string (read only)

*Type of graphics object*. This property contains a string that identifies the class of graphics object. For Axes objects, `Type` is always set to 'axes'.

**`Units`** `inches | centimeters | {normalized} | points | pixels`

*Position units*. The units used to interpret the `Position` property. All units are measured from the lower-left corner of the Figure window. `normalized` units map the lower-left corner of the Figure window to (0,0) and the upper-right corner to (1.0, 1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals $1/72$ of an inch).

**`UserData`** matrix

*User specified data*. This property can be any data you want to associate with the Axes object. The Axes does not use this property, but you can access it using the `set` and `get` functions.

**`View`** Obsolete

The functionality provided by the View property is now controlled by the Axes camera properties – `CameraPosition`, `CameraTarget`, `CameraUpVector`, and `CameraViewAngle`. See the `view` command.

**`Visible`** `{on} | off`

*Visibility of Axes*. By default, Axes are visible. Setting this property to `off` prevents axis lines, tick marks, and labels from being displayed. The visible property does not affect children of Axes.

**`XAxisLocation`** `top | {bottom}`

*Location of x-axis tick marks and labels*. This property controls where MATLAB displays the *x*-axis tick marks and labels. Setting this property to `top` moves the *x*-axis to the top of the plot.

**`YAxisLocation`** `right | {left}`

*Location of y-axis tick marks and labels*. This property controls where MATLAB displays the *y*-axis tick marks and labels. Setting this property to `right` moves the *y*-axis to the right side of the plot.

## Properties That Control the X-, Y-, or Z-Axis

**`XColor, YColor, ZColor`** *ColorSpec*.

*Color of axis lines*. A three-element vector specifying an RGB triple, or a predefined MATLAB color string. This property determines the color of the axis lines, tick marks, tick mark labels, and the axis grid lines of the respective *x*-, *y*-, and *z*-axis. The default axis color is white. See the `ColorSpec` reference page for details on specifying colors.

**XDir, YDir, ZDir** `{normal} | reverse`

*Direction of increasing values*. A mode controlling the direction of increasing axis values. Axes form a right-hand coordinate system. By default,

- *x*-axis values increase from left to right. To reverse the direction of increasing *x* values, set this property to `reverse`.
- *y*-axis values increase from bottom to top (2-D view) or front to back (3-D view). To reverse the direction of increasing *y* values, set this property to `reverse`.
- *z*-axis values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). To reverse the direction of increasing *z* values, set this property to `reverse`.

**XGrid, YGrid, ZGrid** `on | {off}`

*Axis gridline mode*. When you set any of these properties to `on`, MATLAB draws grid lines perpendicular to the respective axis (i.e., along lines of constant *x*, *y*, or *z* values). Use the `grid` command to set all three properties `on` or `off` at once.

**XLabel, YLabel, ZLabel** handle of text object

*Axis labels*. The handle of the Text object used to label the *x*, *y*, or *z*-axis, respectively. To assign values to any of these properties, you must obtain the handle to the text string you want to use as a label. This statement defines a Text object and assigns its hanlde to the `XLabel` property:

```
set(gca,'Xlabel',text('String','axis label'))
```

MATLAB places the string `'axis label'` appropriately for an *x*-axis label. Any Text object whose handle you specify as an `XLabel`, `YLabel`, or `ZLabel` property is moved to the appropriate location for the respective label.

Alternatively, you can use the `xlabel`, `ylabel`, and `zlabel` functions, which generally provide a simpler means to label axis lines.

**XLim, YLim, ZLim** `[minimum maximum]`

*Axis limits*. A two-element vector specifying the minimum and maximum values of the respective axis.

Changing these properties affects the scale of the *x*-, *y*-, or z-dimension as well as the placement of labels and tick marks on the axis. The default values for these properties are [0 1].

**XLimMode, YLimMode, ZLimMode** {auto} | manual

*MATLAB or user-controlled limits*. The axis limits mode determines whether MATLAB calculates axis limits based on the data plotted (i.e., the XData, YData, or ZData of the Axes children) or uses the values explicitly set with the XLim, YLim, or ZLim property, in which case, the respective limits mode is set to manual.

**XScale, YScale, ZScale** {linear} | log

*Axis scaling*. Linear or logarithmic scaling for the respective axis.

**XTick, YTick, ZTick** vector of data values locating tick marks

*Tick spacing*. A vector of *x*-, *y*-, or *z*-data values that determine the location of tick marks along the respective axis. If you do not want tick marks displayed, set the respective property to the empty vector, [ ]. These vectors must contain monotonically increasing values.

**XTickLabel, YTickLabel, ZTickLabel** string

*Tick labels*. A matrix of strings to use as labels for tick marks along the respective axis. These labels replace the numeric labels generated by MATLAB. If you do not specify enough text labels for all the tick marks, MATLAB uses all of the labels specified, then reuses the specified labels.

For example, the statement,

```
set(gca,'XTickLabel',{'One';'Two';'Three';’Four'})
```

labels the first four tick marks on the *x*-axis and then reuses the labels until all ticks are labeled.

Labels can be specified as cell arrays of strings, padded string matrices, string vectors separated by vertical slash characters, or as numeric vectors (where each number is implicitly converted to the equivalent string using num2str). All of the following are equivalent:

```
set(gca,’XTickLabel’,{‘1’;’10’;’100’})
set(gca,’XTickLabel’,‘1|10|100’)
set(gca,’XTickLabel’,[1;10;100])
set(gca,’XTickLabel’,[‘1  ’;’10 ’;’100’])
```

**XTickMode, YTickMode, ZTickMode** {auto} | manual

*MATLAB or user controlled tick spacing*. The axis tick modes determine whether MATLAB calculates the tick mark spacing based on the range of data for the respective axis (auto mode) or uses the values explicitly set for any of the XTick, YTick, and ZTick properties (manual mode). Setting values for the XTick, YTick, or ZTick properties sets the respective axis tick mode to manual.

**XTickLabelMode, YTickLabelMode, ZTickLabelMode**{auto} | manual

*MATLAB or user determined tick labels*. The axis tick mark labeling mode determines whether MATLAB uses numeric tick mark labels that span the range of the plotted data (auto mode) or uses the tick mark labels specified with the XTickLabel, YTickLabel, or ZTickLabel property (manual mode). Setting values for the XTickLabel, YTickLabel, or ZTickLabel property sets the respective axis tick label mode to manual.

## See Also

axis, cla, clf, figure, gca, subplot

# axis

**Purpose** Axis scaling and appearance

**Syntax**

```
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax])
axis auto
axis manual
axis(axis)
v = axis

axis ij
axis xy

axis square
axis equal
axis normal
axis image
axis vis3d

axis off
axis on

[mode,visibility,direction] = axis('state')
```

**Description**

`axis` manipulates commonly used Axes properties. (See Algorithm section.)

`axis([xmin xmax ymin ymax])` sets the limits for the *x*- and *y*-axis of the current Axes.

`axis([xmin xmax ymin ymax zmin zmax])` sets the limits for the *x*-, *y*-, and *z*-axis of the current Axes.

`axis auto` sets MATLAB to its default behavior of computing the current Axes' limits automatically, based on the minimum and maximum values of *x*, *y*, and *z* data. You can restrict this automatic behavior to a specific axis. For example, `axis 'auto x'` computes only the *x*-axis limits automatically; `axis 'auto yz'` computes the *y*- and *z*-axis limits automatically.

`axis manual` and `axis(axis)` freeze the scaling at the current limits, so that if `hold` is on, subsequent plots use the same limits. This sets the `XLimMode`, `YLimMode`, and `ZLimMode` properties to `manual`.

`v = axis` returns a row vector containing scaling factors for the *x*-, *y*-, and *z*-axis. `v` has four or six components depending on whether the current Axes is 2-D or 3-D, respectively. The returned values are the current Axes' `XLim`, `Ylim`, and `ZLim` properties.

`axis ij` places the coordinate system origin in the upper-left corner. The *i*-axis is vertical, with values increasing from top to bottom. The *j*-axis is horizontal with values increasing from left to right.

`axis xy` draws the graph in the default Cartesian axes format with the coordinate system origin in the lower-left corner. The *x*-axis is horizontal with values increasing from left to right. The *y*-axis is vertical with values increasing from bottom to top.

`axis square` makes the current Axes region square (or cubed when three-dimensional). MATLAB adjusts the *x*-axis, *y*-axis, and *z*-axis so that they have equal lengths and adjusts the increments between data units accordingly.

`axis equal` sets the aspect ratio so that the data units are the same in every direction. The aspect ratio of the *x*-, *y*-, and *z*-axis is adjusted automatically according to the range of data units in the *x*, *y*, and *z* directions.

`axis vis3` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill.

`axis normal` automatically adjusts the aspect ratio of the Axes
and the aspect ratio of the data units represented on the Axes to fill the plot box.

`axis tightequal` sets the aspect ratio so that the data units are the same in every direction. This differs from `axis equal` because the plot box aspect ratio automatically adjusts. (Formally `axis image`.)

`axis off` turns off all axis lines, tick marks, and labels.

`axis on` turns on all axis lines, tick marks, and labels.

`[mode,visibility,direction] = axis('state')` returns three strings indicating the current setting of Axes properties:

| **Output Argument** | **Strings Returned** |
|---|---|
| `mode` | `'auto' \| 'manual'` |
| `visibility` | `'on' \| 'off'` |
| `direction` | `'xy' \| 'ij'` |

`mode` is `'auto'` if `XLimMode`, `YLimMode`, and `ZLimMode` are all set to `auto`. If `XLimMode`, `YLimMode`, or `ZLimMode` is `manual`, `mode` is `'manual'`.

## Examples

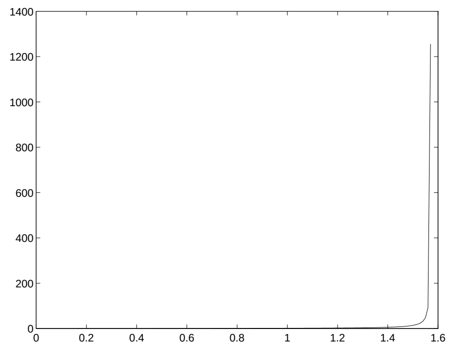
The statements

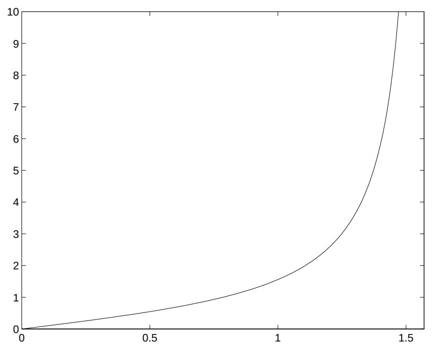
```
x = 0:.01:pi/2;
plot(x,tan(x))
```

use the automatic scaling of the *y*-axis based on `ymax = tan(1.57)`, which is well over 1000, as shown in the left figure.

The right figure shows a more satisfactory plot after typing

```
axis([0  pi/2  0  10])
```





## Algorithm

When you specify minimum and maximum values for the *x*-, *y*-, and *z*-axes, `axis` sets the `XLim`, `Ylim`, and `ZLim` properties for the current Axes to the respective minimum and maximum values in the argument list. Additionally, the

XLimMode, YLimMode, and ZLimMode properties for the current Axes are set to manual.

axis auto sets the current Axes' XLimMode, YLimMode, and ZLimMode properties to 'auto'.

axis manual sets the current Axes' XLimMode, YLimMode, and ZLimMode properties to 'manual'.

The following table shows the values of the Axes properties set by axis equal, axis normal, axis square, and axis image.

| Axes Property | axis equal | axis normal | axis square | axis tightequal |
|---|---|---|---|---|
| DataAspectRatio | [1 1 1] | not set | not set | [1 1 1] |
| DataAspectRatioMode | manual | auto | auto | manual |
| PlotBoxAspectRatio | [3 4 4] | not set | [1 1 1] | auto |
| PlotBoxAspectRatioMode | manual | auto | manual | auto |
| Stretch-to-fill | disabled | active | disabled | disabled |

## See Also

axes, get, set, subplot

Properties of Axes graphics objects.

# bar, barh

**Purpose** Bar chart

**Syntax**

```
bar(Y)
bar(x,Y)
bar(...,width)
bar(...,'style')
bar(...,ColorSpec)
[xb,yb] = bar(...)
h = bar(...)

barh(...)
[xb,yb] = barh(...)
h = barh(...)
```

**Description** A bar chart displays the values in a vector or matrix as horizontal or vertical bars.

`bar(Y)` draws one bar for each element in `Y`. If `Y` is a matrix, `bar` groups together the bars produced by the elements in each row. The *x*-axis scale ranges from 1 to `length(Y)` when `Y` is a vector, and `1` to `size(Y,1)`, which is the number of rows, when `Y` is a matrix.

`bar(x,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `x` is a monotonically increasing vector defining the *x*-axis intervals for the vertical bars. If `Y` is a matrix, `bar` clusters the elements in the same row in `Y` at locations corresponding to an element in `x`.

`bar(...,width)` sets the relative bar width and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, the bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar(...,'`*`style`*`')` specifies the style of the bars. `'`*`style`*`'` is `'group'` or `'stack'`. `'group'` is the default mode of display.

- `'group'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stack'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multi-colored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar(...,`*`LineSpec`*`)` displays all bars using the color specified by *LineSpec*.

`[xb,yb] = bar(...)` returns vectors that you plot using `plot(xb,yb)` or `patch(xb,yb,C)`. This gives you greater control over the appearance of a graph, for example, to incorporate a bar chart into a more elaborate `plot` statement.

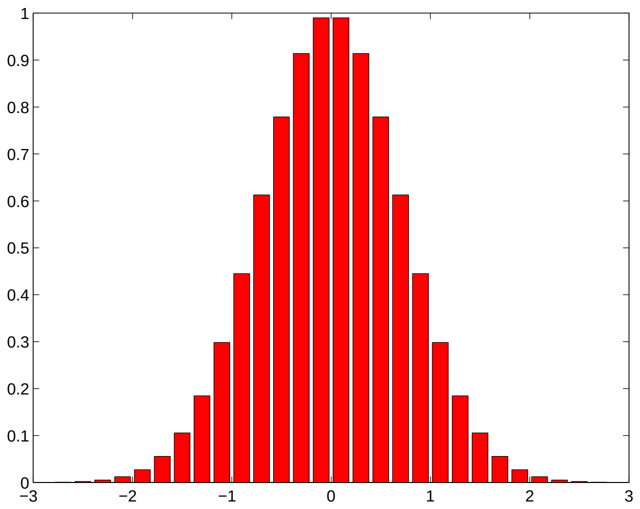
`h = bar(...)` returns a vector of handles to Patch graphics objects. `bar` creates one Patch graphics object per column in `Y`.

`barh(...), [xb,yb] = barh(...)`, and `h = barh(...)` create horizontal bars. `Y` determines the bar length. The vector `x` is a monotonic vector defining the *y*-axis intervals for horizontal bars.

## Examples

Plot a bell shaped curve:

```
x = –2.9:0.2:2.9;
bar(x,exp(–x.*x))
```



Create four subplots showing the effects of some bar arguments:

```
Y = round(rand(5,3)*10);
subplot(2,2,1)
bar(Y,'group')
title 'Group'

subplot(2,2,2)
bar(Y,'stack')
title 'Stack'

subplot(2,2,3)
barh(Y,'stack')
title 'Stack'

subplot(2,2,4)
bar(Y,1.5)
title 'Width = 1.5'
```

## See Also

`bar3`, `ColorSpec`, `patch`, `stairs`, `hist`

## Purpose

Three-dimensional bar chart

## Syntax

```
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineSpec)
h = bar3(...)

bar3h(...)
h = bar3h(...)
```

## Description

`bar3` and `bar3h` draw three-dimensional vertical and horizontal bar charts.

`bar3(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *x*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to `size(Y,2)`, which is the number of columns, and the elements in each row are grouped together.

`bar3(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a monotonic vector defining the *y*-axis intervals for vertical bars. If `Y` is a matrix, `bar3` clusters elements from the same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3(...,'`*`style`*`')` specifies the style of the bars. `'`*`style`*`'` is `'detached'`, `'grouped'`, or `'stacked'`. `'detached'` is the default mode of display.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *x* direction.

- 'grouped' displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in Y. The group contains one bar per column in Y.
- 'stacked' displays one bar for each row in Y. The bar height is the sum of the elements in the row. Each bar is multi-colored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

bar3(...,*LineSpec*) displays all bars using the color specified by *LineSpec*.

h = bar3(...) returns a vector of handles to Patch graphics objects. bar3 creates one Patch object per column in Y.

bar3h(...) and h = bar3h(...) create horizontal bars. Y determines the bar length. The vector x is a monotonic vector defining the *y*-axis intervals for horizontal bars.

## Examples

Create four subplots showing the effects of different arguments for bar3:

```
Y = rand(7,3);
subplot(2,2,1)
bar3(Y,'group')
title('Group')

subplot(2,2,2)
bar3(Y,'stacked')
title('Stacked')

subplot(2,2,3)
bar3(Y,.5)
title('Width =.5')

subplot(2,2,4)
bar3(Y,1.5)
title('Width=1.5')
```

## See Also

bar, LineSpec, patch

**Purpose**

Control Axes border

**Syntax**

```
box on
box off
box
```

**Description**

`box on` displays the boundary of the current Axes.

`box off` rdoes not display the boundary of the current Axes.

`box` toggles the visible state of the current Axes' boundary.

**Algorithm**

The `box` function sets the Axes `Box` property to `on` or `off`.

**See Also**

`axes`

# brighten

**Purpose** Brighten or darken colormap

**Syntax**

```
brighten(beta)
brighten(h,beta)
newmap = brighten(beta)
newmap = brighten(cmap,beta)
```

**Description** `brighten` increases or decreases the color intensities in a colormap. The modified colormap is brighter if `0 < beta < 1` and darker if `–1 < beta < 0`.

`brighten(beta)` replaces the current colormap with a brighter or darker colormap of essentially the same colors. `brighten(beta)`, followed by `brighten(–beta)`, where `beta < 1`, restores the original map.

`brighten(h,beta)` brightens all objects that are children of the Figure having the handle `h`.

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

`newmap = brighten(cmap,beta)` returns a brighter or darker version of the colormap `cmap` without changing the display.

**Examples** Brighten then darken the current colormap:

```
beta = .5; brighten(beta);
beta = –.5; brighten(beta);
```

**Algorithm** The values in the colormap are raised to the power of gamma, where gamma is

$$\gamma = \begin{cases} 1 - \beta, & \beta > 0 \\ \dfrac{1}{1+\beta}, & \beta \le 0 \end{cases}$$

`brighten` has no effect on graphics objects defined with true color.

**See Also** `colormap`, `rgbplot`

**Purpose** Screen capture

**Syntax**

```
capture
capture(h)
[X,cmap] = capture(h)
```

**Description**

`capture` creates a bitmap copy of the contents of the current Figure, including any Uicontrol graphics objects. It creates a new Figure and displays the bitmap copy as an Image graphics object in the new Figure.

`capture(h)` creates a new Figure that contains a copy of the Figure identified by `h`.

`[X,cmap] = capture(h)` returns an image matrix `X` and a colormap. You display this information using the statements

```
colormap(cmap)
image(X)
```

**Remarks** The resolution of a bitmap copy is less than that obtained with the `print` command.

**See Also** `image`, `print`

## caxis

**Purpose** Color axis scaling

**Syntax**

```
caxis([cmin cmax])
caxis auto
caxis manual
caxis(caxis)
v = caxis
```

**Description**

caxis controls the mapping of data values to the colormap. It affects any Surfaces, Patches, and Images with indexed CData and CDataMapping set to scaled . It does not affect Surfaces, Patches, or Images with true color CData or with CDataMapping set to direct.

caxis([cmin cmax]) sets the color limits to specified minimum and maximum values. Data values less than cmin or greater than cmax map to cmin and cmax, respectively. Values between cmin and cmax linearly map to the current colormap.

caxis auto lets MATLAB compute the color limits automatically using the minimum and maximum data values. This is MATLAB's default behavior. Color values set to Inf have the maximum color and values set to –Inf have the minimum color. Faces or edges with color values set to NaN are not drawn.

caxis manual and caxis(caxis) freeze the color axis scaling at the current limits. This enables subsequent plots to use the same limits when hold is on.

v = caxis returns a two-element row vector containing the [cmin cmax] currently in use.

**Examples**

Create (X,Y,Z) data for a sphere of radius 1 and view the data as a Surface:

```
[X,Y,Z] = sphere(32);
C = Z;
surf(X,Y,Z,C)
```

Values of C have the range [–1 1]. Values of C near –1 are assigned the lowest values in the colormap; values of C near +1 are assigned the highest values in the colormap.

Map the top half of the sphere to the highest value in the color table:

```
caxis([-1 0])
```

To use only the bottom half of the color table, enter

```
caxis([-1 3])
```

which maps the lowest CData values to the bottom of the colormap, and the hightest values to the middle of the colormap (by specifying a cmax whose value is equal to cmin plus twice the range of the CData).

The command

```
caxis auto
```

resets axis scaling back to auto-ranging and you see all the colors in the Surface. In this case, entering

```
v = caxis
```

returns

```
v =
     [-1 1]
```

## Algorithm

caxis changes the CLim and CLimMode properties of Axes graphics objects.

Surface, Patch and Image graphics objects with indexed CData and CDataMapping set to scaled map CData values to colors in the Figure colormap each time they render. CData values equal to or less than cmin map to the first color value in the colormap, and CData values equal to or greater than cmax map to the last color value in the colormap. MATLAB performs the following linear transformation on the intermediate values (referred to as C below) to map them to an entry in the colormap (whose length is m, and whose row index is referred to as index below):

```
index = fix((C-cmin)/(cmax-cmin)*m)+1
```

## See Also

axes, axis, colormap, get, mesh, pcolor, set, surf

The CLim and CLimMode properties of Axes graphics objects.

The ColorMap property of Figure graphics objects.

The Axes chapter in the *Graphics User's Guide*.

**Purpose** Clear current Axes

**Syntax**

```
cla
cla reset
```

**Description** `cla` deletes all graphics objects from the current Axes.

`cla reset` deletes all graphics objects from the current Axes and resets all Axes properties, except `Position`, to their default values.

**See Also** `clf`, `hold`, `reset`

**Purpose** Contour plot elevation labels

**Syntax**

```
clabel(C,h)
clabel(C,h,v)
clabel(C,h,'manual')

clabel(C)
clabel(C,v)
clabel(C,'manual')
```

**Description** The `clabel` function adds height labels to a two-dimensional contour plot.

`clabel(C,h)` rotates the labels and inserts them in the contour lines. The function inserts only those labels that fit within the contour, due to the size of the contour.

`clabel(C,h,v)` creates labels only for those contour levels given in vector `v`, then rotates the labels and inserts them in the contour lines.

`clabel(C,h,'manual')` places contour labels at locations you select with a mouse. You press the left mouse button (the only mouse button on a single-button mouse), or the space bar to label a contour at the closest location beneath the center of the cursor. Press the **Return** key while the cursor is within the Figure window to terminate labeling. The labels are rotated and inserted in the contour lines.

`clabel(C)` adds labels to the current contour plot using the contour structure `C` output from `contour`. The function labels all contours displayed and randomly selects label positions.

`clabel(C,v)` labels only those contour levels given in vector `v`.

`clabel(C,'manual')` places contour labels at locations you select with a mouse.

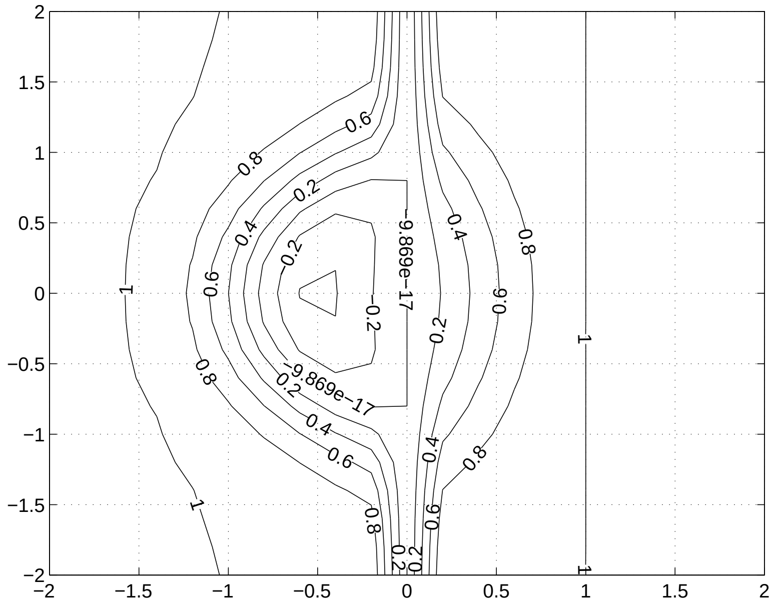
## Remarks

When the syntax includes the argument `h`, this function rotates the labels and inserts them in the contour lines (see Example). Otherwise, the labels are displayed upright and a `'+'` indicates which contour line the label is annotating.

## Examples

Generate, draw, and label a simple contour plot:

```
[x,y] = meshgrid(–2:.2:2);
z = x.^exp(–x.^2–y.^2);
[C,h] = contour(x,y,z);
clabel(C,h);
```



**See Also** `contour`, `contourc`, `contourf`

## Purpose

Clear command window

## Syntax

`clc`

## Description

`clc` clears the command window.

## Examples

Display a sequence of random matrices at the same location in the command window:

```
clc
for i =1:25
   home
   A = rand(5)
end
```

## See Also

`clf`, `home`

**Purpose** Clear current Figure window

**Syntax**

```
clf
clf reset
```

**Description** `clf` deletes all graphics objects from the current Figure.

`clf reset` deletes all graphics objects within the current Figure and resets all Figure properties, except `Position`, to their default values.

**See Also** `cla`, `clc`, `hold`, `reset`

## Purpose

Delete specified Figure

## Syntax

```
close
close(h)
close name
close all
close all hidden
status = close(...)
```

## Description

`close` deletes the current Figure or the specified Figure(s). It optionally returns the status of the close operation.

`close` deletes the current Figure (equivalent to `close(gcf)`).

`close(h)` deletes the Figure identified by `h`. If `h` is a vector or matrix, `close` deletes all Figures identified by `h`.

`close name` deletes the Figure with the specified name.

`close all` deletes all Figures whose handles are not hidden.

`close all hidden` deletes all figures including those with hidden handles.

`status = close(...)` returns `1` if the specified windows have been deleted and `0` otherwise.

## Remarks

The `close` function works by evaluating the specified Figure's `CloseRe-questFcn` property with the statement:

```
eval(get(h,'CloseRequestFcn'))
```

The default `CloseRequestFcn`, `closereq`, deletes the current Figure using `delete(get(0,'CurrentFigure'))`. If you specify multiple Figure handles, `close` executes each Figure's `CloseRequestFcn` in turn. If MATLAB encounters an error that terminates the execution of a `CloseRequestFcn`, the Figure is not deleted. Note that using your computer's window manager (i.e., the **Close** menu item) also calls the Figure's `CloseRequestFcn`.

If a Figure's handle is hidden (i.e., the Figure's `HandleVisibility` property is set to `callback` or `off` and the Root `ShowHiddenHandle` property is set `no`), you must specify the `hidden` option when trying to access a Figure using the `all` option.

To unconditionally delete all Figures, use the statements:

```
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

The delete function does not execute the Figure's `CloseRequestFcn`, it simply deletes the specified Figure.

The Figure `CloseRequestFcn` allows you to either delay or abort the closing of a Figure once the `close` function has been issued. For example, you can display a dialog box to see if the user really want to delete the Figure or save and cleanup before closing.

## See Also

`delete`, `figure`, `gcf`

The Figure `HandleVisibility` property

The Root `ShowHiddenHandle` property

## colorbar

**Purpose** Display colorbar showing the color scale

**Syntax**

```
colorbar
colorbar('vert')
colorbar('horiz')
colorbar(h)
h = colorbar(...)
```

**Description** The `colorbar` function displays the current colormap in the current Figure and resizes the current Axes to accommodate the colorbar.

`colorbar` updates the most recently created colorbar, or when the current Axes does not have a colorbar, `colorbar` adds a new vertical colorbar.

`colorbar('vert')` adds a vertical colorbar to the current Axes.

`colorbar('horiz')` adds a horizontal colorbar to the current Axes.

`colorbar(h)` places a colorbar in the Axes identified by `h`. The colorbar is horizontal if the width of the Axes is greater than its height, as determined by the Axes `Position` property.

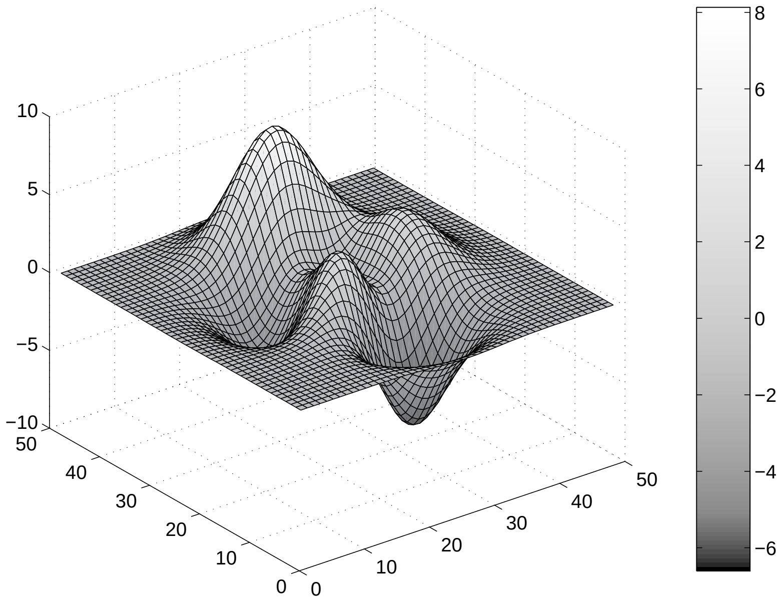
`h = colorbar(...)` returns a handle to the colorbar, which is an Axes graphics object.

**Remarks** `colorbar` works with two-dimensional and three-dimensional plots.

## Examples

Display a colorbar beside the Axes:

```
surf(peaks);
colorbar
```



## See Also

`colormap`

## Purpose

Sets default property values to display different color schemes

## Syntax

```
colordef white
colordef black
colordef none
colordef(fig,color_option)
h = colordef('new',color_option)
```

## Description

`colordef` enables you to select either a white or black background for graphics display. It sets axis lines and labels to show up against the background color.

`colordef white` sets the axis background color to white, the axis lines and labels to black, and the Figure background color to light gray.

`colordef black` sets the axis background color to black, the axis lines and labels to white, and the Figure background color to dark gray.

`colordef none` sets the Figure coloring to that used by MATLAB Version 4 (essentially a black background).

`colordef(fig,color_option)` sets the color scheme of the Figure identified by the handle `fig` to the color option `'white'`, `'black'`, or `'none'`.

`h = colordef('new',color_option)` returns the handle to a new Figure created with the specified color options (i.e., `'white'`, `'black'`, or `'none'`).

## Remarks

`colordef` affects only subsequently drawn Figures, not those currently on the display. This is because `colordef` works by setting default property values (on the Root or Figure level). You can list the currently set default values on the Root level with the statement:

```
get(0,'defaults')
```

You can remove all default values using the `reset` command:

```
reset(0)
```

See the `get` and `reset` references pages for more information.

## See Also

`whitebg`

# colormap

## Purpose

Set and get the current colormap

## Syntax

```
colormap(map)
colormap('default')
cmap = colormap
```

## Description

A colormap is an $m$-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color. The $k^{th}$ row of the colormap defines the $k$-th color, where `map(k,:) = [r(k) g(k) b(k)])` specifies the intensity of red, green, and blue.

`colormap(map)` sets the colormap to the matrix `map`. If any values in `map` are outside the interval [0 1], MATLAB returns the error: `Colormap must have values in [0,1]`

`colormap('default')` sets the current colormap to the default colormap.

`cmap = colormap;` retrieves the current colormap. The values returned are in the interval [0 1].

### Specifying Colormaps

M-files in the `color` directory generate a number of colormaps. Each M-file accepts the colormap size as an argument. For example,

```
colormap(hsv(128))
```

creates an `hsv` colormap with 128 colors. If you do not specify a size, MATLAB creates a colormap the same size as the current colormap.

### Supported Colormaps

MATLAB supports a number of colormaps.

- `autumn` varies smoothly from red, through orange, to yellow.
- `bone` is a grayscale colormap with a higher value for the blue component. This colormap is useful for adding an “electronic” look to grayscale images.
- `colorcube` contains as many regularly spaced colors in RGB colorspace as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
- `cool` consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
- `copper` varies smoothly from black to bright copper.
- `flag` consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
- `gray` returns a linear grayscale colormap.
- `hot` varies smoothly from black, through shades of red, orange, and yellow, to white.
- `hsv` varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions. `hsv(m)` is the same as `hsv2rgb([h ones(m,2)])` where `h` is the linear ramp, `h = (0:m–1)'/m`.
- `jet` ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the `hsv` colormap. The `jet` colormap is associated with an astrophysical fluid jet simulation from the National Center for Su-

percomputer Applications. The following commands display the `flujet` data using the `jet` colormap:

```
load flujet
image(X)
colormap(jet)
```

- `lines` produces a colormap of colors specified by the Axes `ColorOrder` property and a shade of gray.
- `pink` contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
- `prism` repeats the six colors red, orange, yellow, green, blue, and violet.
- `spring` consists of colors that are shades of magenta and yellow.
- `summer` consists of colors that are shades of green and yellow.
- `white` is an all white monochrome colormap.
- `winter` consists of colors that are shades of blue and green.

## Examples

The Images and colormaps demo, `imagedemo`, provides an introduction to colormaps. Select **Color Spiral** from the menu (starts automatically on the Macintosh). This uses the `pcolor` function to display a 16-by-16 matrix whose elements vary from 0 to 255 in a rectilinear spiral. The `hsv` colormap starts with red in the center, then passes through yellow, green, cyan, blue, and magenta before returning to red at the outside end of the spiral. Selecting **Colormap Menu** gives access to a number of other colormaps (except for on the Macintosh).

The `rgbplot` function plots colormap values. Try `rgbplot(hsv)`, `rgbplot(gray)`, and `rgbplot(hot)`.

The demos directory contains a CAT scan image of a human spine. To view the image:

```
load spine
image(X)
colormap bone
```

## Algorithm

Each Figure has its own `ColorMap` property. `colormap` is an M-file that sets and gets this property.

## See Also

`brighten`, `caxis`, `contrast`, `hsv2rgb`, `pcolor`, `rgb2hsv`, `rgbplot`

The `ColorMap` property of Figure graphics objects.

# ColorSpec

**Purpose** Color specification

**Description** `ColorSpec` is not a command; it refers to the three ways in which you specify color in MATLAB:

- RGB triple
- Short name
- Long name

The short names and long names are MATLAB strings that specify one of eight predefined colors. The RGB triple is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB equivalents.

| RGB Value | Short Name | Long Name |
|---|---|---|
| `[1 1 0]` | `y` | `yellow` |
| `[1 0 1]` | `m` | `magenta` |
| `[0 1 1]` | `c` | `cyan` |
| `[1 0 0]` | `r` | `red` |
| `[0 1 0]` | `g` | `green` |
| `[0 0 1]` | `b` | `blue` |
| `[1 1 1]` | `w` | `white` |
| `[0 0 0]` | `k` | `black` |

**Remarks** The eight predefined colors and any colors you specify as RGB values are not part of a Figure's colormap, nor are they affected by changes to the Figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

## Examples

To change the background color of a Figure to green, specify the color with a short name, a long name, or an RGB triple. These statements generate equivalent results:

```
whitebg('g')
whitebg('green')
whitebg([0 1 0]);
```

You can use `ColorSpec` anywhere you need to define a color. For example, this statement changes the Figure background color to pink:

```
set(gcf,'Color',[1 .4 .6])
```

## See Also

`bar`, `bar3`, `colormap`, `fill`, `fill3`, `whitebg`

# comet

## Purpose

Two-dimensional comet plot

## Syntax

```
comet
comet(y)
comet(x,y)
comet(x,y,p)
```

## Description

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet` demonstrates the comet plot.

`comet(y)` displays a comet plot of the vector `y`.

`comet(x,y)` displays a comet plot of vector `y` versus vector `x`.

`comet(x,y,p)` specifies a comet body of length `p*length(y)`. `p` defaults to `0.1`.

## Examples

Create a simple comet plot:

```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```

## See Also

`comet3`

## comet3

**Purpose** Three-dimensional comet plot

**Syntax**

```
comet3
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
```

**Description** A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet3`, with no arguments, demonstrates the three-dimensional comet plot.

`comet3(z)` displays a three-dimensional comet plot of the vector `z`.

`comet3(x,y,z)` displays a comet plot of the curve through the points `[x(i),y(i),z(i)]`.

`comet3(x,y,z,p)` specifies a comet body of length `p*length(y)`.

**Examples** Create a three-dimensional comet plot:

```
t = –10*pi:pi/250:10*pi;
comet3((cos(2*t).^2).*sin(t),(sin(2*t).^2).*cos(t),t);
```

**See Also** `comet`

# compass

**Purpose** Plot arrows emanating from the origin

**Syntax**

```
compass(X,Y)
compass(Z)
compass(...,LineSpec)
h = compass(...)
```

**Description** A compass plot displays direction or velocity vectors as arrows emanating from the origin. `X`, `Y`, and `Z` are in Cartesian coordinates and plotted on a circular grid.

`compass(X,Y)` displays a compass plot having *n* arrows, where *n* is the number of elements in `X` or `Y`. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by `[X(i),Y(i)]`.

`compass(Z)` displays a compass plot having *n* arrows, where *n* is the number of elements in `Z`. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of `Z`. This syntax is equivalent to `compass(real(Z),imag(Z))`.

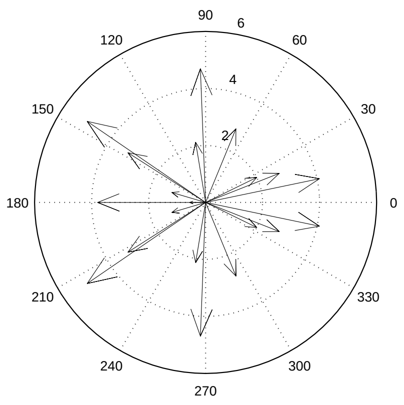
`compass(...,`*`LineSpec`*`)` draws a compass plot using the line type, marker symbol, and color specified by *LineSpec*.

`h = compass(...)` returns handles to Line objects.

## Examples

Draw a compass plot of the eigenvalues of a matrix:

```
Z = eig(randn(20,20));
compass(Z)
```



## See Also

`feather`, `LineSpec`, `rose`

# contour

## Purpose

Two-dimensional contour plot

## Syntax

```
contour(Z)
contour(Z,n)
contour(Z,v)
contour(X,Y,Z)
contour(X,Y,Z,n)
contour(X,Y,Z,v)
contour(...,LineSpec)
[C,h] = contour(...)
```

## Description

A contour plot displays isolines of matrix `Z`. You label the contour lines using `clabel`.

`contour(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix. The number of contour levels and the values of the contour levels are chosen automatically based on the minimum and maximum values of `Z`. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels.

`contour(Z,v)` draws a contour plot of matrix `Z` with contour lines at the data values specified in vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`.

`contour(X,Y,Z)`, `contour(X,Y,Z,n)`, and `contour(X,Y,Z,v)` draw contour plots of `Z`. `X` and `Y` specify the *x*- and *y*-axis limits. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does.

`contour(...,`*`LineSpec`*`)` draws the contours using the line type and color specified by *LineSpec*. Marker symbols are ignored.

`[C,h] = contour(...)` returns the contour matrix `C` (see `contourc`) and a vector of handles to graphics objects. `clabel` uses the contour matrix `C` to create the labels. `contour` creates Patch graphics objects unless you specify *LineSpec*, in which case `contour` creates Line graphics objects.

## Remarks

If you do not specify *LineSpec*, colormap and caxis control the color.

If X or Y is irregularly spaced, contour calculates contours using a regularly spaced contour grid, then transforms the data to X or Y.

## Examples

To view a contour plot of the function

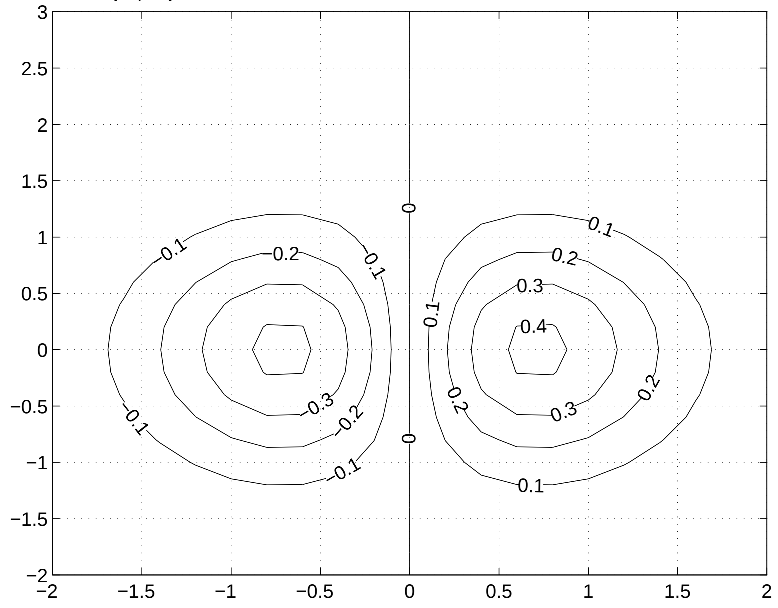
$$z = xe^{(-x^2 - y^2)}$$

over the range $-2 \le x \le 2$, $-2 \le y \le 3$, create matrix Z using the statements

```
xrange = -2:.2:2;
yrange = -2:.2:3;
[X,Y] = meshgrid(xrange,yrange);
Z = X.*exp(-X.^2-Y.^2);
```

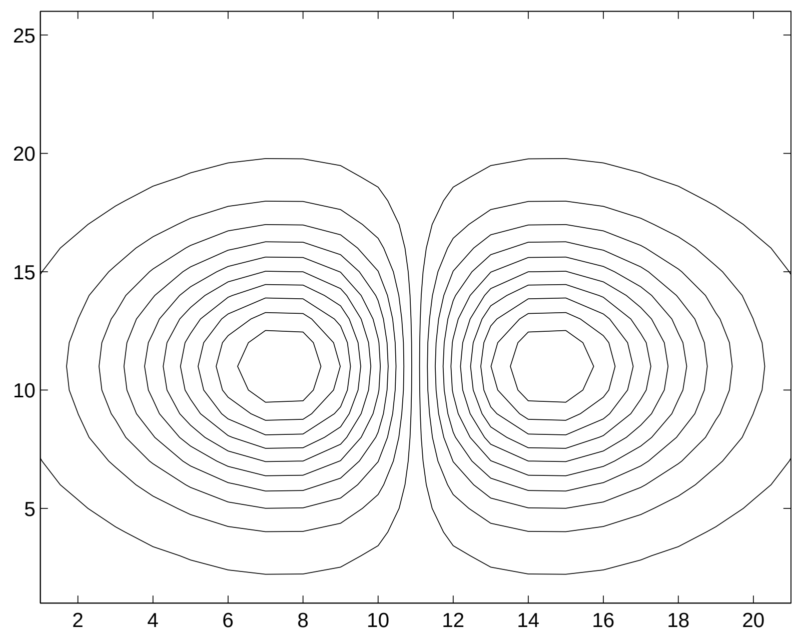
Then, generate a contour plot of Z:

```
[C,h] = contour(X,Y,Z);
clabel(C,h)
```



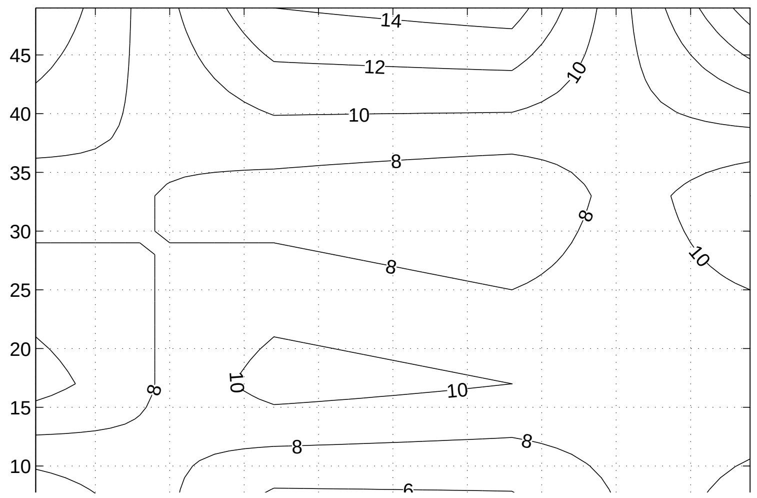
View the same function using the default range and 20 evenly spaced contour lines:

```
contour(Z,20);
```



Use `interp2` and `contour` to create smoother contours:

```
Z = magic(4);
[C,h] = contour(interp2(Z,4));
clabel(C,h)
```



## See Also

`clabel`, `contour3`, `contourc`, `contourf`, `quiver`

The `interp2` function in the *MATLAB Language Reference Manual*.

# contour3

**Purpose** Three-dimensional contour plot

**Syntax**

```
contour3(Z)
contour3(Z,n)
contour3(Z,v)
contour3(X,Y,Z)
contour3(X,Y,Z,n)
contour3(X,Y,Z,v)
contour3(...,LineSpec)
[C,h] = contour3(...)
```

**Description** `contour3` creates a three-dimensional contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a three-dimensional view. `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix. The number of contour levels and the values of contour levels are chosen automatically. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a three-dimensional view.

`contour3(Z,v)` draws a contour plot of matrix `Z` with contour lines at the values specified in vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`.

`contour3(X,Y,Z)`, `contour3(X,Y,Z,n)`, and `contour3(X,Y,Z,v)` use `X` and `Y` to define the *x*- and *y*-axis limits. If `X` is a matrix, `X(1,:)` defines the *x*-axis. If `Y` is a matrix, `Y(:,1)` defines the *y*-axis. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does.

`contour3(...,`*`LineSpec`*`)` draws the contours using the line type and color specified by *LineSpec*.

