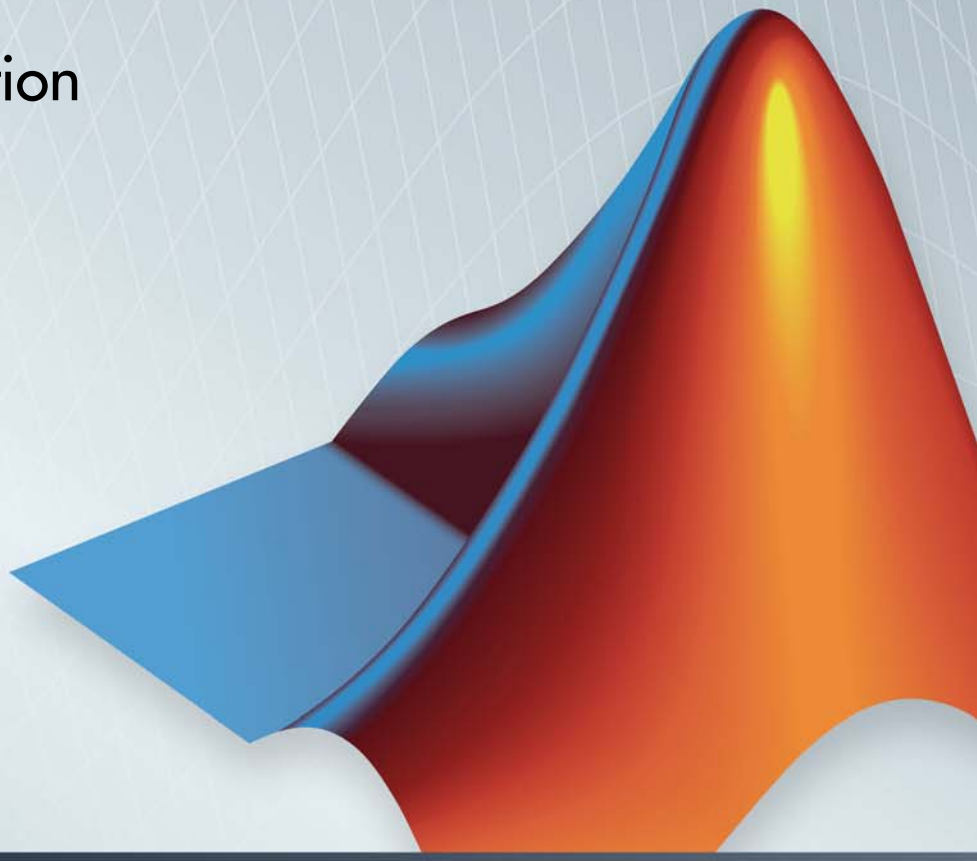
MATLAB®

3-D Visualization

R2013a



MATLAB®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® 3-D Visualization

© COPYRIGHT 1984–2013 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2006	Online only	New for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a) This publication was previously part of the Using MATLAB Graphics User Guide.
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)
April 2011	Online only	Revised for Version 7.12 (Release 2011a)
September 2011	Online only	Revised for Version 7.13 (Release 2011b)
March 2012	Online only	Revised for Version 7.14 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)

1

Creating 3-D Graphs

Typical 3-D Graph

This table illustrates typical steps involved in producing 3-D scenes containing either data graphs or models of 3-D objects. Example applications include pseudocolor surfaces illustrating the values of functions over specific regions and objects drawn with polygons and colored with light sources to produce realism. Usually, you follow either step 4 or step 5 (not both).

Step	Typical Code
1 Prepare your data.	`Z = peaks(20);`
2 Select window and position plot region within window.	`figure(1);subplot(2,1,2)`
3 Call 3-D graphing function.	`h = surf(Z);`

4 Set colormap and shading algorithm.	`colormap hot`
	`shading interp`
	`set(h,'EdgeColor','k')`

5 Add lighting.	`light('Position',[-2,2,20])`
	`lighting phong`
	`material([0.4,0.6,0.5,30])`
	`set(h,'FaceColor',[0.7 0.7 0],...`
	`'BackFaceLighting','lit')`

6 Set viewpoint.	`view([30,25])` `set(gca,'CameraViewAngleMode','Manual')`
7 Set axis limits and tick marks.	`axis([5 15 5 15 -8 8])` `set(gca,'ZTickLabel','Negative||Positive')`

Step	Typical Code
8 Set aspect ratio.	`set(gca,'PlotBoxAspectRatio',[2.5 2.5 1])`

9 Annotate the graph with axis labels, legend, and text.	`xlabel('X Axis')`
	`ylabel('Y Axis')`
	`zlabel('Function Value')`
	`title('Peaks')`

10 Print graph.	`set(gcf,'PaperPositionMode','auto')` `print -dps2`

Line Plots of 3-D Data

In this section...

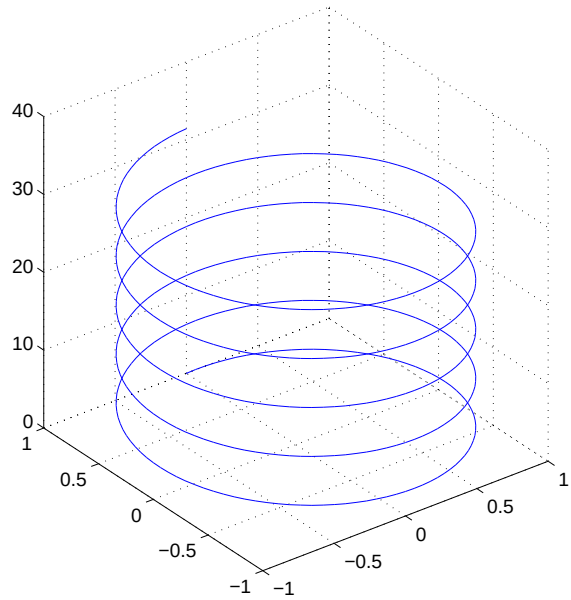
Basic 3-D Plotting: The plot3 function

The 3-D analog of the `plot` function is `plot3`. If `x`, `y`, and `z` are three vectors of the same length,

```
plot3(x,y,z)
```

generates a line in 3-D through the points whose coordinates are the elements of `x`, `y`, and `z` and then produces a 2-D projection of that line on the screen. For example, these statements produce a helix:

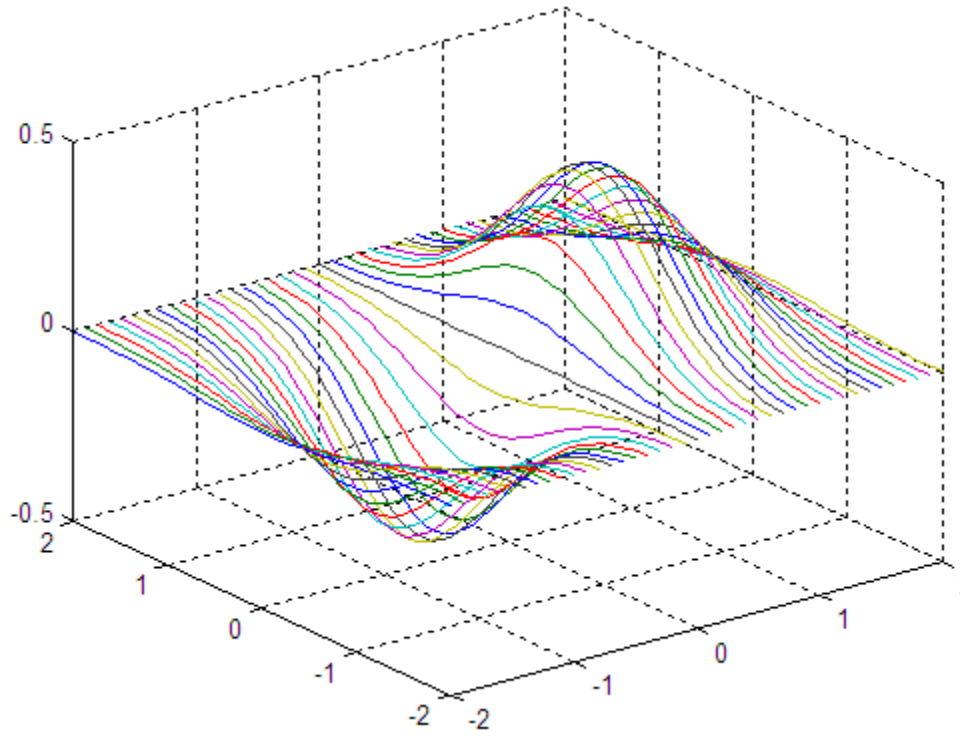
```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
axis square; grid on
```



Plotting Matrix Data

If the arguments to `plot3` are matrices of the same size, lines obtained from the columns of `X`, `Y`, and `Z` are plotted. For example:

```
[X,Y] = meshgrid([-2:0.1:2]);
Z = X.*exp(-X.^2-Y.^2);
plot3(X,Y,Z)
grid on
```



Notice how line colors cycle, based on the axes `ColorOrder` property.

Representing Data as a Surface

In this section...

Functions for Plotting Data Grids

MATLAB® graphics defines a surface by the *z*-coordinates of points above a rectangular grid in the *x*-*y* plane. The plot is formed by joining adjacent points with straight lines. Surface plots are useful for visualizing matrices that are too large to display in numerical form and for graphing functions of two variables.

MATLAB can create different forms of surface plots. Mesh plots are wire-frame surfaces that color only the lines connecting the defining points. Surface plots display both the connecting lines and the faces of the surface in color. This table lists the various forms.

Function	Used to Create
`mesh, surf`	Surface plot
`meshc, surfc`	Surface plot with contour plot beneath it
`meshz`	Surface plot with curtain plot (reference plane)
`pcolor`	Flat surface plot (value is proportional only to color)
`surfl`	Surface plot illuminated from specified direction
`surface`	Low-level function (on which high-level functions are based) for creating surface graphics objects

Functions for Gridding and Interpolating Data

These functions are useful when you need to restructure and interpolate data so that you can represent this data as a surface.

Function	Used to Create
`meshgrid`	Rectangular grid in 2-D and 3-D space
`griddata`	Interpolate scattered data
`griddedInterpolant`	Interpolant for gridded data
`scatteredInterpolant`	Interpolate scattered data

For a discussion of how to interpolate data, see "Interpolating Gridded Data" and "Interpolating Scattered Data".

Mesh and Surface Plots

The `mesh` and `surf` commands create 3-D surface plots of matrix data. If `Z` is a matrix for which the elements `Z(i,j)` define the height of a surface over an underlying `(i,j)` grid, then

```
mesh(Z)
```

generates a colored, wire-frame view of the surface and displays it in a 3-D view. Similarly,

```
surf(Z)
```

generates a colored, faceted view of the surface and displays it in a 3-D view. Ordinarily, the facets are quadrilaterals, each of which is a constant color, outlined with black mesh lines, but the `shading` command allows you to eliminate the mesh lines (`shading flat`) or to select interpolated shading across the facet (`shading interp`).

Surface object properties provide additional control over the visual appearance of the surface. You can specify edge line styles, vertex markers, face coloring, lighting characteristics, and so on.

Visualizing Functions of Two