`[C,h] = contour3(...)` returns the contour matrix `C` as described in the function `contourc` and a column vector containing handles to graphics objects. `contour3` creates Patch graphics objects unless you specify *LineSpec*, in which case `contour3` creates Line graphics objects.

## Remarks

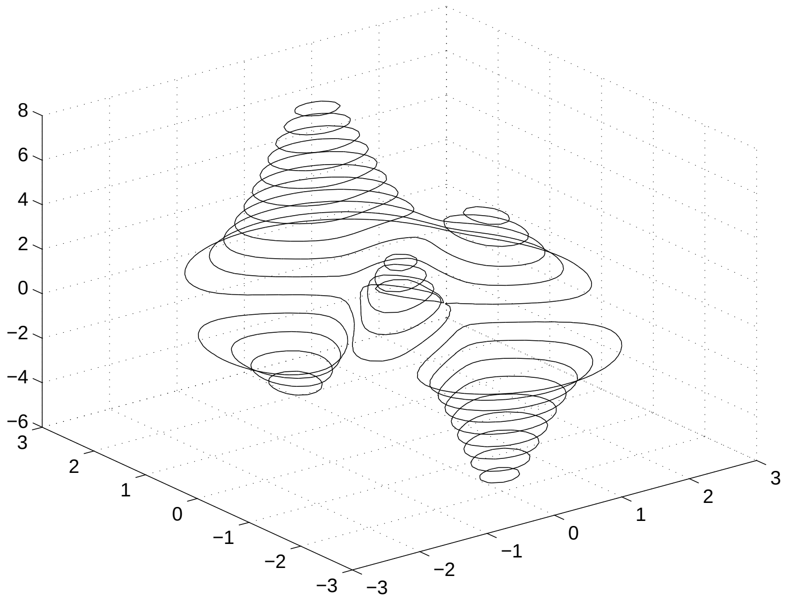
If you do not specify *LineSpec*, `colormap` and `caxis` control the color.

If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, then transforms the data to `X` or `Y`.

## Examples

Plot the three-dimensional contour of the `peaks` function:

```
xrange = -3:.125:3;
yrange = xrange;
[X,Y] = meshgrid(xrange,yrange);
Z = peaks(X,Y);
contour3(X,Y,Z,20);
```



## See Also

`contour`, `contourc`, `meshc`, `meshgrid`, `surfc`

# contourc

## Purpose

Low-level contour plot computation

## Syntax

```
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)
```

## Description

`contourc` calculates the contour matrix `C` used by `contour`, `contour3`, and `contourf`. The values in `Z` determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of `Z`.

`C = contourc(Z)` computes the contour matrix from data in matrix `Z`, where `Z` must be at least a 2-by-2 matrix. The contours are isolines in the units of `Z`. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

`C = contourc(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = contourc(Z,v)` computes contours of matrix `Z` with contour lines at the values specified in vector `v`. The length of `v` determines the number of contour levels. To compute a single contour of level `i`, use `contourc(Z,[i i])`.

`C = contourc(x,y,Z)`, `C = contourc(x,y,Z,n)`, and `C = contourc(x,y,Z,v)` compute contours of `Z` using vectors `x` and `y` to determine the *x*- and *y*-axis limits. `x` and `y` must be monotonically increasing.

## Remarks

`C` is a two-row matrix specifying all the contour lines. Each contour line defined in matrix `C` begins with a column that contains the value of the contour (specified by `v` and used by `clabel`), and the number of `(x,y)` vertices in the contour line. The remaining columns contain the data for the `(x,y)`pairs.

```
C = [value1 xdata(1) xdata(2)...value2 xdata(1) xdata(2)...;
     dim1   ydata(1) ydata(2)...dim2   ydata(1) ydata(2)...]
```

Specifying irregularly spaced `x` and `y` vectors is not the same as contouring irregularly spaced data. If `x` or `y` is irregularly spaced, `contourc` calculates contours using a regularly spaced contour grid, then transforms the data to `x` or `y`.

**See Also** `clabel`, `contour`, `contour3`, `contourf`

# contourf

## Purpose

Filled two-dimensional contour plot

## Syntax

```
contourf(Z)
contourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
[C,h,CF] = contourf(...)
```

## Description

A filled contour plot displays isolines calculated from matrix `Z` and fills the areas between the isolines using constant colors. The color of the filled areas depends on the current Figure's colormap.

`contourf(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to a plane. `Z` must be at least a 2-by-2 matrix. The number of contour lines and the values of the contour lines are chosen automatically.

`contourf(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels.

`contourf(Z,v)` draws a contour plot of matrix `Z` with contour levels at the values specified in vector `v`.

`contourf(X,Y,Z)`, `contourf(X,Y,Z,n)`, and `contourf(X,Y,Z,v)` produce contour plots of `Z` using `X` and `Y` to determine the *x*- and *y*-axis limits. When `X` and `Y` are matrices, they must be the same size as `Z`, in which case they specify a surface as `surf` does.

`[C,h,CF] = contourf(...)` returns the contour matrix `C` as calculated by the function `contourc` and used by `clabel`, a vector of handles `h` to Patch graphics objects, and a contour matrix `CF` for the filled areas.

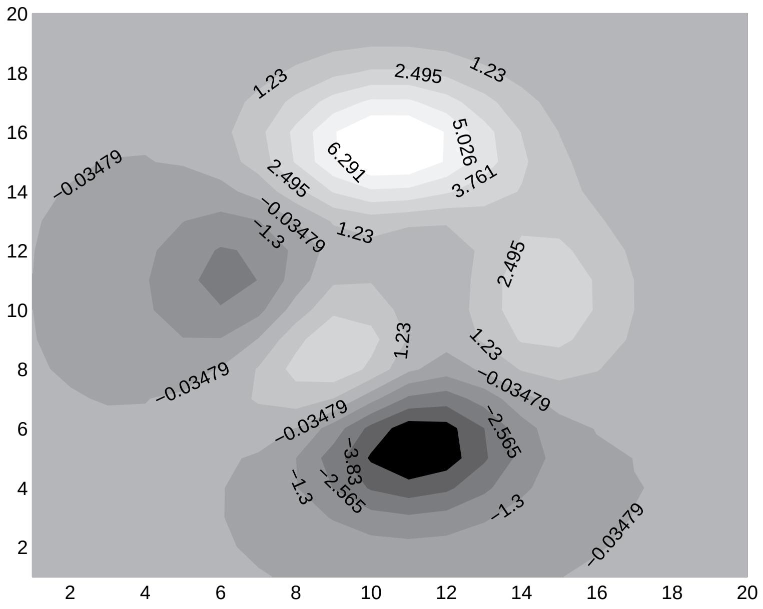
## Remarks

If `X` or `Y` is irregularly spaced, `contourf` calculates contours using a regularly spaced contour grid, then transforms the data to `X` or `Y`.

## Examples

Create a filled contour plot of the peaks function:

```
[C,h] = contourf(peaks(20),10);
clabel(C,h)
```



## See Also

`clabel`, `contour`, `contour3`, `contourc`, `quiver`

## Purpose

Grayscale colormap for contrast enhancement

## Syntax

```
cmap = contrast(X)
cmap = contrast(X,m)
```

## Description

The `contrast` function enhances the contrast of an Image. It creates a new gray colormap, `cmap`, that has an approximately equal intensity distribution. All three elements in each row are identical.

`cmap = contrast(X)` returns a gray colormap that is the same length as the current colormap.

`cmap = contrast(X,m)` returns an `m`-by-3 gray colormap.

## Examples

Add contrast to the clown image defined by `X`:

```
load clown;
cmap = contrast(X);
image(X);
colormap(cmap);
```

## See Also

`brighten`, `gray`, `image`

**Purpose**

Copy graphics objects and their descendants

**Syntax**

```
new_handle = copyobj(h,p)
```

**Description**

`copyobj` creates copies of graphics objects. The copies are identical to the original objects except the copies have different values for their `Parent` property and a new handle. The new parent must be appropriate for the copied object (e.g., you can copy a Line object only to another Axes object).

`new_handle = copyobj(h,p)` copies one or more graphics objects identified by `h` and returns the handle of the new object or a vector of handles to new objects. The new graphics objects are children of the graphics objects specified by `p`.

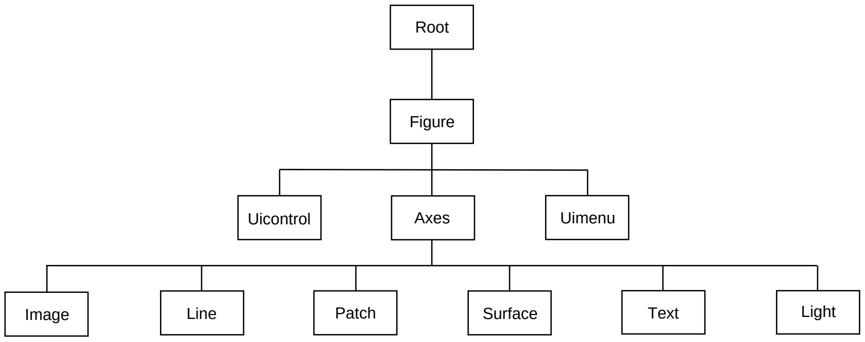
**Remarks**

`h` and `p` can be scalars or vectors. When both are vectors, they must be the same length and the output argument, `new_handle`, is a vector of the same length. In this case, `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p(i)`.

When `h` is a scalar and `p` is a vector, `h` is copied once to each of the parents in `p`. Each `new_handle(i)` is a copy of `h` with its `Parent` property set to `p(i)`, and `length(new_handle)` equals `length(p)`.

When `h` is a vector and `p` is a scalar, each `new-handle(i)` is a copy of `h(i)` with its `Parent` property set to `p`. The length of `new_handle` equals `length(h)`.

Graphics objects are arranged as a hierarchy. Here, each graphics object is shown connected below its appropriate parent object.



**Examples**

Copy a set of Patch handles into a new Figure by assigning the `Parent` property of the new Patch graphics objects to the current Axes:

```
X = rand(5,3);
Y = rand(5,3);
C = rand(1,3);
h = fill(X,Y,C);
figure     % Create a new figure window
axes       % Create an axes object in the figure window
new_handle = copyobj(h,gca);
```

**See Also**

`findobj`, `gcf`, `gca`, `gco`, `get`, `set`

`Parent` property for all graphics objects.

## Purpose

Generate cylinder

## Syntax

```
[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(...)
```

## Description

`cylinder` generates *x*, *y*, and *z* coordinates of a unit cylinder. You can draw the cylindrical object using `surf` or `mesh`, or draw it immediately by not providing output arguments.

`[X,Y,Z] = cylinder` returns the *x*, *y*, and *z* coordinates of a cylinder with a radius equal to `1`. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r)` returns the *x*, *y*, and *z* coordinates of a cylinder using `r` to define a profile curve. `cylinder` treats each element in `r` as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r,n)` returns the *x*, *y*, and *z* coordinates of a cylinder based on the profile curve defined by vector `r`. The cylinder has `n` equally spaced points around its circumference.

`cylinder(...)`, with no output arguments, plots the cylinder using `surf`.

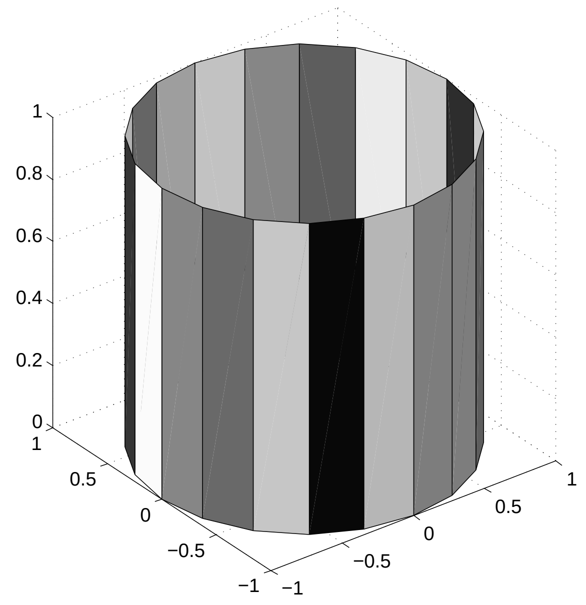
## Remarks

`cylinder` treats its first argument as a profile curve. The resulting Surface graphics object is generated by rotating the curve about the *x*-axis, and then aligning it with the *z*-axis.

## Examples

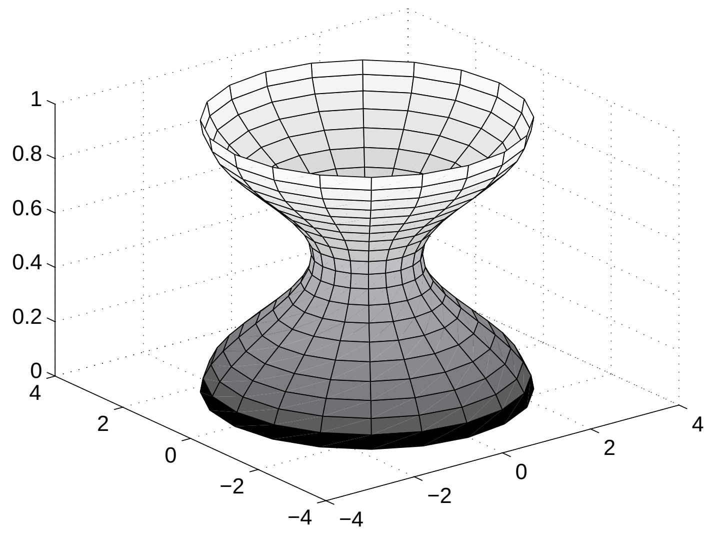
Create a cylinder with randomly colored faces.

```
cylinder
axis square
h = findobj('Type','surface');
set(h,'CData',rand(size(get(h,'CData'))))
```



Generate a cylinder defined by the profile function `2+sin(t)`:

```
t = 0:pi/10:2*pi;
axis square
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
```



**See Also** `sphere`, `surf`

## Purpose

Label tick lines using dates

## Syntax

```
datetick(tickaxis)
datetick(tickaxis,dateform)
```

## Description

`datetick(tickaxis)` labels the tick lines of an axis using dates, replacing the default numeric labels. `tickaxis` is the string `'x'`, `'y'`, or `'z'`. The default is `'x'`. `datetick` selects a label format based on the minimum and maximum limits of the specified axis.

`datetick(tickaxis,dateform)` formats the labels according to the integer `dateform` (see table). To produce correct results, the data for the specified axis must be serial date numbers (as produced by `datenum`).

| Dateform | Format | Example |
|---|---|---|
| 0 | day-month-year hour:minute | 01-Mar-1995 03:45 |
| 1 | day-month-year | 01-Mar-1995 |
| 2 | month/day/year | 03/01/95 |
| 3 | month, three letters | Mar |
| 4 | month, single letter | M |
| 5 | month, numeral | 3 |
| 6 | month/day | 03/01 |
| 7 | day of month | 1 |
| 8 | day of week, three letters | Wed |
| 9 | day of week, single letter | W |
| 10 | year, four digit | 1995 |
| 11 | year, two digit | 95 |

| Dateform | Format | Example |
|---|---|---|
| 12 | month year | Mar95 |
| 13 | hour:minute:second | 15:45:17 |
| 14 | hour:minute:second AM or PM | 03:45:17 |
| 15 | hour:minute | 15:45 |
| 16 | hour:minute AM or PM | 03:45 PM |

## Remarks

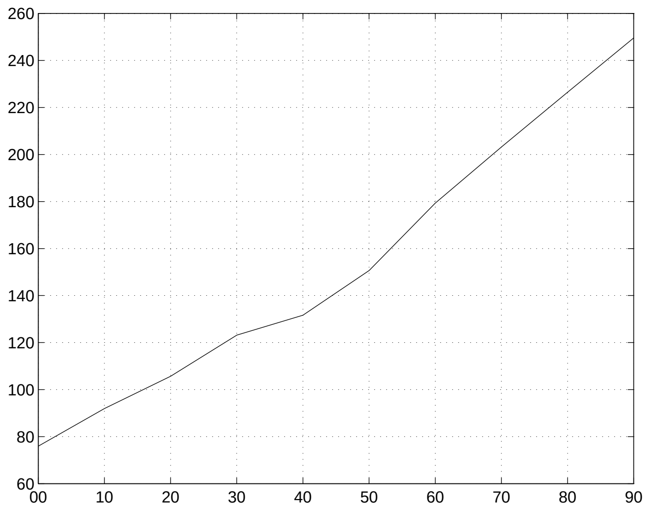
`datetick` calls `datestr` to convert date numbers to date strings.

To change the tick spacing and locations, set the appropriate Axes property (i.e., `XTick`, `YTick`, or `ZTick`) before calling `datetick`.

**Example**

Consider graphing population data based on the 1990 U.S. census:

```
t = (1900:10:1990)';   % Time interval
p = [75.995 91.972 105.711 123.203 131.669 ...
   150.697 179.323 203.212 226.505 249.633]';  % Population
plot(datenum(t,1,1),p) % Convert years to date numbers and plot
datetick('x',11)   % Replace x-axis ticks with 2-digit year
labels
```



**See Also**

The Axes properties `XTick`, `YTick`, and `ZTick`.

The `datenum` and `datestr` functions in the *MATLAB Language Reference Guide*.

**Purpose** MATLAB Version 4.0 Figure and Axes defaults

**Syntax**

```
default4
default4(h)
```

**Description** `default4` sets Figure and Axes defaults to match MATLAB Version 4.0 defaults.

`default4(h)` only affects the Figure with handle `h`.

**See Also** `wdefault`, `kdefault`

# dialog

**Purpose** Create and display dialog box

**Syntax**

```
h = dialog('PropertyName',PropertyValue,...)
```

**Description** `h = dialog('`*`PropertyName`*`',PropertyValue,...)` returns a handle to a dialog box. This function creates a Figure graphics object and sets the Figure properties recommended for dialog boxes. You can specify any valid Figure property value.

**See Also** `errordlg`, `figure`, `helpdlg`, `inputdlg`, `questdlg`, `uiwait`, `uiresume`, `warndlg`

**Purpose**

Drag rectangles with mouse

**Syntax**

```
[finalRect] = dragrect(initialRect)
[finalRect] = dragrect(initialRect,stepSize)
```

**Description**

`[finalRect] = dragrect(initialRect)` tracks one or more rectangles anywhere on the screen. The *n*-by-4 matrix `rect` defines the rectangles. Each row of `rect` must contain the initial rectangle position as `[left bottom width height]` values. `dragrect` returns the final position of the rectangles in `finalRect`.

`[finalRect] = dragrect(initialRect,stepSize)` moves the rectangles in increments of STEPSIZE. The lower-left corner of the first rectangle is constrained to a grid of size STEPSIZE starting at the lower-left corner of the figure, and all other rectangles maintain their original offset from the first rectangle. `[finalRect] = dragrect(...)` returns the final positions of the rectangles when the mouse button is released. The default stepsize is `1`.

**Remarks**

`dragrect` returns immediately if a mouse button is not currently pressed. Use `dragrect` in a ButtonDownFcn, or from the commandline in conjunction with `waitforbuttonpress`, to ensure that the mouse button is down when `dragrect` is called. `dragrect` returns when you release the mouse button.

**Example**

Drag a rectangle that is 50 pixels wide and 100 pixels in height.

```
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

**See Also**

`rbbox`, `waitforbuttonpress`

# drawnow

**Purpose** Complete pending drawing events

**Synopsis** `drawnow`

**Description** `drawnow` flushes the event queue and updates the Figure window.

**Remarks** Other events that cause MATLAB to flush the event queue and draw the Figure windows include returning to the MATLAB prompt, a `pause` statement, a `waitforbuttonpress` statement, a `waitfor` statement, a `getframe` statement, and a `figure` statement.

**Examples** Executing the statements

```
x = –pi:pi/20:pi;
plot(x,cos(x))
drawnow
title('A Short Title')
grid
```

as an M-file updates the current Figure after executing the `drawnow` function and after executing the final statement.

**See Also** `waitfor`, `pause`, `waitforbuttonpress`

## Purpose

Plot error bars along a curve

## Syntax

```
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineSpec)
h = errorbar(...)
```

## Description

Error bars show the confidence level of data or the deviation along a curve.

`errorbar(Y,E)` plots `Y` and draws an error bar at each element of `Y`. The error bar is a distance of `E(i)` above and below the curve so that each bar is symmetric and `2*E(i)` long.

`errorbar(X,Y,E)` plots `X` versus `Y` with symmetric error bars `2*E(i)` long. `X`, `Y`, `E` must be the same size. When they are vectors, each error bar is a distance of `E(i)` above and below the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `E(i,j)` above and below the point defined by `(X(i,j),Y(i,j))`.

`errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars `L(i)+U(i)` long specifying the lower and upper error bars. `X`, `Y`, `L`, and `U` must be the same size. When they are vectors, each error bar is a distance of `L(i)` below and `U(i)` above the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `L(i,j)` below and `U(i,j)` above the point defined by `(X(i,j),Y(i,j))`.

`errorbar(...,`*LineSpec*`)` draws the error bars using the line type, marker symbol, and color specified by *LineSpec*.

`h = errorbar(...)` returns a vector of handles to Line graphics objects.

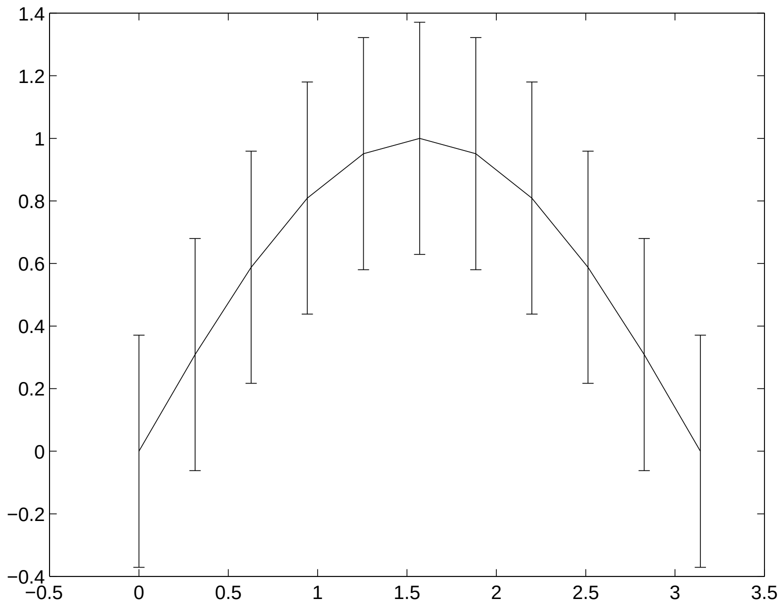
## Remarks

When the arguments are all matrices, `errorbar` draws one line per matrix column. If `X` and `Y` are vectors, they specify one curve.

## Examples

Draw symmetric error bars that are two standard deviation units in length:

```
X = 0:pi/10:pi;
Y = sin(X);
E = std(Y)*ones(size(X));
errorbar(X,Y,E)
```



## See Also

`LineSpec`, `plot`

The `std` function in the online MATLAB Function Reference for more informatoin.

## errordlg

**Purpose** Create and display an error dialog box

**Syntax**

```
errordlg
errordlg('errorstring')
errordlg('errorstring','dlgname')
errordlg('errorstring','dlgname','on')
h = errordlg(...)
```

**Description** `errordlg` creates an error dialog box, or if the named dialog exists, `errordlg` pops the named dialog in front of other windows.

`errordlg` displays a dialog box named `'Error Dialog'` and contains the string `'This is the default error string.'`

`errordlg('`*`errorstring`*`')` displays a dialog box named `'Error Dialog'` that contains the string `'`*`errorstring`*`'`.

`errordlg('`*`errorstring`*`','`*`dlgname`*`')` displays a dialog box named `'`*`dlgname`*`'` that contains the string `'`*`errorstring`*`'`.

`errordlg('`*`errorstring`*`','`*`dlgname`*`','on')` specifies whether to replace an existing dialog box having the same name. `'on'` brings an existing error dialog having the same name to the foreground. In this case, `errordlg` does not create a new dialog.

`h = errordlg(...)` returns the handle of the dialog box.

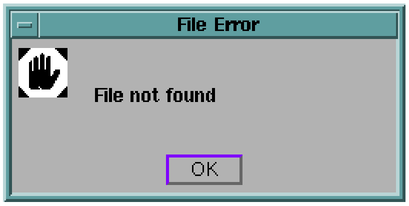
**Remarks** MATLAB sizes the dialog box to fit the string `'`*`errorstring`*`'`. The error dialog box has an OK pushbutton and remains on the screen until you press the OK button or the **Return** key. After pressing the button, the error dialog box disappears.

The appearance of the dialog box depends on the windowing system you use.

**Examples** The function

```
errordlg('File not found','File Error');
```

displays the following dialog box on a UNIX system:



File not found

**See Also** `dialog`, `helpdlg`, `msgbox`, `questdlg`, `warndlg`

## Purpose

Easy to use function plotter.

## Syntax

```
ezplot(f)
ezplot(f,[xmin xmax])
ezplot(f,[xmin xmax],fig)
```

## Description

`ezplot(f)` plots a graph of *f(x)*, where `f` is a symbolic expression representing a mathematical expression involving a single symbolic variable, say *x*. The domain on the *x*-axis is usually `[–2*pi, 2*pi]`.

`ezplot(f,[xmin xmax])` uses the specified *x*-domain instead of the default `[–2*pi, 2*pi]`.

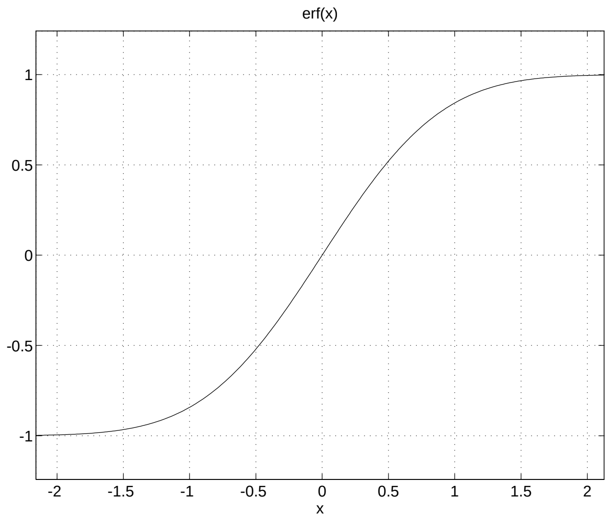
`ezplot(f,[xmin xmax],fig)` uses the specified Figure number instead of the current Figure. It also omits the title of the graph.

## Examples

Either of the following commands,

```
ezplot('erf(x)')
ezplot erf(x)
```

plot a graph of the error function::



**Algorithm**

ezplot determines the interval of the *x*-axis by sampling the function between -2*pi and 2*pi and then selecting a subinterval where the variation is significant. For the range of the *y*-axis, ezplot omits extreme values associated with singularities.

**See Also**

fplot

## Purpose

Plot velocity vectors

## Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
```

## Description

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the *x* components as relative coordinates, and `V` contains the *y* components as relative coordinates.

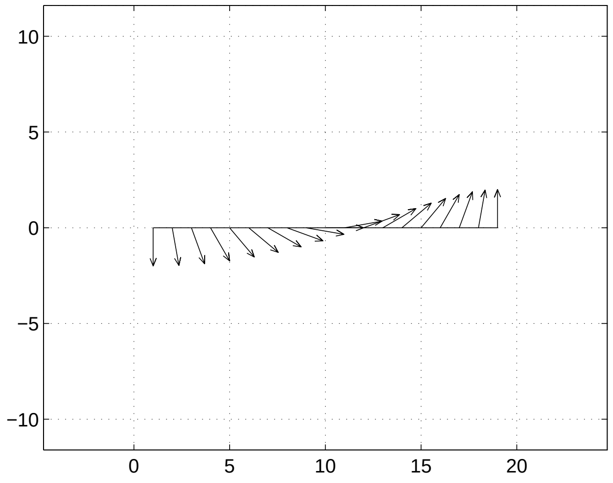
`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z),imag(Z))`.

`feather(...,`*`LineSpec`*`)` draws a feather plot using the line type, marker symbol, and color specified by *LineSpec*.

## Examples

Create a feather plot showing the direction of `theta`:

```
theta = (-90:10:90)*pi/180;
r = 2*ones(size(theta));
[u,v] = pol2cart(theta,r);
feather(u,v);
axis equal
```



**See Also** `compass`, `LineSpec`, `rose`

## Purpose

Test if Figure is on screen

## Syntax

```
[flag] = figflag('figurename')
[flag,fig] = figflag('figurename')
[...] = figflag('figurename',silent)
```

## Description

Use `figflag` to determine if a particular Figure exists, bring a Figure to the foreground, or set the window focus to a Figure.

`[flag] = figflag('`*`figurename`*`')` returns a `1` if the Figure named `'`*`figure-name`*`'` exists and pops the Figure to the foreground, otherwise this function returns `0`.

`[flag,fig] = figflag('`*`figurename`*`')` returns a `1` in `flag`, returns the Figure's handle in `fig`, and pops the Figure to the foreground, if the Figure named `'`*`figurename`*`'` exists. Otherwise this function returns `0`.

`[...] = figflag('`*`figurename`*`',silent)` pops the Figure window to the foreground if `silent` is `0`, and leaves the Figure in its current position if `silent` is `1`.

## Examples

To determine if a Figure window named `'Fluid Jet Simulation'` exists, type

```
[flag,fig] = figflag('Fluid Jet Simulation')
```

If two Figures with handles 1 and 3 have the name `'Fluid Jet Simulation'`, MATLAB returns:

```
flag =
   1

fig =
   1 3
```

## See Also

`figure`

# figure

## Purpose

Create a Figure graphics object

## Syntax

```
figure
figure('PropertyName',PropertyValue,...)
figure(h)
h = figure(...)
```

## Description

`figure` is the function for creating Figure graphics objects. Figure objects are the individual windows on the screen in which MATLAB displays graphical output.

`figure` creates a new Figure object using default property values.

`figure('`*`PropertyName`*`',PropertyValue,...)` creates a new Figure object using the values of the properties specified. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`figure(h)` does one of two things, depending on whether or not a Figure with handle `h` exists. If `h` is the handle to an existing Figure, `figure(h)` makes the Figure identified by `h` the current Figure, makes it visible, and raises it above all other Figures on the screen. The current Figure is the target for graphics output. If `h` is not the handle to an existing Figure, but is an integer, `figure(h)` creates a Figure, and assigns it the handle `h`. `figure(h)` where `h` is not the handle to a Figure, and is not an integer, is an error.

`h = figure(...)` returns the handle to the Figure object.

## Remarks

To create a Figure object, MATLAB creates a new window whose characteristics are controlled by default Figure properties (both factory installed and user defined) and properties specified as arguments. See the "Figure Properties" section for a description of these properties.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Use `set` to modify the properties of an existing Figure or `get` to query the current values of Figure properties.

The `gcf` command returns the handle to the current Figure.

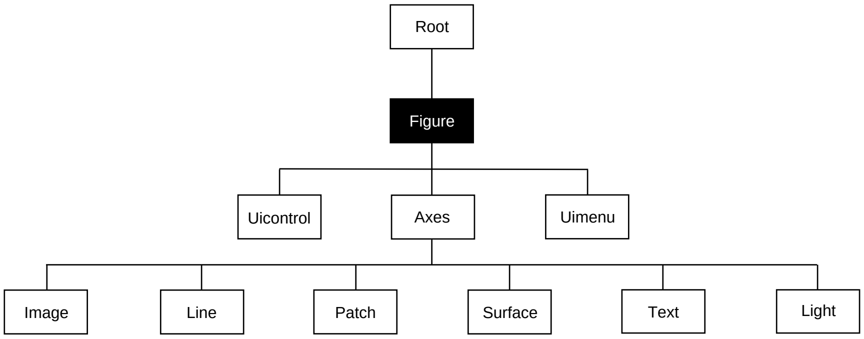
## Example

To create a Figure one quarter the size of your screen, positioned in the upper-left corner, use the Root object's `ScreenSize` property to determine the size:

```
scrsz = get(0,'ScreenSize');
figure('Position',[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2])
```

`ScreenSize` is a four-element vector: `[left, bottom, width, height]`.

## Object Hierarchy



### Setting Default Properties

You can set default Figure properties only on the Root level:

```
set(0,'DefaultFigureProperty',PropertyValue...)
```

Where *Property* is the name of the Figure property and `PropertyValue` is the value you are specifying.

## Figure Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`BackingStore`** `{on} | off`

*Off screen pixel buffer*. When `BackingStore` is on, MATLAB stores a copy of the Figure window in an off-screen pixel buffer. When obscured parts of the Figure window are exposed, MATLAB copies the window contents from this buffer rather than regenerating the objects on the screen. This increases the speed with which the screen is redrawn.

While refreshing the screen quickly is generally desirable, the buffers required do consume system memory. If memory limitations occur, you can set `BackingStore` to `off` to disable this feature and release the memory used by the buffers. If your computer does not support backingstore, setting the `BackingStore` property results in a warning message, but has no other effect.

Setting `BackingStore` to `off` can increase the speed of animations because it eliminates the need to draw into both an off-screen buffer and the Figure window.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

*Button press callback function*. A callback routine that executes whenever you press a mouse button while the pointer is in the Figure window, but not over descendent object (i.e., Uicontrol, Axes or Axes child). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`Children`** vector of handles

Children of the Figure. A vector containing the handles of all Axes, Uicontrol, and Uimenu objects displayed within the Figure. You can change the order of the handles and thereby change the stacking of the objects on the display.

**`Clipping`** `{on} | off`

This property has no effect on Figures.

**CloseRequestFcn** string

*Callback executed on Figure close*. This property defines a callback routine that MATLAB executes whenever you issue the `close` command (either a `close(fig_handle)` or a `close all`) or close a Figure window from the computer's window manager menu. This provides an opportunity for the Figure to, for example, display a `yes/no/cancel` dialog box before closing, to abort the deletion of the Figure, or to perform "clean up" before closing. The `delete` command unconditionally closes the Figure. The default callback is `closereq`, which uses:

```
delete(get(0,'CurrentFigure'))
```

**Color** ColorSpec

*Background color*. This property controls the Figure window background color. You can specify a color using a three-element vector of RGB values or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information.

**Colormap** m-by-3 matrix of RGB values

*Figure colormap*. This property is an m-by-3 array of red, green, and blue (RGB) intensity values that define m individual colors. MATLAB accesses colors by their row number. For example, an index of 1 specifies the first RGB triplet, an index of 2 specifies the second RGB triplet, and so on. Colormaps can be any length (up to 256 only on MS-Windows and Macintosh), but must be three columns wide. The default Figure colormap contains 64 predefined colors.

Colormaps affect the rendering of Surface, Image, and Patch objects, but generally do not affect other graphics objects. See the `colormap` and `ColorSpec` reference pages for more information.

CreateFcn string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Figure object. You must define this property as a default value for Figures. For example, the statement,

```
set(0,'DefaultFigureCreateFcn','set(gcbo,''IntegerHandle'',''off
'')')
```

defines a default value on the Root level that causes the created Figure to use noninteger handles whenever you (or MATLAB) creates Figure. MATLAB executes this routine after setting all properties for the Figure. Setting this property on an existing Figure object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`CurrentAxes`** handle of current Axes

*Target Axes in this Figure.* MATLAB sets this property to the handle of the Figure's current Axes (i.e., the handle returned by the `gca` function when this Figure is the current Figure). In all Figures for which Axes children exist, there is always a current Axes. The current Axes does not have to be the topmost axes, and setting an Axes to be the `CurrentAxes` does not restack it above all other Axes.

You can make an Axes current using the `axes` and `set` commands. For example, `axes(`*axes_handle*`)` and `set(gcf,'CurrentAxes',`*axes_handle*`)` both make the Axes identified by the handle *axes_handle* the current Axes. However, `axes(`*axes_handle*`)` also restacks the Axes above all other Axes in the Figure.

If a Figure contains no Axes, `get(gcf,'CurrentAxes')` returns the empty matrix. Note that the `gca` function actual creates an Axes if one does not exist.

**`CurrentCharacter`** single character (read only)

*Last key pressed.* MATLAB sets this property to the last key pressed in the Figure window. `CurrentCharacter` is useful for obtaining user input.

**`CurrentMenu`** (Obsolete)

This property produces a warning message when queried. It has been superseded by the Root `CallbackObject` property.

**`CurrentObject`** object handle

*Handle of current object.* MATLAB sets this property to the handle of the object that is under the current point (see the `CurrentPoint` property). This object is the front-most object in the stacking order. You can use this property to determine which object a user has selected. The function gco provides a convenient way to retrieve the `CurrentObject` of the `CurrentFigure`.

**`CurrentPoint`** two-element vector: `[x-coordinate, y-coordinate]`

*Location of last button click in this Figure.* MATLAB sets this property to the location of the pointer at the time of the most recent mouse button press.

MATLAB updates this property whenever you press the mouse button while the pointer is in a Figure window.

The `CurrentPoint` is measured from the lower-left corner of the Figure window, in units determined by the `Units` property.

**`DeleteFcn`** string

*Delete Figure callback routine*. A callback routine that executes when the Figure object is deleted (e.g., when you issue a `delete` or a `close` command). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`Dithermap`** m-by-3 matrix of RGB values

*Colormap used for true-color data on pseudocolor displays*. This property defines a colormap that MATLAB uses to dither true-color `CData` for display on pseudocolor (8-bit or less) displays. MATLAB maps each RGB color defined as true-color `CData` to the closest color in the dithermap. The default `Dithermap` contains colors that span the full spectrum so any color values map reasonably well.

However, if the true-color data contains wide range of shades in one color, you may achieve better results by defining your own dithermap. See the `Dither-mapMode` property.

**`DithermapMode`** `auto | {manual}`

*MATLAB generated dithermap*. In `manual` mode, MATLAB uses the colormap defined in the `Dithermap` property to display direct color on pseudocolor displays. When `DithermapMode` is `auto`, MATLAB generates a dithermap based on the colors currently displayed. This is useful if the default dithermap does not produce satisfactory results.

The process of generating the dithermap can be quite time consuming and is repeated whenever MATLAB re-renders the display (e.g., when you add a new object or resize the window). You can avoid unnecessary regeneration by setting this property back to `manual` and save the generated dithermap (which MATLAB loaded into the `Dithermap` property).

**FixedColors** m-by-3 matrix of RGB values (read only)

Non-colormap colors. Fixed colors define all colors appearing in a Figure window that are not obtained from the Figure colormap. These colors include axis lines and labels, the color of Line, Text, Uicontrol, and Uimenu objects, and any colors that you explicitly define, for example, with a statement like:

```
set(gcf,'Color',[.3 .7 .9]).
```

Fixed color definitions reside in the system color table and do not appear in the Figure colormap. For this reason, fixed colors can limit the number of simultaneously displayed colors if the number of fixed colors plus the number of entries in the Figure colormap exceed your system's maximum number of colors.

(See the `ScreenDepth` property of the Root for information on determining the total number of colors supported on your system. See the `MinColorMap` and `ShareColors` properties for information on how MATLAB shares colors between applications.)

**HandleVisibility** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to `on` to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`IntegerHandle`** `{on} | off`

*Figure handle mode*. Figure object handles are integers by default. When creating a new Figure, MATLAB uses the lowest integer that is not used by an existing Figure. If you delete a Figure, its integer handle can be reused.

If you set this property to `off`, MATLAB assigns nonreusable real-number handles (e.g., 67.0001221) instead of integers. This feature is designed for dialog boxes where removing the handle from integer values reduces the likelihood of inadvertently drawing into the dialog box.

**`Interruptible`** `{on} | off`

*Callback routine interruption mode*. The `Interruptible` property controls whether a Figure callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn`, `KeyPressFcn`, `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `EventQueue` property for related information.

**`InvertHardcopy`** `{on} | off`

*Change hardcopy to black objects on white background*. This property affects only printed output. Printing a Figure having a background color (`Color` property) that is not white results in poor contrast between graphics objects and the Figure background and also consumes a lot of printer toner.

When `InvertHardCopy` is on, MATLAB eliminates this effect by changing the color of the Figure and Axes to white and the axis lines, tick marks, axis labels, etc., to black. Lines, Text, and the edges of Patches and Surfaces may be changed depending on the `print` command options specified.

If you set `InvertHardCopy` to `off` and specify the `–exact` option with the `print` command, the printed output matches the colors displayed on the screen (which may be dithered on black and white devices).

See the `print` reference page for more information on printing MATLAB Figures.

**`KeyPressFcn`** string

Key press callback function. A callback routine invoked by a key press occurring in the Figure window. You can define `KeyPressFcn` as any legal MATLAB expression or the name of an M-file.

The callback routine can query the Figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

The callback routine can also query the Root object's `PointerWindow` property to determine in which Figure the key was pressed. Note that pressing a key while the pointer is in a particular Figure window does not make that Figure the current Figure (i.e., the one referred by the `gcf` command).

**`MenuBar`** `none | {figure}`

*Enable-disable Figure menu bar*. This property allows you to display or hide the menu bar placed at the top of a Figure window. Note that not all systems support Figure window menu bars. However, for those that do, the default is to display the menu.

**`MinColormap`** scalar (default = 64)

*Minimum number of color table entries used*. This property specifies the minimum number of system color table entries used by MATLAB to store the colormap defined for the Figure (see the `ColorMap` property). In certain situations, you may need to increase this value to ensure proper use of colors.

For example, suppose you are running color-intensive applications in addition to MATLAB and have defined a large Figure colormap (e.g., 150 to 200 colors). MATLAB may select colors that are close, but not exact from the existing colors in the system color table because there aren't enough slots available to define all the colors you specified.

To ensure MATLAB uses exactly the colors you define in the Figure colormap, set `MinColorMap` equal to the length of the colormap:

```
set(gcf,'MinColormap',length(get(gcf,'ColorMap')))
```

Note that the larger the value of `MinColorMap`, the greater the likelihood other windows (including other MATLAB Figure windows) will display in false colors.

**`Name`** string

*Figure window title.* This property specifies the title displayed in the Figure window. By default `Name` is empty and the Figure title is displayed as `Figure No. 1`, `Figure No. 2`, and so on. When you set this parameter to a string, the Figure title becomes `Figure No. 1: <`*string*`>`. See the `NumberTitle` property.

**`NextPlot`** `{add} | replace | replacechildren`

*How to add next plot.* `NextPlot` determines which Figure MATLAB uses to display graphics output. If the value of the current Figure is:

- `add` — use the current Figure to display graphics (the default).
- `replace` — reset all Figure properties, except `Position`, to their defaults and delete all Figure children before displaying graphics (equivalent to `clf reset`).
- `replacechildren` — remove all child objects, but do not reset Figure properties (equivalent to `clf`).

The `newplot` function provides an easy way to handle the `NextPlot` property. Also see the `NextPlot` property of Axes.

**`NumberTitle`** `{on} | off`

*Figure window title number*. This property determines whether the string `Figure No. N` (where `N` is the Figure number) is prefixed to the Figure window title. See the `Name` property.

**`PaperOrientation`** `{portrait} | landscape`

*Horizontal or vertical paper orientation*. This property determines how printed Figures are oriented on the page. `portrait` orients the longest page dimension vertically; `landscape` orients the longest page dimension horizontally.

**PaperPosition** 4-element rect vector

*Location on printed page*. A rectangle that determines the location of the Figure on the printed page. Specify this rectangle with a vector of the form

```
    rect = [left, bottom, width, height]
```

where `left` specifies the distance from the left side of the paper to the left side of the rectangle and `bottom` specifies the distance from the bottom of the page to the bottom of the rectangle. Together these distances define the lower-left corner of the rectangle. `width` and `height` define the dimensions of the rectangle. The `PaperUnits` property specifies the units used to define this rectangle.

**PaperPositionMode** `auto | {manual}`

*WYSIWYG printing of Figure*. In `manual` mode, MATLAB honors the value specified by the `PaperPosition` property. In `auto` mode, MATLAB prints the Figure the same size as it appears on the computer screen, centered on the page.

**PaperSize** `[width height]` (read only)

*Paper size*. This property contains the size of the current `PaperType`, measured in `PaperUnits`.

**PaperType** `{usletter} | uslegal | a3 | a4letter | a5 b4 | tabloid`

*Selection of standard paper size*. This property sets the `PaperSize` to the one of seven standard sizes. In inches, these sizes are:

- `usletter`: width = 8.5, height = 11 inches
- `uslegal`: width = 11, height = 14 inches
- `a3`: width = 297, height = 420 mm
- `a4letter`: width = 210, height = 297 mm
- `a5`: width = 148, height = 210 mm
- `b4`: width = 250, height = 354 mm
- tabloid: width = 11, height = 17 inches (also called “C” size)

**PaperUnits** `normalized | {inches} | centimeters | points`

*Hardcopy measurement units*. This property specifies the units used to define the `PaperPosition` and `PaperSize` properties. All units are measured from the lower-left corner of the page. `normalized` units map the lower-left corner of the page to (0,0) and the upper-right corner to (1.0,1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).

If you change the value of `PaperUnits`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `PaperUnits` is set to the default value.

**`Parent`** handle

*Handle of Figure's parent*. The parent of a Figure object is the Root object. The handle to the Root is always 0.

**`Pointer`**

```
crosshair | {arrow} | watch | topl | topr
| botl | botr | circle | cross | fleur
| left | right | top | bottom | fullcrosshair
| Ibeam | custom
```

*Pointer symbol selection*. This property determines the symbol used to indicate the pointer (cursor) position in the Figure window.

Setting `Pointer` to `custom` allows you to define your own pointer symbol. See the `PointerShapeCData` property for more information.

**`PointerShapeCData`** 16-by-16 matrix

*User-defined pointer*. This property defines the pointer that is used when you set the `Pointer` property to `custom`. It is a 16-by-16 element matrix defining the 16-by-16 pixel pointer using the following values:

- `1` – color pixel black
- `2` – color pixel white
- `NaN` – make pixel transparent (underlying screen shows through)

Element (1,1) of the `PointerShapeCData` matrix corresponds to the upper-left corner of the pointer. Setting the `Pointer` property to one of the predefined pointer symbols does not change the value of the `PointerShapeCData`. Computer systems supporting 32-by-32 pixel pointers fill only one quarter of the available pixmap.

**`PointerShapeHotSpot`** 2-element vector

*Pointer active area*. A two-element vector specifying the row and column indices in the `PointerShapeCData` matrix defining the pixel indicating the pointer location. The location is contained in the `CurrentPoint` property and the Root object's `PointerLocation` property. The default value is element (1,1), which is the upper-left corner.

**`Position`** 4-element vector

*Figure position*. This property specifies the size and location on the screen of the Figure window. Specify the position rectangle with a 4-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower-left corner of the screen to the lower-left corner of the Figure window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

You can use the `get` function to obtain this property and determine the position of the Figure and you can use the `set` function to resize and move the Figure to a new location.

**`Renderer`** `painters | zbuffer`

*Rendering method used for screen and printing*. This property enables you to select the method used to render MATLAB graphics. The choices are:

- `painters` – MATLAB's original rendering method is faster when the Figure contains only simple or small graphics objects.
- `zbuffer` – MATLAB draws graphics object faster and more accurately because objects are colored on a per pixel basis and MATLAB renders only those pixels that are visible in the scene (thus eliminating front-to-back sorting errors). Note that this method can consume a lot of system memory if MATLAB is displaying a complex scene.

**`RendererMode`** `{auto} | manual`

*Automatic, or user selection of Renderer*. This property enables you to specify whether MATLAB should choose the `Renderer` based on the contents of the figure window, or whether the `Renderer` should remain unchanged.

When the `RendererMode` property is set to `auto`, MATLAB selects the rendering method for printing as well as for the screen based on the size and complexity of the graphics objects in the Figure. For printing, MATLAB switches to `zbuffer` at a greater scene complexity than for screen rendering because printing from a Z-buffered Figure can be considerably slower than one using the painters rendering method, and can result in large PostScript files.

When the RendererMode property is set to manual, MATLAB does not change the Renderer, regardless of changes to the Figure contents.

**Resize** {on} | off

*Window resize mode*. This property determines if you can resize the Figure window with the mouse. on means you can resize the window, off means you cannot. When Resize is off, the Figure window doesn't display any resizing controls (such as boxes at the corners) to indicate the absence of resizeability.

**ResizeFcn** string

*Window resize callback routine*. MATLAB executes the specified callback routine whenever you resize the Figure window. The Figure's Position property can be queried to determine the new size and position of the Figure window. The handle to the Figure being resized is only accessible through the Root CallbackObject property, which can be queried using gcbo.

ResizeFcn can be used to maintain a GUI layout that is not directly supported by MATLAB's Position/Units paradigm, such as keeping an object a constant height in pixels, and attached to the top of the Figure, but always matching the width of the Figure. For example, the following ResizeFcn will keep a Uicontrol whose Tag is 'StatusBar' 20 pixels high, as wide as the Figure, and attached to the top of the Figure. Note the use of the Tag property to retrieve the Uicontrol handle, and the gcbo function to retrieve the Figure handle. Also note the defensive programming regarding Figure Units, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards:

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

The Figure Position may be changed from within the ResizeFcn callback, however the ResizeFcn will not be called again as a result.

**`Selected`**  `on | off`

*Is object selected*. This property indicates whether the Figure is selected. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**`SelectionHighlight`**  `{on} | off`

Figures do not indicate selection.

**`SelectionType`**  `{normal} | extended | alt | open` (read only)

Mouse selection type. MATLAB maintains this property to provide information about the last mouse button press that occurred within the Figure window. This information indicates the type of selection made. Selection types are particular actions that are generally associated with particular responses from the user interface software (e.g., single clicking on a graphics object places it in move or resize mode; double-clicking on a filename opens it, etc.).

The physical action required to make these selections varies on different platforms. However, all selection types exist on all platforms.

| Selection Type | MS-Windows | X-Windows | Macintosh |
|---|---|---|---|
| Normal | Click left mouse button | Click left mouse button | Click mouse button |
| Extended | **Shift** - click left mouse button or both left and right mouse buttons | **Shift** - click left mouse button or middle mouse button | **Shift** - click mouse button |
| Alternate | **Control** - click left mouse button or right mouse button | **Control** - click left mouse button or right mouse button | **Option** - click mouse button |
| Open | Double click any mouse button | Double click any mouse button | Double click mouse button |

Note that the `ListBox` style of Uicontrols set the Figure `SelectionType` property to `normal` to indicate a single mouse click or to `open` to indicate a double mouse click.

**`ShareColors`** `{on} | off`

*Share slots in system colortable with like colors*. This property affects the way MATLAB stores the Figure colormap in the system color table. By default, MATLAB looks at colors already defined and uses those slots to assign pixel colors. This leads to an efficient use of color resources (which are limited on systems capable of displaying 256 or less colors) and extends the number of Figure windows that can simultaneously display correct colors.

However, in situations where you want to change the Figure colormap quickly without causing MATLAB to re-render the displayed graphics objects, you should disable color sharing (set `ShareColors` to `off`). In this case, MATLAB can swap one colormap for another without changing pixel color assignments since all the slots in the system color table used for the first colormap are replaced with the corresponding color in the second colormap. (Note that is applies only in cases where both colormaps are the same length and where the computer hardware allows user modification of the system color table.)

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you want to direct all graphics output from an M-file to a particular Figure, regardless of user actions that may have changed the current Figure. To do this, identify the Figure with a `Tag`:

```
figure('Tag','Plotting Figure')
```

Then make that Figure the current Figure before drawing by searching for the `Tag` with `findobj`:

```
figure(findobj('Tag','Plotting Figure'))
```

**Type** string (read only)

*Object class*. This property identifies the kind of graphics object. For Figure objects, `Type` is always the string "`figure`".

**Units** `{pixels} | normal | inches | centimeters | points`

*Units of measurement*. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower-left corner of the window. Normalized (`normal`) units map the lower-left corner of the Figure window to (0,0) and the upper-right corner to (1.0,1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch). The size of a `pixel` depends on screen resolution.

This property affects the `CurrentPoint` and `Position` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

When specifying the units as property/value pairs during object creation, you must set the `Units` property before specifying the properties that you want to use these units.

**UserData** matrix

User specified data. You can specify `UserData` as any matrix you want to associate with the Figure object. The object does not use this data, but you can access it using the `set` and `get` commands.

**Visible** `{on} | off`

*Object visibility*. The `Visible` property determines whether an object is displayed on the screen. If the `Visible` property of a Figure is `off`, the entire Figure window is invisible.

**WindowButtonDownFcn** string

*Button press callback function*. Use this property to define a callback routine that MATLAB executes whenever you press a mouse button while the pointer is in the Figure window. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`WindowButtonMotionFcn`** string

*Mouse motion callback function*. Use this property to define a callback routine that MATLAB executes whenever you move the pointer within the Figure window. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`WindowButtonUpFcn`** string

*Button release callback function*. Use this property to define a callback routine that MATLAB executes whenever you release a mouse button. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

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

If the callback routines defined by `WindowButtonDownFcn` or `WindowButtonMotionFcn` contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is set to `off`, the `WindowButtonUpFcn` may not be called. You can prevent this problem by setting `Interruptible` to `on`.

**`WindowStyle`** `{normal} | modal`

*Normal or modal window behavior*. When `WindowStyle` is set to `modal`, the Figure window traps all keyboard and mouse events over all MATLAB windows as long as it is visible. Windows belonging to other applications other than MATLAB are unaffected. Modal Figures remain stacked above all normal Figures and the MATLAB command window. When multiple modal windows exist, the most recently created window will keep focus and stay above all other windows until it becomes invisible, or is returned to `WindowStyle normal`, or is deleted. At that time, focus reverts to the window which last had the focus.

Figures with `WindowStyle modal` and `Visible off` do not behave modally until they are made visible, so it is acceptable to hide a `modal` window instead of destroying it, for efficiency, when a dialog that is being dismissed may be used again.

The WindowStyle of a Figure may be changed at any time, including when the figure is visible, and contains children. However, on some systems this may cause the figure to flash, or even to disappear and reappear, depending on the windowing-system's implementation of normal and modal windows. For best visual results, WindowStyle should be set at creation time, or when the figure is invisible.

`modal` Figures do not display Uimenu children or built-in menus, but it is not an error to create Uimenus in a `modal` Figure, or to change `WindowStyle` to `modal` on a Figure with Uimenu children. The Uimenu objects exist, and their handles are retained by the Figure. If the Figure's `WindowStyle` is returned to `normal`, the Uimenus will again be displayed.

Modal Figures are used to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Control** C at the MATLAB prompt causes all Figures with `WindowStyle` `modal` to revert to `WindowStyle` `normal`, to allow typing at the command line.

## See Also

`axes`, `uicontrol`, `uimenu`, `close`, `clf`, `gcf`, `root`

## Purpose

Filled two-dimensional polygons

## Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

## Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,`*`ColorSpec`*`)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by *ColorSpec*.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'`*`PropertyName`*`',PropertyValue)` allows you to specify property names and values for a Patch graphics object.

`h = fill(...)` returns a vector of handles to Patch graphics objects, one handle per Patch object.

## Algorithm

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

The type of color shading depends on how you specify color in the argument list. If you specify color using *ColorSpec*, `fill` generates flat-shaded polygons by setting the Patch object's `FaceColor` property to the corresponding RGB triple.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the Axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each Patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`-th Patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the Patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective Patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

## Examples

Create a red octagon:

```
t = (1/16:1/8:1)'*2*pi;
x = sin(t);
y = cos(t);
fill(x,y,'r')
axis square
```

## See Also

`axis`, `caxis`, `colormap`, `ColorSpec`, `fill3`, `patch`

**Purpose** Filled three-dimensional polygons

**Syntax**

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

**Description** The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates *n* polygons, where *n* is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,`*`ColorSpec`*`)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by *ColorSpec*.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'`*`PropertyName`*`',PropertyValue)` allows you to set values for specific Patch properties.

`h = fill3(...)` returns a vector of handles to Patch graphics objects, one handle per Patch.

**Algorithm** If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If `X`, `Y`, or `Z` is a matrix, `fill3` replicates any column vector argument to produce matrices of the required size.

If you specify color using `ColorSpec`, `fill3` generates flat-shaded polygons and sets the Patch object `FaceColor` property to an RGB triple.

If you specify color using C, fill3 scales the elements of C by the Axes property CLim, which specifies the color axis scaling parameters, before indexing the current colormap.

If C is a row vector, fill3 generates flat-shaded polygons and sets the FaceColor property of the Patch objects to 'flat'. Each element becomes the CData property value for the respective Patch object.

If C is a column vector or a matrix, fill3 generates polygons with interpolated colors and sets the patch object FaceColor property to 'interp'. fill3 uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the CData property value for the respective Patch object. If C is a column vector, fill3 replicates the column vector to produce the required sized matrix.

## Examples

Create four triangles with interpolated colors.

```
colormap(cool)
X = rand(3,4); Y = rand(3,4); Z = rand(3,4)
C = rand(3,4);
fill3(X,Y,Z,C)
```

## See Also

axis, caxis, colormap, ColorSpec, fill, patch

**Purpose** Locate graphics objects

**Syntax**

```
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h = findobj(objhandles,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)
```

**Description** `findobj` locates graphics objects and returns their handles. You can limit the search to objects with particular property values and along specific branches of the hierarchy.

`h = findobj` returns the handles of the Root object and all its descendents.

`h = findobj('`*`PropertyName`*`',PropertyValue,...)` returns the handles of all graphics objects having the property *PropertyName*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendents.

`h = findobj(objhandles,'flat','`*`PropertyName`*`',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendents.

**Remarks** `findobj` returns an error if a handle refers to a non-existent graphics object.

When you specify a property value, use the same format as `get` returns. For example, you must use the RGB format to specify a color value and when the value is a string, you must specify the entire character string.

When a graphics object is a descendent of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in its handle being returned multiple times.

**Examples** Find all Line objects in the current Axes:

```
h = findobj(gca,'Type','line')
```

**See Also**

copyobj, gcf, gca, gco, get, set

axes, figure, image, light, line, patch, surface, text, uicontrol, uimenu

**Purpose** Plot a function between specified limits

**Syntax**

```
fplot('function',limits)
fplot('function',limits,LineSpec)
fplot('function',limits,tol)
fplot('function',limits,tol,LineSpec)
[x,Y] = fplot(...)
```

**Description** `fplot` plots a function between specified limits. The function must be of the form
`y = f(x)`, where `x` is a vector whose range specifies the limits, and `y` is a vector the same size as `x` and contains the function's value at the points in `x` (see the first example). If the function returns more than one value for a given `x`, then `y` is a matrix whose columns contain each component of `f(x)` (see the second example).

`fplot('`*`function`*`',limits)` plots `'`*`function`*`'` between the limits specified by `limits`. `limits` is a vector specifying the *x*-axis limits (`[xmin xmax]`), or the *x*- and *y*-axis limits, (`[xmin xmax ymin ymax]`).

`fplot('`*`function`*`',limits,`*`LineSpec`*`)` plots `'`*`function`*`'` using the line specification `LineSpec`. `'`*`function`*`'` is a name of a MATLAB M-file or a string containing the variable `x`.

`fplot('`*`function`*`',limits,tol)` plots `'`*`function`*`'` using the relative error tolerance `tol` (default is 2e–3). The maximum number of x steps is `(1/tol)+1`.

`fplot('`*`function`*`',limits,tol,`*`LineSpec`*`)` plots `'`*`function`*`'` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color.

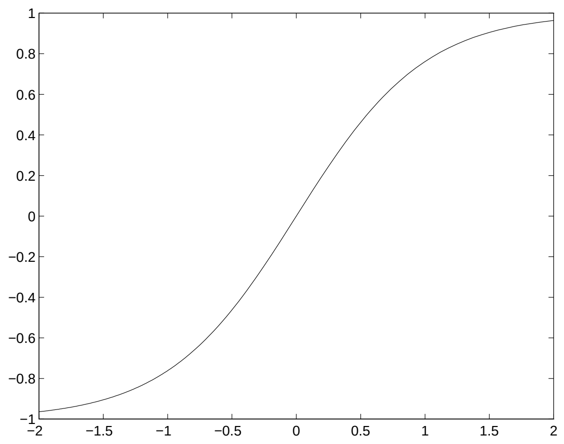
`[x,Y] = fplot(...)` returns the abscissas and ordinates for `'`*`function`*`'` in `x` and `Y`. No plot is drawn on the screen. You plot the function using `plot(x,Y)`.

**Remarks** `fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

## Examples

Plot the hyperbolic tangent function from -2 to 2:

```
fplot('tanh',[-2 2])
```

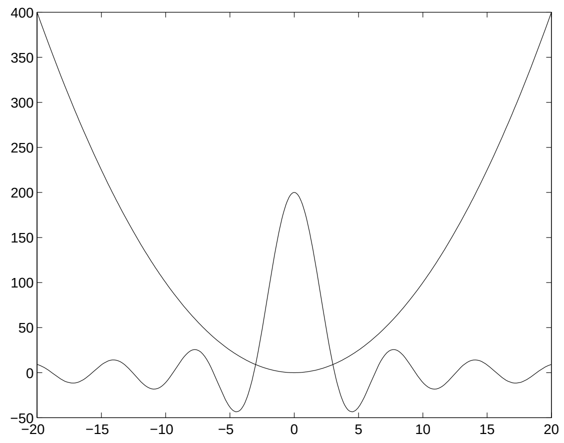


Create an M-file, myfun, that returns a two column matrix:

```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./
x(:);
Y(:,2) = x(:).^2;
```

Plot the function with the statement:

```
fplot('myfun',[–20 20]
```



## See Also

`LineSpec`, `plot`

The `eval` and `feval` functions in the online MATLAB Function Reference.

## frame2im

**Purpose** Convert movie frame to indexed image

**Syntax** `[X,Map] = frame2im(F)`

**Description** `[X,Map] = frame2im(F)` converts the single movie frame `F` into the indexed image `X` and associated colormap `Map`. A single column of a movie matrix is one movie frame. The functions `getframe` and `im2frame` create a movie frame.

**See Also** `capture`, `im2frame`, `movie`

**Purpose** Get current Axes handle

**Syntax**

```
h = gca
```

**Description** `h = gca` returns the handle to the current Axes for the current Figure. If no Axes exists, MATLAB creates one and returns its handle. You can use the statement,

```
get(gcf,'CurrentAxes')
```

if you do not what MATLAB to create an Axes if one does not alread exist.

The current Axes is the target for graphics output when you create Axes children. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current Axes. Changing the current Figure also changes the current Axes.

**See Also** `axes`, `cla`, `delete`, `gcf`, `gcbo`, `gco`, `hold`, `subplot`, `findobj`

Figure `CurrentAxes` property

**Purpose** Return the handle of the object whose callback is currently executing

**Syntax**

```
h = gcbo
[h, figure] = gcbo
```

**Description**

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h, figure] = gcbo` returns the handle of the current callback object and the handle of the Figure containing this object.

**Remarks**

MATLAB stores the handle of the object whose callback is executing in the Root's `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The Root `CallbackObject` property is read-only, so its value is always valid at any time during callback execution. The Root `CurrentFigure` property, and the Figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco` respectively) are user settable, so they can change during the execution of a callback, especially if that callback is inter rupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

`gcbo` must be used in conjunction with `CreateFcn` and `DeleteFcn` callbacks, and with the Figure `ResizeFcn` callback, since those callbacks do not update the Root's `CurrentFigure` property, or the Figure's `CurrentObject` or `CurrentAxis` properties, but only update the Root's `CallbackObject` property.

When no callbacks are executing, `gcbo` returns `[]`.

**See Also** `gca`, `gcf`, `gco`, `root`

# gcf

**Purpose** Get current Figure handle

**Syntax**

```
h = gcf
```

**Description** `h = gcf` returns the handle of the current Figure. The current Figure is the Figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. If no Figure exists, MATLAB creates one and returns its handle. You can use the statement,

```
get(0,'CurrentFigure')
```

if you do not what MATLAB to create a Figure if one does not alread exist.

**See Also** `axes`, `clf`, `close`, `delete`, `figure`, `gca`, `gcbo`, `gco`, `subplot`

Root `CurrentFigure` property

## Purpose

Return handle of current object

## Syntax

```
h = gco
h = gco(h)
```

## Description

`h = gco` returns the handle of the last graphics object you clicked on with the mouse or the last graphics object created.

`h = gco(h)` returns the value of the current object for the Figure specified by `h`.

## Remarks

MATLAB stores the handle of the current object in the Figure's `CurrentObject` property.

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback` intentionally do not update `CurrentFigure` or `CurrentObject`. `gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the callback function, regardless of the type of callback or of any previous interruptions.

## Examples

Return the handle to the current graphics object in Figure 2:

```
h = gco(2)
```

## See Also

`gca`, `gcbo`, `gcf`, `root`

**Purpose** Get object properties

**Syntax**

```
get(h)
get(h,'PropertyName')
<m-by-n value cell array> = get(H,<property cell array>)
a = get(h)
a = get(0,'Factory')
a = get(0,'FactoryObjectTypePropertyName')
a = get(h,'Default')
a = get(h,'DefaultObjectTypePropertyName')
```

**Description**

`get(h)` returns all properties and their current values of the graphics object identified by the handle `h`.

`get(h,'`*`PropertyName`*`')` returns the value of the property `'`*`PropertyName`*`'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns *n* property values for *m* graphics objects in the *m*-by-*n* cell array, where `m = length(H)` and *n* is equal to the number of property names contained in `pn`.

`a = get(h)` returns a structure whose field names are the object's property names and whose values are the current values of the corresponding properties. `h` must be a scalar. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory')` returns the factory-defined values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory`*`ObjectTypePropertyName`*`')` returns the factory-defined value of the named property for the specified object type. The argument, *FactoryObjectTypePropertyName* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

`a = get(h,'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h,'Default`*`ObjectTypePropertyName`*`')` returns the factory-defined value of the named property for the specified object type. The argument, *DefaultObjectTypePropertyName* is the word `Default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

## Examples

You can obtain the default value of the `LineWidth` property for Line graphics objects defined on the Root level with the statement:

```
get(0,'DefaultLineLineWidth')

ans =
    0.5000
```

To query a set of properties on all Axes children define a cell array of property names:

```
props = {'HandleVisibility', 'Interruptible';
       'SelectionHighlight', 'Type'};
output = get(get(gca,'Children'),props);
```

The variable output is a cell array of dimension: `length(get(gca,'Children')` by 4.

For example, type:

```
patch;surface;text;line
output = get(get(gca,'Children'),props)
output =

   'off'    'on'    'off'    'line'
   'off'    'on'    'off'    'text'
   'off'    'on'    'off'    'surface'
   'off'    'on'    'off'    'patch'
```

## See Also

`findobj`, `gca`, `gcf`, `gco`, `set`

**Purpose**

Get movie frame

**Synopsis**

```
M = getframe
M = getframe(h)
M = getframe(h,rect)
[X,Map] = getframe(...)
```

**Description**

`getframe` returns a column vector containing one movie frame. The frame is a snapshot (pixmap) of the current Axes.

`M = getframe` gets a frame from the current Axes.

`M = getframe(h)` gets a frame from the Figure or Axes graphics object identified by `h`.

`M = getframe(h,rect)` specifies a rectangular area from which to copy the pixmap. `rect` is relative to the lower-left corner of the Figure or Axes graphics object identified by `h`, in pixel units. `rect` is a four-element vector in the form `[left bottom width height]`, where `width` and `height` define the dimensions of the rectangle.

`[X,Map] = getframe(...)` returns the frame as an indexed image matrix `X` and a colormap `Map`. In this case, `h` is a handle to a Figure or Axes object. You display the image matrix using `image` or `imagesc`.

**Remarks**

Usually, `getframe` is put in a `for` loop to assemble movie matrix `M` for playback using `movie`. To prevent excessive memory use, use `moviein` to allocate movie matrix `M` before building the movie. This generates an appropriate size matrix of zeros.

## Examples

Make the peaks function vibrate:

```
Z = peaks; surf(Z)
axis manual    % Freeze Axes limits
set(gca,'nextplot','replacechildren');
M = moviein(20);
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    M(:,j) = getframe;
end
movie(M,20) % Play the movie twenty times
```

## See Also

movie, moviein

## ginput

**Purpose**

Input data using the mouse

**Syntax**

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

**Description**

`ginput` allows you to select points from the Figure using the mouse or arrow keys for cursor positioning. The Figure must have focus before `ginput` receives input.

`[x,y] = ginput(n)` allows you to select `n` points from the current Axes and returns the *x*- and *y*-coordinates in the column vectors `x` and `y`, respectively. You can press the **Return** key to terminate the input before entering `n` points.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

`[x,y,button] = ginput(...)` returns the *x*-coordinates, the *y*-coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

**Examples**

Pick 10 two-dimensional points from the Figure window.

```
[x,y] = ginput(10)
```

Position the cursor with the mouse (or the arrow keys on terminals without a mouse, such as Tektronix emulators). Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 10 points, press the **Return** key.

**See Also**

`gtext`

## Purpose

Plot set of nodes using an adjacency matrix

## Synopsis

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineSpec)
```

## Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the *n*-by-*n* adjacency matrix `A`, where *n* is the number of nodes. `Coordinates` is an *n*-by-2 or an *n*-by-3 matrix, where *n* is the number of nodes and each coordinate pair or triple represents one node.

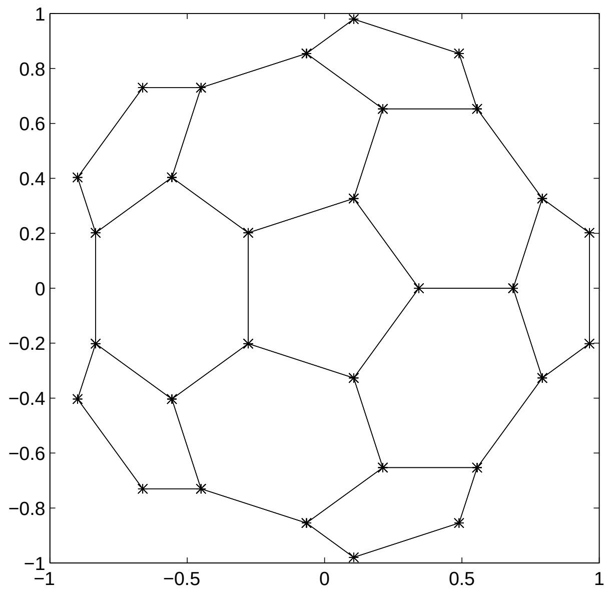
`gplot(A,Coordinates,`*`LineSpec`*`)` plots the nodes using the line type, marker symbol, and color specified by *LineSpec*.

## Remarks

For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node `i`, and `Coordinates(j,:) = [x(j) y(j)]` denotes node `j`. If node `i` and node `j` are joined, `A(i,j)` or `A(j,i)` are nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

**Examples** To draw half of a Bucky ball with asterisks at each node:

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



**See Also** `LineSpec`, `spy`

The `sparse` function in the online MATLAB Function Reference.

## graymon

**Purpose** Set default Figure properties for grayscale monitors

**Syntax**

```
graymon
```

**Description** `graymon` sets defaults for graphics properties to produce more legible displays for gray-scale monitors.

**See Also** `axes`, `figure`

# grid

**Purpose** Grid lines for two- and three-dimensional plots

**Syntax**

```
grid on
grid off
grid
```

**Description** The `grid` function turns the current Axes' grid lines on and off.

`grid on` adds grid lines to the current Axes.

`grid off` removes grid lines from the current Axes.

`grid` toggles the grid visibility state.

**Algorithm** `grid` sets the `XGrid`, `YGrid`, and `ZGrid` properties of the current Axes.

**See Also** `axes`, `plot`

The `XGrid`, `YGrid`, and `ZGrid` properties of Axes objects.

## gtext

**Purpose**

Mouse placement of text in two-dimensional view

**Syntax**

```
gtext('string')
h = gtext('string')
```

**Description**

`gtext` displays a text string in the current Figure window after you select a location with the mouse.

`gtext('`*`string`*`')` waits for you to press a mouse button or keyboard key while the pointer is within a Figure window. Pressing a mouse button or any key places `'`*`string`*`'` on the plot at the selected location.

`h = gtext('`*`string`*`')` returns a handle to a Text graphics objects after you place `'`*`string`*`'` on the plot at the selected location.

**Remarks**

As you move the pointer into a Figure window, the pointer becomes a crosshair to indicate that `gtext` is waiting for you to select a location.

**Algorithm**

`gtext` uses the functions `ginput` and `text`.

**Examples**

Place a label on the current plot:

```
gtext('Note this divergence!')
```

**See Also**

`ginput`, `text`

# helpdlg

**Purpose** Create a help dialog box

**Syntax**

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

**Description** `helpdlg` creates a help dialog box or brings the named help dialog box to the front.

`helpdlg` displays a dialog box named `'Help Dialog'` containing the string `'This is the default help string.'`

`helpdlg('`*`helpstring`*`')` displays a dialog box named '`Help Dialog`' containing the string specified by `'`*`helpstring`*`'`.

`helpdlg('`*`helpstring`*`','`*`dlgname`*`')` displays a dialog box named `'`*`dlgname`*`'` containing the string `'`*`helpstring`*`'`.

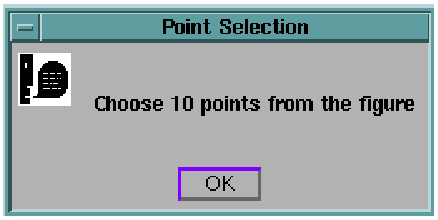
`h = helpdlg(...)` returns the handle of the dialog box.

**Remarks** MATLAB wraps the text in `'`*`helpstring`*`'` to fit the width of the dialog box. The dialog box remains on your screen until you press the OK button or the **Return** key. After pressing the button, the help dialog box disappears.

**Examples** The statement,

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays the following dialog box:



Choose 10 points from the figure

**See Also** `dialog`, `errordlg`, `questdlg`, `warndlg`

## Purpose

Remove hidden lines from a mesh plot

## Syntax

```
hidden on
hidden off
hidden
```

## Description

Hidden line removal draws only those lines that are not obscured by other objects in the field of view.

`hidden on` turns on hidden line removal for the current graph so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current graph.

`hidden` toggles the hidden line removal state.

## Algorithm

`hidden on` sets the `FaceColor` property of a Surface graphics object to `BackgroundColor`, which is usually `black`. `hidden off` sets the `FaceColor` property to `none`.

## Examples

Set hidden line removal off and on while displaying the peaks function:

```
mesh(peaks)
hidden off
hidden on
```

## See Also

`shading`

The Surface properties `FaceColor` and `EdgeColor`.

## Purpose

Histogram plot

## Syntax

```
hist(Y)
hist(Y,x)
hist(Y,nbins)
[n,xout] = hist(...)
```

## Description

A histogram shows the distribution of data values.

`hist(Y)` draws a histogram of the elements in `Y`. `hist` distributes the bins along the *x*-axis between the minimum and maximum values of `Y`.

`hist(Y,x)` draws a histogram using *n* bins, where *n* is `length(x)`. `x` also specifies the locations on the *x*-axis where `hist` places the bins. For example, if `x` is a 5-element vector, `hist` distributes the elements of `Y` into five bins centered on the *x*-axis at the elements in `x`.

`hist(Y,nbins)` draws a histogram with no more bins than `nbins`.

`[n,xout] = hist(...)` returns vectors `n` and `xout` containing the frequency counts and the bin locations. This syntax does not generate a plot. This is useful when you need more control over the appearance of a graph, for example, to combine a histogram into a more elaborate plot. You can use `bar(xout,n)` to plot the histogram.

## Remarks

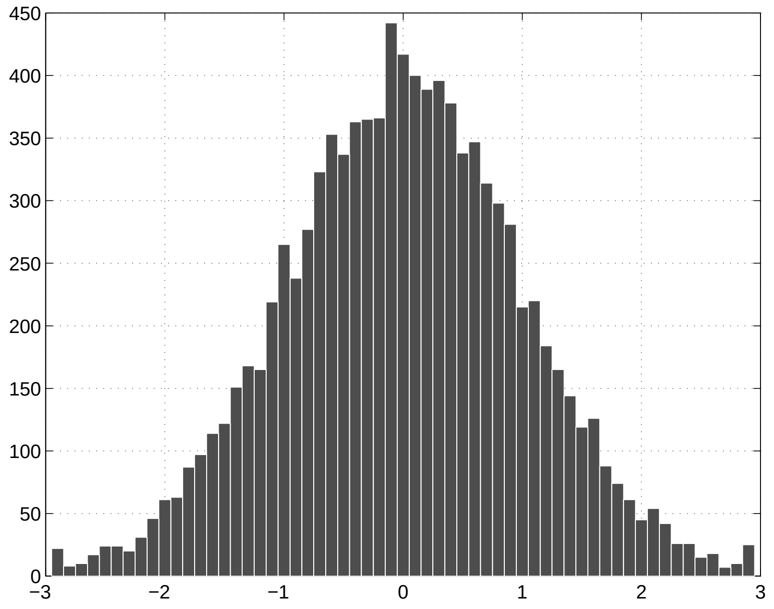
All elements in vector `Y` or in one column of matrix `Y` are grouped according to their numeric range. Each group is shown as one bin.

The histogram's *x*-axis reflects the range of values in `Y`. The histogram's *y*-axis shows the number of elements that fall within the groups; therefore, the *y*-axis ranges from `0` to the greatest number of elements deposited in any bin.

## Examples

Generate a bell-curve histogram from Gaussian data.

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
hist(y,x)
```



## See Also

`bar`, `stairs`

# hold

**Purpose**

Hold current graph in the Figure

**Syntax**

```
hold on
hold off
hold
```

**Description**

The `hold` function determines whether new graphics objects are added to the graph or replace objects in the graph.

`hold on` retains the current plot and certain Axes properties so that subsequent graphing commands add to the existing graph.

`hold off` resets Axes properties to their defaults before drawing new plots. `hold off` is the default.

`hold` toggles the hold state between adding to the graph and replacing the graph.

**Remarks**

You test the hold state using the `ishold` function.

Although the hold state is `on`, some Axes properties change to accommodate additional graphics objects. For example, the Axes' limits increase when the data requires them to do so.

**Algorithm**

The `hold` function sets the `NextPlot` property of the current Figure and the current Axes. If several Axes objects exist in a Figure window, each Axes has its own hold state. `hold` also creates an Axes if one does not exist.

`hold on` sets the `NextPlot` property of the current Figure and Axes to `add`.

`hold off` sets the `NextPlot` property of the current Axes to `replace`.

`hold` toggles the `NextPlot` property between the `add` and `replace` states.

**See Also**

`axis`, `cla`, `ishold`, `newplot`

The `NextPlot` property of Axes and Figure graphics objects.

**Purpose** Send the cursor home

**Syntax**

```
home
```

**Description** `home` returns the cursor to the upper-left corner of the command window.

**Examples** Display a sequence of random matrices at the same location in the command window:

```
clc
for i =1:25
    home
    A = rand(5)
end
```

**See Also** `clc`

# hsv2rgb

**Purpose** Convert HSV colormap to RGB

**Syntax** `M = hsv2rgb(H)`

**Description** `M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an *m*-by-3 matrix, where *m* is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an *m*-by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

**Remarks** As `H(:,1)` varies from 0 to 1, the resulting color varies from red, through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([hue saturation value])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

**See Also** `brighten`, `colormap`, `rgb2hsv`

## Purpose

Convert indexed image into movie frame

## Syntax

```
F = im2frame(X,Map)
```

## Description

`F = im2frame(X,Map)` converts the indexed image `X` and associated colormap `Map` into a movie frame F. You can use `im2frame` to convert a sequence of images into a movie.

## Example

You can use `im2frame` to convert a sequence of images into a movie.

```
M = moviein(n);
M(:,1) = im2frame(X1,map);
M(:,2) = im2frame(X2,map);
...
M(:,n) = im2frame(Xn,map);
movie(M)
```

## See Also

`capture`, `frame2im`, `movie`, `moviein`

## Purpose

Display Image object

## Syntax

```
image(C)
image(x,y,C)
image(...,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)
```
Formal synatx – PN/PV pairs only
```
handle = image(...)
```

## Description

`image` creates an Image graphics object by interpreting each element in a matrix as an index into the Figure's colormap or directly as RGB values, depending on the data specified.

The `image` function has two forms:

- A high-level function that calls `newplot` to determine where to draw the graphics objects and sets the following Axes properties:

  `XLim` and `YLim` to enclose the Image

  `Layer` to `top` to place the Image in front of the tick marks and grid lines

  `YDir` to `reverse`

  `View` to `[0 90]`
- A low-level function that adds the Image to the current Axes without calling `newplot`. The low-level function argument list can contain only property name/property value pairs.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

`image(C)` displays matrix `C` as an Image. Each element of `C` specifies the color of a rectangular segment in the Image.

`image(x,y,C)` where `x` and `y` are two-element vectors, specifies the range of the *x*- and *y*-axis labels, but produces the same Image as `image(C)`. This can be useful, for example, if you want the axis tick labels to correspond to real physical dimensions represented by the image.

`image(x,y,C,'`*`PropertyName`*`',PropertyValue,...)` is a high-level function that also specifies property name/property value pairs. This syntax calls `newplot` before drawing the Image.

`image('`*`PropertyName`*`',PropertyValue,...)` is the low-level syntax of the `image` function. It specifies only property name/property value pairs as input arguments.

`handle = image(...)` returns the handle of the Image object it creates. You can obtain the handle with all forms of the `image` function.

**Remarks**

Image data can be either indexed or true color. An indexed image stores colors as an array of indices into the Figure colormap. A true color image does not use a colormap; instead, the color values for each pixel are stored directly as RGB triplets. In MATLAB , the `CData` property of a truecolor Image object is a three-dimensional (`m`-by-`n`-by-3) array. This array consists of three `m`-by-`n` matrices (representing the red, green, and blue color planes) concatenated along the third dimension.

The `imread` function reads image data into MATLAB arrays from graphics files in various standard formats, such as TIFF. You can write MATLAB image data to graphics files using the `imwrite` function. `imread` and `imwrite` both support a variety of graphics file formats and compression schemes.

When you read image data into MATLAB using `imread`, the data is stored as an array of 8-bit integers. This is a much more efficient storage method than the double-precision (64-bit) floating-point numbers that MATLAB typically uses.

However, it is necessary for MATLAB to interpret 8-bit image data differently from 64-bit data. This table summarizes these differences:

| Image type | Double-precision data (double array) | 8-bit data (uint8 array) |
|---|---|---|
| indexed (colormap) | Image is stored as a two-dimensional (m-by-n) array of integers in the range [1, length(colormap)]; colormap is an m-by-3 array of floating-point values in the range [0, 1] | Image is stored as a two-dimensional (m-by-n) array of integers in the range [0, 255]; colormap is an m-by-3 array of floating-point values in the range [0, 1] |
| truecolor (RGB) | Image is stored as a three-dimensional (m-by-n-by-3) array of floating-point values in the range [0, 1] | Image is stored as a three-dimensional (m-by-n-by-3) array of integers in the range [0, 255] |

### Indexed images

In an indexed image of class double, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a uint8 indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The uint8 convention is also used in graphics file formats, and enables 8-bit indexed images to support up to 256 colors. Note that when you read in an indexed image with imread, the resulting image array is always of class uint8. (The colormap, however, is of class double; see below.)

If you want to convert a uint8 indexed image to double, you need to add 1to the result. For example:

```
X64 = double(X8) + 1;
```

To convert from double to uint8, you need to first subtract 1, and then use round to ensure all the values are integers:

```
X8 = uint8(round(X64 – 1));
```

The order of the operations must be as shown in these examples, because you cannot perform mathematical operations on `uint8` arrays.

When you write an indexed image using `imwrite`, MATLAB automatically converts the values if necessary.

### Colormaps

Colormaps in MATLAB are alway `m`-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB does not support colormaps with integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

### True Color Images

In a truecolor image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a truecolor image of class `uint8`, the data values are integers in the range [0, 255].

If you want to convert a truecolor image from one data type to the other, you must rescale the data. For example, this call converts a `uint8` truecolor image to `double`:

```
RGB64 = double(RGB8)/255;
```

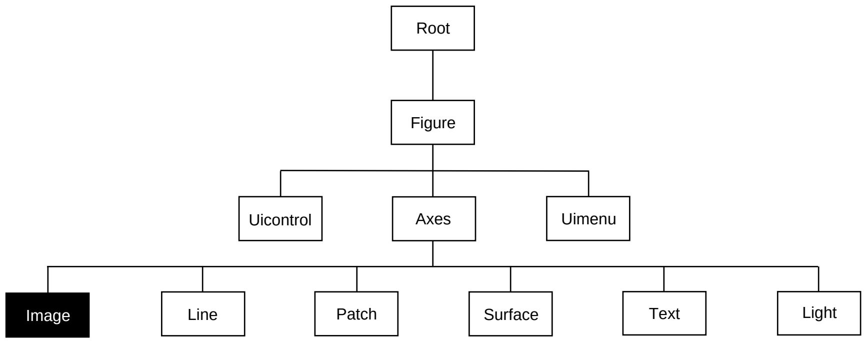
This statement converts a `double` truecolor image to `uint8`:

```
RGB8 = uint8(round(RGB*255));
```

The order of the operations must be as shown in these examples, because you cannot perform mathematical operations on `uint8` arrays.

When you write a truecolor image using `imwrite`, MATLAB automatically converts the values if necessary.

## Object Hierarchy



### Setting Default Properties

You can set default Image properties on the Axes, Figure, and Root levels:

```
set(0,'DefaultImageProperty',PropertyValue...)
set(gcf,'DefaultImageProperty',PropertyValue...)
set(gca,'DefaultImageProperty',PropertyValue...)
```

Where *Property* is the name of the Image property and *PropertyValue* is the value you are specifying.

## Image Properties

This section lists property names along with the type of values each property accepts.

**BusyAction** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked call-back routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn** string

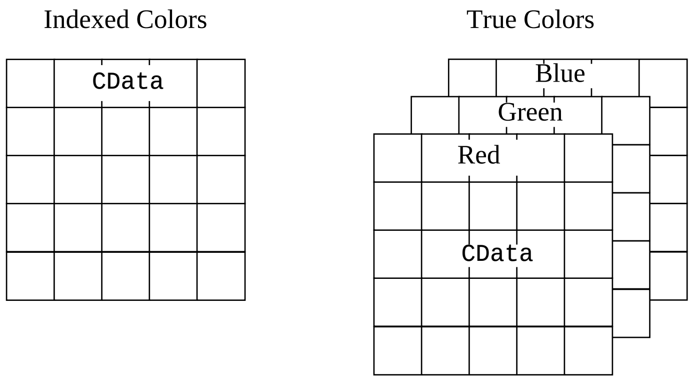
*Button press callback routine.* A callback routine that executes whenever you press a mouse button while the pointer is over the Image object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**CData** matrix or m-by-n-by-3 array

*The Image data.* A matrix of values specifying the color of each rectangular area defining the Image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the Image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default)
- Scaling the elements of `CData` to range between the values `min(get(gca,'CLim'))` and `max(get(gca,'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

A true color specification for `CData` requires an m-by-n-by-3 array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the Image. RGB values range from 0 to 1. The following picture illustrates the relative dimensions of CData for the two color models:



**`CDataMapping`** `scaled | {direct}`

*Direct or scaled indexed colors*. This property determines whether MATLAB interprets the values in `CData` as indices into the Figure colormap (the default) or scales the values according to the Axes `CLim` property.

When `CDataMapping` is `direct`, the values of `CData` should be in the range 1 to `length(get(gcf,'Colormap'))`. If you use true color specification for `CData`, this property has no effect.

**`Children`** handles

The empty matrix; Image objects have no children.

**`Clipping`** `on | off`

*Clipping mode*. By default, MATLAB clips Images to the Axes rectangle. If you set `Clipping` to `off`, the Image can display outside the Axes rectangle. For example, if you create an Image, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger Image, it extends beyond the axis limits.

**`CreateFcn`** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates an Image object. You must define this property as a default value for Images. For example, the statement,

```
set(0,'DefaultImageCreateFcn','axis image')
```

defines a default value on the Root level that sets the aspect ratio and the axis limits so the Image has square pixels. MATLAB executes this routine after setting all Image properties. Setting this property on an existing Image object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Image callback routine*. A callback routine that executes when you delete the Image object (i.e., when you issue a `delete` command or clear the Axes or Figure containing the Image). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`EraseMode`** `{normal} | none | xor | background`

*Erase mode*. This property controls the technique MATLAB uses to draw and erase Image objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` – Do not erase the Image when it is moved or changed.
- `xor` – Draw and erase the Image by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the Image. However, the Image's color depends on the color of whatever is beneath it on the display.
- `background` – Erase the Image by drawing it in the Axes' background color. This damages objects that are behind the erased Image, but Images are always properly colored.

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, newplot, cla, clf, and close. When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, Figures do not appear in the Root's CurrentFigure property, objects do not appear in the Root's CallbackObject property or in the Figure's CurrentObject property, and Axes do not appear in their parent's CurrentAxes property.

The Root ShowHiddenHandles property can be set to on to temporarily make all handles visible, regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**Interruptible** {on} | off

*Callback routine interruption mode*. The Interruptible property controls whether an Image callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the ButtonDownFcn are affected by the Interruptible property. MATLAB checks for events that can interrupt a callback routine only when it encounters a drawnow, figure, getframe, or pause command in the routine.

**Parent** handle of parent Axes

*Image's parent*. The handle of the Image object's parent Axes. You can move an Image object to another Axes by changing this property to the new Axes handle.

**Selected** on | off

*Is object selected*. When this property is on. MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects highlight when selected*. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

| | |
|---|---|
| **`Tag`** | string |

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

| | |
|---|---|
| **`Type`** | string (read only) |

*Type of graphics object*. This property contains a string that identifies the class of graphics object. For Image objects, `Type` is always '`image`'.

| | |
|---|---|
| **`UserData`** | matrix |

*User specified data*. This property can be any data you want to associate with the Image object. The Image does not use this property, but you can access it using `set` and `get`.

| | |
|---|---|
| **`Visible`** | `on | off` |

*Image visibility*. By default, Image objects are visible. Setting this property to `off` prevents the Image from being displayed. However, the object still exists and you can set and query its properties.

| | |
|---|---|
| **`XData`** | `[1 size(C,2)]` |

*X-axis range*. A two-element vector specifying the *x*-coordinates spanned by the Image, along the *x*-axis. By default, the second element in `XData` is equal to the number of columns in the Image `CData` property.

| | |
|---|---|
| **`YData`** | `[1 size(C,1)]` |

*Y-axis range*. A two-element vector specifying the *y*-coordinates spanned by the Image, along the *y*-axis. By default, the second element in `YData` is equal to the number of rows in the Image `CData` property.

## See Also

`colormap`, `iminfo`, `imread`, `imwrite`, `pcolor`,`newplot`,`surface`

# imagesc

**Purpose** Scale data and display an Image

**Syntax**

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
h = imagesc(...)
```

**Description** The `imagesc` function scales image data to the full range of the current colormap and displays an Image. (See illustration on the following page.)

`imagesc(C)` displays `C` as an Image. Each element of `C` corresponds to a rectangular area in the Image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an Image and specifies the bounds of the *x*- and *y*-axis with vectors `x` and `y`.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an Image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`h = imagesc(...)` returns the handle for an Image graphics object.

**Remarks** `x` and `y` do not affect the elements in `C`; they only affect the annotation of the Axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

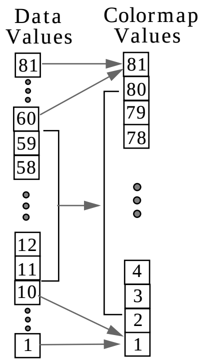
**Algorithm** `imagesc` creates an image with `CDataMapping` set to `scaled`, and sets the axes `CLim` to the value passed in `clims`.

## Examples

If the size of the current colormap is 81-by-3, the statements

```
clims = [10 60]
imagesc(C,clims)
```

map the data values in C to the colormap, as shown to the right.

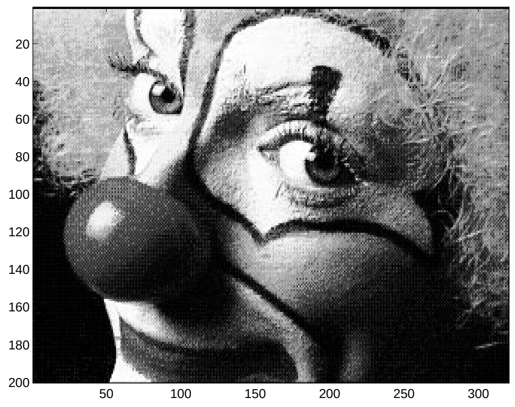


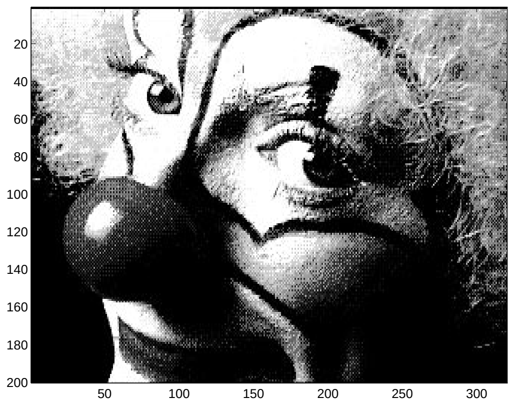
The left Image maps to the gray colormap using the statements

```
load clown
imagesc(X)
colormap(gray)
```

The right Image has values between `10` and `60` scaled to the full range of the gray colormap using the statements

```
load clown
clims = [10 60];
imagesc(X,clims)
colormap(gray)
```





**See Also** `image`, `colorbar`

**Purpose** Return information about a graphics file

**Synopsis**

```
info = imfinfo(filename,fmt)
info = imfinfo(filename)
```

**Description** `info = imfinfo(filename,fmt)` returns a structure whose fields contain information about an image in a graphics file. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory on the MATLAB path. If `imfinfo` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for `fmt`:

| Format | File type |
|---|---|
| `'bmp'` | Windows Bitmap (BMP) |
| `'hdf'` | Hierarchical Data Format (HDF) |
| `'jpg'` or `'jpeg'` | Joint Photographic Experts Group (JPEG) |
| `'pcx'` | Windows Paintbrush (PCX) |
| `'tif'` or `'tiff'` | Tagged Image File Format (TIFF) |
| `'xwd'` | X Windows Dump (XWD) |

If `filename` is a TIFF or HDF file containing more than one image, `info` is a structure array with one element (i.e., an individual structure) for each image in the file. For example, `info(3)` would contain information about the third image in the file.

The set of fields in `info` depends on the individual file and its format. However, the first seven fields are always the same. This table lists these fields and describes their values:

| Field | Value |
|---|---|
| `Filename` | A string containing the name of the file; if the file is not in the current directory, the string contains the full pathname of the file |
| `Format` | A string containing the file format, as specified by `fmt`; for JPEG and TIFF files, the three-letter variant is returned |
| `FormatVersion` | A string or number describing the version of the format |
| `Width` | An integer indicating the width of the image in pixels |
| `Height` | An integer indicating the height of the image in pixels |
| `BitDepth` | An integer indicating the number of bits per pixel |
| `ColorType` | A string indicating the type of image; either `'truecolor'` for a truecolor RGB image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image |

`info = imfinfo(filename)` attempts to infer the format of the file from its content.

## Example

```
info = imfinfo('flower.bmp')

info =

             Filename: 'flower.bmp'
               Format: 'bmp'
        FormatVersion: 'Version 3 (Microsoft Windows 3.x)'
                Width: 227
               Height: 149
             BitDepth: 8
            ColorType: 'indexed'
      FormatSignature: 'BM'
   NumColormapEntries: 256
             Colormap: [256x3  double]
              RedMask: []
            GreenMask: []
             BlueMask: []
             FileSize: 35050
      ImageDataOffset: 1078
     BitmapHeaderSize: 40
            NumPlanes: 1
      CompressionType: 'none'
           BitmapSize: 33972
       HorzResolution: 0
       VertResolution: 0
        NumColorsUsed: 256
   NumImportantColors: 0
```

## See Also

`imread`, `imwrite`

# imread

## Purpose

Read image from graphics file

## Synopsis

```
A = imread(filename,fmt)
[X,map] = imread(filename,fmt)
[...] = imread(filename)
[...] = imread(...,idx)
[...] = imread(...,ref)
```

## Description

`A = imread(filename,fmt)` reads the image in `filename` into `A`, whose class is `uint8`. If the file contains a grayscale intensity image, `A` is a two-dimensional array. If the file contains a truecolor (RGB) image, `A` is a three-dimensional (m-by-n-by-3) array. `filename` is a string that specifies the name of the graphics file, and `fmt` is a string that specifies the format of the file. The file must be in the current directory or in a directory in the MATLAB path. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

This table lists the possible values for `fmt`:

| Format | File type |
|---|---|
| `'bmp'` | Windows Bitmap (BMP) |
| `'hdf'` | Hierarchical Data Format (HDF) |
| `'jpg'` or `'jpeg'` | Joint Photographic Experts Group (JPEG) |
| `'pcx'` | Windows Paintbrush (PCX) |
| `'tif'` or `'tiff'` | Tagged Image File Format (TIFF) |
| `'xwd'` | X Windows Dump (XWD) |

`[X,map] = imread(filename,fmt)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. `X` is of class `uint8`, and `map` is of class `double`. The colormap values are rescaled when they are read to have the range [0, 1].

`[...] = imfread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(...,idx)` reads in one image from a multi-image TIFF file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is 3, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[...] = imread(...,ref)` reads in one image from a multi-image HDF file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is 12, `imread` reads the image whose reference number is 12. (Note that in an HDF file the reference numbers do not necessarily correspond with the order of the images in the file.) If you omit this argument, `imread` reads the first image in the file.

This table summarizes the types of images that `imread` can read:

| Format | Variants |
|---|---|
| BMP | 1-bit, 4-bit, 8-bit, and 24-bit uncompressed images; 4-bit and 8-bit run-length encoded (RLE) images |
| HDF | 8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets |
| JPEG | Any baseline JPEG image; JPEG images with some commonly used extensions |
| PCX | 1-bit, 8-bit, and 24-bit images |
| TIFF | Any baseline TIFF image, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression |
| XWD | 1-bit and 8-bit ZPixmaps; XYBitmaps; 1-bit XYPixmaps |

## Examples

This example reads the sixth image in a TIFF file:

```
[X,map] = imread('flower.tif',6);
```

This example reads the fourth image in an HDF file:

```
info = imfinfo('skull.hdf');
[X,map] = imread('skull.hdf',info(4).Reference);
```

## See Also

`imfinfo`, `imwrite`

## Purpose

Write an image to a graphics file

## Synopsis

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Parameter,Value,...)
```

## Description

`imwrite(A,filename,fmt)` writes the image in `A` to `filename`. `filename` is a string that specifies the name of the output file, and `fmt` is a string that specifies the format of the file. If `A` is a grayscale intensity image or a truecolor (RGB) image of class `uint8`, `imwrite` writes the actual values in the array to the file. If `A` is of class `double`, `imwrite` rescales the values in the array before writing, using `uint8(round(255*A))`. This operation converts the floating-point numbers in the range [0, 1] to 8-bit integers in the range [0, 255].

This table lists the possible values for `fmt`:

| **Format** | **File type** |
|---|---|
| `'bmp'` | Windows Bitmap (BMP) |
| `'hdf'` | Hierarchical Data Format (HDF) |
| `'jpg'` or `'jpeg'` | Joint Photographers Expert Group (JPEG) |
| `'pcx'` | Windows Paintbrush (PCX) |
| `'tif'` or `'tiff'` | Tagged Image File Format (TIFF) |
| `'xwd'` | X Windows Dump (XWD) |

`imwrite(X,map,filename,fmt)` writes the indexed image in `X`, and its associated colormap `map`, to `filename`. If `X` is of class `uint8`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X–1)`. `map` must be of class `double`; `imwrite` rescales the values in `map` using `uint8(round(255*map))`.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the legal values for `fmt`.

`imwrite(...,Parameter,Value,...)` specifies parameters that control various characteristics of the output file. Parameters are currently supported for HDF, JPEG, and TIFF files.

This table describes the available parameters for HDF files:

| Parameter | Values | Default |
|---|---|---|
| `'Compression'` | One of these strings: `'none'`, `'rle'`, `'jpeg'` | `'rle'` |
| `'Quality'` | A number between 0 and 100; parameter applies only if `'Compression'` is `'jpeg'`; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger | 75 |
| `'WriteMode'` | One of these strings: `'overwrite'`, `'append'` | `'overwrite'` |

This table describes the available parameters for JPEG files:

| Parameter | Values | Default |
|---|---|---|
| `'Quality'` | A number between 0 and 100; higher numbers mean quality is better (less image degradation due to compression), but the resulting file size is larger | 75 |

This table describes the available parameters for TIFF files:

| Parameter | Values | Default |
|---|---|---|
| 'Compression' | One of these strings: 'none', 'packbits', 'ccitt'; 'ccitt' is valid for binary images only | 'ccitt' for binary images; 'packbits' for all other images |
| 'Description' | Any string; fills in the ImageDescription field returned by imfinfo | empty |

This table summarizes the types of images that imwrite can write:

| Format | Variants |
|---|---|
| BMP | 8-bit and 24-bit uncompressed images |
| HDF | 8-bit raster image datasets, with or without associated colormap; 24-bit raster image datasets |
| JPEG | Baseline JPEG images |
| PCX | 8-bit images |
| TIFF | Baseline TIFF images, including 1-bit, 8-bit, and 24-bit uncompressed images; 1-bit, 8-bit, and 24-bit images with packbit compression; 1-bit images with CCITT compression |
| XWD | 8-bit ZPixmaps |

## Example

```
imwrite(X,map,'eggs.hdf','Compression','none','WriteMode','appe
nd')
```

## See Also

imfinfo, imread

## Purpose

Create input dialog

## Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,title)
answer = inputdlg(prompt,title,lineNo)
answer = inputdlg(prompt,title,lineNo,defAns)
```

## Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array `answer`. `prompt` is a cell array containing prompt strings.

`answer = inputdlg(prompt,title)` specifies a title for the dialog.

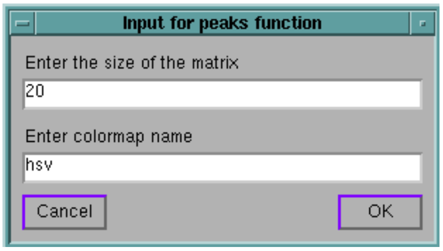
`answer = inputdlg(prompt,title,lineNo)` specifies the number of lines for each answer in `lineNo`, which is a scalar value applying to all prompts, or a vector having one element per prompt.

`answer = inputdlg(prompt,title,lineNo,defAns)` specifies the default answer to display for each question. `defAns` must contain the same number of elements as `prompt`.

## Example

Create an input dialog to input an integer and colormap name:

```
prompt = {'Enter the size of the matrix','Enter colormap name'};
def    = {20,'hsv'}
title  = 'Input for peaks function'
lineNo = 1;
answer = inputdlg(prompt,title,lineNo,def);
```


Input for peaks function
Enter the size of the matrix
20
Enter colormap name
hsv

## See Also

`textwrap`, `dialog`, `warndlg`, `helpdlg`, `questdlg`, `errdlg`

# ishandle

**Purpose** Determines if values are valid graphics object handles

**Syntax**

```
array = ishandle(h)
```

**Description** `array = ishandle(h)` returns an array that contains `1`'s where the elements of `h` are valid graphics handles and `0`'s where they are not.

**Examples** Determine whether the handles previously returned by `fill` remain handles of existing graphical objects:

```
X = rand(4); Y = rand(4);
h = fill(X,Y,'blue')
.
.
.
delete(h(3))
.
.
.
ishandle(h)
ans =
    1
    1
    0
    1
```

**See Also** `findobj`

**Purpose** Return hold state

**Syntax**

```
k = ishold
```

**Description** `k = ishold` returns the hold state of the current Axes. If `hold` is `on` `k = 1`, if `hold` is `off`, `k` = `0`.

**Examples** `ishold` is useful in graphics M-files where you want to perform a particular action only if `hold` is not `on`. For example, these statements set the view to 3-D only if `hold` is `off`:

```
if ~ishold
   view(3);
end
```

**See Also** `axes`, `figure`, `hold`, `newplot`

## legend

**Purpose** Display a legend for an Axes

**Syntax**

```
legend('string1','string2',...)
legend(Strings)
legend(h,Strings)
legend('off')
legend(h,...)
legend(...,pos)
h = legend(...)
```

**Description**

`legend` places a legend on a graph. For each line in the plot, the legend shows a sample of the line type, marker symbol, and color beside the text label you specify. When plotting filled areas, the legend contains a sample of the face color next to the text label. After the legend appears, you can move it using the mouse.

`legend('`*`string1`*`','`*`string2`*`',...)` displays a legend in the current Axes using the specified strings to label each set of data.

`legend(Strings)` adds a legend containing the rows of the matrix `Strings` as labels. This is the same as `legend(Strings(1,:),Strings(2,:),...)`.

`legend(h,Strings)` associates each row of the matrix `Strings` with the corresponding graphics object in the vector `h`.

`legend('off')` removes the legend from the current Axes or the Axes specified by `h`.

`legend(h,...)` specifies the legend for the Axes specified by `h`.

`legend(...,pos)` uses `pos` to determine where to place the legend.

- `pos = -1` places the legend outside the Axes boundary.
- `pos = 0` places the legend inside the Axes boundary, obscuring as few points as possible.
- `pos = 1` places the legend in the upper-left corner of the Axes.
- `pos = 2` places the legend in the upper-right corner of the Axes.
- `pos = 3` places the legend in the lower-left corner of the Axes.
- `pos = 4` places the legend in the lower-right corner of the Axes.
- `pos = [XlowerLeft YlowerLeft]` explicitly specifies the lower-left legend position in normalized coordinates.

`h = legend(...)` returns a handle to the legend, which is an Axes graphics object.

## Remarks

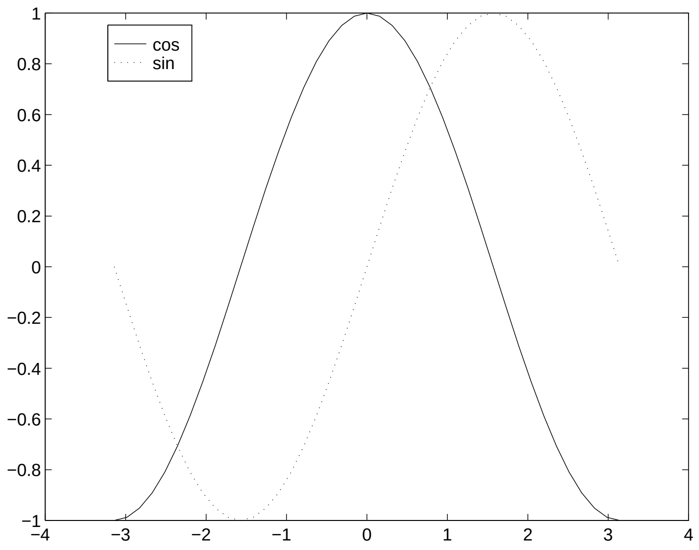
`legend` associates strings with the objects in the Axes in the same order that they are listed in the Axes `Children` property. By default, the legend annotates the current Axes.

MATLAB displays only one legend per Axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures. You move the legend by pressing the mouse button while the cursor is over the legend and dragging the legend to a new location. If your mouse has more than one button, you press the left mouse button.

## Examples

Add a legend to a plot showing a sine and cosine function:

```
x = -pi:pi/20:pi;
plot(x,cos(x),x,sin(x),':')
grid on
h = legend('cos','sin');
```



## See Also

`LineSpec`, `plot`

## Purpose

Create a Light object

## Syntax

```
light('PropertyName',PropertyValue,...)
handle = light(...)
```

## Description

`light` creates a Light object in the current Axes. Lights affect only Patch and Surface object.

`light('`*`PropertyName`*`',PropertyValue,...)` creates a Light object using the specified values for the named properties. MATLAB parents the Light to the current Axes unless you specify another Axes with the `Parent` property.

## Remarks

You cannot see a Light object *per se*, but you can see the effects of the light source on Patch and Surface objects. You can also specify an Axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one Light object is present and visible in the Axes.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

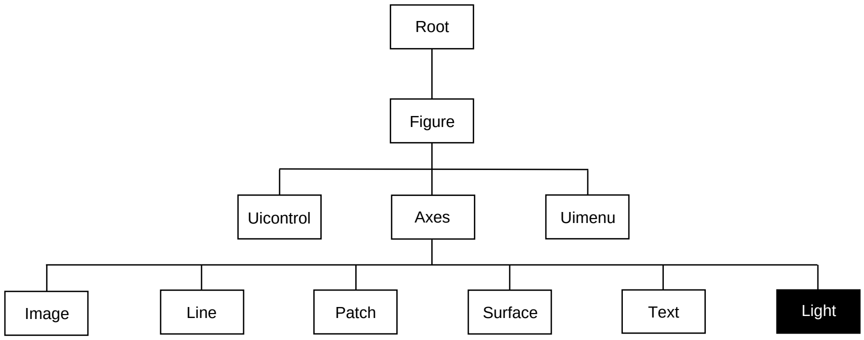
See also the Patch and Surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties.

## Examples

Light the `peaks` Surface with a light source located at infinity and oriented along the direction defined by the vector `[1 0 0]`, that is, along the *x*-axis.

```
h = surf(peaks);
set(h,’FaceLighting’,’phong’)
light(‘Position’,[1 0 0],’Style’,’infinite’);
```

## Object Hierarchy



### Setting Default Properties

You can set default Light properties on the Axes, Figure, and Root levels:

```
set(0,'DefaultLightProperty',PropertyValue...)
set(gcf,'DefaultLightProperty',PropertyValue...)
set(gca,'DefaultLightProperty',PropertyValue...)
```

Where *Property* is the name of the Light property and `PropertyValue` is the value you are specifying.

## Light Properties

This section lists property names along with the type of values each accepts.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

This property is not useful on Lights.

`Children` handles

The empty matrix; Light objects have no children.

`Clipping` `on | off`

`Clipping` has no effect on Light objects.

**`Color`** `ColorSpec`

*Color of Light*. This property defines the color of the light emanating from the Light object. Define it as three-element RGB vector or one of MATLAB's predefined names. See the `ColorSpec` reference page for more information.

**`CreateFcn`** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Light object. You must define this property as a default value for Lights. For example, the statement,

```
set(0,'DefaultLightCreateFcn','set(gcf,''Colormap'',hsv)')
```

sets the current Figure colormap to `hsv` whenever you create a Light object. MATLAB executes this routine after setting all Light properties. Setting this property on an existing Light object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Light callback routine*. A callback routine that executes when you delete the Light object (i.e., when you issue a `delete` command or clear the Axes or Figure containing the Light). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisi-`

bility is callback, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting HandleVisibility to off makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including get, findobj, gca, gcf, gco, newplot, cla, clf, and close. When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, Figures do not appear in the Root's CurrentFigure property, objects do not appear in the Root's CallbackObject property or in the Figure's CurrentObject property, and Axes do not appear in their parent's CurrentAxes property.

The Root ShowHiddenHandles property can be set to on to temporarily make all handles visible, regardless of their HandleVisibility settings (this does not affect the values of the HandleVisibility properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**Interruptible** {on} | off

*Callback routine interruption mode*. Light object callback routines defined for the DeleteFcn property are not affected by the Interruptible property.

**Style** {infinite} | local

*Parallel or divergent light source*. This property determines whether MATLAB places the Light object at infinity, in which case the light rays are parallel, or at the location specified by the Position property, in which case the light rays diverge in all directions. See the Position property.

**`Parent`** handle of parent Axes

*Light objects parent*. The handle of the Light object's parent Axes. You can move a Light object to another Axes by changing this property to the new Axes handle.

**`Position`** `[x,y,z]` in Axes data units

*Location of Light object*. This property specifies a vector defining the location of the Light object. The vector is defined from the origin to the specified *x*, *y*, and *z* coordinates. The placement of the Light depends on the setting of the `Style` property:

- If the `Style` property is set to `local`, `Position` specifies the actual location of the Light (which is then a point source that radiates from the location in all directions).
- If the `Style` property is set to `infinite`, `Position` specifies the direction from which the light shines in parallel rays.

**`Selected`** `on | off`

This property is not used by Light objects.

**`SelectionHighlight`** `{on} | off`

This property is not used by Light objects.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**`Type`** string (read only)

*Type of graphics object*. This property contains a string that identifies the class of graphics object. For Light objects, `Type` is always `'light'`.

**`UserData`** matrix

*User specified data*. This property can be any data you want to associate with the Light object. The Light does not use this property, but you can access it using `set` and `get`.

**`Visible`** `{on} | off`

*Light visibility*. While Light objects themselves are not visible, you can see the light on Patch and Surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one Light object in the Axes whose `Visible` property is on for any lighting features to be enabled (including the Axes `AmbientLightColor` and Patch and Surface `AmbientStrength`).

## See Also

`lighting`, `material`, `patch`, `surface`

## Purpose

Select the lighting algorithm

## Syntax

```
lighting flat
lighting gouraud
lighting phong
lighting none
```

## Description

`lighting` selects the algorithm used to calculate the effects of Light objects on all Surface and Patch objects in the current Axes.

`lighting flat` selects flat lighting.

`lighting gouraund` selects gouraud lighting.

`lighting phong` selects phong lighting.

`lighting none` turns off lighting.

## Remarks

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of Surfaces and Patches appropriately for the graphics object.

## See Also

`light`, `material`, `patch`, `surface`

# line

## Purpose

Create Line object

## Syntax

```
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',PropertyValue,...)
line('PropertyName',PropertyValue,...)  Formal–PN/PV pairs only
h = line(...)
```

## Description

`line` creates a Line object in the current Axes. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify matrix coordinate data using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the Axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.
- Purely low-level behavior. When you call `line` with only property name/property value pairs,

  ```
  line('XData',x,'YData',y,'ZData',z)
  ```

  MATLAB draws a Line object in the current Axes using the default Line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the Line defined in vectors `X` and `Y` to the current Axes. If `X` and `Y` are matrices of the same size, `line` draws one Line per column.

`line(X,Y,Z)` creates Lines in three-dimensional coordinates.

`line(X,Y,Z,'`*`PropertyName`*`',PropertyValue,...)` creates a Line using the values for the property name/property value pairs specified and default values for all other properties.

`line('XData',x,'YData',y,'ZData',z,'`*`PropertyName`*`',PropertyValue,...)` creates a Line in the current Axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of handles corresponding to each Line object the function creates.

## Remarks

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the `X`, `Y`, and `Z` coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z,'Color','r','LineWidth',4)
```

The low-level form of the `line` function can have arguments that are only property name/property value paris. For example,

```
line('XData',x,'YData',y,'ZData',z,'Color','r','LineWidth',4)
```

Line properties control various aspects of the Line object and are described in the “Line Properties” section. You can also set and query property values after creating the Line using `set` and `get`.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Unlike high-level functions such as `plot`, `line` does not respect the setting of the Figure and Axes `NextPlot` properties. It simply adds Line objects to the current Axes. However, Axes properties that are under automatic control such as the axis limits can change to accommodate the Line within the current Axes.

## Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the Line’s handle:

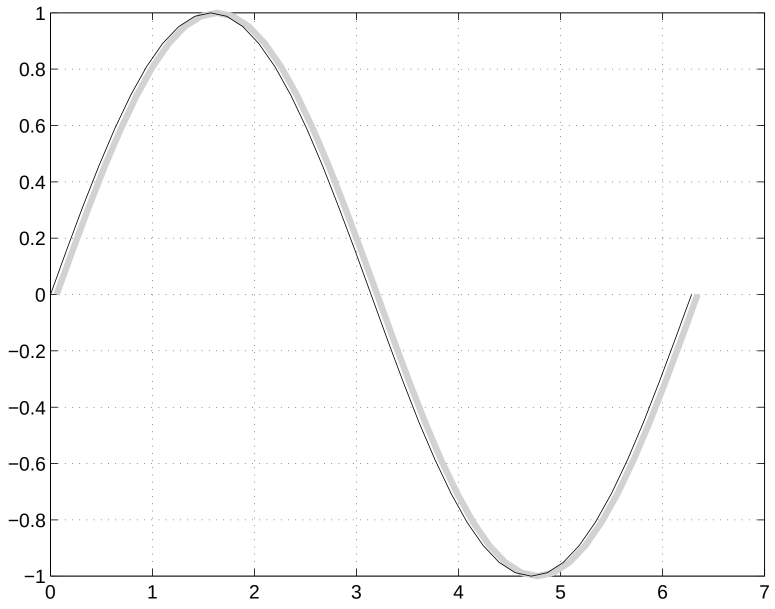
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),’k’);
```

Next, add a shadow by offsetting the *x* coordinates. Make the shadow Line light gray and wider than the default `LineWidth`:

```
hline2 = line(t+.06,sin(t),'LineWidth',4,'Color',[.8 .8 .8]);
```

Finally, pop the first Line to the front:

```
set(gca,'Children',[hline1 hline2])
```



### Input Argument Dimensions – Informal Form

This statement reuses the one column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to:

```
line(rand(4,1),rand(4,1),rand(4,1))
```

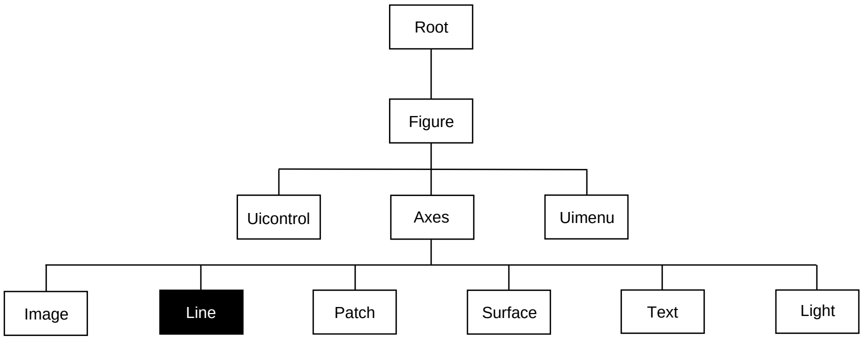
This also applies to the case when just one or two matrices have one row. For example, the statement,

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to:

```
line(rand(4,2),rand(4,2),rand(4,1))
```

## Object Hierarchy



### Setting Default Properties

You can set default Line properties on the Axes, Figure, and Root levels:

```
set(0,'DefaultLinePropertyName',PropertyValue,...)
set(gcf,'DefaultLinePropertyName',PropertyValue,...)
set(gca,'DefaultLinePropertyName',PropertyValue,...)
```

Where *PropertyName* is the name of the Line property and `PropertyValue` is the value you are specifying.

## Line Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

*Button press callback routine.* A callback routine that executes whenever you press a mouse button while the pointer is over the Line object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`Children`** vector of handles

The empty matrix; Line objects have no children.

**`Clipping`** `{on} | off`

*Clipping mode.* MATLAB clips Lines to the Axes plot box by default. If you set `Clipping` to `off`, Lines display outside the Axes plot box. This can occur if you create a Line, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a longer Line.

**`Color`** `ColorSpec`

*Line color.* A three-element RGB vector or one of MATLAB's predefined names, specifying the Line color. See the `ColorSpec` reference page for more information on specifying color.

**`CreateFcn`** string

*Callback routine executed during object creation.* This property defines a callback routine that executes when MATLAB creates a Line object. You must define this property as a default value for Lines. For example, the statement,

```
set(0,'DefaultLineCreateFcn','set(gca,''LineStyleOrder'',''-.|-
-'')')
```

defines a default value on the Root level that sets the Axes `LineStyleOrder` whenever you create a Line object. MATLAB executes this routine after setting all Line properties. Setting this property on an existing Line object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Line callback routine*. A callback routine that executes when you delete the Line object (e.g., when you issue a `delete` command or clear the Axes or Figure). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`EraseMode`** `{normal} | none | xor | background`

*Erase mode*. This property controls the technique MATLAB uses to draw and erase Line objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` (the default) — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` – Do not erase the Line when it is moved or destroyed.
- `xor` – Draw and erase the Line by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the Line. However, the Line's color depends on the color of whatever is beneath it on the display.
- `background` – Erase the Line by drawing it in the Axes' background color. This damages objects that are behind the erased Line, but Lines are always properly colored.

**HandleVisibility** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to on to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**Interruptible** `{on} | off`

*Callback routine interruption mode.* The `Interruptible` property controls whether a Line callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

**LineStyle** `{-} | - - | : | -. | none`

*Linestyle*. This property specifies the line style. The available line styles are:

| Symbol | Line Style |
|---|---|
| `–` | solid line (default) |
| `--` | dashed line |
| `:` | dotted line |
| `–.` | dash-dot line |
| `none` | no line |

You can use `LineStyle` `none` when you want to place a marker at each point, but do not want the points connected with a Line (see the `Marker` property).

**LineWidth** scalar

*The width of the Line object*. Specify this value in points (1 point = 1/72 inch). The default `LineWidth` is 0.5 points.

**Marker** character (see table)

Marker symbol. The Marker property specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include:

| Marker Specifier | Description |
|---|---|
| `+` | plus sign |
| `o` | circle |
| `*` | asterisk |
| `.` | point |
| `x` | cross |
| `square` | square |
| `diamond` | diamond |

| Marker Specifier | Description |
|---|---|
| `^` | upward pointing triangle |
| `v` | downward pointing triangle |
| `>` | right pointing triangle |
| `<` | left pointing triangle |
| `pentagram` | five-pointed star |
| `hexagram` | six-pointed star |
| `none` | no marker (default) |

**`MarkerEdgeColor`** `ColorSpec | none | {auto}`

*Marker edge color*. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the Line's `Color` property.

**`MarkerFaceColor`** `ColorSpec | {none} | auto`

*Marker face color*. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the Axes color, or the Figure color, if the Axes `Color` property is set to `none` (which is the default for Axes).

**`MarkerSize`** size in points

*Marker size*. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker at one-third the specified size.

**`Parent`** handle

*Line's parent*. The handle of the Line object's parent Axes. You can move a Line object to another Axes by changing this property to the new Axes handle.

**`Selected`** `on | off`

*Is object selected*. When this property is `on`. MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**`SelectionHighlight`** `{on} | off`

*Objects highlight when selected*. When the `Selected` property is on, MATLAB indicates the selected state by drawing handles at each vertex. When `Selec-tionHighlight` is `off`, MATLAB does not draw the handles.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**`Type`** string (read only)

*Class of graphics object*. For Line objects, `Type` is always the string `'line'`.

**`UserData`** matrix

*User-specified data*. Any data you want to associate with the Line object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

**`Visible`** `{on} | off`

*Line visibility*. By default, all Lines are visible. When set to `off`, the Line is not visible, but still exists and you can `get` and `set` its properties.

**`XData`** vector of coordinates

*X-coordinates*. A vector of *x*-coordinates defining the Line. `YData` and `ZData` must have the same number of rows. (See “Examples”).

**`YData`** vector or matrix of coordinates

*Y-coordinates*. A vector of *y*-coordinates defining the Line. `XData` and `ZData` must have the same number of rows. (See “Examples”).

**`ZData`** vector of coordinates

*Z-coordinates*. A vector of *z*-coordinates defining the Line. `XData` and `YData` must have the same number of rows. (See “Examples”).

**See Also** `axes`,`newplot`, `plot`, `plot3`

**Purpose** Line specification syntax

**Description** `LineSpec` is not a command. It refers to the three components used to specify linestyles in MATLAB :

- Line Style
- Marker Symbol
- Color

The line type, marker symbol, and color are MATLAB strings that specify a line style. You create a one-, two-, three-, or four-character string from the characters in the following table. The LineSpec argument to the `plot` command can contain up to one element from each column. Each element of the Axes `LineStyleOrder` property can contain up to one element from each of the first two columns (but can not contain Color). The order of characters is unimportant.

The `LineStyle` properties of Line, Surface, and Patch, and the `GridLineStyle` property of Axes are specified using symbols in the first column, while the `Marker` properties of Line, Surface, and Patch are specified with symbols from the second column.

| **Line Style** | | **Marker Symbol** | | **Color** | |
|---|---|---|---|---|---|
| `-` | solid line | `.` | point | `y` | yellow |
| `:` | dotted line | `o` | circle | `m` | magenta |
| `-.` | dashdot line | `x` | cross | `c` | cyan |
| `--` | dashed line | `+` | plus | `r` | red |
| | | `*` | asterisk | `g` | green |
| | | `s` | square | `b` | blue |
| | | `d` | diamond | `w` | white |
| | | `^` | up arrow | `k` | black |

| Line Style | | Marker Symbol | | Color | |
|---|---|---|---|---|---|
| | | `v` | down arrow | | |
| | | `>` | right arrow | | |
| | | `<` | left arrow | | |
| | | `p` | pentagram | | |
| | | `h` | hexagram | | |

## Examples

Create a plot that displays an asterisk at each point and connects the points with solid blue lines:

```
plot(rand(10,1),'-*b')
```

## See Also

`line`, `plot`, `surface`, `patch`, Axes `LineStyleOrder`.

## Purpose

Log-log scale plot

## Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
```

## Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`loglog(X1,Y1,`*`LineSpec`*`,...)` plots all lines defined by the `Xn,Yn,`*`LineSpec`* triples, where *`LineSpec`* determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,`*`LineSpec`* triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all Line graphics objects created by `loglog`. See the `line` reference page for more information.

`h = loglog(...)` returns a column vector of handles to Line graphics objects, one handle per Line.

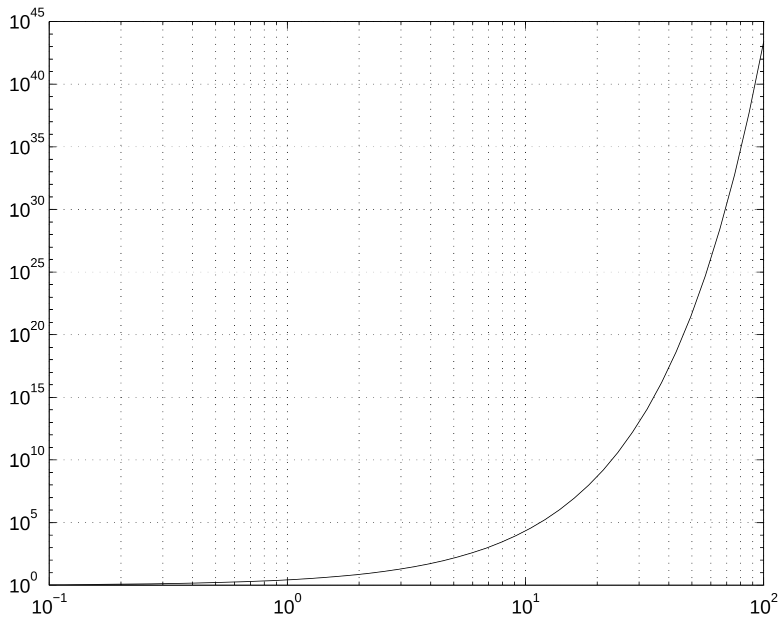
## Remarks

If you do not specify a color when plotting more than one Line, `loglog` automatically cycles through the colors and line styles in the order specified by the current Axes.

## Examples

Create a simple loglog plot:

```
x = logspace(-1,2);
loglog(x,exp(x))
```



## See Also

line, LineSpec, plot, semilogx, semilogy

## Purpose

Controls the reflectance properties of Surfaces and Patches

## Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

## Description

`material` sets the lighting characteristics of Surface and Patch objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative the diffuse and ambient light and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light, has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent and specular color reflectance of the objects.

`material metal` sets the ambient/diffuse/specular strength, specular exponent and specular color reflectance of the objects to their defaults.

## Remarks

The `material` command sets the `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, and `SpecularColorReflectance` properties of all Surface and Patch objects in the Axes. There must be visible Light objects in the Axes for lighting to be enabled. Look at the `materal.m` M-file to see the actual values set.

**See Also** `light`, `lighting`, `patch`, `surface`

## Purpose

Mesh plots

## Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
meshc(...)
meshz(...)
h = mesh(...)
h = meshc(...)
h = meshz(...)
```

## Description

`mesh`, `meshc`, and `meshz` create wireframe parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `C`.

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i, j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i, j), Y(i, j), Z(i, j))$ are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`meshc(...)` draws a contour plot beneath the mesh.

`meshz(...)` draws a curtain plot (i.e., a reference plane), around the mesh.

`h = mesh(...)`, `h = meshc(...)`, and `h = meshz(...)` return a handle to a Surface graphics object.

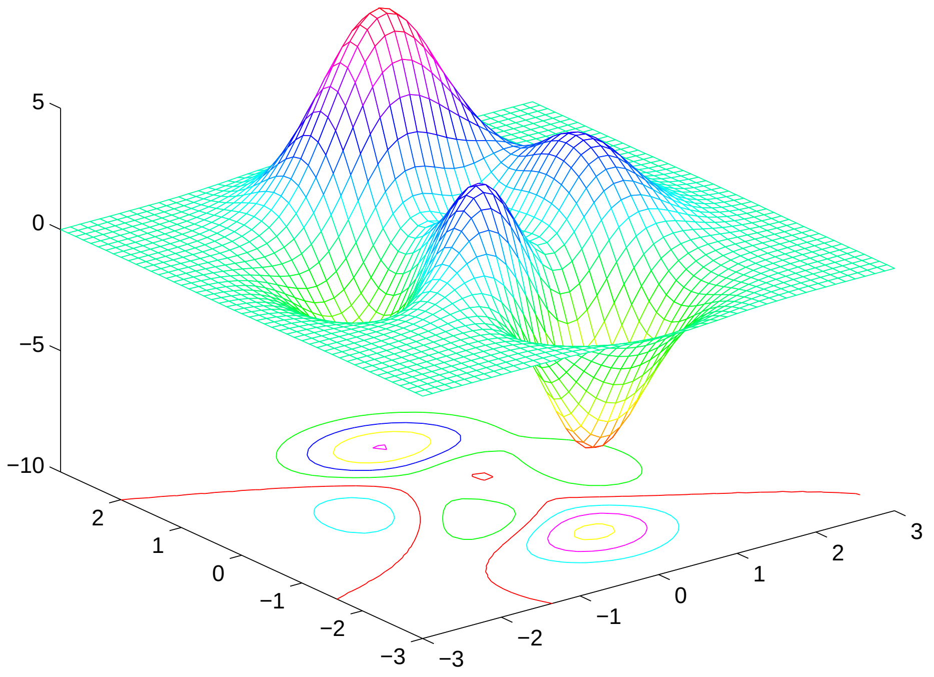
## Remarks

A mesh is drawn as a Surface graphics object with the view point specified by `view(3)`. The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or none when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` function controls the simulation of hidden-surface elimination in the mesh, and the `shading` function controls the shading model.

**Examples**

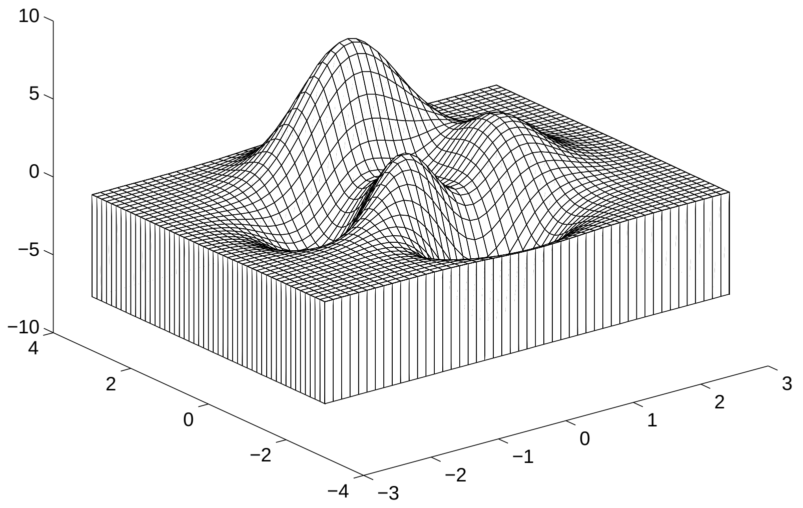
Produce a combination mesh and contour plot of the peaks surface:

```
[X,Y] = meshgrid(–3:.125:3);
Z = peaks(X,Y);
meshc(X,Y,Z);
axis([–3 3 –3 3 –10 5])
```



Generate the curtain plot for the peaks function:

```
[X,Y] = meshgrid(–3:.125:3);
Z = peaks(X,Y);
meshz(X,Y,Z)
```



## Algorithm

The range of X, Y, and Z, or the current setting of the Axes XLimMode, YlimMode, and ZlimMode properties determine the axis limits. axis sets these properties.

The range of C, or the current setting of the Axes CLim and ClimMode properties (also set by the caxis function) determine the color scaling. The scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array), onto the current colormap. MATLAB's default behavior computes the color limits automatically using the minimum and maximum data values (also set using caxis auto). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

meshc calls mesh, turns hold on, and then calls contour and positions the contour on the *x*-*y* plane. For additional control over the appearance of the contours, you can issue these commands directly. You can combine other types of graphs in this manner, for example surf and pcolor plots.

meshc assumes that X and Y are monotonically increasing. If X or Y is irregularly spaced, contour3 calculates contours using a regularly spaced contour grid, then transforms the data to X or Y.

## See Also

contour, hidden, meshgrid, surf, surfc, surfl, waterfall

axis, caxis, colormap, hold, shading, and view set graphics object properties that affect mesh, meshc, and meshz.

For a discussion of parametric surfaces plots, refer to surf.

## Purpose

Play recorded movie frames

## Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

## Description

`movie` plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once.

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If n is a vector, the first element is the number of times the movie is played, and the second through last elements specify the order in which to play the frames. For example, if `M` has three columns, `n = [10 3 2 1]` plays the movie backwards 10 times.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie in the Figure or Axes identified by `h`.

`movie(h,M,n,fps,loc)` specifies a four-element location vector, [x y 0 0], where the lower-left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower-left corner of the Figure or Axes specified by handle and in units of pixels, regardless of the object's `Units` property.

## Remarks

The movie function displays each frame as it loads the data into memory, and then plays the movie. This eliminates long delays with a blank screen when you load a memory-intensive movie. The movie's load cycle is not considered one of the movie repetitions.

## Examples

Animate the peaks function as you scale the values of Z:

```
Z = peaks;
surf(Z);
M = moviein(20);

% Freeze Axes limits
axis manual
set(gca,'nextplot','replacechildren');

% Record the movie
for j = 1:20
   surf(sin(2*pi*j/20)*Z,Z)
   M(:,j) = getframe;
end

% Play the movie twenty times
movie(M,20)
```

## See Also

getframe, moviein

**Purpose** Create matrix for movie frames

**Syntax**

```
M = moviein(n)
M = moviein(n,h)
M = moviein(n,h,rect)
```

**Description** `moviein` allocates an appropriately sized matrix for the `getframe` function.

`M = moviein(n)` creates matrix `M` having `n` columns to store `n` frames of a movie based on the size of the current Axes.

`M = moviein(n,h)` specifies a handle for a valid Figure or Axes graphics object on which to base the memory requirement.

`M = moviein(n,h,rect)` specifies the rectangular area from which to copy the bitmap, relative to the lower-left corner of the Figure or Axes graphics object identified by `h`.
`rect = [left bottom width height]`, where left and bottom specify the lower-left corner of the rectangle, and width and height specify the dimensions of the rectangle. Components of `rect` are in pixel units.

**Examples** Use `moviein` to allocate a matrix for the movie frames and `getframe` to create the movie:

```
Z = peaks;
surf(Z);
M = moviein(20);

% Freeze Axes limits
axis manual
set(gca,'nextplot','replacechildren');

% Record the movie
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    M(:,j) = getframe;
end

% Play the movie twenty times
movie(M,20)
```

**See Also** `getframe`, `movie`

## msgbox

**Purpose** Display message box

**Syntax**

```
msgbox(message)
msgbox(message,title)
msgbox(message,title,'icon')
msgbox(message,title,'custom',iconData,iconCmap)
msgbox(...,'createMode');
h = msgbox(...)
```

**Description**

`msgbox(message)` creates a message box that automatically wraps `message` to fit an appropriately sized Figure. `message` is a string vector, string matrix, or cell array.

`msgbox(message,title)` specifies the title of the message box.

`msgbox(message,title,'`*`icon`*`')` specifies which icon to display in the message box. `'`*`icon`*`'` is `'none'`, `'error'`, `'help'`, `'warn'`, or `'custom'`. The default is `'none'`.

Error Icon     Help Icon     Warning Icon

`msgbox(message,title,'custom',iconData,iconCmap)` defines a customized icon. `iconData` contains image data defining the icon; `iconCmap` is the colormap used for the image.

`msgbox(...,'`*`createMode`*`')` specifies whether the message box is modal or nonmodal, and if it is nonmodal, whether to replace another message box with the same `title`. Valid values for `'`*`createMode`*`'` are `'modal'`, `'non-modal'`, and `'replace'`.

`h = msgbox(...)` returns the handle of the box in `h`, which is a handle to a Figure graphics object.

**See Also**

`dialog`, `errordlg`, `questdlg`, `inputdlg`, `helpdlg`, `textwrap`, `warndlg`

## Purpose

Determine where to draw graphics objects

## Syntax

```
newplot
h = newplot
```

## Description

`newplot` is used at the beginning of high-level graphics M-files to determine in which Figure and Axes to draw subsequent graphics objects. Calling `newplot` can change the current Figure and current Axes.

`newplot` prepares a Figure and Axes for subsequent graphics commands.

`h = newplot` prepares a Figure and Axes for subsequent graphics commands and returns a handle to the current Axes.

## Algorithm

First, `newplot` reads the current Figure's `NextPlot` property and acts accordingly:

| NextPlot | Description |
|---|---|
| `add` | Draw to the current Figure without clearing any graphics objects already present. |
| `replacechildren` | Remove all child objects, but do not reset Figure properties to their defaults. This clears the current Figure like the `clf` command. |
| `replace` | Remove all child objects and reset Figure properties to their defaults. This clears and resets the current Figure like the `clf reset` command. |

After newplot establishes which Figure to draw in, it reads the current Axes' NextPlot property and acts accordingly:

| NextPlot | Description |
|---|---|
| add | Draw to the current Axes, retaining all graphics objects already present. |
| replacechildren | Remove all child objects, but do not reset Axes properties. This clears the current Axes like the cla command. |
| replace | Removes all child objects and resets Axes properties to their defaults. This clears and resets the current Axes like the cla reset command. |

## See Also

axes, cla, clf, figure, hold, ishold

The NextPlot property for Figure and Axes graphics objects.

## Purpose

Hardcopy paper orientation

## Syntax

```
orient
orient portrait
orient landscape
orient tall
```

## Description

`orient` returns a string with the current paper orientation, either `portrait`, `landscape`, or `tall`.

`orient portrait` sets the paper orientation for the current Figure to portrait mode. Output from subsequent `print` operations have a 4-to-3 aspect ratio and are centered in the middle of the page. This syntax orients the longest page dimension vertically. This is the default.

`orient landscape` sets the paper orientation for the current Figure to full-page landscape orientation. This syntax orients the longest page dimension horizontally.

`orient tall` maps the current Figure to the entire page in portrait orientation.

## Algorithm

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current Figure. Subsequent print operations use these properties.

## See Also

`print`

`PaperOrientation`, `PaperPosition`, `PaperSize`, `PaperType`, and `PaperUnits` properties of Figure graphics objects.

# pareto

**Purpose** Draw Pareto chart

**Syntax**

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
```

**Description** Parento charts display the values in the vector `Y` as bars drawn in descending order.

`pareto(Y)` labels each bar with its element index in `Y`.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

`H = pareto(...)` returns a combination of Patch and Line object handles.

**See Also** `hist`, `bar`

## Purpose

Create Patch graphics object

## Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(...'PropertyName',PropertyValue...)
patch('PropertyName',PropertyValue...)  PN/PV pairs only
handle = patch(...)
```

## Description

`patch` is the low-level graphics function for creating Patch graphics objects. A Patch object is one or more polygons defined by the coordinates of its vertices. You can specify the coloring and lighting of the Patch.

`patch(X,Y,C)` adds the filled two-dimensional polygon to the current Axes. The elements of `X` and `Y` specify the vertices of the polygon. If `X` and `Y` are matrices, MATLAB draws one polygon per column. `C` determines the color of the Patch. It can be a single `ColorSpec`, one color per face, or one color per vertex (see “Remarks”).

`patch(X,Y,Z,C)` creates a Patch in three-dimensional coordinates.

`patch(...'`*`PropertyName`*`',PropertyValue...)` follows the `X`, `Y`, (`Z`), and `C` arguments with property name/property value pairs to specify additional Patch properties.

`patch('`*`PropertyName`*`',PropertyValue,...)` specifies all properties using property name/property value pairs. This form allows you to omit the color specification because MATLAB uses the default face color and edge color, unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also allows you to specify the Patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and *z*-coordinates. See the “Examples” section for more information.

`handle = patch(...)` returns the handle of the Patch object it creates.

## Remarks

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the Figure and Axes `NextPlot` properties. It simply adds the Patch object to the current Axes.

If the coordinate data do not define closed polygons, `patch` closes the polygons. The points in `X`, `Y`, (and `Z`) can define concave or self-intersecting polygons.

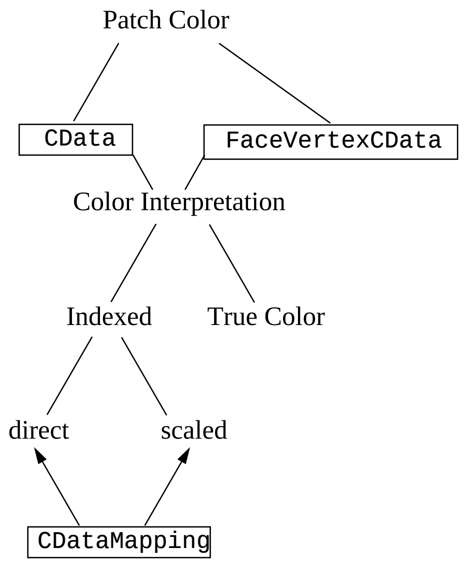
You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

There are two Patch properties that specify color:

- `CData` – use when specifying *x*-, *y*-, and *z*-coordinates (`XData`, `YData`, `ZData`).
- `FaceVertexCData` – use when specifying vertices and connection matrix (`Vertices` and `Faces`).

The `CData` and `FaceVertexCData` properties accept color data as indexed or true color (RGB) values. See the `CData` and `FaceVertexCData` property descriptions for information on how to specify color.

Indexed color data can represent either direct indices into the colormap or scaled values that map the data linearly to the entire colormap (see the `caxis` function for more information on this scaling). The `CDataMapping` property determines how MATLAB interprets indexed color data:



### Color Data Interpretation

You can specify Patch colors as:

- A single color for all faces
- One color for each face enabling flat coloring
- One color for each vertex enabling interpolated coloring

The following tables summarize how MATLAB interprets color data defined by the `CData` and `FaceVertexCData` properties.

**Table 1-1: Interpretation of the CData Property**

| [X,Y,Z]Data Dimensions | CData Required for Indexed | True Color | Results Obtained |
|---|---|---|---|
| `m-by-n` | `scalar` | `1-by-1-by-3` | Use the single color specified for all Patch faces. Edges can be only a single color. |
| `m-by-n` | `1-by-n` | `1-by-n-by-3` | Use one color for each Patch face. Edges can be only a single color. |
| `m-by-n` | `m-by-n` | `m-by-n-3` | Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated. |

**Table 2-1: Interpretation of the FaceVertexCData Property**

| Vertices Dimensions | Faces Dimensions | FaceVertexCData Required for Indexed | True Color | Results Obtained |
|---|---|---|---|---|
| m-by-n | `k-by-3` | `scalar` | `1-by-3` | Use the single color specified for all Patch faces. Edges can be only a single color. |

Table 2-1: Interpretation of the FaceVertexCData Property

| Vertices Dimensions | Faces Dimensions | FaceVertexCData Required for Indexed | True Color | Results Obtained |
|---|---|---|---|---|
| m-by-n | `k-by-3` | `k-by-1` | `k-by-3` | Use one color for each Patch face. Edges can be only a single color. |
| m-by-n | `k-by-3` | `m-by-1` | `m-by-3` | Assign a color to each vertex. Patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated. |

## Examples

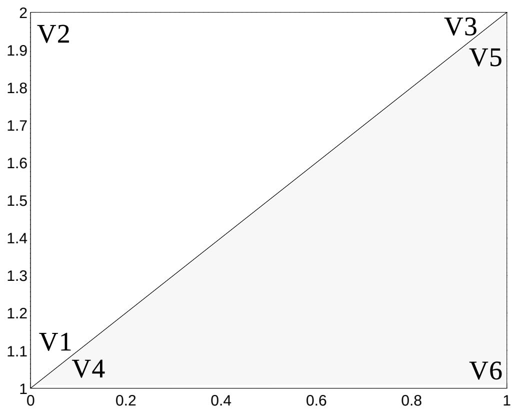
This example creates a Patch object using two different methods:

- Specifying *x*-, *y*-, and *z*-coordinates and color data (`XData`, `YData`, `ZData`, and `CData` properties).
- Specifying vertices, the connection matrix, and color data (`Vertices`, `Faces`, and `FaceVertexCData` properties).

### Specifying X, Y, and Z Coordinates

The first approach specifies the coordinates of each vertex. In this example, the coordinate data defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray:

```
x = [0 1;1 1;0 0];
y = [2 2;2 1;1 1];
z = [1 1;1 1;1 1];
tcolor(1,1,1:3) = [1 1 1];
tcolor(1,2,1:3) = [.7 .7 .7];
patch(x,y,z,tcolor)
```



Notice that each face shares two vertices with the other face ($V_1$-$V_4$ and $V_3$-$V_5$).

### Specifying Vertices and Faces

The `Vertices` property contains the coordinates of each *unique* vertex defining the Patch. The `Faces` property specifies how to connect these vertices to form each face of the Patch. For this particular example, two vertices share the same location so you need to specify only four of the six vertices. Each row contains the *x*, *y*, and *z*-coordinates of each vertex:

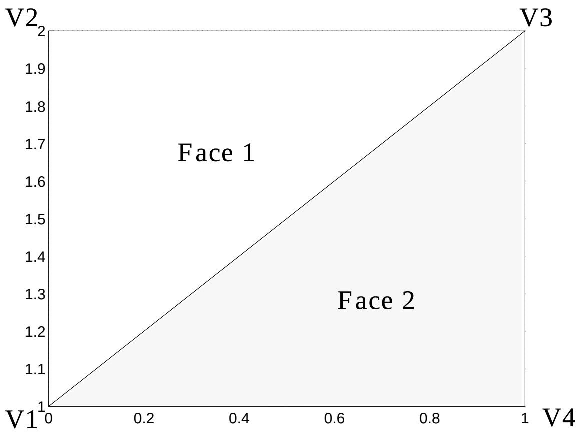
```
vert = [0 1 1;0 2 1;1 2 1;1 1 1];
```

There are only two faces, defined by connecting the vertices in the order indicated:

```
fac = [1 2 3;1 3 4];
```

Create the Patch by specifying the `Faces`, `Vertices`, and `FaceVertexCData` properties, using the same values for `tcolor` as the previous example:

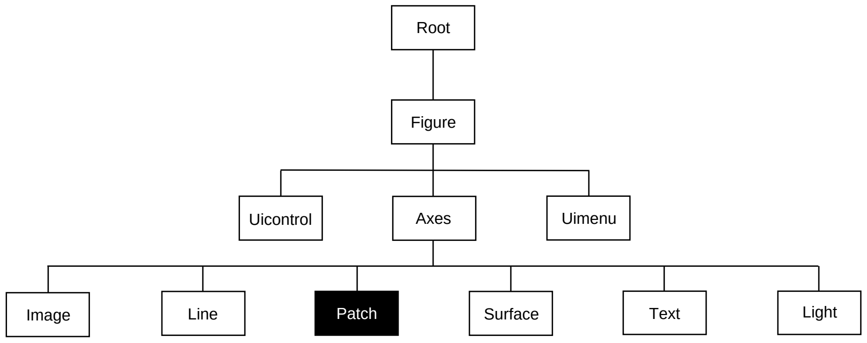
```
patch('faces',fac,'vertices',vert,'FaceVertexCData',tcolor)
```



Specifying only unique vertices and their connection matrix can reduce the size of the data considerably for Patches having many faces. See the descriptions of the `Faces`, `Vertices`, and `FaceVertexCData` properties for information on how to define them.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the `Faces` matrix with `NaN`s. To define a Patch with faces that do not close, add one or more `NaN` to the row in the `Vertices` matrix that defines the vertex you do not want connected.

## Object Hierarchy



### Setting Default Properties

You can set default Patch properties on the Axes, Figure, and Root levels:

```
set(0,'DefaultPatchPropertyName',PropertyValue...)
set(gcf,'DefaultPatchPropertyName',PropertyValue...)
set(gca,'DefaultPatchPropertyName',PropertyValue...)
```

Where *PropertyName* is the name of the Patch property and `PropertyValue` is the value you are specifying.

## Patch Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`AmbientStrength`** scalar >= 0 and <= 1

*Strength of ambient light*. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible Light object in the Axes for the ambient light to be visible. The Axes `AmbientColor` property sets the color of the ambient light, which is therefore the same on all objects in the Axes.

You can also set the strength of the diffuse and specular contribution of Light objects. See the `DiffuseStrength` and `SpecularStrength` properties.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

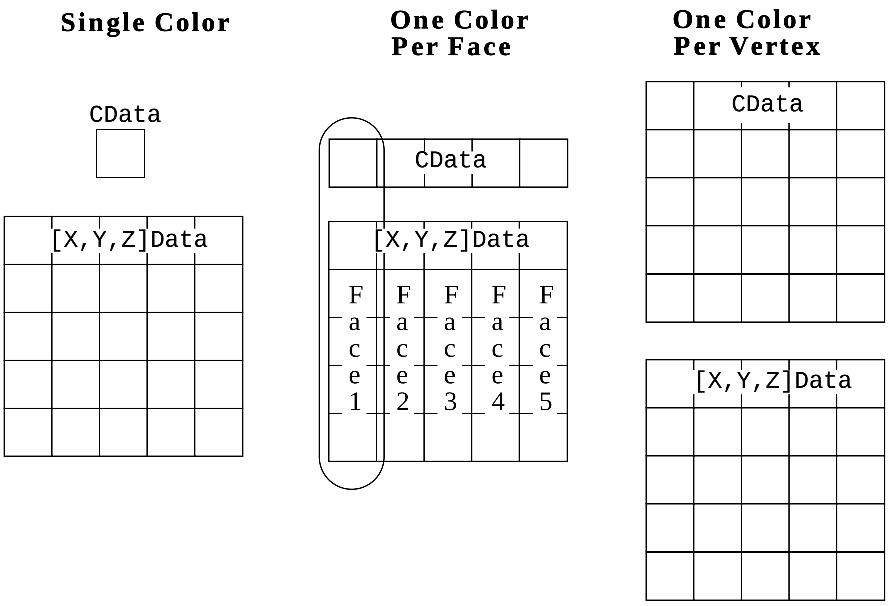
**`ButtonDownFcn`** string

*Button press callback routine.* A callback routine that executes whenever you press a mouse button while the pointer is over the Patch object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

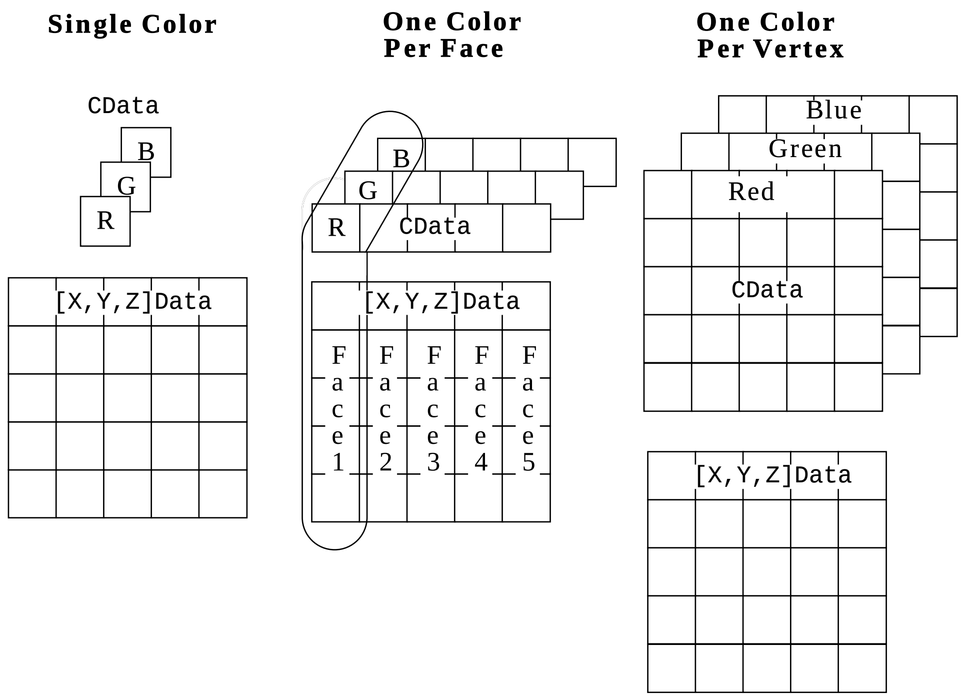
**`CData`** scalar, vector, or matrix

*Patch colors.* This property specifies the color of the Patch. You can specify color for each vertex, each face, or a single color for the entire Patch. The way MATLAB interprets `CData` depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples. On pseudocolor systems, MATLAB uses dithering to approximate the RGB triples using the colors in the figure's `Colormap` and `Dithermap`.

The following two diagrams illustrate the dimensions of `CData` with respect to the coordinate data arrays, `XData`, `YData`, and `ZData`. The first diagram illustrates the use of indexed color:


Single Color
CData
[X,Y,Z]Data
One Color Per Face
CData
[X,Y,Z]Data
Face 1
Face 2
Face 3
Face 4
Face 5
One Color Per Vertex
CData
[X,Y,Z]Data

The second diagram illustrates the use of true color. True color requires *m*-by-*n*-by-3 arrays to define red, green, and blue components for each color.



Note that if `CData` contains NaNs, MATLAB does not color the faces.

See also the `Faces`, `Vertices`, and `FaceVertexCData` properties for an alternative method of Patch definition.

**`CDataMapping`** `{scaled} | direct`

*Direct or scaled color mapping*. This property determines how MATLAB interprets indexed color data used to color the Patch. (If you use true color specification for `CData` or `FaceVertexCData`, this property has no effect.)

- `scaled` – transform the color data to span the portion of the colormap indicated by the Axes `CLim` property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.
- `direct` – use the color data as indices directly into the colormap. When not scaled, the data are usually integer values ranging from 1 to

`length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest, lower integer.

**`Children`** matrix of handles

Always the empty matrix; Patch objects have no children.

**`Clipping`** `{on} | off`

*Clipping to Axes rectangle*. When `Clipping` is `on`, MATLAB does not display any portion of the Patch outside the Axes rectangle.

**`CreateFcn`** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Patch object. You must define this property as a default value for Patches. For example, the statement,

```
set(0,'DefaultPatchCreateFcn','set(gcf,''DitherMap'',my_dither_
map)')
```

defines a default value on the Root level that sets the Figure `DitherMap` property whenever you create a Patch object. MATLAB executes this routine after setting all properties for the Patch created. Setting this property on an existing Patch object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Patch callback routine*. A callback routine that executes when you delete the Patch object (e.g., when you issue a `delete` command or clear the Axes (`cla`) or Figure (`clf`) containing the Patch). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**DiffuseStrength** scalar >= 0 and <= 1

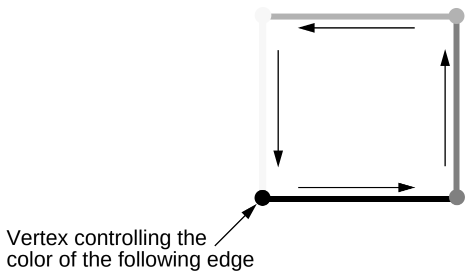
*Intensity of diffuse light*. This property sets the intensity of the diffuse component of the light falling on the Patch. Diffuse light comes from Light objects in the Axes.

You can also set the intensity of the ambient and specular components of the light on the Patch object. See the `AmbientStrength` and `SpecularStrength` properties.

**EdgeColor** `{ColorSpec} | none | flat | interp`

*Color of the Patch edge*. This property determines how MATLAB colors the edges of the individual faces that make up the Patch.

- `ColorSpec` – A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for edges. The default edge color is black. See the `ColorSpec` reference page for more information on specifying color.
- `none` – Edges are not drawn.
- `flat` – The color of each vertex controls the color of the edge that follows it. This means `flat` edge coloring is dependent on the order you specify the vertices:



- `interp` – Linear interpolation of the `CData` or `FaceVertexCData` values at the vertices determines the edge color.

**`EdgeLighting`** `{none} | flat | gouraud | phong`

*Algorithm used for lighting calculations*. This property selects the algorithm used to calculate the effect of Light objects on Patch edges. Choices are:

- `none` – Lights do not affect the edges of this object.
- `flat` – The effect of Light objects is uniform across each edge of the Patch.
- `gouraud` – The effect of Light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` – The effect of Light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**`EraseMode`** `{normal} | none | xor | background`

*Erase mode*. This property controls the technique MATLAB uses to draw and erase Patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` – Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` – Do not erase the Patch when it is moved or destroyed.
- `xor`– Draw and erase the Patch by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. Erasing the Patch does not damage the color of the objects beneath it. However, Patch color depends on the color of the screen beneath.
- `background` – Erase the Patch by drawing it in the Axes' background color. This damages objects that are behind the erased Patch, but the Patch is always properly colored.

**FaceColor** `{ColorSpec} | none | flat | interp`

*Color of the Patch face.* This property can be any of the following:

- `ColorSpec` – A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for faces. See the `ColorSpec` reference page for more information on specifying color.
- `none` – Do not draw faces. Note that edges are drawn independently of faces.
- `flat` – The values of `CData` or `FaceVertexCData` determine the color for each face in the Patch. The color data at the first vertex determines the color of the entire face.
- `interp` – Bilinear interpolation of the color at each vertex determines the coloring of each face.

**FaceLighting** `{none} | flat | gouraud | phong`

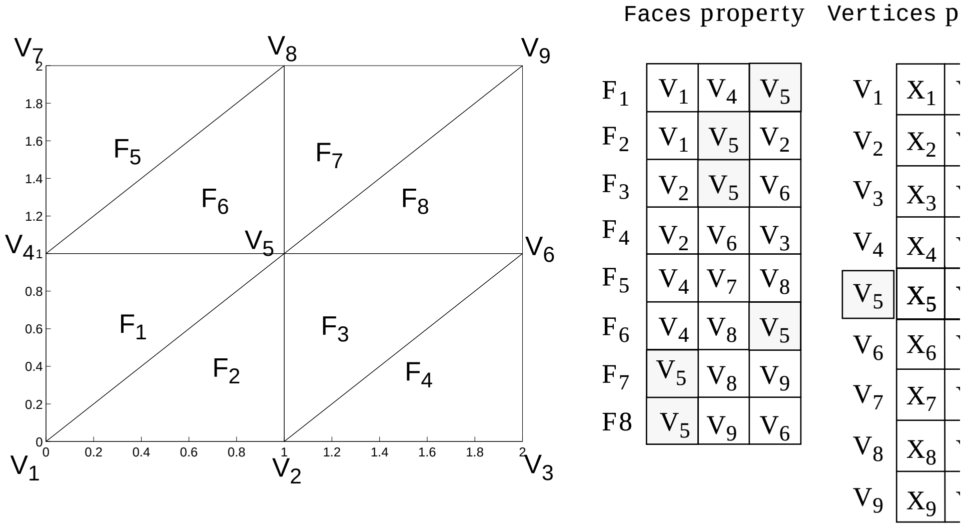
*Algorithm used for lighting calculations.* This property selects the algorithm used to calculate the effect of Light objects on Patch faces. Choices are:

- `none` – Lights do not affect the faces of this object.
- `flat` – The effect of Light objects is uniform across the faces of the Patch. Select this choice to view faceted objects.
- `gouraud` – The effect of Light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` – The effect of Light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**Faces** m-by-n matrix

*Vertex connection defining each face.* This property is the connection matrix specifying which vertices in the `Vertices` property are connected. The `Faces` matrix defines *m* faces with up to *n* vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not NaN defines the number of vertices for that face.

The `Faces` and `Vertices` properties provide an alternative way to specify a Patch that can be more efficient in most cases. For example, consider the following Patch. It is composed of eight triangular faces defined by nine vertices:



`Faces` property

| | | | |
|---|---|---|---|
| $F_1$ | $V_1$ | $V_4$ | $V_5$ |
| $F_2$ | $V_1$ | $V_5$ | $V_2$ |
| $F_3$ | $V_2$ | $V_5$ | $V_6$ |
| $F_4$ | $V_2$ | $V_6$ | $V_3$ |
| $F_5$ | $V_4$ | $V_7$ | $V_8$ |
| $F_6$ | $V_4$ | $V_8$ | $V_5$ |
| $F_7$ | $V_5$ | $V_8$ | $V_9$ |
| F8 | $V_5$ | $V_9$ | $V_6$ |

`Vertices` p

| | |
|---|---|
| $V_1$ | $X_1$ |
| $V_2$ | $X_2$ |
| $V_3$ | $X_3$ |
| $V_4$ | $X_4$ |
| $V_5$ | $X_5$ |
| $V_6$ | $X_6$ |
| $V_7$ | $X_7$ |
| $V_8$ | $X_8$ |
| $V_9$ | $X_9$ |

The corresponding `Faces` and `Vertices` properties are shown to the right of the Patch. Note how some faces share vertices with other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same Patch requires 24 vertex definitions.

**`FaceVertexCData`** matrix

*Face and vertex colors*. The `FaceVertexCData` property specifies the color of Patches defined by the Faces and Vertices properties, and the values are used when `FaceColor`, `EdgeColor`, `MarkerFaceColor`, or `MarkerEdgeColor` are set appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data:

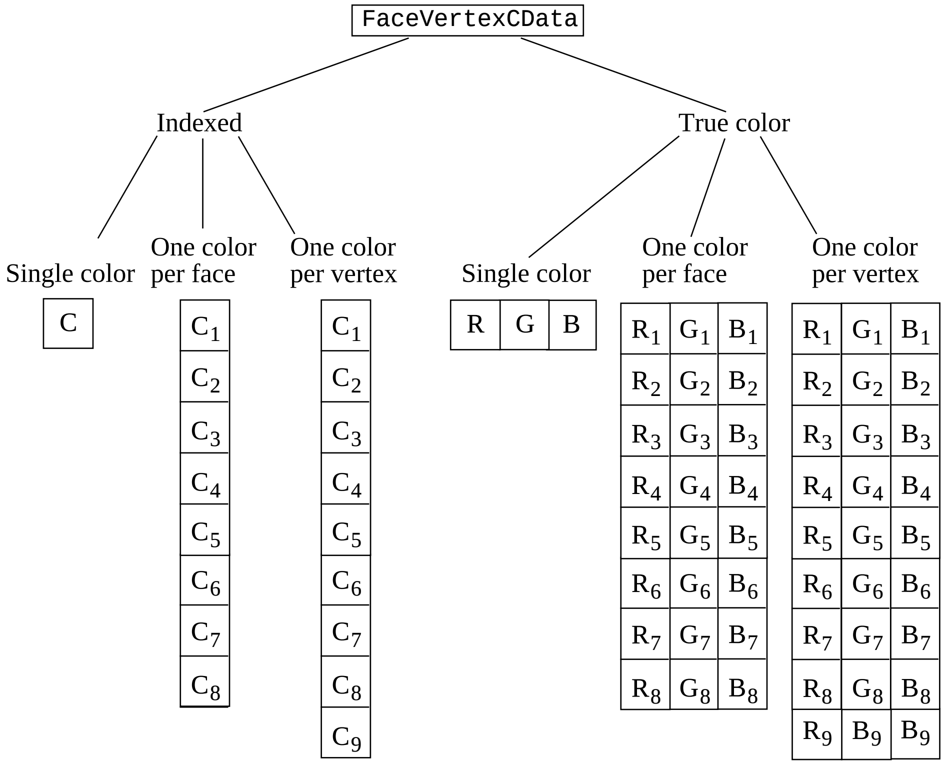
For indexed colors, `FaceVertexCData` can be:

- A single value, which applies a single color to the entire Patch
- An *n*-by-1 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face
- An *n*-by-1 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex

For true colors, FaceVertexCData can be:

- A 1-by-3 matrix , which applies a single color to the entire Patch
- An *n*-by-3 matrix, where *n* is the number of rows in the Faces property, which specifies one color per face
- An *n*-by-3 matrix, where *n* is the number of rows in the Vertices property, which specifies one color per vertex

The following diagram illustrates the various forms of the FaceVertexCData property for a Patch having eight faces and nine vertices. The CDataMapping property determines how MATLAB interprets the FaceVertexCData property when you specify indexed colors.



**HandleVisibility** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs.* This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to `on` to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**Interruptible** `{on} | off`

*Callback routine interruption mode.* The `Interruptible` property controls whether a Patch callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, getframe, or pause command in the routine. See the EventQueue property for related information.

**LineStyle** {-} | - - | : | -. | none

*Edge linestyle*. This property specifies the line style of the Patch edges. The available line styles are:

| Symbol | Line Style |
|---|---|
| – | solid line (default) |
| - - | dashed line |
| : | dotted line |
| –. | dash-dot line |
| none | no line |

You can use LineStyle none when you want to place a marker at each point, but do not want the points connected with a line (see the Marker property).

**LineWidth** scalar

*Edge line width*. The width, in points, of the Patch edges (1 point = 1/72 inch). The default LineWidth is 0.5 points.

**Marker** character (see table)

*Marker symbol*. The `Marker` property specifies marks that locate vertices. You can set values for the `Marker` property independently from the `LineStyle` property. Supported markers include:

| **Marker Specifier** | **Description** |
|---|---|
| + | plus sign |
| o | circle |
| * | asterisk |
| . | point |
| x | cross |
| square | square |
| diamond | diamond |
| ^ | upward pointing triangle |
| v | downward pointing triangle |
| > | right pointing triangle |
| < | left pointing triangle |
| pentagram | five-pointed star |
| hexagram | six-pointed star |
| none | no marker (default) |

**MarkerEdgeColor** `ColorSpec | none | {auto} | flat`

*Marker edge color*. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). *ColorSpec* defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `EdgeColor` property.

**MarkerFaceColor** ColorSpec | {none} | auto | flat

*Marker face color*. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). *ColorSpec* defines the color to use. none makes the interior of the marker transparent, allowing the background to show through. auto sets the fill color to the Axes color, or the Figure color, if the Axes Color property is set to none.

**MarkerSize** size in points

*Marker size*. A scalar specifying the size of the marker, in points. The default value for MarkerSize is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 of the specified size.

**NormalMode** {auto} | manual

*MATLAB-generated or user-specified normal vectors*. When this property is auto, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the VertexNormals property.

**Parent** Axes handle

*Patch's parent*. The handle of the Patch's parent object. The parent of a Patch object is the Axes in which it is displayed. You can move a Patch object to another Axes by setting this property to the handle of the new parent.

**Selected** on | off

*Is object selected*. When this property is on. MATLAB displays selection handles or a dashed box (depending on the number of faces) if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects highlight when selected*. When the Selected property is on, MATLAB indicates the selected state by:

- Drawing handles at each vertex for a single-faced Patch.
- Drawing a dashed bounding box for a multi-faced Patch.

When SelectionHighlight is off, MATLAB does not draw the handles.

**SpecularColorReflectance** scalar in the range 0 to 1

*Color of specularly reflected light*. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the Light object `Color` property). The proportions vary linearly for values in between.

**SpecularExponent** scalar >= 1

*Harshness of specular reflection*. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**SpecularStrength** scalar >= 0 and <= 1

*Intensity of specular light*. This property sets the intensity of the specular component of the light falling on the Patch. Specular light comes from Light objects in the Axes.

You can also set the intensity of the ambient and diffuse components of the light on the Patch object. See the `AmbientStrength` and `DiffuseStrength` properties.

**Tag** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you use Patch objects to create borders for a group of Uicontrol objects and want to change the color of the borders in a Uicontrol's callback routine. You can specify a `Tag` with the Patch definition:

```
patch(X,Y,'k','Tag','PatchBorder')
```

Then use `findobj` in the Uicontrol's callback routine to obtain the handle of the Patch and set its `FaceColor` property:

```
set(findobj('Tag','PatchBorder'),'FaceColor','w')
```

**Type** string (read only)

*Class of the graphics object*. For Patch objects, `Type` is always the string `'patch'`.

**`UserData`** matrix

*User-specified data*. Any matrix you want to associate with the Patch object. MATLAB does not use this data, but you can access it using `set` and `get`.

**`VertexNormals`** matrix

*Surface normal vectors*. This property contains the vertex normals for the Patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

**`Vertices`** matrix

*Vertex coordinates*. A matrix containing the *x*-, *y*-, *z*-coordinates for each vertex. See the `Faces` property for more information.

**`Visible`** `{on} | off`

*Patch object visibility*. By default, all Patches are visible. When set to `off`, the Patch is not visible, but still exists and you can query and set its properties.

**`XData`** vector or matrix

*X-coordinates*. The *x*-coordinates of the points at the vertices of the Patch. If `XData` is a matrix, each column represents the *x*-coordinates of a single face of the Patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

**`YData`** vector or matrix

*Y-coordinates*. The *y*-coordinates of the points at the vertices of the Patch. If `YData` is a matrix, each column represents the *y*-coordinates of a single face of the Patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

**`ZData`** vector or matrix

*Z-coordinates*. The *z*-coordinates of the points at the vertices of the Patch. If `ZData` is a matrix, each column represents the *z*-coordinates of a single face of the Patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

## See Also

`area`,`caxis`,`fill`,`fill3`,`surface`

## pcolor

**Purpose** Pseudocolor plot

**Syntax**

```
pcolor(C)
pcolor(X,Y,C)
h = pcolor(...)
```

**Description** A pseudocolor plot is a rectangular array of cells with colors determined by `C`. MATLAB creates a pseudocolor plot by using each set of four adjacent points in `C` to define a Surface patch (i.e., cell).

`pcolor(C)` draws a pseudocolor plot. The elements of `C` are linearly mapped to an index into the current colormap. The mapping from `C` to the current colormap is defined by `colormap` and `caxis`.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of `C` at the locations specified by `X` and `Y`. The plot is a logically rectangular, two-dimensional grid with vertices at the points `[X(i,j), Y(i,j)]`. `X` and `Y` are vectors or matrices that specify the spacing of the grid lines. If `X` and `Y` are vectors, `X` corresponds to the columns of `C` and `Y` corresponds to the rows. If `X` and `Y` are matrices, they must be the same size as `C`.

`h = pcolor(...)` returns a handle to a Surface graphics object.

**Remarks** A pseudocolor plot is a flat Surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,0*Z,C)` using `view([0 90])`.

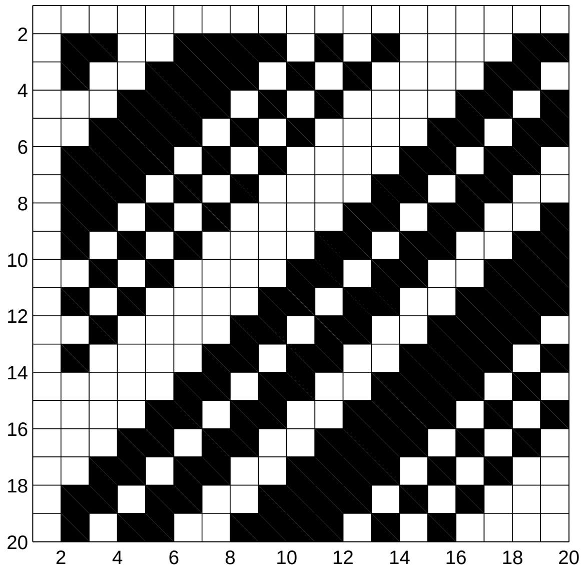
Using `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x-y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of `C` are not used.

Using `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices and all elements of `C` are used.

## Examples

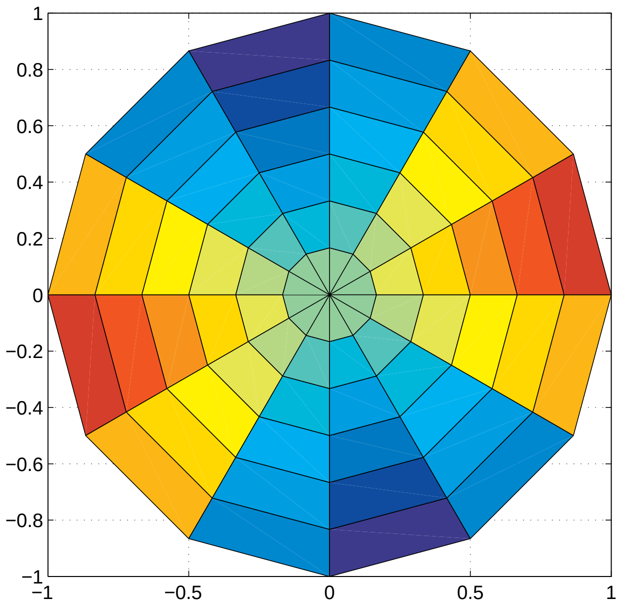
A Hadamard matrix has elements that are +1 and –1. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix:

```
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```



A simple color wheel illustrates a polar coordinate system:

```
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
axis equal
```



## Algorithm

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the Axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

## See Also

`caxis`, `image`, `mesh`, `shading`, `surf`, `view`

# pie

## Purpose

Pie chart

## Syntax

```
pie(X)
pie(X,Explode)
h = pie(...)
```

## Description

`pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,Explode)` offsets a slice from the pie. `Explode` is a vector or matrix of `0`'s and nonzeros that correspond to `X`. A non-zero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `Explode(i,j)` is nonzero. `Explode` must be the same size as `X`.

`h = pie(...)` returns a vector of handles to Patch and Text graphics objects.

## Remarks

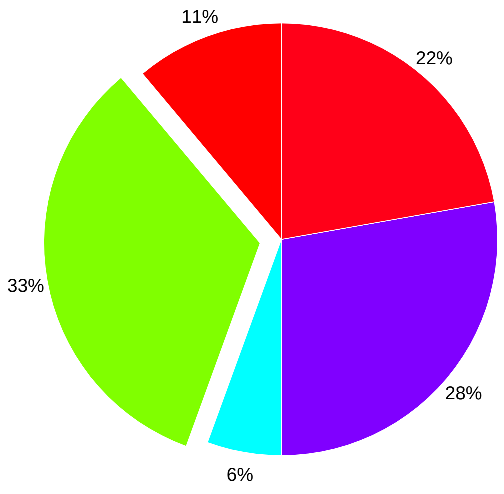
If `sum(X)` 1`pie` normalizes the `X` values so that each slice has an area of $X_i$/ $sum(X_i)$, where $X_i$ is an element of `X`. The normalized value specifies the fractional part of each pie slice. If `sum(X) < 1`, `pie` does not normalize the elements of `X`. `pie` draws a partial pie when `sum(X) < 1`.

## Examples

Emphasize the second slice in the chart by exploding it:

element to `1`:

```
x = [1 3 0.5 2.5 2]
explode = [0 1 0 0 0]
pie(x,explode)
```



## See Also

`pie3`

## Purpose

Three-dimensional pie chart

## Syntax

```
pie3(X)
pie3(X,Explode)
h = pie3(...)
```

## Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie3(X,Explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `Explode(i,j)` is nonzero. `Explode` must be the same size as `X`.

`h = pie(...)` returns a vector of handles to Patch, Surface, and Text graphics objects.

## Remarks

If `sum(X)` `1`, `pie3` normalizes the `X` values so that each slice has an area of $X_i$/ $sum(X_i)$, where $X_i$ is an element of `X`. The normalized value specifies the fractional part of each pie slice. If `sum(X) < 1`, `pie3` does not normalize the elements of `X`. `pie3` draws a partial pie when `sum(X) < 1`.

## Examples

A slice in the pie chart is offset by setting the corresponding `explode` element to `1`:

```
x = [1 3 0.5 2.5 2]
explode = [0 1 0 0 0]
pie3(x,explode)
```

## See Also

`pie`

## Purpose

Linear 2–D plot

## Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
h = plot(...)
```

## Description

`plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

`plot(X1,Y1,...)` plots all lines defined by `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, the vector is plotted versus the rows or columns of the matrix, depending whether the vector's row or column dimension matches the matrix.

`plot(X1,Y1,`*LineSpec*`,...)` plots all lines defined by the `Xn,Yn,`*LineSpec* triples, where *LineSpec* is a line specification that determines line type, marker symbol, and color of the plotted lines.

`plot(...,'`*PropertyName*`',PropertyValue,...)` sets properties to the specified property values for all Line graphics objects created by `plot`.

`h = plot(...)` returns a column vector of handles to Line graphics objects, one handle per Line.

## Remarks

If you do not specify a color when plotting more than one line, `plot` automatically cycles through the colors and line styles in the order specified by the current Axes.

You can mix `Xn,Yn,`*LineSpec* triples with `Xn,Yn` pairs, for example,

```
plot(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

## Examples

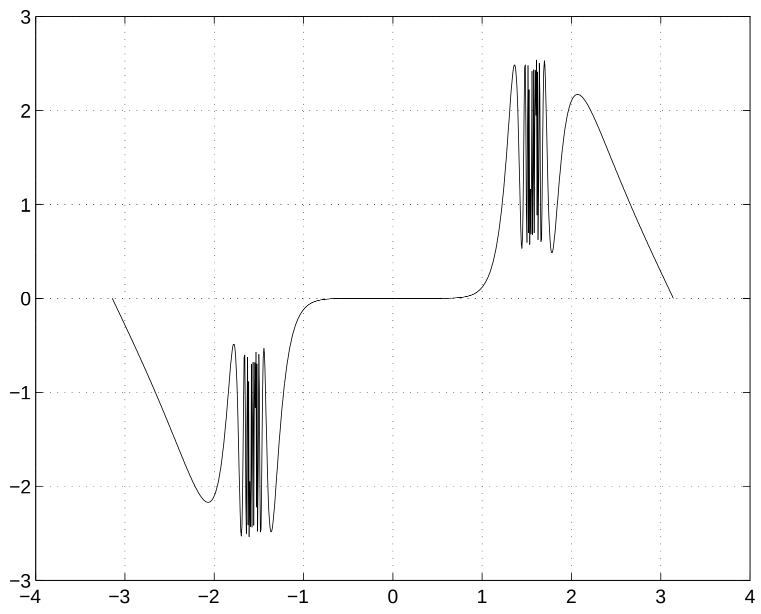
`plot(X,Y,'c+')` plots a cyan-colored plus sign at each data point.

`plot(X,Y,'r-',X,Y,'go')` plots a solid red line connecting the data points and green circles showing the location of each data point.

The statements

```
x = –pi:pi/500:pi;
y = tan(sin(x)) – sin(tan(x));
plot(x,y)
```

produce



**See Also** axis, grid, line, LineSpec, loglog, plotyy, semilogx, semilogy

## Purpose

Linear 3-D plot

## Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

## Description

The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,`*`LineSpec`*`,...)` creates and displays all lines defined by the `Xn,Yn,Zn,`*`LineSpec`* quads, where *LineSpec* is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'`*`PropertyName`*`',PropertyValue,...)` sets properties to the specified property values for all Line graphics objects created by `plot3`.

`plot3(...,'`*`PropertyName`*`',PropertyValue,...)` sets properties to the specified property values for all Line graphics objects created by `plot3`.

`h = plot3(...)` returns a column vector of handles to Line graphics objects, with one handle per Line.

## Remarks

If one or more of `X1`, `Y1`, `Z1` is a vector, the vectors are plotted versus the rows or columns of the matrix, depending if the vectors' length equals the number of rows or the number of columns.

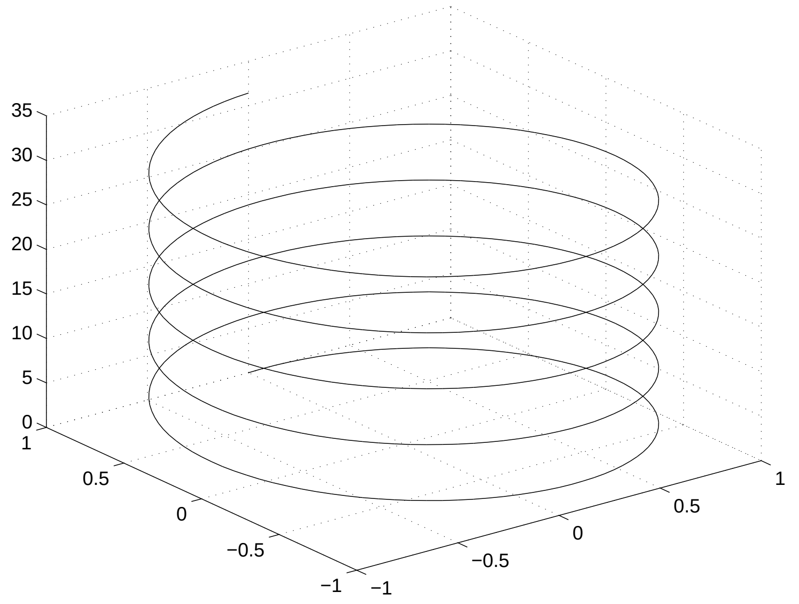
You can mix `Xn,Yn,Zn` triples with `Xn,Yn,Zn,`*`LineSpec`* quads, for example,

```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

## Examples

Plot a three-dimensional helix:

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
```



**See Also** `axis`, `grid`, `line`, `LineSpec`, `loglog`, `semilogx`, `semilogy`

## Purpose

Draw scatter plots

## Syntax

```
plotmatrix(X,Y)
plotmatrix(...,'LineSpec')
[H,AX,BigAx,P] = plotmatrix(...)
```

## Description

`plotmatrix(X,Y)` scatter plots the columns of `X` against the columns of `Y`. If `X` is *p*-by-*m* and `Y` is *p*-by-*n*, `plotmatrix` produces an *n*-by-*m* matrix of Axes. `plotmatrix(Y)` is the same as `plotmatrix(Y,Y)` except that the diagonal is replaced by `hist(Y(:,i))`.

`plotmatrix(...,'LineSpec')` uses the line specification in the string `'LineSpec'`; `'.'` is the default (see `plot` for possibilities).

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in `H`, a matrix of handles to the individual subaxes in `AX`, a handle to a big (invisible) Axes which frames the subaxes in `BigAx`, and a matrix of handles for the histogram plots in `P`. `BigAx` is left as the current Axes so that a subsequent `title`, `xlabel`, or `ylabel` commands are centered with respect to the matrix of Axes.

**Examples**

Generate plots of random data.

```
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3;]';
plotmatrix(y)
```

**Purpose**

Create graphs with y axes on both left and right side

**Syntax**

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,'function')
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)
```

**Description**

`plotyy(X1,Y1,X2,Y2)` plots `X1` versus `Y1` with y-axis labeling on the left and plots `X2` versus `Y2` with y-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,'function')` uses the plotting function specified by the string '`function`' instead of `plot` to produce each plot. '`function`' can be `plot`, `semilogx`, `semilogy`, `loglog`, `stem` or any MATLAB function that accepts the syntax:

```
h = function(x,y)
```

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function1(X2,Y2)` to plot the data for the right axis.

`[AX,H1,H2] = plotyy(...)` returns the handles of the two Axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left Axes and `AX(2)` is the right Axes.

**See Also**

`plot`

**Purpose**

Plot polar coordinates

**Syntax**

`polar(theta,rho)`
`polar(theta,rho,`*`LineSpec`*`)`

## Description

The polar function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

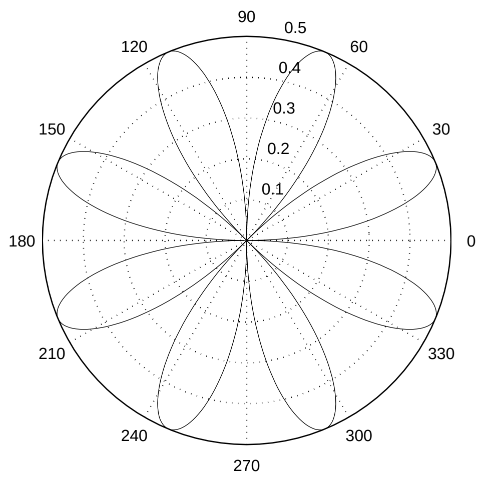
`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,`*LineSpec*`)` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

## Examples

Create a simple polar plot:

```
t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t))
```



## See Also

`cart2pol`, `compass`, `plot`, `pol2cart`, `rose`

## Purpose

Create hardcopy output

## Syntax

```
print
print –devicetype –options filename
[pcmd,dev] = printopt
```

# print, printopt

## Description

`print` and `printopt` produce hardcopy output. All arguments to the `print` command are optional. You can use them in any combination or order.

`print` sends the contents of the current Figure, including any user interface controls, to the printer using the device and system print command defined by `printopt`.

`print –`*devicetype* specifies a device type, overriding the value returned by `printopt`. The “Devices” section lists all supported device types.

`print –`*options* specifies print options that modify the action of the `print` command. (For example, the `–noui` option suppresses printing of user interface controls.) The “Options” section lists available options.

`print` *filename* directs the output to the file designated by *filename*. If *filename* does not include an extension, `print` appends an appropriate extension, depending on the device (e.g., `.eps`). If you omit *filename*, `print` sends the file to the default output device (except for `-dmeta` and `-dbitmap`, which place their output on the clipboard).

`[pcmd,dev] = printopt` returns strings containing the current system-dependent print command and output device. `printopt` is an M-file used by `print` to produce the hardcopy output. You can edit the M-file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the device options for the `print` command. Their defaults are platform-dependent.

| Platform | pcmd | dev |
|---|---|---|
| UNIX (except Silicon Graphics) | `lpr –r –s` | `–dps2` |
| Silicon Graphics | `lp` | `–dps2` |
| VMS | `PRINT/DELETE` | `–dps2` |
| Windows | `COPY /B %s LPT1:` | `–dwin` |
| Macintosh | (not applicable) | `–dps2` |

## Devices

The table below lists device types supported by MATLAB's built-in drivers. Generally, Level 2 PostScript files are smaller and render more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those devices.

| Device | Description |
|---|---|
| `-dps` | Level 1 black and white PostScript |
| `-dpsc` | Level 1 color PostScript |
| `-dps2` | Level 2 black and white PostScript |
| `-dpsc2` | Level 2 color PostScript |
| `-deps` | Level 1 black and white Encapsulated PostScript (EPS) |
| `-depsc` | Level 1 color Encapsulated PostScript (EPS) |
| `-deps2` | Level 2 black and white Encapsulated PostScript (EPS) |
| `-depsc2` | Level 2 color Encapsulated PostScript (EPS) |
| `-dhpgl` | HPGL compatible with HP 7475A plotter |
| `-dill` | Adobe Illustrator 88 compatible illustration file |
| `-dmfile` | M-file, and MAT-file when appropriate, containing Handle Graphics commands to re-create the Figure and its children |

This table lists additional devices supported via the Ghostscript post-processor, which converts PostScript files into other formats. (This feature is not available on Macintosh systems.)

| Device | Description |
|---|---|
| `-dlaserjet` | HP LaserJet |
| `-dljetplus` | HP LaserJet+ |
| `-dljet2p` | HP LaserJet IIP |
| `-dljet3` | HP LaserJet III |
| `-dljet4` | HP LaserJet 4 (defaults to 600 dpi) |
| `-ddeskjet` | HP DeskJet and DeskJet Plus |
| `-ddjet500` | HP Deskjet 500 |
| `-dcdeskjet` | HP DeskJet 500C with 1 bit/pixel color |
| `-dcdjmono` | HP DeskJet 500C printing black only |
| `-dcdjcolor` | HP DeskJet 500C with 24 bit/pixel color and high-quality color (Floyd-Steinberg) dithering |
| `-dcdj500` | HP DeskJet 500C |
| `-dcdj550` | HP Deskjet 550C |
| `-dpaintjet` | HP PaintJet color printer |
| `-dpjxl` | HP PaintJet XL color printer |
| `-dpjetxl` | HP PaintJet XL color printer |
| `-dpjxl300` | HP PaintJet XL300 color printer |
| `-ddnj650c` | HP DesignJet 650C |
| `-dbj10e` | Canon BubbleJet BJ10e |
| `-dbj200` | Canon BubbleJet BJ200 |

| Device | Description |
|---|---|
| `-dbjc600` | Canon Color BubbleJet BJC-600 and BJC-4000 |
| `-dln03` | DEC LN03 printer |
| `-depson` | Epson-compatible dot matrix printers (9- or 24-pin) |
| `-depsonc` | Epson LQ-2550 and Fujitsu 3400/2400/1200 |
| `-deps9high` | Epson-compatible 9-pin, interleaved lines (triple resolution) |
| `-dibmpro` | IBM 9-pin Proprinter |
| `-dbmp256` | 8-bit (256-color) BMP file format |
| `-dbmp16m` | 24-bit BMP file format |
| `-dpcxmono` | Monochrome PCX file format |
| `-dpcx16` | Older color PCX file format (EGA/VGA, 16-color) |
| `-dpcx256` | Newer color PCX file format (256-color) |
| `-dpcx24b` | 24-bit color PCX file format, three 8-bit planes |
| `-dpbm` | Portable Bitmap (plain format) |
| `-dpbmraw` | Portable Bitmap (raw format) |
| `-dpgm` | Portable Graymap (plain format) |
| `-dpgmraw` | Portable Graymap (raw format) |
| `-dppm` | Portable Pixmap (plain format) |
| `-dppmraw` | Portable Pixmap (raw format) |
| `-dbit` | A plain “bit bucket” device |
| `-dbitrgb` | Plain bits, RGB |
| `-dbitcmyk` | Plain bits, CMYK |

This table summarizes additional devices available on Windows systems.

| Device | Description |
|---|---|
| `-dwin` | Use Windows printing services (black and white) |
| `-dwinc` | Use Windows printing services (color) |
| `-dmeta` | Copy to clipboard in Enhanced Windows metafile format |
| `-dbitmap` | Copy to clipboard in Windows bitmap (BMP) format |
| `-dsetup` | Display **Print Setup** dialog box, but do not print |
| `-v` | Verbose mode to display **Print** dialog box (suppressed by default) |

This table summarizes additional devices available on Macintosh systems.

| Device | Description |
|---|---|
| `-dpict` | Create PICT file |
| `-v` | Verbose mode to display **Print** dialog box (suppressed by default) |

## Options

This table summarizes printing options that you can specify when you enter the `print` command.

| Option | Description |
|---|---|
| `-epsi` | Add 1-bit deep EPSI preview to EPS |
| `-loose` | Use loose bounding box for EPS and PS |
| `-cmyk` | Use CMYK colors in PostScript instead of RGB |
| `-append` | Append to existing PostScript file without overwriting |
| `-r`*number* | Specify resolution in dots per inch |
| `-adobecset` | Use PostScript default character set encoding |
| `-P`*printer* | Specify printer to use |
| `-f`*handle* | Handle of a Figure graphics object to print |
| `-s`*windowtitle* | Name of SIMULINK system window to print |
| `-painters` | Render using painter's algorithm |
| `-zbuffer` | Render using Z-buffer |
| `-noui` | Suppress printing of user interface controls |

## Example

This command saves the contents of the current Figure as Level 2 color Encapsulated PostScript in the file called `meshdata.eps`:

```
print –depsc2 meshdata
```

## See Also

`orient`, `figure`

See the *Using MATLAB Graphics* manual for detailed information about printing in MATLAB.

## Purpose

Write QuickTime movie file

## Syntax

```
qtwrite(D,size,Map,'filename')
qtwrite(M,Map,'filename')
qtwrite(...,options)
```

## Description

`qtwrite(D,size,Map,'`*`filename`*`')` writes the indexed image deck D with size size and colormap `Map` to the QuickTime movie file `'`*`filename`*`'`. If `'`*`filename`*`'` exists, it is replaced.

`qtwrite(M,Map,'`*`filename`*`')` writes the MATLAB movie matrix `M` with colormap `Map` to the QuickTime movie file `'`*`filename`*`'`.

`qtwrite(...,options)` sets the frame rate, spacial quality, and compressor type:

| Option | Description |
|---|---|
| `options(1)` | `Frame rate in frames per second. The default is 10.` |
| `options(2)` | `Compressor type:`<br>• 1 is video (default)<br>• 2 is jpeg<br>• 3 is animation |
| `options(3)` | Spacial quality:<br>• 1 - minimum<br>• 2 - low<br>• 3 - normal (default)<br>• 4 - high<br>• 5 - maximum<br>• 6 - lossless |

## Remarks

`qtwrite` requires QuickTime and works only on the Macintosh.

## Purpose

Create and display question dialog box

## Syntax

```
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title','default')
button = questdlg('qstring','title','str1','str2','default')
button =
   questdlg('qstring','title','str1','str2','str3','default')
```

## Description

`button = questdlg('`*`qstring`*`')` displays a modal dialog presenting the question `'`*`qstring`*`'`. The dialog has three default buttons—**No**, **Cancel**, and **Yes**. `'`*`qstring`*`'` is a cell array or a string that automatically wraps to fit within the dialog box. `button` contains the name of the button pressed.

`button = questdlg('`*`qstring`*`','`*`title`*`')` displays a question dialog with `'`*`title`*`'` displayed in the dialog's title bar.

`button = questdlg('`*`qstring`*`','`*`title`*`','`*`default`*`')` specifies which push button is the default in the event that the **Return** key is pressed. `'`*`default`*`'` must be `'Yes'`, `'No'`, or `'Cancel'`.

`button = questdlg('`*`qstring`*`','`*`title`*`','`*`str1`*`','`*`str2`*`','`*`default`*`')` creates a question dialog box with two push buttons labeled `'`*`str1`*`'` and `'`*`str2`*`'`. `'`*`default`*`'` specifies the default button selection and must be `'`*`str1`*`'` or `'`*`str2`*`'`.

`button =`
`questdlg('`*`qstring`*`','`*`title`*`','`*`str1`*`','`*`str2`*`','`*`str3`*`','`*`default`*`')` creates a question dialog box with three push buttons labeled `'`*`str1`*`'`, `'`*`str2`*`'`, and `'`*`str3`*`'`. `'`*`default`*`'` specifies the default button selection and must be `'`*`str1`*`'`, `'`*`str2`*`'`, or `'`*`str3`*`'`.

## Example

Create a question dialog asking the user whether to continue a hypothetical operation

```
button=questdlg('Do you want to continue?','Continue Operation',
...
               'Yes','No','Help','No');

if strcmp(buttonName,'Yes'), disp('Creating file');
elseif strcmp(buttonName,'No'), disp('Cancelled file operation')
elseif strcmp(buttonName,'Help'), disp('Sorry, no help
available');
end
```

## See Also

`dialog`, `errordlg`, `helpdlg`, `inputdlg`, `msgbox`, `warndlg`

## Purpose

Quiver or velocity plot

## Syntax

```
quiver(U,V)
quiver(X,Y,U,V)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')

h = quiver(...)
```

## Description

A quiver plot displays vectors with components (u,v) at the points (x,y).

`quiver(U,V)` draws vectors specified by `U` and `V` at the coordinates defined by `x = 1:n` and `y = 1:m`, where `[m,n] = size(U) = size(V)`. This syntax plots `U` and `V` over a geometrically rectangular grid. `quiver` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver(X,Y,U,V)` draws vectors at each pair of elements in `X` and `Y`. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(U) = size(V)`. The vector `X` corresponds to the columns of `U` and `V`, and vector `Y` corresponds to the rows of `U` and `V`.

`quiver(...,scale)` automatically scales the vectors to prevent them from overlapping, then multiplies them by `scale`. `scale = 2` doubles their relative length and `scale = 0.5` halves them. Use `scale = 0` to plot the velocity vectors without the automatic scaling.

`quiver(...,`*LineSpec*`)` specifies line style, marker symbol, and color using any valid line specification. `quiver` draws the markers at the origin of the vectors.

`quiver(...,`*LineSpec*`,'filled')` fills markers specified by *LineSpec*.

`h = quiver(...)` returns a vector of Line handles.

## Remarks

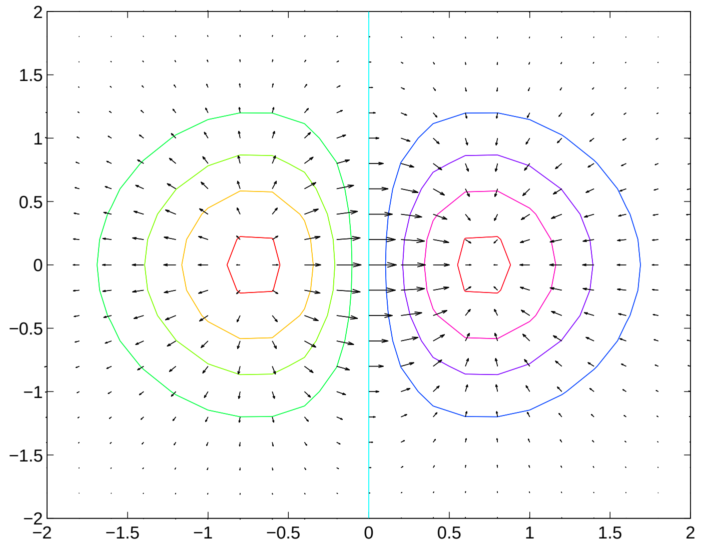
If `X` and `Y` are vectors, this function behaves as

```
[X,Y] = meshgrid(x,y)
quiver(X,Y,U,V)
```

## Examples

Plot the gradient field of the function $z = xe^{(-x^2 - y^2)}$:

```
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
grid off
hold off
```



## See Also

`contour`, `LineSpec`, `plot`, `quiver3`

## Purpose

Three-dimensional velocity plot

## Syntax

```
quiver3(Z,U,V,W)
quiver3(X,Y,Z,U,V,W)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
h = quiver3(...)
```

## Description

A three-dimensional quiver plot displays vectors with components (u,v,w) at the points (x,y,z).

`quiver3(Z,U,V,W)` plots the vectors at the equally spaced surface points specified by matrix `Z`. `quiver3` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver3(X,Y,Z,U,V,W)` plots vectors with components (u,v,w) at the points (x,y,z). The matrices `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size and contain the corresponding position and vector components.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, then multiplies them by `scale`. `scale = 2` doubles their relative length and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,`*LineSpec*`)` specify line type and color using any valid line specification.

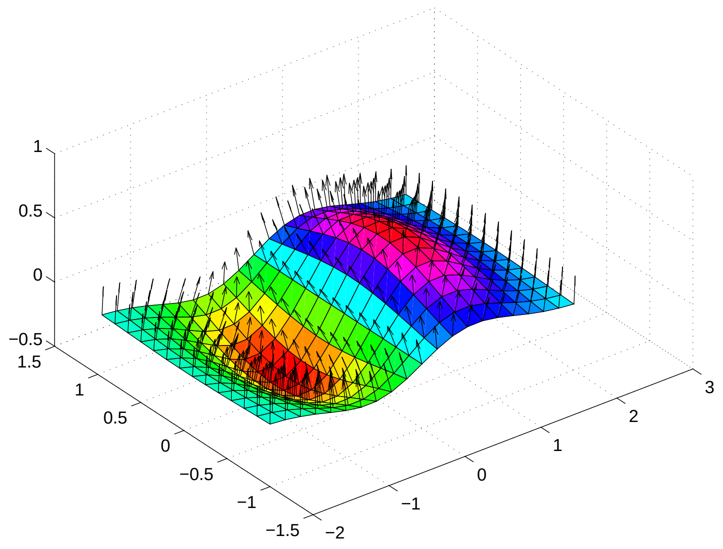
`quiver3(...,`*LineSpec*`,'filled')` fills markers specified by *LineSpec*.

`h = quiver3(...)` returns a vector of Line handles.

## Examples

Plot the surface normals of the function $z = xe^{(-x^2 - y^2)}$:

```
[X,Y] = meshgrid(-2:.2:2,-1:.15:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W);
hold on
surf(X,Y,Z);
grid on
hold off
```



## See Also

`contour`, `LineSpec`, `plot`, `plot3`, `quiver`

**Purpose** Rubberband box for area selection

**Synopsis**

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

**Description**

`rbbox` initializes and tracks a rubberband box in the current Figure. It sets the initial rectangular size of the box to 0, anchors the box at the Figure's `CurrentPoint`, and begins tracking at the Figure's `CurrentPoint`.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower-left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current Figure's `Units` property, and measured from the lower-left corner of the Figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current Figure's `Units` property, and measured from the lower-left corner of the Figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` Figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower-left corner of the box, and `width` and `height` are the dimensions of the box.

## Remarks

rbbox is useful for defining and resizing a rectangular region:

- For box definition, initialRect is [x y 0 0], where (x,y) is the Figure's CurrentPoint.
- For box resizing, initialRect defines the rectangular region that you resize (e.g., a legend). fixedPoint is the corner diametrically opposite the tracking point.

rbbox returns immediately if a button is not currently pressed. Therefore, you use rbbox with waitforbuttonpress so that the mouse button is down when rbbox is called. rbbox returns when you release the mouse button.

## Examples

Assuming the current view is view(2), use the current Axes' CurrentPoint property to determine the extent of the rectangle in dataspace units:

```
k = waitforbuttonpress

point1 = get(gca,'CurrentPoint')% button down detected
finalRect = rbbox % return Figure units
point2 = get(gca,'CurrentPoint')% button up detected

point1 = point1(1,1:2)% extract x and y
point2 = point2(1,1:2)

p1 = min(point1,point2)% calculate locations
offset = abs(point1-point2)% and dimensions

x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)]
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)]

hold on
axis manual
plot(x,y)% redraw in dataspace units
```

## See Also

dragrect, waitforbuttonpress

## refresh

**Purpose** Redraw current Figure

**Syntax**

```
refresh
refresh(h)
```

**Description** `refresh` erases and redraws the current Figure.

`refresh(h)` redraws the Figure identified by `h`.

## Purpose

Reset graphics object properties to their defaults

## Syntax

```
reset(h)
```

## Description

`reset(h)` resets all properties having factory defaults on the object identified by `h`. To see the list of factory defaults, use the statement,

```
get(0,'factory')
```

If `h` is a Figure, MATLAB does not reset `Position`, `Units`, `PaperPosition`, and `PaperUnits`. If `h` is an Axes, MATLAB does not reset `Position` and `Units`.

## Examples

`reset(gca)` resets the properties of the current Axes.

`reset(gcf)` resets the properties of the current Figure.

## See Also

`cla`, `clf`, `gca`, `gcf`, `hold`

# rgb2hsv

**Purpose** Convert RGB colormap to HSV colormap

**Syntax** `cmap = rgb2hsv(M)`

**Description** `cmap = rgb2hsv(M)` converts a RGB colormap, `M`, to a HSV colormap, `cmap`. Both colormaps are *m*-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix, `M`, represent intensities of red, green, and blue, respectively. The columns of the output matrix, `cmap`, represent hue, saturation, and value, respectively.

**See Also** `brighten`, `colormap`, `hsv2rgb`, `rgbplot`

**Purpose**

Plot colormap

**Syntax**

```
rgbplot(cmap)
```

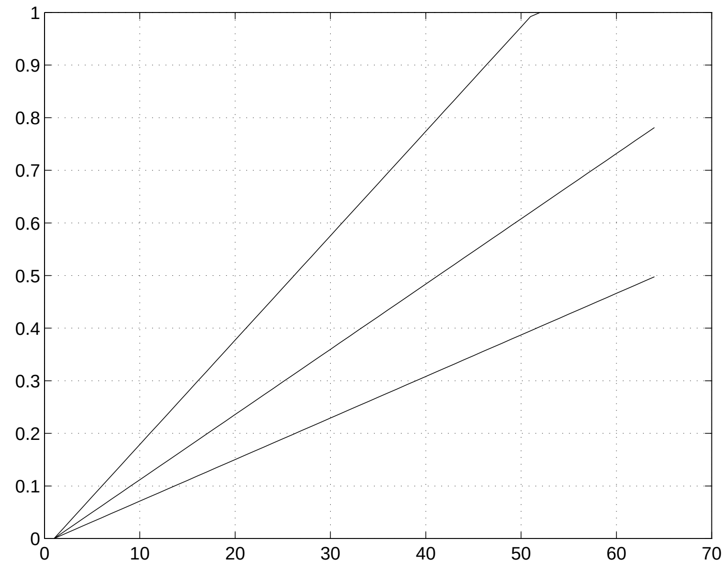
**Description**

`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an *m*-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

**Examples**

Plot the RGB values of the `copper` colormap:

```
rgbplot(copper)
```



**See Also**

`colormap`

# ribbon

**Purpose**

Ribbon plot

**Syntax**

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
h = ribbon(...)
```

**Description**

`ribbon(Y)` plots the columns of `Y` as separate three-dimensional ribbons using `X = 1:size(Y,1)`.

`ribbon(X,Y)` plots `X` versus the columns of `Y` as three-dimensional strips. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

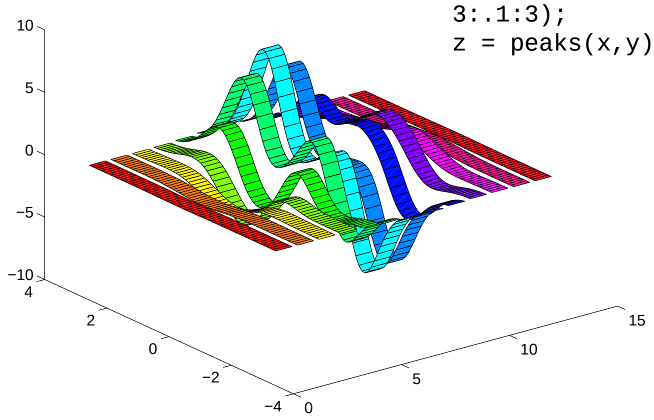
`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`.

`h = ribbon(...)` returns a vector of handles to Surface graphics objects. `ribbon` returns one handle per strip.

**Examples**

Create a ribbon plot of the peaks function:

```
[x,y] = meshgrid(–3:.5:3,–
3:.1:3);
z = peaks(x,y);
```



**See Also**

`plot`, `plot3`, `surface`

## Purpose

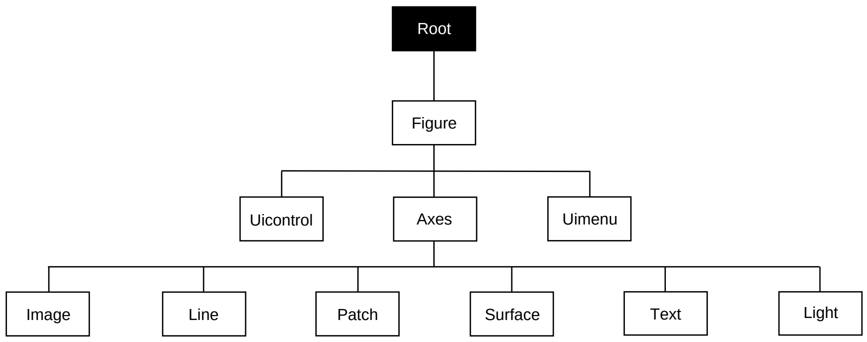
Root object properties

## Description

The Root is a graphics object that corresponds to the computer screen. There is only one Root object and it has no parent. The children of the Root object are Figures.

The Root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the Root properties, which are described in the "Root Properties" section.

## Object Hierarchy



### Root Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**BusyAction** `cancel | {queue}`

Not used by the Root object.

**ButtonDownFcn**string

Not used by the Root object.

**CallbackObject** handle (read only)

*Handle of current callback's object*. This property contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains the empty matrix [ ]. See also the `gco` command.

**CaptureMatrix** (obsolete)

This property has been superseded by the `getframe` command.

**CaptureRect** (obsolete)

This property has been superseded by the `getframe` command.

**Children** vector of handles

*Handles of child objects*. A vector containing the handles of all non-hidden Figure objects. You can change the order of the handles and thereby change the stacking order of the Figures on the display.

**Clipping** `{on} | off`

Clipping has no effect on the Root object.

**CreateFcn**

The Root does not use this property.

**CurrentFigure** Figure handle

Handle of the current Figure window, which is the one most recently created, clicked in, or made current with the statement:

```
figure(h)
```

which restacks the Figure to the top of the screen, or

```
set(0,'CurrentFigure',h)
```

which does not restack the Figures. In these statements, `h` is the handle of an existing Figure. If there are no Figure objects,

```
get(0,'CurrentFigure')
```

returns the empty matrix. Note, however, that `gcf` always returns a Figure handle, and creates one if there are no Figure objects.

**DeleteFcn** string

Since you cannot delete the Root object, this property is not used.

**Diary** `on | {off}`

*Diary file mode*. When this property is `on`, MATLAB maintains a file (whose name is specified by the `DiaryFile` property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

| | |
|---|---|
| **DiaryFile** | string |

*Diary filename*. The name of the diary file. The default name is `diary`.

| | |
|---|---|
| **Echo** | `on | {off}` |

*Script echoing mode*. When `Echo` is `on`, MATLAB displays each line of a script file as it executes. See also the `echo` command.

| | |
|---|---|
| **ErrorMessage** | string |

*Text of last error message*. This property contains the last error message issued by MATLAB.

| | |
|---|---|
| **Format** | `short | {shortE} | long | longE | blank | hex | + | rat` |

*Output format mode*. This property sets the format used to display numbers. See also the `format` command.

- `short` – Fixed-point format with 5 digits.
- `shortE` – Floating-point format with 5 digits.
- `shortG` – Fixed- or floating-point format displaying as many significant figures as possible with 5 digits.
- `long` – Scaled fixed-point format with 15 digits.
- `longE` – Floating-point format with 15 digits.
- `longG` – Fixed- or floating-point format displaying as many significant figures as possible with 15 digits.
- `bank` – Fixed-format of dollars and cents.
- `hex` – Hexadecimal format.
- `+` – Displays + and – symbols.
- `rat` – Approximation by ratio of small integers.

| | |
|---|---|
| **FormatSpacing** | `compact | {loose}` |

*Output format spacing (see also `format` command).*

- `compact` — Suppress extra line feeds for more compact display.
- `loose` — Display extra line feeds for a more readable display.

**HandleVisibility** `{on} | callback | off`

This property is not useful on the Root object.

**Interruptible** `{on} | off`

This property is not useful on the Root object.

**Parent** handle

*Handle of parent object*. This property always contains the empty matrix, as the Root object has no parent.

**PointerLocation** `[x,y]`

*Current location of pointer*. A vector containing the *x*- and *y*-coordinates of the pointer position, measured from the lower-left corner of the screen. You can move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

This property always contains the instantaneous pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the `PointerLocation` can get a different value than the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

**PointerWindow** handle (read only)

*Handle of window containing the pointer*. MATLAB sets this property to the handle of the Figure window containing the pointer. If the pointer is not in a MATLAB window, the value of this property is 0. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

**Profile** `on | {off}`

*M-file profiler on or off*. Setting this property to `on` activates the profiler when you execute the M-files named in `ProfileFile`. The profiler determines what percentage of time MATLAB spends executing each line of the M-file. See also the `profile` command.

**ProfileFile** M-file name

*M-file to profile*. This property contains the full path name of the M-file to profile.

**`ProfileCount`** vector

*Profiler output*. This property is a *n*-by-1 vector, where *n* is the number of lines of code in the profiled M-file. Each element in this vector represents the number of times the profiler found MATLAB executing a particular line of code. The `ProfileInterval` property determines how often MATLAB profiles (i.e., determines which line is executing).

**`ProfileInterval`** scalar

*Time increment to profile M-file*. This property sets the time interval at which the profiler checks to see what line in the M-file is executing.

**`ScreenDepth`** bits per pixel

*Screen depth*. The depth of the display bitmap (i.e., the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is 2 raised to this power.

`ScreenDepth` supersedes the `BlackAndWhite` property. To override automatic hardware checking, set this property to 1. This value causes MATLAB to assume the display is monochrome. This is useful if MATLAB is running on color hardware, but is displaying on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

**`ScreenSize`** 4-element rectangle vector (read only)

*Screen size*. A four-element vector,

```
[left,bottom,width,height]
```

that defines the display size. `left` and `bottom` are 0 for all `Units` except `pixels`, in which case `left` and `bottom` are 1. `width` and `height` are the screen dimensions in units specified by the `Units` property.

**`Selected`** `on | off`

This property has no effect on the Root level.

`SelectionHighlight` `{on} | off`

This property has no effect on the Root level.

**`ShowHiddenHandles`** `on | {off}`

*Show or hide handles marked as hidden*. When set to `on`, this property disables handle hiding and exposes all object handles, regardless of the setting of an object's `HandleVisibility` property. When set to `off`, all objects so marked remain hidden within the graphics hierarchy.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can set `Tag` to any string.

**`TerminalHideGraphCommand`** string X-Windows only

*Hide graph window command*. 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 used only by the terminal graphics driver. Consult your terminal manual for the correct escape sequence.

**`TerminalOneWindow`** `{on} | off` X-Windows only

*One window terminal*. This property indicates whether 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 used only by the terminal graphics driver.

**`TerminalDimensions`** pixels X-Windows only

*Size of default terminal*. This property defines the size of the terminal.

**`TerminalProtocol`** `none | x | tek401x | tek410x` X-Windows only

*Type of terminal*. 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 is automatically set to `x`.

Once this property is set, you cannot change it unless you quit and restart MATLAB.

**`TerminalShowGraphCommand`** string X-Windows only

*Display graph window command*. 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.

**`Type`** string (read only)

Class of graphics object. For the Root object, `Type` is always `'root'`.

**`Units`** `{pixels} | normalized | inches | centimeters | points`

*Unit of measurement*. This property specifies the units MATLAB uses to interpret size and location data. All units are measure from the lower-left corner of the screen. Normalized units map the lower-left corner of the screen to (0,0) and the upper right corner to (1.0,1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).

This property affects the `PointerLocation` and `ScreenSize` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your operation so as not to affect other functions that assume `Units` is set to the default value.

**`UserData`** matrix

*User specified data*. This property can be any data you want to associate with the Root object. MATLAB does not use this property, but you can access it using the `set` and `get` functions.

**`Visible`** `{on} | off`

*Object visibility*. This property has no effect on the Root object.

## See Also

`diary`, `echo`, `figure`, `format`, `gcf`, `get`, `set`

# rose

## Purpose

Angle histogram

## Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
[tout,rout] = rose(...)
```

## Description

`rose` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range. Each group is shown as one bin.

`rose(theta)` plots an angle histogram showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle from the origin of each bin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

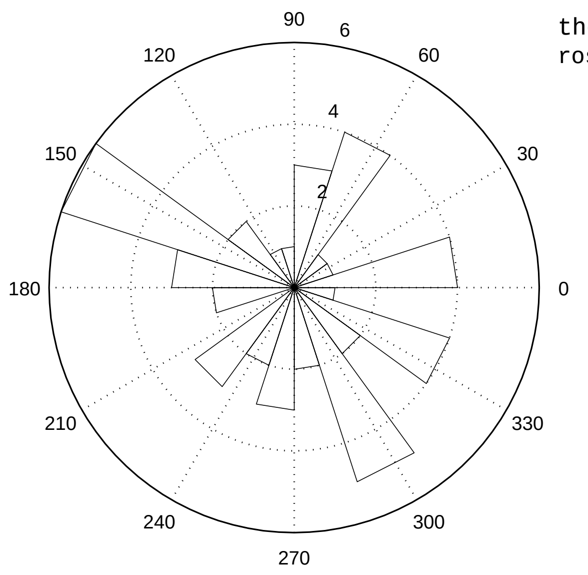
`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is `20`.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

## Example

Create a rose plot showing the distribution 50 random numbers.



```
theta = 2*pi*rand(1,50)
rose(theta)
```

## See Also

`compass`, `feather`, `hist`, `polar`

# rotate

**Purpose** Rotate object about a specified direction

**Syntax**

```
rotate(h,direction,alpha)
rotate(...,origin)
```

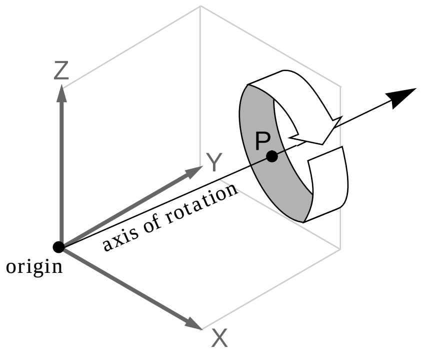
**Description** The `rotate` function rotates a graphics object in three-dimensional space, according to the right-hand rule.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

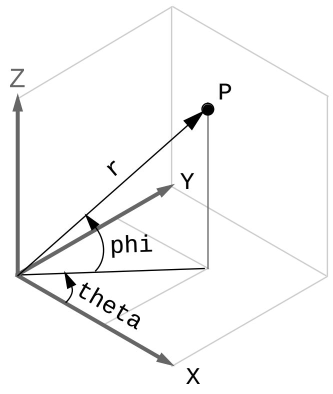
`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector. The default is `[0 0 0]`.

**Remarks** The graphics object you want rotated must be a child of an Axes graphics object. The object's data is modified by the rotation transformation. This is in contrast to `view` and `rotate3d`, which only modify the viewpoint.

The axis of rotation is defined by an origin and a point *P* relative to the origin. *P* is expressed as the spherical coordinates `[theta phi]`, or as Cartesian coordinates.



The two-element form for `direction` specifies the axis direction using the spherical coordinates `[theta phi]`. `theta` is the angle in the *xy* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *xy* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to (X,Y,Z).

## Examples

Rotate a graphics object 180° about the *x*-axis:

```
h = surf(peaks(20))
rotate(h,[1 0 0],180)
```

Rotate a Surface graphics object 45° about its center in the *z* direction:

```
h = surf(peaks(20))
zdir = [0 0 1]
center = [10 10 0]
rotate(h,zdir,45,center)
```

## Algorithm

`rotate` changes the `Xdata`, `Ydata`, and `Zdata` properties of the appropriate graphics object.

## See Also

`rotate3d`, `sph2cart`, `view`

# rotate3d

**Purpose** Rotate Axes using mouse

**Syntax**

```
rotate3d
rotate3d on
rotate3d off
```

**Description** `rotate3d on` enables interactive Axes rotation within the current figure using the mouse. When interactive Axes rotation is enabled, clicking on an Axes draws an animated box, which rotates as the mouse is dragged, showing the `View` that will result when the mouse button is released. A numeric readout appears in the lower-left corner of the figure during this time, showing the current Azimuth and Elevation of the animated box. Releasing the mouse button removes the animated box and the readout, and changes the `View` of the Axes to correspond to the last orientation of the animated box.

`rotate3d off` disables interactive Axes rotation in the current Figure.

`rotate3d` toggles interactive Axes rotation in the current Figure.

**See Also** `rotate`, `view`

## Purpose

Selecting, moving, resizing, or copying graphics objects

## Syntax

```
object_creation_fcn('ButtonDownFcn','selectmoveresize')
set(h,'ButtonDownFcn','selectmoveresize')
A = selectmoveresize;
```

## Description

`selectmoveresize` is a function that you can use as the callback routine for any graphics object's button down function. When executed, it selects graphics objects and allows you to move, resize, and copy them.

`A = selectmoveresize` returns a structure array containing:

- `A.Type`: a sting containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`
- `A.Handles`: a list of the selected handles or for a `Copy` an Mx2 matrix containing the original handles in the first column and the new handles in the second column.

## See Also

The `ButtonDownFcn` of all graphics objects.

# semilogx, semilogy

**Purpose** Semi-logarithmic plots

**Syntax**

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)

semilogy(...)
h = semilogy(...)
```

**Description**

`semilogx` and `semilogy` plot data as logarithmic scales for the *x*- and *y*-axis, respectively.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y),imag(Y))` if `Y` contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogx(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`semilogx(X1,Y1,`*`LineSpec`*`,...)` plots all lines defined by the `Xn,Yn,`*`LineSpec`* triples. *`LineSpec`* determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all Line graphics objects created by `semilogx`. See the `line` reference page for more information.

`semilogy(...)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis.

`h = semilogx(...)` and `h = semilogy(...)` return a vector of handles to Line graphics objects, one handle per Line.

## Remarks

If you do not specify a color when plotting more than one line, `semilogx` and `semilogy` automatically cycle through the colors and line styles in the order specified by the current Axes `ColorOrder` and `LineStyleOrder` properties.

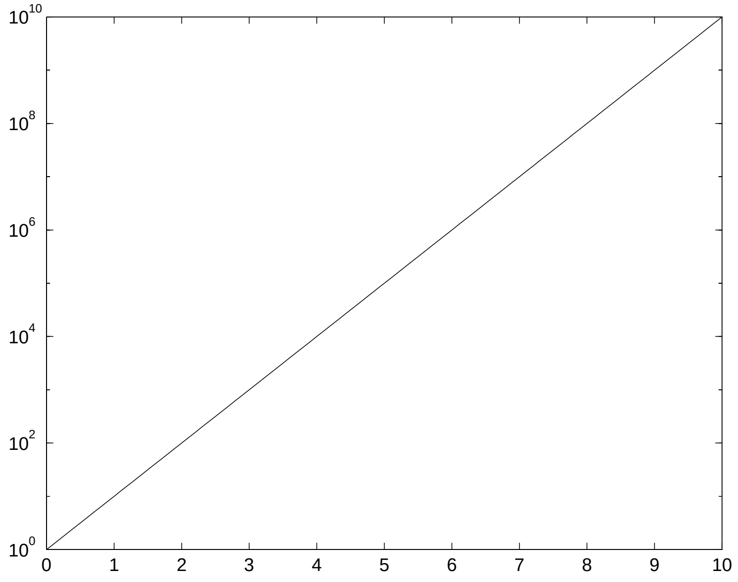
You can mix `Xn,Yn` pairs with `Xn,Yn,`*LineSpec* triples, for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

## Examples

A simple `semilogy` plot is:

```
x = 0:.1:10;
semilogy(x,10.^x)
```



## See Also

`line`, `LineSpec`, `loglog`, `plot`

**Purpose** Set object properties

**Syntax**

```
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv...)
set(H,pn,<m-by-n cell array>)
a= set(h)
a= set(0,'Factory')
a= set(0,'FactoryObjectTypePropertyName')
a= set(h,'Default')
a= set(h,'DefaultObjectTypePropertyName')
<cell array> = set(h,'PropertyName')
```

**Description**

`set(H,'`*PropertyName*`',PropertyValue,...)` sets the named properties to the specified values on the object(s) identified by `H`.

`set(H,a)` sets the named properties to the specified values on the object(s) identified by `H`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

`set(H,pn,pv,...)` sets the named properties specified in the cell array `pn` to the corresponding value in the cell array `pv` for all objects identified in `H`.

`set(H,pn,<m-by-n cell array>)` sets *n* property values on each of *m* graphics objects, where *m* `= length(H)` and *n* is equal to the number of property names contained in the cell array `pn`. This allows you to set a given group of properties to different values on each object.

`a = set(h)` returns the user-settable properties and possible values for the objectidentified by `h`. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

`a = set(0,'Factory')` returns the properties whose defaults are user settable for all objects and lists possible values for each property. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = set(0,'Factory`*`ObjectTypePropertyName`*`')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `Factory`*`ObjectTypePropertyName`* is the word `Factory` concatenated with the object type (e.g., `Axes`) and the property name (e.g., `CameraPosition`).

`a = set(h,'Default')` returns the names of properties having default values set on the object identified by `h`. `set` also returns the possible values if they are strings. `h` must be scalar.

`a = set(h,'Default`*`ObjectTypePropertyName`*`')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `Default`*`ObjectTypePropertyName`* is the word `Default` concatenated with the object type (e.g., `Axes`) and the property name (e.g., `CameraPosition`). For example, `DefaultAxesCameraPosition`. `h` must be scalar.

`pv = set(h,'`*`PropertyName`*`')` returns the possible values for the named property. If the possible values are strings, `set` returns each in a cell of the cell array, `pv`. For other properties, `set` returns an empty cell array. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

## Remarks

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

## Examples

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called `active`, which contains a set of property definitions used for the Uicontrol objects in a particular Figure. When this Figure becomes the current Figure, MATLAB changes colors and enables the controls:

```
active.BackgroundColor = [.7 .7 .7];
active.Enable = 'on';
active.ForegroundColor = [0 0 0];

if gcf == control_fig_handle
   set(findobj(control_fig_handle,'Type','uicontrol'),active)
end
```

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties:

```
PropName(1) = {'BackgroundColor'};
PropName(2) = {'Enable'};
PropName(3) = {'ForegroundColor'};
```

These statements define a cell array containing three values for each of three objects. (i.e., a 3-by-3 cell array):

```
PropVal(1,1) = {[.5 .5 .5]};
PropVal(1,2) = {'off'};
PropVal(1,3) = {[.9 .9 .9]};

PropVal(2,1) = {[1 0 0]};
PropVal(2,2) = {'on'};
PropVal(2,3) = {[1 1 1]};

PropVal(3,1) = {[.7 .7 .7]};
PropVal(3,2) = {'on'};
PropVal(3,3) = {[0 0 0]};
```

Now pass the arguments to set,

```
set(H,PropName,PropVal)
```

where `length(H) == 3` and each element is the handle to a Uicontrol.

**See Also** `findobj`, `gca`, `gcf`, `gco`, `gcbo`, `get`

## Purpose

Set color shading properties

## Syntax

```
shading flat
shading faceted
shading interp
```

## Description

The shading function controls the color shading of Surface and Patch graphics objects.

`shading flat` sets each mesh line segment, Surface face, or Patch face to a constant color determined by the color values at the end points of the segment, or the corners of the Surface face or Patch.

`shading faceted` sets the shading to `flat` with individual faces outlined in black. This is the default shading mode.

`shading interp` varies the color in each line segment, Surface face, or Patch face by interpolating the colormap index or true color value across the face or line.

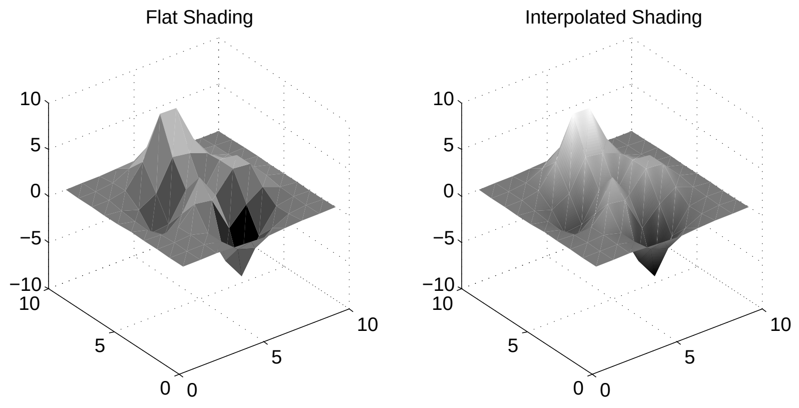
## Examples

Compare a flat-shaded sphere with a Gouraud-shaded sphere:

```
colormap gray

subplot(1,2,1)
surf(peaks(10));
axis square
shading flat
title('Flat Shading')

subplot(1,2,2)
surf(peaks(10));
axis square
shading interp
title('Interpolated Shading')
```



**Algorithm**

`shading` sets the `EdgeColor` and `FaceColor` properties of all Surface and Patch graphics objects in the current Axes. `shading` sets the appropriate values, depending on whether the Surface or Patch objects represent meshes or solid surfaces.

**See Also**

`fill`, `fill3`, `hidden`, `mesh`, `patch`, `pcolor`, `surf`

The `EdgeColor` and `FaceColor` properties for Surface and Patch graphics objects.

## Purpose

Volumetric slice plot

## Syntax

```
slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
h = slice(...)
```

## Description

`slice` displays volumetric data. You indicate the portion of the data you want to view by specifying a slice plane or surface.

`slice(V,sx,sy,sz)` draws data in the volume `V` for the slices defined by `sx`, `sy`, and `sz`. `V` is an *m*-by-*n*-by-*p* volume array containing data values at the default location `X = 1:n`, `Y = 1:m`, `Z = 1:p`. Each element in the vectors `sx`, `sy`, and `sz` defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

`slice(X,Y,Z,V,sx,sy,sz)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (e.g., produced by the function `meshgrid`).

`slice(V,XI,YI,ZI)` draws data in the volume `V` for the slices defined by `XI`, `YI`, and `ZI`. `XI`, `YI`, and `ZI` are matrices that define a surface and the volume is evaluated at the surface points. `XI`, `YI`, and `ZI` must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (e.g., produced by the function `meshgrid`).

`slice(...,'`*`method`*`')` specifies the interpolation method. `'`*`method`*`'` is `'linear'`, `'cubic'`, or `'nearest'`. `'linear'` is the default.

- `'linear'` specifies trilinear interpolation.
- `'cubic'` specifies tricubic interpolation.
- `'nearest'` specifies nearest neighbor interpolation.

`h = slice(...)` returns a vector of handles to Surface graphics objects.

## Remarks

The color drawn at each point is determined by interpolation into the volume `V`.

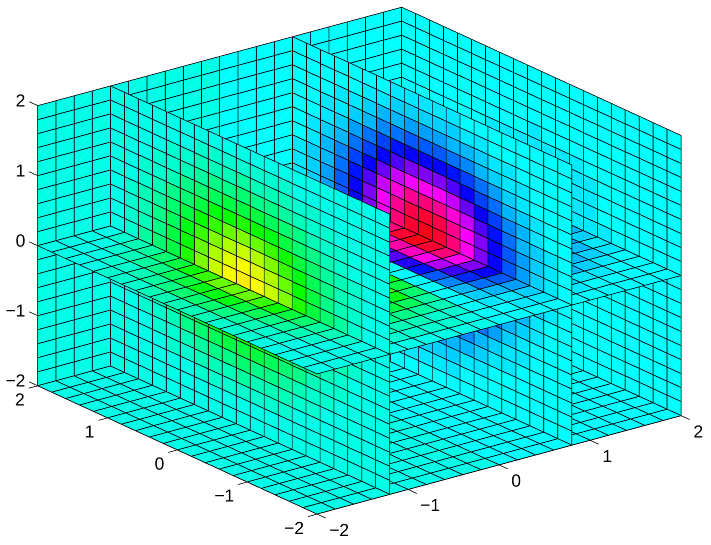
## Examples

Visualize the function

$$v = xe^{(-x^2 - y^2 - z^2)}$$

over the range $-2 \le x \le 2$, $-2 \le y \le 2$, $-2 \le z \le 2$:

```
[x,y,z] = meshgrid(-2:.2:2, -2:.25:2, -2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2 .8 2]; yslice = 2; zslice = [-2 0];
slice(x,y,z,v,xslice,yslice,zslice)
```



## See Also

`meshgrid`

The `interp3` function in the online MATLAB Function Reference.

# sphere

**Purpose** Generate sphere

**Syntax**

```
sphere
sphere(n)
[X,Y,Z] = sphere(...)
```

**Description** The `sphere` function generates the *x*-, *y*-, and *z*--coordinates of a unit sphere for use with `surf` and `mesh`.

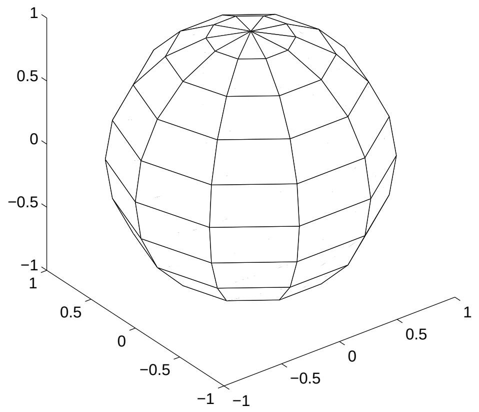
`sphere` generates a sphere consisting of 20-by-20 faces.

`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current Figure.

`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are `(n+1)`–by–`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

**Examples** Generate and plot a sphere:

```
[X,Y,Z] = sphere(10);
mesh(X,Y,Z)
```



**See Also** `cylinder`

## Purpose

Spin colormap

## Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

## Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately 10∗`t` seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB over a network).

`spinmap(t,inc)` rotates the colormap for approximately 10∗`t` seconds and specifies an increment `inc` by which the colormap shifts. When `inc` is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., –2) rotates the colormap in a negative direction.

`spinmap('inf')` rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl-C**.

## See Also

`colormap`

# stairs

**Purpose** Stairstep plot

**Syntax**

```
stairs(Y)
stairs(X,Y)
stairs(...,LineSpec)
[xb,yb] = stairs(Y)
[xb,yb] = stairs(X,Y)
```

**Description** Stairstep plots are useful for drawing time-history plots of digitally sampled data systems.

`stairs(Y)` draws a stairstep plot of the elements of `Y`. When `Y` is a vector, the *x*-axis scale ranges from `1` to `size(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to the number of rows in `Y`.

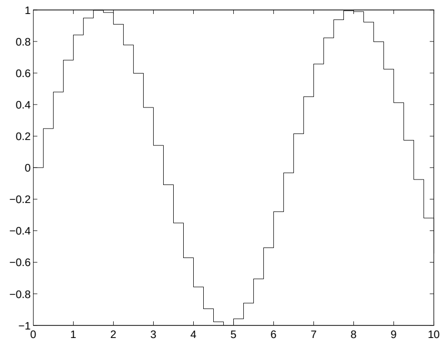
`stairs(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

`stairs(...,`*`LineSpec`*`)` specifies a line style, marker symbol, and color for the plot.

`[xb,yb] = stairs(Y)` and `[xb,yb] = stairs(x,Y)` do not draw graphs, but return vectors `xb` and `yb` such that `plot(xb,yb)` plots the stairstep graph.

**Examples** Create a stairstep plot of a sine wave:

```
x = 0:.25:10;
stairs(x,sin(x))
```



**See Also** `bar`, `hist`

## Purpose

Plot discrete sequence data

## Syntax

```
stem(Y)
stem(X,Y)
stem(...,'fill')
stem(...,LineSpec)
h = stem(...)
```

## Description

A two-dimensional stem plot displays data as lines extending from the *x*-axis. A circle (the default) or other marker symbol whose *y*-position represents the data value, terminates each stem.

`stem(Y)` plots the data sequence `Y` as stems that extend from equally spaced and automatically generated values along the *x*-axis. When `Y` is a matrix, `stem` plots all elements in a row against the same *x* value.

`stem(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

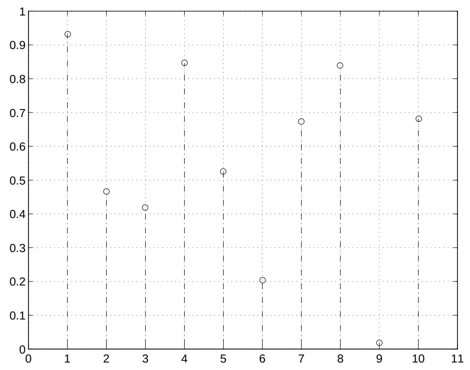
`stem(...,`*`LineSpec`*`)` specifies the line style, marker symbol, and color for the stem plot.

`h = stem(...)` returns handles to Line graphics objects.

## Examples

Create a stem plot of 10 random numbers:

```
Y = rand(1,10)
stem(Y,'—.')
axis ([0 11 0 1])
```



## See Also

`bar`, `plot`, `stairs`, `stem3`

## Purpose

Plot three-dimensional discrete sequence data

## Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineSpec)
h = stem3(...)
```

## Description

Three-dimensional stem plots display lines extending from the *xy*-plane. A circle (the default) or other marker symbol whose *z*-position represents the data value, terminates each stem.

`stem3(Z)` plots the data sequence `Z` as stems that extend from the *xy*-plane. *x* and *y* are generated automatically. When `Z` is a row vector, `stem3` plots all elements at equally spaced *x* values against the same *y* value. When `Z` is a column vector, `stem3` plots all elements at equally spaced *y* values against the same *x* value.

`stem3(X,Y,Z)` plots the data sequence `Z` at values specified by `X` and `Y`. `X`, `Y`, and `Z` must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the circle at the end of the stem.

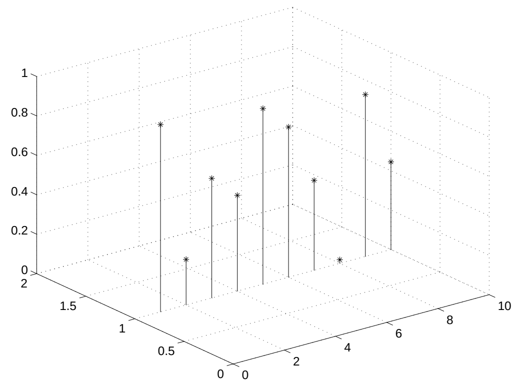
`stem3(...,`*`LineSpec`*`)` specifies the line style, marker symbol, and color for the stems.

`h = stem3(...)` returns handles to Line graphics objects.

## Examples

Create a three-dimensional stem plot of 10 random numbers:

```
Z = rand(1,10)
stem3(Z,'—*')
```



## See Also

`bar`, `plot`, `stairs`, `stem`

## Purpose

Create and control multiple Axes

## Syntax

```
subplot(m,n,p)
subplot(h)
subplot('Position',[left bottom width height])
h = subplot(...)
```

## Description

`subplot` divides the current Figure into rectangular panes that are numbered row-wise. Each pane contains an Axes. Subsequent plots are output to the current pane.

`subplot(m,n,p)` creates an Axes in the `p`-th pane of a Figure divided into an `m`-by-`n` matrix of rectangular panes. The new Axes becomes the current Axes.

`subplot(h)` makes the Axeswith handle `h` current for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an Axes at the position specified by a four-element vector. `left`, `bottom`, `width`, and `height` are in normalized coordinates in the range from 0.0 to 1.0.

`h = subplot(...)` returns the handle to the new Axes.

## Remarks

If a `subplot` specification causes a new Axes to overlap an existing Axes, `subplot` deletes the existing Axes. `subplot(1,1,1)` or `clf` deletes all Axes objects and returns to the default `subplot(1,1,1)` configuration.

## Examples

To plot `income` in the top half of a Figure and `outgo` in the bottom half,

```
income = [3.2 4.1 5.0 5.6];
outgo = [2.5 4.0 3.35 4.9];
subplot(2,1,1); plot(income)
subplot(2,1,2); plot(outgo)
```

## See Also

`axes`, `cla`, `clf`, `figure`, `gca`

## surf, surfc

**Purpose**

3-D shaded surface plot

**Syntax**

```
surf(Z)
surf(X,Y,Z)
surf(...,C)
surfc(...)

h = surf(...)
h = surfc(...)
```

**Description**

You use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `Z` or `C`.

`surf(Z)` creates a a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data as well as Surface height, so color is proportional to surface height.

`surf(X,Y,Z)` creates a shaded Surface using `Z` for the color data as well as Surface height. `X` and `Y` are vectors or matrices defining the *x* and *y* components of a Surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the Surface faces are $(X(j), Y(i), Z(i, j))$ triples.

`surf(...,C)` creates a shaded surface, with color defined by `C`. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surfc(...)` draws a contour plot beneath the Surface.

`h = surf(...)` and `h = surfc(...)` return a handle to a Surface graphics object.

**Algorithm**

Abstractly, a parametric surface is parametrized by two independent variables, `i` and `j`, which vary continuously over a rectangle, for example, `1 ≤ i ≤ m` and `1 ≤ j ≤ n`. The three functions, `x(i,j)`, `y(i,j)`, and `z(i,j)` specify the surface. When `i` and `j` are integer values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y` and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors:

```
          i–1,j
            |
 i,j–1 – i,j – i,j+1
            |
          i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways – at the vertices or at the centers of each patch. In this general setting, the surface need not be a single valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
  (i,j)    –   (i,j+1)
    |    C(i,j)  |
 (i+1,j)  –  (i+1,j+1)
```

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored, Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surf` and `surfc` functions specify the view point using `view(3)`.

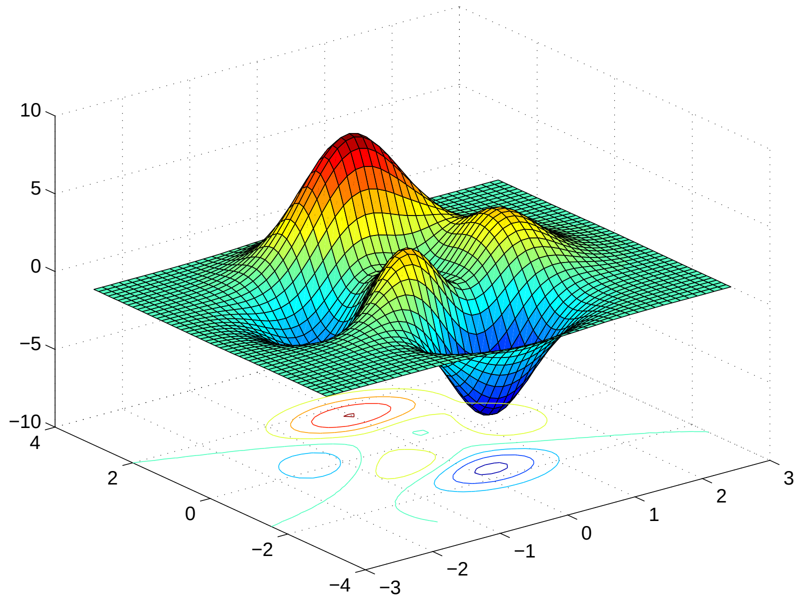
The range of X, Y, and Z, or the current setting of the Axes XLimMode, YlimMode, and ZlimMode properties (also set by the axis function) determine the axis labels.

The range of C, or the current setting of the Axes CLim and ClimMode properties (also set by the caxis function) determine the color scaling. The scaled color values are used as indices into the current colormap.

## Examples

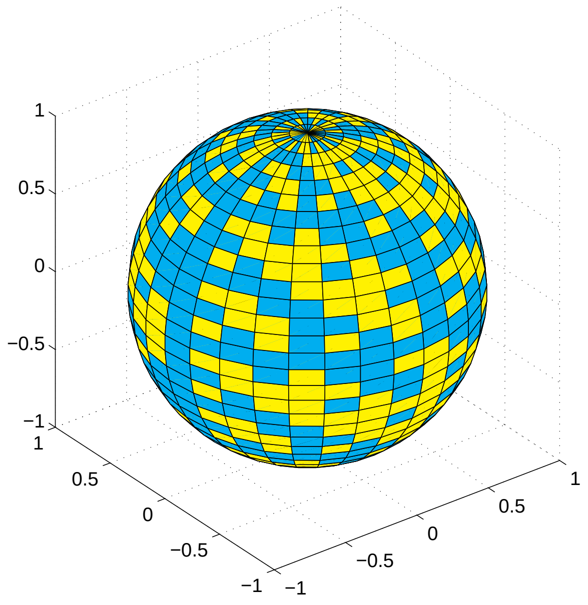
Display a surface and contour plot of the peaks surface:

```
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
surfc(X,Y,Z)
axis([-3 3 -3 3 -10 5])
```



Color a sphere with the pattern of +1s and -1s in a Hadamard matrix:

```
k = 5;
n = 2^k–1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
surf(x,y,z,c);
colormap([1  1  0; 0  1  1])
set(gca,'Stretch','off')
```



**See Also**

axis, caxis, colormap, contour, mesh, pcolor, shading, view

Properties for Surface graphics objects.

# surface

## Purpose

Create Surface object

## Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(...'PropertyName',PropertyValue,...)
h = surface(...)
```

## Description

`surface` is the low-level function for creating Surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the *x*- and *y*- coordinates and the value of each element as the z-coordinate.

`surface(Z)` plots the Surface specified by the matrix `Z`. Here, `Z` is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the Surface specified by `Z` and colors it according to the data in `C` (see "Examples").

`surface(X,Y,Z,C)` plots the parametric surface specified by `X`, `Y` and `Z`, with color specified by `C`.

`surface(X,Y,Z)` uses `C = Z`, so color is proportional to surface height above the *x*-*y* plane.

`surface(x,y,Z)`, `surface(x,y,Z,C)` replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the Surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` reference page.

`surface(...'`*`PropertyName`*`',PropertyValue,...)` follows the `X`, `Y`, `Z`, and `C` arguments with property name/property value pairs to specify additional Surface properties. These properties are described in the "Surface Properties" section.

`h = surface(...)` returns a handle to the created Surface object.

**Remarks**

Unlike high-level area creation functions, such as surf or mesh, surface does not respect the settings of the Figure and Axes NextPlot properties. It simply adds the Surface object to the current Axes.

If you do not specify separate color data (C), MATLAB uses the matrix (Z) to determine the coloring of the Surface. In this case, color is proportional to values of Z. You can specify a separate matrix to color the Surface independently of the data defining the area of the Surface.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

surface provides convenience forms that allow you to omit the property name for the XData, YData, ZData, and CData properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to:

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified,

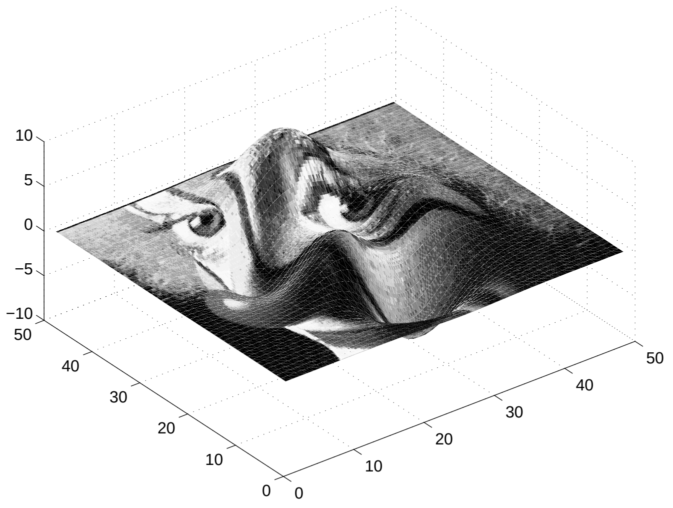
```
surface('XData',[1:size(Z,2)],...
        'YData',[1:size(Z,1)],...
        'ZData',Z,...
        'CData',Z)
```

The axis, caxis, colormap, hold, shading, and view commands set graphics properties that affect Surfaces. You can also set and query Surface property values after creating them using the set and get commands.

**Example**

This example creates a Surface using the peaks M-file to generate the data and colors it using the clown Image. The ZData is a 49-by-49 element matrix, while the CData is a 200-by-320 matrix. You must set the FaceColor to texturemap to use ZData and CData of different dimensions.

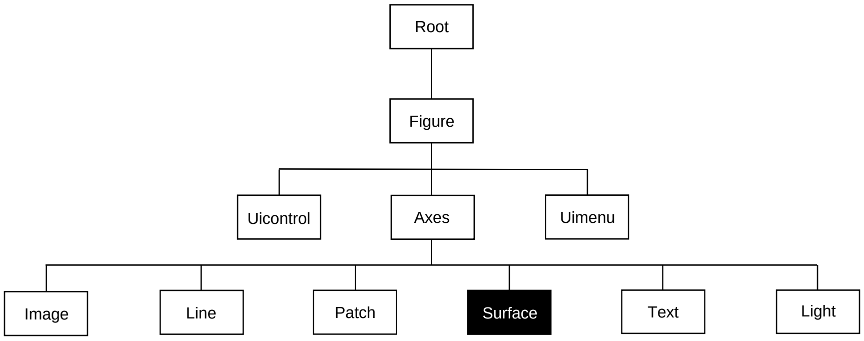
```
load clown
surface(peaks,flipud(X),...
        'FaceColor','texturemap',...
        'EdgeColor','none',...
        'CDataMapping','direct')
colormap(map)
view(3)
```



Note the use of the surface(Z,C) convenience form combined with property name/property value pairs.

Since the clown data (X) is typically viewed as an Image, which MATLAB normally displays with 'ij' axis numbering and direct CDataMapping, this example reverses the data in the vertical direction using flipud and sets the CDataMapping property to direct.

## Object Hierarchy



### Setting Default Properties

You can set default Surface properties on the Axes, Figure, and Root levels:

```
set(0,'DefaultSurfaceProperty',PropertyValue...)
set(gcf,'DefaultSurfaceProperty',PropertyValue...)
set(gca,'DefaultSurfaceProperty',PropertyValue...)
```

Where *Property* is the name of the Surface property whose default value you want to set and `PropertyValue` is the value you are specifying.

## Surface Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`AmbientStrength`** scalar >= 0 and <= 1

*Strength of ambient light*. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible Light object in the Axes for the ambient light to be visible. The Axes `AmbientColor` property sets the color of the ambient light, which is therefore the same on all objects in the Axes.

You can also set the strength of the diffuse and specular contribution of Light objects. See the `DiffuseStrength` and `SpecularStrength` properties.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

*Button press callback routine.* A callback routine that executes whenever you press a mouse button while the pointer is over the Surface object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`CData`** matrix

*Vertex colors*. A matrix of values that specify the color at every point in `ZData`. If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`. In this case, MATLAB maps `CData` to conform the Surface defined by `ZData`.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to linearly map into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property.

True color defines an RGB value for each vertex. If the coordinate data (`XData` for example) are contained in an *m*-by-*n* matrix, then `CData` must be an *m*-by-*n*-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

On computer displays that cannot display true color (e.g., 8-bit displays), MATLAB uses dithering to approximate the RGB triples using the colors in the Figure's `Colormap` and `Dithermap`, which defaults to `colorcube(64)`. You can also specify your own dithermap.

**CDataMapping** {scaled} | direct

*Direct or scaled color mapping*. This property determines how MATLAB interprets indexed color data used to color the Surface. (If you use true color specification for CData, this property has no effect.)

- scaled – transform the color data to span the portion of the colormap indicated by the Axes CLim property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.
- direct – use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to length(colormap). MATLAB maps values less than 1 to the first color in the colormap, and values greater than length(colormap) to the last color in the colormap. Values with a decimal portion are fixed to the nearest, lower integer.

**Children** matrix of handles

Always the empty matrix; Surface objects have no children.

**Clipping** {on} | off

*Clipping to Axes rectangle*. When Clipping is on, MATLAB does not display any portion of the Surface that is outside the Axes rectangle.

**CreateFcn** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Surface object. You must define this property as a default value for Surfaces. For example, the statement,

```
set(0,'DefaultSurfaceCreateFcn',...
      'set(gcf,''DitherMap'',my_dithermap)')
```

defines a default value on the Root level that sets the Figure DitherMap property whenever you create a Surface object. MATLAB executes this routine after setting all Surface properties. Setting this property on an existing Surface object has no effect.

The handle of the object whose CreateFcn is being executed is accessible only through the Root CallbackObject property, which can be queried using gcbo.

**DeleteFcn** string

*Delete Surface callback routine*. A callback routine that executes when you delete the Surface object (e.g., when you issue a `delete` command or clear the Axes or Figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**DiffuseStrength** scalar >= 0 and <= 1

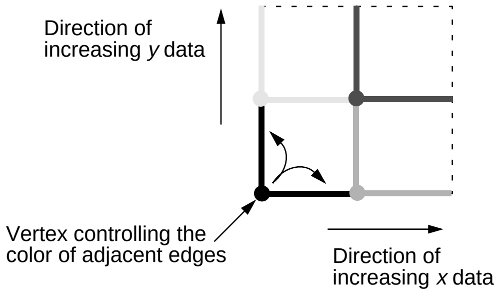
*Intensity of diffuse light*. This property sets the intensity of the diffuse component of the light falling on the Surface. Diffuse light comes from Light objects in the Axes.

You can also set the intensity of the ambient and specular components of the light on the Surface object. See the `AmbientStrength` and `SpecularStrength` properties.

**EdgeColor** `{ColorSpec} | none | flat | interp`

*Color of the Surface edge*. This property determines how MATLAB colors the edges of the individual faces that make up the Surface:

- `ColorSpec` — A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for edges. The default `EdgeColor` is black. See the *ColorSpec* reference page for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge:


Direction of increasing *y* data
Vertex controlling the color of adjacent edges
Direction of increasing *x* data

- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

**EdgeLighting** `{none} | flat | gouraud | phong`

*Algorithm used for lighting calculations*. This property selects the algorithm used to calculate the effect of Light objects on Patch edges. Choices are:

- `none` – Lights do not affect the edges of this object.
- `flat` – The effect of Light objects is uniform across each edge of the Surface.
- `gouraud` – The effect of Light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` – The effect of Light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**EraseMode** `{normal} | none | xor | background`

*Erase mode*. This property controls the technique MATLAB uses to draw and erase Surface objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the Surface when it is moved or destroyed.
- `xor` — Draw and erase the Surface by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. Erasing the Surface does not damage the color of the objects beneath it. However, Surface color depends on the color of the screen beneath it and is correctly colored only when over the Axes background color, or Figure background color if the Axes color is set to `none`.
- `background` — Erase the Surface by drawing it in the Axes' background color. This damages objects that are behind the erased object, but Surface objects are always properly colored.

**FaceColor** `ColorSpec | none | {flat} | interp`

*Color of the Surface face*. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for faces. See the `ColorSpec` reference page for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the Surface. The color data at the first vertex determines the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the Surface. MATLAB transforms the color data so that it conforms to the Surface. (See "Examples")

**FaceLighting** `{none} | flat | gouraud | phong`

*Algorithm used for lighting calculations*. This property selects the algorithm used to calculate the effect of Light objects on the Surface. Choices are:

- `none` – Lights do not affect the faces of this object.
- `flat` – The effect of Light objects is uniform across the faces of the Surface. Select this choice to view faceted objects.
- `gouraud` – The effect of Light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` – The effect of Light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

**HandleVisibility** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to `on` to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`Interruptible`** `{on} | off`

*Callback routine interruption mode*. The `Interruptible` property controls whether a Surface callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in theroutine. See the `EventQueue` property for related information.

**`LineStyle`** `{–} | –– | : | –. | none`

*Edge line type*. This property determines the line style used to draw Surface edges. The available line styles are:

| Symbol | Line Style |
|---|---|
| – | solid line (default) |
| - - | dashed line |
| : | dotted line |
| –. | dash-dot line |
| none | no line |

**`LineWidth`** scalar

*Edge line width*. The width of the lines in points used to draw Surface edges. The default width is 0.5 points (1 point = 1/72 inch).

**`Marker`** marker symbol (see table)

Marker symbol. The `Marker` property specifies symbols that display at vertices. You can set values for the `Marker` property independently from the `LineStyle` property.

The available markers are:

| Marker Specifier | Description |
|---|---|
| + | plus sign |
| o | circle |
| * | asterisk |
| . | point |
| x | cross |
| square | square |
| diamond | diamond |
| ^ | upward pointing triangle |
| v | downward pointing triangle |
| > | right pointing triangle |
| < | left pointing triangle |
| pentagram | five-pointed star |
| hexagram | six-pointed star |
| none | no marker (default) |

**MarkerEdgeColor** `ColorSpec | none | {auto}`

*Marker edge color*. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` defines a single color to use for the edge (see the `ColorSpec` reference page).
- `none` specifies no color, which makes nonfilled markers invisible.
- `auto` uses the same color as the `EdgeColor` property.

**MarkerFaceColor** `ColorSpec | {none} | auto`

*Marker face color*. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` defines a single color to use for all marker on the Surface (see the `ColorSpec` reference page).
- `none` makes the interior of the marker transparent, allowing the background to show through.
- `auto` uses the `CData` for the vertex located by the marker to determine the color.

**MarkerSize** size in points.

*Marker size*. A scalar specifying the marker size, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

**MeshStyle** `{both} | row | column`

*Row and column lines*. This property specifies whether to draw all edge lines or just row or column edge lines.

- `both` draws edges for both rows and columns.
- `row` draws row edges only.
- `column` draws column edges only.

**NormalMode** `{auto} | manual`

*MATLAB-generated or user-specified normal vectors*. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

**`Parent`** handle

*Surface's parent object*. The parent of a Surface object is the Axes in which it is displayed. You can move a Surface object to another Axes by setting this property to the handle of the new parent.

**`Selected`** `on | off`

*Is object selected*. When this property is `on`. MATLAB displays a dashed bounding box around the Surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**`SelectionHighlight`** `{on} | off`

*Objects highlight when selected*. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the Surface. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

**`SpecularColorReflectance`**scalar in the range 0 to 1

*Color of specularly reflected light*. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the Light object `Color` property). The proportions vary linearly for values in between.

**`SpecularExponent`** scalar >= 1

*Harshness of specular reflection*. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

**`SpecularStrength`** scalar >= 0 and <= 1

*Intensity of specular light*. This property sets the intensity of the specular component of the light falling on the Surface. Specular light comes from Light objects in the Axes.

You can also set the intensity of the ambient and diffuse components of the light on the Surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**`Type`** string (read only)

*Class of the graphics object*. The class of the graphics object. For Surface objects, `Type` is always the string `'surface'`.

**`UserData`** matrix

*User-specified data*. Any matrix you want to associate with the Surface object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

**`VertexNormals`** vector or matrix

*Surface normal vectors*. This property contains the vertex normals for the Surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

**`Visible`** `{on} | off`

*Surface object visibility*. By default, all Surfaces are visible. When set to `off`, the Surface is not visible, but still exists and you can query and set its properties.

**`XData`** vector or matrix

X-coordinates. The *x*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of columns as `ZData`.

**`YData`** vector or matrix

*Y-coordinates*. The *y*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of rows as `ZData`.

**`ZData`** vector or matrix

*Z-coordinates*. *Z*-position of the surface points. See the “Description” section for more information.

## See Also

`ColorSpec`, `mesh`, `patch`, `pcolor`, `surf`

## Purpose

Surface plot with colormap-based lighting

## Syntax

```
surfl(Z)
surfl(X,Y,Z)
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

## Description

The `surfl` function displays a shaded Surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and `surfl(X,Y,Z)` create three-dimensional shaded Surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the *x*, *y*, and *z* components of a Surface.

`surfl(...,’light’)` produces a colored lighted surface using the Light object. This produces different results than the default lighting method, surfl(...,'cdata'), which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a Surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka kd ks shine]` and defaults to `[.55,.6,.4,10]`.

`h = surfl(...)` returns a handle to a Surface graphics object.

## Remarks

For smoother color transitions, use colormaps that have linear intensity variations (e.g., `gray`, `copper`, `bone`, `pink`).

The ordering of points in the `X`, `Y`, and `Z` matrices define the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use `surfl(X',Y',Z')`. Due to the way surface normal vectors are computed, `surfl` requires matrices that are at least 3-by-3.

## Examples

View the peaks function using colormap-based lighting:

```
[x,y] = meshgrid(–3:1/8:3);
z = peaks(x,y);
surfl(x,y,z);
shading interp
colormap(gray);
axis([–3  3  –3  3  –8  8])
```

To plot a lighted surface from a view direction other than the default:

```
cla
hold on
view([10 10])
surfl(peaks)
shading interp
colormap(gray)
hold off
```

## See Also

`colormap`, `shading`, `light`

## Purpose

Compute and display 3-D surface normals

## Syntax

```
surfnorm(Z)
surfnorm(X,Y,Z)
[Nx,Ny,Nz] = surfnorm(...)
```

## Description

The `surfnorm` function computes surface normals for the Surface defined by `X`, `Y`, and `Z`. The surface normals are unnormalized and valid at each vertex. Normals are not shown for Surface elements that face away from the viewer.

`surfnorm(Z)` and `surfnorm(X,Y,Z)` plot a Surface and its surface normals. `Z` is a matrix that defines the z component of the Surface. `X` and `Y` are vectors or matrices that define the *x* and *y* components of the Surface.

`[Nx,Ny,Nz] = surfnorm(...)` returns the components of the three-dimensional surface normals for the Surface.

## Remarks

The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

```
surfnorm(X',Y',Z')
```

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a Surface.

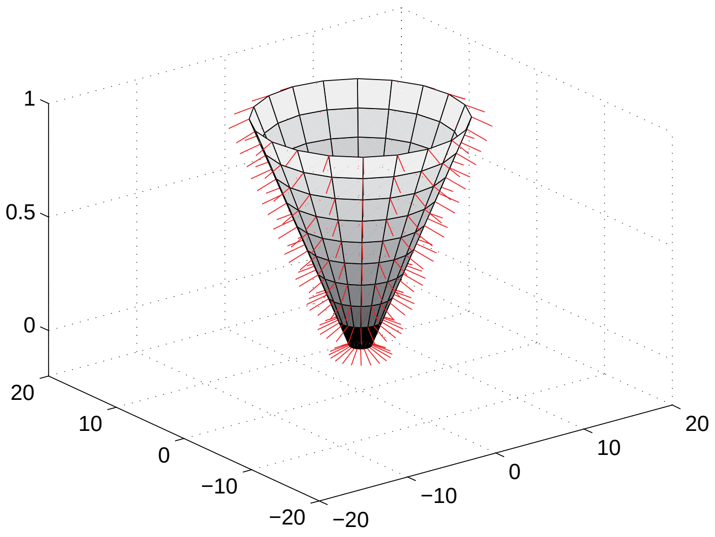
## Algorithm

The surface normals are based on a bicubic fit of the data in `X`, `Y`, and `Z`. For each vertex, diagonal vectors are computed and crossed to form the normal.

## Examples

Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
```



**See Also** `surfl`

## Purpose

Set graphics terminal type

## Syntax

```
terminal
terminal('type')
```

## Description

To add terminal-specific settings (e.g., escape characters, line length), edit the file `terminal.m`.

`terminal` displays a menu of graphics terminal types, prompts for a choice, then configures MATLAB to run on the specified terminal.

`terminal('type')` accepts a terminal type string. Valid '*type*' strings are

| Type | Description |
|---|---|
| `tek401x` | Tektronix 4010/4014 |
| `tek4100` | Tektronix 4100 |
| `tek4105` | Tektronix 4105 |
| `retro` | Retrographics card |
| `sg100` | Selanar Graphics 100 |
| `sg200` | Selanar Graphics 200 |
| `vt240tek` | VT240 & VT340 Tektronix mode |
| `ergo` | Ergo terminal |
| `graphon` | Graphon terminal |
| `citoh` | C.Itoh terminal |
| `xtermtek` | xterm, Tektronix graphics |
| `wyse` | Wyse WY-99GT |
| `kermit` | MS-DOS Kermit 2.23 |

| Type | Description (Continued) |
|---|---|
| hp2647 | Hewlett-Packard 2647 |
| versa | Macintosh with VersaTerm (Tektronix 4010/4014) |
| versa4100 | Macintosh with VersaTerm (Tektronix 4100) |
| versa4105 | Color/grayscale Macintosh with VersaTerm (Tektronix 4105) |
| hds | Human Designed Systems |

## Purpose

Create Text object in current Axes

## Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(...'PropertyName',PropertyValue...)
h = text(...)
```

## Description

`text` is the low-level function for creating Text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point (`x,y`).

`text(x,y,z,'string')` adds the string in 3-D coordinates.

`text(x,y,z,'string','`*`PropertyName`*`',PropertyValue....)` adds the string in quotes to location defined by the coordinates and uses the values for the specified Text properties.

`text('`*`PropertyName`*`',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

`h = text(..)` returns a column vector of handles to Text objects, one handle per object. All forms of the `text` function optionally return this output argument.

## Remarks

Specify the Text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current Axes (see "Examples"). The `Extent`, `VerticalAlignment`, and `HorizontalAlignment` properties control the positioning of the character string with regard to the Text location point.

If the coordinates are vectors, `text` writes the string at all locations defined by the list of points. If the character string is an array the same length as `x`, `y`, and `z`, `text` writes the corresponding row of the string array at each point specified.

When specifying strings for multiple Text objects, string can be a cell array of strings, a padded string matrix, or a string vector using vertical slash characters (‘`|`’) as separators, and each Text object will be assigned a different element of the specified string. When specifying the string for a single Text object, cell arrays of strings and padded string matrices result in a Text object with a multiline string, while vertical slash characters are not interpreted as separators, and result in a single line string containing vertical slashes.

While text is a low-level function that accepts property name/property value pairs as input arguments, the convince form,

```
text(x,y,z,'string')
```

is equivalent to:

```
text('XData',x,'YData',y,'ZData',z,'String','string')
```

You can specify other properties only as property name/property value pairs. See the “Text Properties” section for a description of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

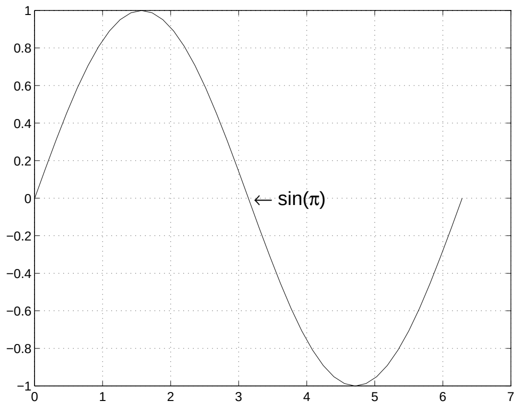
`text` does not respect the setting of the Figure or Axes `NextPlot` property. This allows you to add Text objects to an existing Axes without setting `hold` to `on`.

## Examples

The statements,

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow sin(\pi)','FontSize',18)
```

annotate the point at `(pi,0)` with the string ¨`sin(π)`:



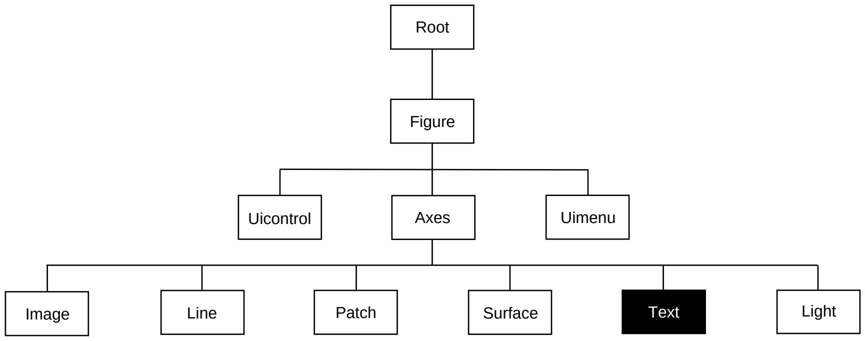
The statement,

```
text(x,y,'\ite^{i\omega\tau} = cos(\omega\tau) + i
sin(\omega\tau)')
```

uses imbedded LaTeX sequences to produce:

$$e^{i\omega\tau} = cos(\omega\tau) + i\ sin(\omega\tau)$$

## Object Hierarchy



### Setting Default Properties

You can set default Text properties on the Axes, Figure, and Root levels:

```
set(0,'DefaulttextProperty',PropertyValue...)
set(gcf,'DefaulttextProperty',PropertyValue...)
set(gca,'DefaulttextProperty',PropertyValue...)
```

Where *Property* is the name of the Text property and PropertyValue is the value you are specifying.

## Text Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**BusyAction** cancel | {queue}

*Callback routine interruption*. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

*Button press callback routine.* A callback routine that executes whenever you press a mouse button while the pointer is over the Text object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

**`Children`** matrix (read only)

The empty matrix; Text objects have no children.

**`Clipping`** `on | {off}`

*Clipping mode.* When `Clipping` is `on`, MATLAB does not display any portion of the Text that is outside the Axes.

**`Color`** `ColorSpec`

*Text color*. A three-element RGB vector or one of MATLAB's predefined names, specifying the Text color. The default value for `Color` is white. See the `ColorSpec` reference page for more information on specifying color.

**`CreateFcn`** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Text object. You must define this property as a default value for Text. For example, the statement,

```
set(0,'DefaultTextCreateFcn',...
    'set(gcf,''Pointer'',’'crosshair'')')
```

defines a default value on the Root level that sets the Figure `Pointer` property to a crosshair whenever you create a Text object. MATLAB executes this routine after setting all Text properties. Setting this property on an existing Text object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Text callback routine.* A callback routine that executes when you delete the Text object (e.g., when you issue a `delete` command or clear the Axes or Figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`EraseMode`** `{normal} | none | xor | background`

*Erase mode.* This property controls the technique MATLAB uses to draw and erase Text objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual object redraw is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the Text when it is moved or destroyed.
- `xor` — Draw and erase the Text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the Text is erased, it does not damage the objects beneath it. However, when Text is drawn in `xor` mode, its color depends on the color of the screen beneath it and is correctly colored only when over the Axes background color.
- `background` — Erase the Text by drawing it in the background color. This damages objects that are behind the erased Text, but Text is always properly colored.

**`Extent`** position rectangle (read only)

*Position and size of Text*. A four-element read-only vector that defines the size and position of the Text string:

```
[left,bottom,width,height]
```

`left` and `bottom` are the distance from the lower-left corner of the Axes rectangle to the lower-left corner of the Text `Extent` rectangle. `width` and `height` are the dimensions of the `Extent` rectangle. All measurements are in units specified by the `Units` property.

**`FontAngle`** `{normal} | italic | oblique`

*Character slant*. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

**`FontName`** string

*Font family*. A string specifying the name of the font to use for the Text object. To display and print properly, this must be a font that your system supports. The default font is Helvetica.

**`FontSize`** size in `FontUnits`

*Font size*. An integer specifying the font size to use for Text, in units determined by the `FontUnits` property. The default point size is 10 (1 point = 1/72 inch).

**`FontWeight`** `light | {normal} | demi | bold`

*Weight of Text characters*. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

**`FontUnits`** `{points} | normalized | inches | centimeters | pixels`

*Font size units*. MATLAB uses this property to determine the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the parent Axes. When you resize the Axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to `on` to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`HorizontalAlignment`** `{left} | center | right`

*Horizontal alignment of Text*. This property specifies the horizontal justification of the Text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property.

**Interpreter** {latex} | none

*Interpret LaTex instructions*. This property controls whether MATLAB interprets certain characters in the String property as LaTex instructions (default) or displays all characters literally. See the String property for a list of support LaTex instructions.

**Interruptible** {on} | off

*Callback routine interruption mode*. The Interruptible property controls whether a Text callback routine can be interrupted by subsequently invoked callback routines. Text objects have four properties that define callback routines: ButtonDownFcn, CreateFcn, and DeleteFcn. See the Executionqueue property for information on how MATLAB executes callback routines.

**Parent** handle

*Text object's parent*. The handle of the Text object's parent object. The parent of a Text object is the Axes in which it is displayed. You can move a Text object to another Axes by setting this property to the handle of the new parent.

**Position** [x,y,[z]]

*Location of Text*. A two- or three-element vector, [x y [z]], that specifies the location of the text in three dimensions. If you omit the z value, it defaults to 0. All measurements are in units specified by the Units property. Initial value is [0 0 0].

**Rotation** scalar (default = 0)

Text orientation. This property determines the orientation of the Text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

**Selected** on | {off}

*Is object selected*. When this property is on. MATLAB displays selection handles if the SelectionHighlight property is also on. You can, for example, define the ButtonDownFcn to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects highlight when selected*. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

**`String`** `string`

*The Text string*. Specify this property as a quoted string for single-line strings, or as a cell array of strings or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as linebreaks in Text strings, and are drawn as part of the Text string.

When the Text `Interpreter` property is `Tex` (the default), you can use a subset of Tex commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequence used to define them.

| Character Sequence | Symbol | Character Sequence | Symbol | Character Sequence | Symbol |
|---|---|---|---|---|---|
| \alpha | α | \upsilon | υ | \sim | ~ |
| \beta | β | \phi | φ | \leq | ≤ |
| \gamma | γ | \chi | χ | \infty | ∞ |
| \delta | δ | \psi | ψ | \clubsuit | ♣ |
| \epsilon | ε | \omega | ω | \diamondsuit | ♦ |
| \zeta | ζ | \Gamma | Γ | \heartsuit | ♥ |
| \eta | η | \Delta | Δ | \spadesuit | ♠ |
| \theta | θ | \Theta | Θ | \leftrightarrow | ↔ |
| \vartheta | υ | \Lambda | Λ | \leftarrow | ← |
| \iota | ι | \Xi | Ξ | \uparrow | ↑ |
| \kappa | κ | \Pi | Π | \rightarrow | → |
| \lambda | λ | \Sigma | Σ | \downarrow | ↓ |
| \mu | µ | \Upsilon | ϒ | \circ | ° |

| Character Sequence | Symbol | Character Sequence | Symbol | Character Sequence | Symbol |
|---|---|---|---|---|---|
| \nu | ν | \Phi | Φ | \pm | ± |
| \xi | ξ | \Psi | Ψ | \geq | ≥ |
| \pi | π | \Omega | Ω | \propto | ∝ |
| \rho | ρ | \forall | ∀ | \partial | ∂ |
| \sigma | σ | \exist | ∃ | \bullet | • |
| \varsigma | ς | \ni | ∍ | \div | ÷ |
| \tau | τ | \cong | ≅ | \neq | ≠ |
| \equiv | ≡ | \approx | ≈ | \aleph | ℵ |
| \Im | ℑ | \Re | ℜ | \wp | ℘ |
| \otimes | ⊗ | \oplus | ⊕ | \oslash | ∅ |
| \cap | ∩ | \cup | ∪ | \supseteq | ⊇ |
| \supset | ⊃ | \subseteq | ⊆ | \subset | ⊂ |
| \int | ∫ | \in | ∈ | \o | ο |

You can also specify stream modifiers that control the font used. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

- `\bf` — bold font
- `\it` — italics font
- `\sl` — oblique font (rarely available)
- `\rm` — normal font
- `\fontname{fontname}` — specify the name of the font family to use.

Stream modifiers remain in effect until the end of the string or only within the context defined by braces { }.

The subscript character "_" and the superscript character "^" modify the character or substring defined in braces immediately following.

To print the special characters used to define the Tex strings when `Interpreter` is `Tex`, prefix them with the backslash "\" character: \\, \{, \} _, \^. See the "Example" section for more information.

When `Interpreter` is `none`, no characters in the `String` are interpreted, and all are displayed when the text is drawn.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**`Type`** string (read only)

*Class of graphics object*. For Text objects, `Type` is always the string `'text'`.

**`Units`** `pixels | normalized | inches | centimeters | points | {data}`

*Units of measurement*. This property specifies the units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the Axes plotbox. `Normalized` units map the lower-left corner of the rectangle defined by the Axes to (0,0) and the upper-right corner to (1.0,1.0). `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch). `data` refers to the data units of the parent Axes.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

**`UserData`** matrix

*User-specified data*. Any data you want to associate with the Text object. MATLAB does not use this data, but you can access it using `set` and `get`.

**VerticalAlignment** `top | cap | {middle} | baseline | bottom`

*Vertical alignment of Text*. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean:

- `top` – Place string at the top of the specified *y*-position.
- `cap` – Place the capital letter height at the specified *y*-position.
- `middle` – Place string at the middle of the specified *y*-position.
- `baseline` – Place font baseline at the specified *y*-position.
- `bottom` – Place the string at the bottom of the specified *y*-position.

**Visible** `{on} | off`

*Text visibility*. By default, all Text is visible. When set to `off`, the Text is not visible, but still exists and you can query and set its properties.

## See Also

`gtext`, `int2str`, `num2str`, `title`, `xlabel`, `ylabel`, `zlabel`

## Purpose

Return wrapped string matrix for given UI control

## Syntax

```
outstring = textwrap(h,instring)
[outstring,position] = textwrap(h,instring)
```

## Description

`outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the Uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`[outstring,position]=textwrap(h,instring)` returns the recommended position of the Uicontrol in the units of the Uicontrol. `position` considers the extent of the multi-line text in the *x* and *y* directions.

## Example

Place a textwrapped string in a Uicontrol:

```
pos = [10 10 100 10]
h = uicontrol('Style','Text','Position',pos);

string = {'This is a string for the uicontrol.',
'It should be correctly wrapped inside.'};

[outstring,newpos] = textwrap(h,string);

pos(4) = newpos(4)
set(h,'String',outstring,'Position',[pos(1) pos(2) pos(3)+10
pos(4)])
```

## See Also

`uicontrol`

## Purpose

Add title to current Axes

## Syntax

```
title('string')
title(fname)
title(...,'PropertyName',PropertyValue,...)
h = title(...)
```

## Description

Each Axes graphics object can have one title. The title is located at the top and in the center of the Axes.

`title('`*`string`*`')` outputs the string at the top and in the center of the current Axes.

`title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current Axes.

`title(...,'`*`PropertyName`*`',PropertyValue,...)` specifies property name and property value pairs for the Text graphics object that `title` creates.

`h = title(...)` returns the handle to the text object used as the title.

## Examples

Display today's date in the current Axes:

```
title(date)
```

Include a variable's value in a title:

```
f = 70;
c = (f–32)/1.8;
title(['Temperature is ',num2str(c),'C'])
```

Include a variable's value in a title and set the color of the title to yellow:

```
n = 3
title(['Case number #',int2str(n)],'Color','y')
```

## Algorithm

`title` sets the `Title` property of the current Axes graphics object to a new `Text` graphics object.

## See Also

`gtext`, `int2str`, `num2str`, `plot`, `text`, `xlabel`, `ylabel`, `zlabel`

## trimesh

**Purpose** Triangular mesh plot

**Syntax**

```
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y,Z,C)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

**Description**

`trimesh(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trimesh(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trimesh(...'`*`PropertyName`*`',PropertyValue...)` specifies additional Patch property names and values for the Patch graphics object created by the function.

`h = trimesh(...)` returns a handle to a Patch graphics object.

**Example** Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x–3,6*x–3);
tri = delaunay(x,y);
trimesh(tri,x,y,z)
```

**See Also** `patch`, `trisurf`

The `delauney` function in the *MATLAB Language Reference Manual*.

# trisurf

**Purpose** Triangular surface plot

**Syntax**

```
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

**Description**

`trisurf(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trisurf(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trisurf(...'`*`PropertyName`*`',PropertyValue...)` specifies additional Patch property names and values for the Patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

**Example** Create vertex vectors and a face matrix, then create a triangular surface plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x–3,6*x–3);
tri = delaunay(x,y);
trisurf(tri,x,y,z)
```

**See Also** `patch`, `surf`, `trimesh`

The `delauney` function in *MATLAB Language Reference Manual*.

## Purpose

Create user interface control object.

## Syntax

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

## Description

`uicontrol` is the function for creating Uicontrol graphics objects. Uicontrols (user interface controls) implement graphical user interfaces. When selected, most Uicontrol objects perform a predefined action. MATLAB supports nine styles of Uicontrols, each of which is suited for a different purpose:

- Push buttons
- Check boxes
- Pop-up menus
- Radio buttons
- Sliders
- Editable text
- Static text
- Frames
- List boxes

*Push buttons* are analogous to the buttons on a telephone – they generate an action with each press, but do not remain in a pressed state. To activate a push button, press and release the mouse button on the object. Push buttons are useful when the action you want to perform is not related to any other action executable by the user interface (for example, an "OK" button).

*Check boxes* also generate an action when pressed, but remain in a pressed state until pressed a second time. These devices are useful when providing the user with a number of independent choices, each toggling between two states. To activate a check box, press and release the mouse button on the object. The state of the device is indicated on the display.

*Pop-up menus* open to display a list of choices when pressed. When not activated, they display a single button with text indicating their current setting. Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons require.

*Radio buttons* are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons (i.e., only one is in a pressed state at any given time). To activate a radio button, press and release the mouse button on the object. The state of the device is indicated on the display. Note that your code can implement the mutually exclusive behavior of radio buttons.

*Sliders* accept numeric input within some specific range by allowing the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the mouse over the bar, or by clicking in the trough or on an arrow. The location of the bar indicates a numeric value, which is selected by releasing the mouse button. You can set the minimum, maximum, and current values of the slider.

*Editable text* are boxes containing text users can modify. After typing in the desired text, press **Control-Return** (for multiline), **Return** (for single line) or move the focus off the object to execute its `Callback`. Use editable text when you want text as input.

*Static text* are boxes that display lines of text. It is typically used to label a group of other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.

*Frames* are boxes that enclose regions of a figure window. Frames can make a user interface easier to understand by grouping related controls. Frames have no callback routines associated with them.

*List boxes* display a list of strings and allow users to select individual list entries or multiple, noncontiguous, list entries. The `Min` and `Max` properties control this selection mode. The `Value` property contains the indices into the list of strings. `Value` is a vector if multiple selections are made. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the `Value` property. Therefore, you may need to add a "Done" button to delay action caused by multiple clicks on list items.

List boxes differentiate between single and double clicks and set the Figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box's `Callback` property.

## Remarks

The `uicontrol` function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. The “Uicontrol Properties” section describes these properties. You can also set and query property values after creating the object using the `set` and `get` functions.

Uicontrol objects are children of Figures and therefore do not require an Axes to exist when being placed in a Figure window.

## Examples

The following statement creates a push button that clears the current axes when pressed:

```
h = uicontrol('Style','Pushbutton','Position',...
    [20 150 100 70], 'Callback','cla','String','Clear');
```

You can create a Uicontrol object that changes Figure colormaps by specifying a pop-up menu and supplying an M-file as the object’s `Callback`:

```
hpop = uicontrol('Style','Popup','String',...
       'hsv|hot|cool|gray','Position',[20 320 100 50],...
       'Callback','setmap')
```

This call to `uicontrol` defines four individual choices in the menu: `hsv`, `hot`, `cool`, and `gray`. You specify these choices with the `String` property, separating each with the “|” character.

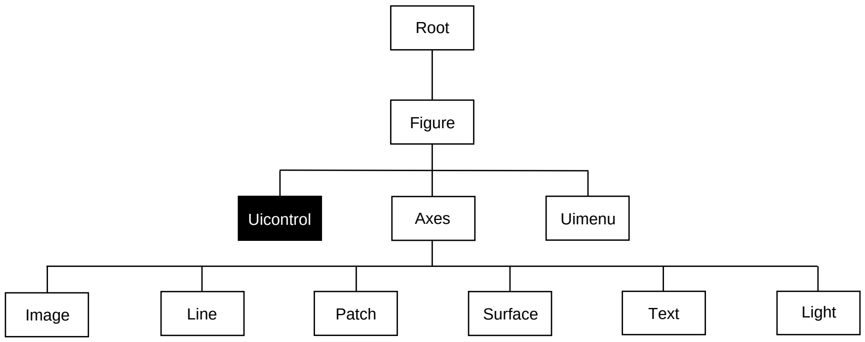
The `Callback`, in this case `setmap`, is the name of an M-file that defines a more complicated set of instructions than a single MATLAB command. `setmap` contains:

```
val = get(hpop,'Value');
if val == 1
    colormap(hsv)
elseif val == 2
    colormap(hot)
elseif val == 3
    colormap(cool)
elseif val == 4
    colormap(gray)
end
```

The `Value` property contains a number that indicates which choice you selected. The choices are numbered sequentially from one to four. The `setmap`

M-file can get and then test the contents of the Value property to determine what action to take.

## Object Hierarchy



### Setting Default Properties

You can set default Uicontrol properties on the Figure and Root levels:

```
set(0,'DefaultUicontrolProperty',PropertyValue...)
set(gcf,'DefaultUicontrolProperty',PropertyValue...)
```

Where *Property* is the name of the Uicontrol property whose default value you want to set and PropertyValue is the value you are specifying.

## Uicontrol Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**BackgroundColor** ColorSpec

*Object background color*. The color used to fill the rectangle defined by the Uicontrol. Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default color is light gray. See the ColorSpec reference page for more information on specifying color.

**BusyAction** cancel | {queue}

*Callback routine interruption*. The BusyAction property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked call-back routes always attempt to interrupt it. If the Interruptible property of the object whose callback is executing is set to on (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**`ButtonDownFcn`** string

*Button press callback routine*. A callback routine that executes whenever you press a mouse button while the pointer is in a five-pixel wide border around the Uicontrol. When the Uicontrol's `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the mouse in the five-pixel border or on the control itself. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using `selectmoveresize`, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

The `Callback` property defines the callback routine that executes when you activate the enabled Uicontrol (e.g., click on a push button).

**`Callback`** string

*Control action*. A callback routine that executes whenever you activate the Uicontrol object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace. Note that Frames and Static Text do not define actions to interactively invoke their callback routines.

**`Children`** matrix

The empty matrix; Uicontrol objects have no children.

**`Clipping`** `{on} | off`

This property has no effect on Uicontrols.

**CreateFcn** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Uicontrol object. You must define this property as a default value for Uicontrols. For example, the statement,

```
set(0,'DefaultUicontrolCreateFcn','set(gcf,''IntegerHandle'',’'o
ff'')')
```

defines a default value on the Root level that sets the Figure `IntegerHandle` property to `off` whenever you create a Uicontrol object. MATLAB executes this routine after setting all property values for the Uicontrol. Setting this property on an existing Uicontrol object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**DeleteFcn** string

*Delete Uicontrol callback routine*. A callback routine that executes when you delete the Uicontrol object (e.g., when you issue a `delete` command or clear the Figure containing the Uicontrol). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**Enable** `{on} | inactive | off`

*Enable or disable the Uicontrol*. This property controls how Uicontrols respond to mouse button clicks.

- `on` – The Uicontrol is operational. When you activate the Uicontrol (generally by clicking on it) MATLAB executes the callback routine defined by the `Callback` property. When you click the mouse within a 5-pixel border outside the Uicontrol, MATLAB executes the callback routine defined by the `ButtonDownFcn`.
- `inactive` – The Uicontrol is not operational, but it is not dimmed (i.e., it looks the same as when `Enable` is `on`). MATLAB executes the `ButtonDownFcn`

if you click the mouse on or within a 5-pixel border around the Uicontrol, and does not execute the `Callback`.

- `off` – The Uicontrol does not respond visually to mouse actions, does not execute its `Callback` routine, and its label (`string` property) is grayed out. MATLAB executes the `ButtonDownFcn` if you click the mouse on or within a 5-pixel border around the Uicontrol.

Setting this property to `inactive` or `off` enables you to implement object "dragging" via the `ButtonDownFcn` callback routine.

**`Extent`** position rectangle (read only)

*Size of Uicontrol character string*. A four-element vector that defines the size and position of the character string used to label the Uicontrol. It has the form:

```
[0,0,width,height]
```

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the Uicontrol itself, you can use this property to determine proper sizing for the Uicontrol with regard to its label. Do this by,

- Defining the `String` property and selecting the font using the `Font`*nnn* properties.
- Getting the value of the `Extent` property.
- Defining the `width` and `height` of the `Position` property to be somewhat larger than the `width` and `height` of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the Extent is returned as a single line, even if the string wraps when displayed on the control.

**`FontAngle`** `{normal} | italic | oblique`

*Character slant*. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

**`FontName`** string

*Font family*. The name of the font in which to display the `String`. To display and print properly, this must be a font that your system supports. The default font is system dependent.

**`FontSize`** size in `FontUnits`

*Font size*. A number specifying the size of the font in which to display the `String`, in units determined by the `FontUnits` property. The default point size is system dependent.

**`FontUnits`** `{points} | normalized | inches | centimeters | pixels`

*Font size units*. MATLAB uses this property to determine the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the Uicontrol. When you resize the Uicontrol, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

**`FontWeight`** `light | {normal} | demi | bold`

*Weight of Text characters*. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

**`ForegroundColor`** `ColorSpec`

*Color of text*. This property determines the color of the text defined for the `String` property (the Uicontrol label). Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default text color is black. See the `ColorSpec` reference page for more information on specifying color.

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to `on` to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`HorizontalAlignment`**  `left | {center} | right`

*Horizontal alignment of label string*. This property determines the justification of the text defined for the `String` property (the Uicontrol label):

- `left` — Text is left justified with respect to the Uicontrol.
- `center` — Text is centered with respect to the Uicontrol.
- `right` — Text is right justified with respect to the Uicontrol.

On MS-Windows and Macintosh systems, this property affects only `edit` and `text` Uicontrols.

**`Interruptible`**  `{on} | off`

*Callback routine interruption mode*. The `Interruptible` property controls whether a Uicontrol callback routine can be interrupted by subsequently invoked callback routines. By default (`off`), a callback routine executes to completion before another can begin.

Only callback routines defined for the `ButtonDownFcn` and `Callback` properties are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

**`ListboxTop`** scalar

*Index of top-most string displayed in list box*. This property applies only to the `listbox` style of Uicontrol. It specifies which string occupies the top-most position in the list box. Define `ListboxTop` as an index into the array of strings defined by the `String` property. Noninteger values are fixed to the next lowest integer.

**`Max`** scalar

*Maximum value*. This property specifies the largest value allowed for the `Value` property. Different `Styles` of Uicontrols interpret `Max` differently:

- Radio buttons and check boxes (on/off switches) – `Max` is the setting of the `Value` property while the Uicontrol is in the on position.
- Sliders – `Max` is the largest value you can select and must be greater than the `Min` property. The default maximum is 1.
- Editable text – If `Max` – `Min` > 1, then editable text boxes accept multiline input. If `Max` – `Min` <= 1, then editable text boxes accept only single line input.
- List boxes – If `Max` – `Min` > 1, then list boxes allow multiple item selection. If `Max` – `Min` <= 1, then list boxes do not allow multiple item selection.
- Frames, pop-up menus, and static text do not use the `Max` property.

**`Min`** scalar

*Minimum value*. This property specifies the smallest value allowed for the `Value` property. Different `Styles` of Uicontrols interpret `Min` differently:

- Radio buttons and check boxes (on/off switches) — `Min` is the setting of the `Value` property while the Uicontrol is in the `off` position.
- Sliders – `Min` is the smallest value you can select and must be less than `Max`. The default minimum is 0.
- Editable text – If `Max` – `Min` > 1, then editable text boxes accept multiline input. If
  `Max` – `Min` <= 1, then editable text boxes accept only single line input.
- List boxes – If `Max` – `Min` > 1, then list boxes allow multiple item selection. If `Max` – `Min` <= 1, then list boxes allow only single item selection.

**`Parent`** handle

*Uicontrol's parent*. The handle of the Uicontrol's parent object. The parent of a Uicontrol object is the Figure in which it displays. You can move a Uicontrol object to another Figure by setting this property to the handle of the new parent.

**`Position`** position rectangle

*Size and location of Uicontrol*. The rectangle defined by this property specifies the size and location of the control within the Figure window. Specify `Position` as

```
[left,bottom,width,height]
```

`left` and `bottom` are the distance from the lower-left corner of the Figure window to the lower-left corner of the Uicontrol object. `width` and `height` are the dimensions of the Uicontrol rectangle. All measurements are in units specified by the `Units` property.

**`Selected`** `on | {off}`

*Is object selected*. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

**SelectionHighlight** {on} | off

*Objects highlight when selected*. When the Selected property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When SelectionHighlight is off, MATLAB does not draw the handles.

**SliderStep** [min_step max_step]

*Slider step size*. This property controls the percentage (of maximum slider value) change in the slider's current value when you click the mouse on the slider trough (max_step) or on its arrow button (min_step). Specify SliderStep as a two-element vector whose elements MATLAB converts to percents. The default, [0.01 0.10], provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough.

**String** string

*Uicontrol label*. A string specifying the text displayed on push buttons, radio buttons, check boxes, static text, editable text, listboxes, and pop-up menus.

For multiple items on a pop-up menu or a list box, items can be specified as a cell array of strings, a padded string matrix, or within a string vector separated by vertical slash ('|') characters.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, each cell of a cell array of strings, and after any \n characters embedded in the string. Vertical slash ('|') characters are not interpreted as linebreaks, and instead show up in the text displayed in the uicontrol.

For the remaining uicontrol styles, which display only one line of text, only the first string of a cell array of string or of a padded string matrix is displayed, and all the rest are ignored. Vertical slash ('|') characters are not interpreted as linebreaks, and instead show up in the text displayed in the uicontrol.

For editable text, this property is set to the string typed in by the user.

**Style** {pushbutton} | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu

*Style of Uicontrol object to create*. The Style property selects the style of Uicontrol to create. See the "Description" section for information on each type.

**`Tag`** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**`Type`** string (read only)

*Class of graphics object*. For Uicontrol objects, `Type` is always the string `'uicontrol'`.

**`Units`** `{pixels} | normalized | inches | centimeters | points`

*Units of measurement*. The units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the Figure window. `Normalized` units map the lower-left corner of the Figure window to (0,0) and the upper-right corner to (1.0,1.0). `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

**`UserData`** matrix

*User-specified data*. Any data you want to associate with the Uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

**`Value`** scalar or vector

*Current value of Uicontrol*. The possible values a Uicontrol can take on depend on its `Style` property:

- Radio buttons and check boxes set `Value` to `Max` (usually 1) when they are on (when the indicator is filled) and `Min` (usually 0) when off (not filled).
- Sliders set `Value` to the number indicated by the slider bar, which is within the range established by `Min` and `Max`.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item on the menu. The "Examples" section shows how to use the `Value` property to determine which item has been selected.
- List boxes set `Value` to a vector of indices corresponding to the highlighted items displayed in the box, where 1 corresponds to the first item in the list.
- Push buttons, editable text, static text, and frames do not set this property.

Set the `Value` property either interactively with the mouse or through a call to the `set` function. The display reflects changes made to `Value`.

**`Visible`** `{on} | off`

*Uicontrol visibility*. By default, all Uicontrols are visible. When set to `off`, the Uicontrol is not visible, but still exists and you can query and set its properties.

## See Also

`textwrap`, `uimenu`

## Purpose

Interactively retrieve a filename

## Syntax

```
uigetfile
uigetfile('filterSpec')
uigetfile('filterSpec','dialogTitle')
uigetfile('filterSpec','dialogTitle',x)
uigetfile('filterSpec','dialogTitle',x,y)
[fname,pname] = uigetfile(...)
```

## Description

`uigetfile` displays a dialog box used to retrieve a file. The dialog lists the subdirectories in your current directory. The default position of the dialog box is the upper-left corner of your monitor.

`uigetfile('`*`filterSpec`*`')` displays a dialog box that lists the files in the current directory specified by `'`*`filterSpec`*`'`. `'`*`filterSpec`*`'` is a full filename or includes wildcards. A wildcard specification such as `'*.m'` does not provide a default file and the scroll box lists only files with the `.m` extension.

`uigetfile('`*`filterSpec`*`','`*`dialogTitle`*`')` displays a dialog box that has the title `'`*`dialogTitle`*`'`.

`uigetfile('`*`filterSpec`*`','`*`dialogTitle`*`',x)` positions the upper-left corner of the dialog box at (`x`,`0`), where `x` is in pixel units. (Some platforms may not support dialog box placement.)

`uigetfile('`*`filterSpec`*`','`*`dialogTitle`*`',x,y)` positions the upper-left corner of the dialog box. `x` and `y` are the *x*- and *y*-position, in pixels, of the dialog box. (Some platforms may not support dialog box placement.)

`[fname,pname] = uigetfile(...)` returns the filename and pathname (or folder) selected in the dialog box. After you press the **Done** button, `fname` contains the name of the file selected and `pname` contains the name of the path selected. If you press the **Cancel** button or if an error occurs, `fname` and `pname` are set to `0`.

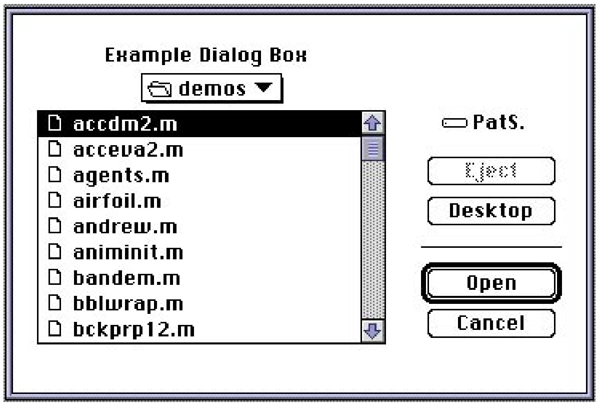
## Remarks

If you select a file that does not exist, an error dialog informs you that the file does not exist. You can then enter another filename, or press the **Cancel** button.

**Examples**

Retrieve a filename using `uigetfile` to list all MATLAB M-files within a selected directory (note that the figure shows the dialog box on a Macintosh):

```
[fname,pname] = uigetfile('*.m','Example Dialog Box')
```



The exact appearance of the dialog box depends on your windowing system.

**See Also**

`uiputfile`

## Purpose

Create menus on a Figure window

## Syntax

```
handle = uimenu('PropertyName',PropertyValue,...)
handle = uimenu(parent,'PropertyName',PropertyValue,...)
```

## Description

`uimenu` creates a hierarchy of menus and submenus that display in the Figure window's menu bar.

`handle = uimenu('`*`PropertyName`*`',PropertyValue,...)` creates a menu in the current Figure's menu bar using the values of the specified properties.

`handle = uimenu(parent,'`*`PropertyName`*`',PropertyValue,...)` creates a submenu of the parent menu specified by `parent`. If `parent` refers to a Figure instead of another Uimenu object, MATLAB creates a new menu on the referenced Figure's menubar.

## Remarks

MATLAB adds the new menu to the existing menu bar. Each menu choice can itself be a menu that displays its submenu when selected.

`uimenu` accepts property name/property value pairs, structures, and cell arrays as input arguments. The Uimenu `Callback` property defines the action taken when you activate the menu. The "Uimenu Properties" section describes these properties. `uimenu` optionally returns the handle to the created Uimenu object.

Uimenus only appear in Figures whose `WindowStyle` is `normal`. If a Figure containing Uimenu children is changed to `WindowStyle` `modal`, the Uimenu children will still exist, and be contained in the `Children` list of the Figure, but will not be displayed until the `WindowStyle` reverts to `normal`.

The value of the Figure `MenuBar` property affects the location of Uimenu children of the Figure on the menubar. When `MenuBar` is `none`, Uimenus are the only items on the Figure menubar. When `MenuBar` is `figure`, a set of built-in menus precedes the Uimenus on the menubar (but MATLAB controls those built-in menus, and their handles can not be obtained by the user).

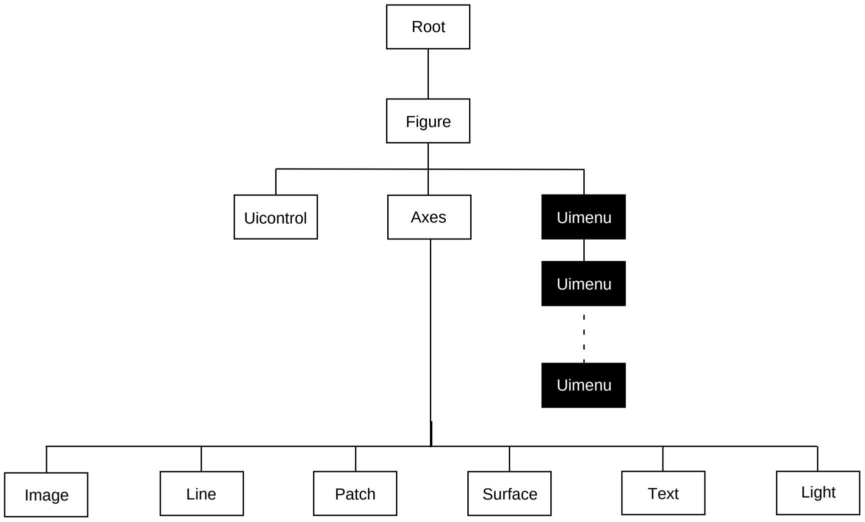
You can set and query property values after creating the menu using `set` and `get`.

## Examples

This example creates a menu labeled **Workspace** whose choices allow users to create a new Figure window, save workspace variables, and exit out of MATLAB. In addition, it defines an accelerator key for the quit option.

```
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','figure');
    uimenu(f,'Label','Save','Callback','save');
    uimenu(f,'Label','Quit','Callback','exit',...
          'Separator','on', 'Accelerator', 'Q');
```

## Object Hierarchy



### Setting Default Properties

You can set default Uimenu properties on the Figure and Root levels:

```
set(0,'DefaultUimenuPropertyName',PropertyValue...)
set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
set(menu_handle,'DefaultUimenuProperty',PropertyValue...)
```

Where *PropertyName* is the name of the Uimenu property and `PropertyValue` is the value you are specifying.

## Object Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

**`Accelerator`** character

*Keyboard equivalent*. A character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

- For X-Windows and MS-Windows systems, the sequence is **Control-**`Accelerator`.
- For Macintosh systems, the sequence is **Command-**`Accelerator`.

You can define an accelerator only for menu items that do not have children menus. Accelerators work only for menu items that directly execute a callback routine, not items that bring up other menus.

Note that the menu item does not have to be displayed (e.g., a submenu) for the accelerator key to work. However, the window focus must be in the Figure when the key sequence is entered.

**`BackgroundColor`** (obsolete)

The background color of menu items is determined by the system.

**`BusyAction`** `cancel | {queue}`

*Callback routine interruption*. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

**ButtonDownFcn** string

The button down function is not used for Uimenus.

**Callback** string

*Menu action*. A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you *release* the mouse button (i.e., on the button up event).

**Checked** `on | {off}`

*Menu check indicator*. Setting this property to `on` places a check mark next to the corresponding menu item. Setting it to `off` removes the check mark. You can use this feature to create menus that indicate the state of a particular option. Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected.

**Children** vector of handles

*Handles of submenus*. A vector containing the handles of all children of the Uimenu object. The children objects of Uimenus are other Uimenus, which function as submenus. You can use this property to re-order the menus.

`Clipping` `{on} | off`

`Clipping` has no effect on Uimenus.

**CreateFcn** string

*Callback routine executed during object creation*. This property defines a callback routine that executes when MATLAB creates a Uimenu object. You must define this property as a default value for Uimenus. For example, the statement,

```
set(0,'DefaultUimenuCreateFcn','set(gcf,''IntegerHandle'',’’off''’)
```

defines a default value on the Root level that sets the Figure `IntegerHandle` property to `off` whenever you create a Uimenu object. Setting this property on an existing Uimenu object has no effect. MATLAB executes this routine after setting all property values for the Uimenu.

The handle of the object whose `CreateFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`DeleteFcn`** string

*Delete Uimenu callback routine*. A callback routine that executes when you delete the Uimenu object (e.g., when you issue a `delete` command or cause the Figure containing the Uimenu to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the Root `CallbackObject` property, which can be queried using `gcbo`.

**`Enable`** `{on} | off`

*Enable or disable the Uimenu*. This property controls the selectability of a menu item. When not enabled (set to `off`), the menu `Label` appears dimmed, indicating you cannot select it.

**`ForegroundColor`** `ColorSpec` X-Windows only

*Color of menu label string*. This property determines color of the text defined for the `Label` property. Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default text color is black. See the `ColorSpec` reference page for more information on specifying color.

**`HandleVisibility`** `{on} | callback | off`

*Control access to object's handle by command-line users and GUIs*. This property determines when an object's handle is visible in its parent's list of children. Handles are always visible when `HandleVisibility` is `on`. When `HandleVisibility` is `callback`, handles are visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line - a useful way to protect GUIs from command-line users, while permitting their callbacks complete access to their own handles. Setting `HandleVisibility` to `off` makes handles invisible at all times - which is occasionally necessary when a callback needs to invoke a function that might potentially damage the UI, and so wants to temporarily hide its own handles during the execution of that function. When a handle is not visible in its parent's list of children, it can not be returned by any functions which obtain handles by searching the object hierarchy or querying handle properties, including `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, Figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the Figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

The Root `ShowHiddenHandles` property can be set to on to temporarily make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles. This property is useful for preventing command-line users from accidently drawing into or deleting a Figure that contains only user interface devices (such as a dialog box).

**`Interruptible`** `{on} | off`

*Callback routine interruption mode.* The `Interruptible` property controls whether a Uimenu callback routine can be interrupted by subsequently invoked callback routines. By default (`off`), a callback routine executes to completion before another can begin. Only the `Callback` Uimenu property is affected by the Interruptible property.

**`Label`** string

*Menu label.* A string specifying the text label on the menu item. You can specify a mnemonic using the "&" character. Whatever character follows the "&" in the string appears underlined and selects the menu item when you type that character while the menu is visible. The "&" character is not displayed. On Macintosh systems, MATLAB ignores (and does not print) the "&" character. To display the "&" character in a label, use two "&" characters in the string:

`'O&pen selection'` yeilds **O<u>p</u>en selection**

`'Save && Go'` yeilds **Save & Go**

**`Parent`** handle

*Uimenu's parent.* The handle of the Uimenu's parent object. The parent of a Uimenu object is the Figure on whose menu bar it displays, or the Uimenu of which it is a submenu. You can move a Uimenu object to another Figure by setting this property to the handle of the new parent.

**Position** scalar

*Relative menu position*. The value of `Position` indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their `Position` property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their `Position` property, with 1 representing the top-most position.

**Selected** `on | {off}`

This property is not useful for Uimenus.

**SelectionHighlight** `on | off`

This property is not useful for Uimenus.

**Separator** `on | {off}`

*Separator line mode*. Setting this property to `on` draws a dividing line above the menu item.

**Tag** string

*User-specified object label*. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

**Type** string (read only)

*Class of graphics object*. For Uimenu objects, `Type` is always the string `'uimenu'`.

**UserData** matrix

*User-specified data*. Any matrix you want to associate with the Uimenu object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

**Visible** `{on} | off`

*Uimenu visibility*. By default, all Uimenus are visible. When set to `off`, the Uimenu is not visible, but still exists and you can query and set its properties.

## See Also

`uicontrol`, `gcbo`, `set`, `get`, `figure`

# uiputfile

**Purpose** Interactively select a file for writing

**Syntax**

```
uiputfile
uiputfile('filterSpec')
uiputfile('filterSpec','dialogTitle')
uiputfile('filterSpec','dialogTitle',x)
uiputfile('filterSpec','dialogTitle',x,y)
[fname,pname] = uiputfile(...)
```

**Description**

`uiputfile` displays a dialog box used to select a file for writing. The dialog lists the directories in your current directory. The default position of the dialog box is the upper-left corner of your monitor.

`uiputfile('`*`filterSpec`*`')` displays a dialog box that lists the files in the current directory specified by `'`*`filterSpec`*`'`. `'`*`filterSpec`*`'` is a full filename or includes wildcards. A wildcard specification such as `'*.m'` does not provide a default file and the scroll box lists only files with the `.m` extension.

`uiputfile('`*`filterSpec`*`','`*`dialogTitle`*`')` displays a dialog box that has the title `'`*`dialogTitle`*`'`.

`uiputfile('`*`filterSpec`*`','`*`dialogTitle`*`',x)` positions the upper-left corner of the dialog box at (`x`,`0`), where `x` is in pixel units. Note that positioning may not work on all platforms.

`uiputfile('`*`filterSpec`*`','`*`dialogTitle`*`',x,y)` positions the upper-left corner of the dialog box. `x` and `y` are the *x*- and *y*-position, in pixels, of the dialog box. Note that positioning may not work on all platforms.

`[fname,pname] = uiputfile(...)` returns the filename and pathname (or folder) selected in the dialog box. After you press the **Done** button, `fname` contains the name of the file selected and `pname` contains the name of the path selected. If you press the **Cancel** button or if an error occurs, `fname` and `pname` are set to `0`.

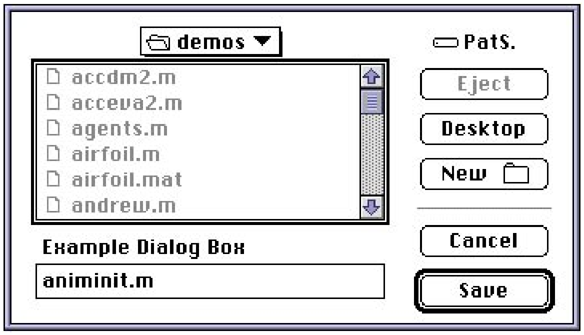
**Remarks**

If you select a file that already exists, a prompt asks whether you want to overwrite the file. If you select **OK**, the function successfully returns but does not delete the existing file (which is the responsibility of the calling routines). If you select **Cancel**, the function returns control back to the dialog box so that you can enter another filename.

## Examples

Display a dialog box titled `'Example Dialog Box'` (the exact appearance of the dialog box depends on your windowing system):

```
[newfile,newpath] = uiputfile('animinit.m','Example Dialog
Box');
```



Example Dialog Box

animinit.m

## See Also

`uigetfile`

# uiresume, uiwait

**Purpose** Control program execution

**Syntax**

```
uiwait(h)
uiwait
uiresume(h)
```

**Description** The `uiwait` and `uiresume` functions block and resume MATLAB program execution.

`uiwait` blocks execution until `uiresume` is called or the current Figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the Figure `h` is deleted.

`uiresume(h)` resumes the M-file execution that `uiwait` suspended.

**Remarks** When creating a dialog, you should have a `uicontrol` with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

`uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the M-file that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, `uiwait/uiresume` can block the execution of the MFile *and* restrict user interaction to the dialog only.

**See Also** `uicontrol`, `uimenu`, `waitfor`, `figure`, `dialog`

## uisetcolor

**Purpose** Interactively set an object's `ColorSpec` from a dialog box (MS-Windows and Mac only)

**Syntax**

```
c = uisetcolor(h_or_c, 'dialogTitle')
```

**Description** `uisetcolor` displays a dialog box for the user to fill in, then applies the selected color to the appropriate property of the graphics object identified by the first argument.

`h_or_c` can be either a handle to a graphics object or an RGB triple. If you specify a handle, it must specify a graphics object that supports color. If you specify a color, it must be a valid RGB triple (e.g., [1 0 0] for red). The color specified is used to initialize the dialog box. If no initial RGB is specified, the dialog box initializes the color to black.

`dialogTitle` is a string that is used as the title of the dialog box.

`c` is the RGB value selected by the user. If the user presses **Cancel** from the dialog box, or if any error occurs, `c` is set to the input RGB triple, if provided; otherwise, it is set to 0.

**See Also** `ColorSpec`

## Purpose

Interactively select a font

## Syntax

```
uisetfont
uisetfont(handleIn)
uisetfont('dialogTitle')
uisetfont(handleIn,'dialogTitle')
handleOut = uisetfont(...)
```

## Description

`uisetfont` displays a dialog box and creates a Text graphics object with the font properties selected in the dialog box.

`uisetfont(handleIn)` displays a dialog box and applies the selected font attributes to the Text or Axes graphics object specified by `handleIn`. `uisetfont` uses the font properties currently assigned to this object to initialize the dialog box.

`uisetfont('`*`dialogTitle`*`')` displays a dialog box with the title `'`*`dialog-Title`*`'` and creates a Text graphics object with the font properties selected in the dialog box.

`uisetfont(handleIn,'`*`dialogTitle`*`')` applies the selected font attributes to the Text or Axes graphics object specified by `handleIn` and assigns the title `'`*`dialogTitle`*`'` to the dialog box. The arguments can appear in any order.

`handleOut = uisetfont(...)` returns the handle `handleOut`. If you specify `handleIn`, `handleOut` is identical to `handleIn`. If you do not specify `handleIn`, `uisetfont` creates a new Text object using the selected font properties, and returns its handle. If you press the **Cancel** button or an error occurs, `handleOut` is set to `handleIn`, if provided, or to 0.

## Example

Interactively change the font for a Text graphics object by displaying a dialog to update the font:

```
h = text(.5,.5,'Figure Annotation')
uisetfont(h,'Update Font')
```

## See Also

`axes`, `text`, `uicontrol`

## Purpose

Viewpoint specification

## Syntax

```
view(az,el)
view([az,el])
view([x,y,z])
view(2)
view(3)
view(T)

[az,el] = view
T = view
```

## Description

The position of the viewer (the viewpoint) determines the orientation of the Axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counter-clockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0, el = 90`.

`view(3)` sets the default three-dimensional view, `az = –37.5, el = 30`.

`view(T)` sets the view according to the transformation matrix `T`, which is a 4-by-4 matrix such as a perspective transformation generated by `viewmtx`.

`[az,el] = view` returns the current azimuth and elevation.

`T = view` returns the current 4-by-4 transformation matrix.

## Examples

View the object from directly overhead:

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the *y*-axis, with the *x*-axis extending horizontally and the *z*-axis extending vertically in the Figure:

```
view([0 0]);
```

Rotate the view about the *z*-axis by 180°:

```
az = 180;
el = 90;
view(az, el);
```

## See Also

`viewmtx`, `axes`

Axes graphics object properties: `CameraPosition`, `CameraTarget`, `CameraViewAngle, Projection`.

## Purpose

View transformation matrices

## Syntax

```
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

## Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion:

| **Phi** | **Description** |
|---|---|
| 0 degrees | Orthographic projection |
| 10 degrees | Similar to telephoto lens |
| 25 degrees | Similar to normal lens |
| 60 degrees | Similar to wide angle lens |

You can use the matrix returned to set the view transformation with `view(T)`. The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the form *(x,y,z,w)*, where *w* is not equal to 1. The *x*- and *y*-components of the normalized vector (*x/w*, *y/w*, *z/w*, 1) are the desired two-dimensional components (see example below).

`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

## Remarks

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, `[x,y,z,1]` is the four-dimensional vector corresponding to the three-dimensional point `[x,y,z]`.

## Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,–3.0) using the default view direction. Note that the point is a column vector.

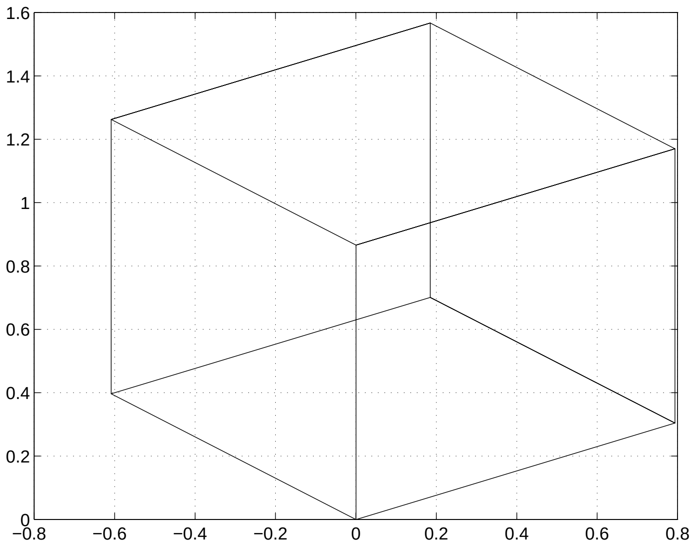
```
A = viewmtx(–37.5,30);
x4d = [.5  0  –3  1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
     0.3967
    –2.4459
```

Vectors that trace the edges of a unit cube are

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object:

```
A = viewmtx(–37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
plot(x2,y2)
```



Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(–37.5,30,25);
x4d = [.5  0  –3  1]';
x2d = A∗x4d;
x2d = x2d(1:2)/x2d(4);   % Normalize
x2d =
     0.1777
    –1.8858
```

Transform the cube vectors to the screen and plot the object:

```
A = viewmtx(–37.5,30,2);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
plot(x2,y2)
```

## See Also

`view`

**Purpose**

Display waitbar

**Syntax**

```
h = waitbar(x,'title')
```

**Description**

A waitbar shows what percentage of a calculation is complete, as the calculation proceeds.

`h = waitbar(x,'title')` creates and displays a waitbar of fractional length `x`. The handle to the waitbar Figure is returned in `h`. `x` should be between 0 and 1. Each subsequent call to waitbar, `waitbar(x)`, extends the length of the bar to the new position `x`.

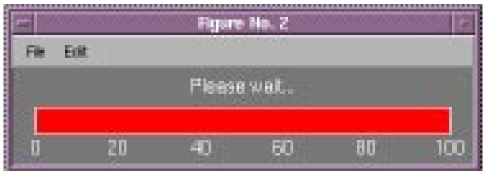
**Example**

`waitbar` is typically used inside a `for` loop that performs a lengthy computation. For example,

```
h = waitbar(0,'Please wait...');

for i=1:100, % computation here %
waitbar(i/100)
end

close(h)
```



# waitfor

## Purpose

Wait for condition

## Syntax

```
waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)
```

## Description

The `waitfor` function blocks the caller's execution stream so that command-line expressions, callbacks, and statements in the blocked M-file do not execute until a specified condition is satisfied.

`waitfor(h)` returns when the graphics object identified by `h` is deleted or when a **Ctrl-C** is typed in the command window. If `h` does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'`*`PropertyName`*`')`, in addition to the conditions in the previous syntax, returns when the value of `'`*`PropertyName`*`'` for the graphics object `h` changes. If `'`*`PropertyName`*`'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'`*`PropertyName`*`',PropertyValue)`, in addition to the conditions in the previous syntax, `waitfor` returns when the value of `'`*`PropertyName`*`'` for the graphics object `h` changes to `PropertyValue`. `waitfor` returns immediately without processing any events if `'`*`PropertyName`*`'` is set to `PropertyValue`.

## Remarks

While `waitfor` blocks an execution stream, other execution streams in the form of callbacks may execute as a result of various events (e.g., pressing a mouse button).

`waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can itself invoke `waitfor`.

## See Also

`uiresume`, `uiwait`

**Purpose** Wait for key or mouse button press

**Syntax** `k = waitforbuttonpress`

**Description** `k = waitforbuttonpress` blocks the caller's execution stream until `waitforbuttonpress` detects a mouse button or key press while the cursor is over a Figure window. The function returns `0` if it detects a mouse button press or `1` if it detects a key press. Additional information about the event that resumes execution is available through the Figure's `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

**See Also** `dragrect`, `figure`, `gcf`, `ginput`, `rbbox`, `waitfor`

# warndlg

**Purpose** Warning dialog box

**Syntax** `h = warndlg('`*`warningstring`*`','`*`dlgname`*`')`

**Description** `warndlg` displays a dialog box named `'Warning Dialog'` containing the string `'This is the default warning string.'` The warning dialog disappears after you press the **OK** push button.

`warndlg('`*`warningstring`*`')` displays a dialog box named `'Warning Dialog'` containing the string specified by `'`*`warningstring`*`'`.

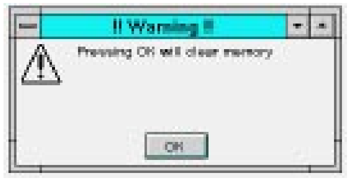
`warndlg('`*`warningstring`*`','`*`dlgname`*`')` displays a dialog box named `'`*`dlgname`*`'` containing the string `'`*`warningstring`*`'`.

`h = warndlg(...)` returns the handle of the dialog box.

**Examples** The function

```
warndlg('Pressing OK will clear memory','!! Warning !!');
```

displays the following dialog box:


!! Warning !!

Pressing OK will clear memory

**See Also** `dialog`, `errordlg`, `helpdlg`, `msgbox`

## Purpose

Waterfall plot

## Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)

h = waterfall(...)
```

## Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,1)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z` and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x* and *y* coordinates of the plot. `Z` is a matrix that defines the *z* coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`h = waterfall(...)` returns the handle of the Patch graphics object used to draw the plot.

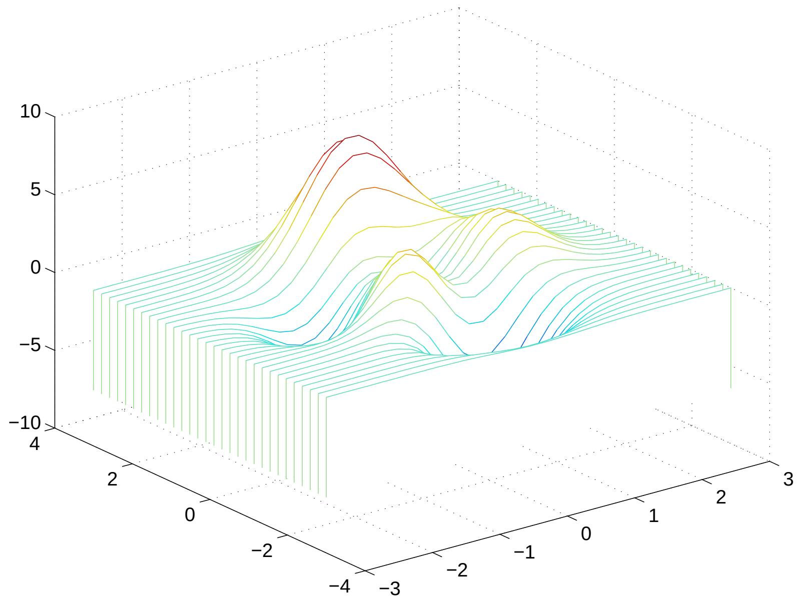
## Remarks

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

## Examples

Produce a waterfall plot of the peaks function:

```
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```



## Algorithm

The range of X, Y, and Z, or the current setting of the Axes Xlim, Ylim, and Zlim properties, determines the range of the Axes (also set by axis). The range of C, or the current setting of the Axes Clim property, determines the color scaling (also set by caxis).

The CData property for the Patch graphics objects specifies the color at every point along the edge of the Patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface, however, it is a Patch graphics object. To create a Surface plot similar to waterfall, use the meshz function and set the MeshStyle property of the Surface to 'Row'. For a discussion of parametric surfaces and related color properties, see surf.

## See Also

axes, axis, caxis, meshz, surf

Properties for Patch graphics objects.

## Purpose

Change Axes background color

## Syntax

`whitebg`
`whitebg(h)`
`whitebg(`*ColorSpec*`)`
`whitebg(h,`*ColorSpec*`)`

## Description

`whitebg` complements the colors in the current Figure.

`whitebg(h)` complements colors in all Figures specified in the vector `h`.

`whitebg(`*ColorSpec*`)` and `whitebg(h,`*ColorSpec*`)` change the color of the Axes, which are children of the Figure, to the color specified by *ColorSpec*.

## Remarks

`whitebg` changes the colors of the Figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties of the Root window such that all subsequent Figure plots use the new background color.

## Examples

Set the background color to blue-gray:

```
whitebg([0 .5 .6])
```

Set the background color to blue:

```
whitebg('blue')
```

## See Also

`ColorSpec`

The Figure graphics object property `InvertHardCopy`.

# xlabel, ylabel, zlabel

## Purpose

Label the *x*-, *y*-, and *z*-axis

## Syntax

```
xlabel('string')
xlabel(fname)
xlabel(...,'PropertyName',PropertyValue,...)
h = xlabel(...)

ylabel(...)
h = ylabel(...)

zlabel(...)
h = zlabel(...)
```

## Description

Each Axes graphics object can have one label for the *x*-, *y*-, and *z*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`xlabel('`*`string`*`')` labels the *x*-axis of the current Axes.

`xlabel(fname)` evaluates the function `fname`, which must return a string, then displays the string beside the *x*-axis.

`xlabel(...,'`*`PropertName`*`',PropertyValue,...)` specifies property name and property value pairs for the Text graphics object created by `xlabel`.

`h = xlabel(...)`, `h = ylabel(...)`, and `h = zlabel(...)` return the handle to the text object used as the label.

`ylabel(...)` and `zlabel(...)` label the *y*-axis and *z*-axis, respectively, of the current Axes.

## Remarks

Re-issuing an `xlabel`, `ylabel`, or `zlabel` command causes the new label to replace the old label.

## Algorithm

For three-dimensional graphics, MATLAB puts the label in the front or side, so that it is never hidden by the plot.

## See Also

`text`, `title`

## Purpose

Zoom in and out on a 2-D plot

## Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
```

## Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a Figure, pressing a mouse button while your cursor is within an Axes zooms into the point or out from the point beneath the mouse. Zooming changes the Axes limits.

- For a single-button mouse, zoom in by pressing the mouse button and zoom out by simultaneously pressing **Shift** and the mouse button.
- For a two- or three-button mouse, zoom in by pressing the left mouse button and zoom out by pressing the right mouse button.

Clicking and dragging over an Axes when interactive zooming is enabled draws a rubber-band box. When the mouse button is released, the Axes zoom in to the region enclosed by the rubber-band box.

Double-clicking over an Axes returns the Axes to its initial zoom setting.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status.

`zoom xon` and `zoom yon` sets `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the above options can be specified on a figure other than the current figure using this syntax.

**Remarks**

`zoom` changes the Axes limits by a factor of two (in or out) each time you press the mouse button while the cursor is within an Axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

# Index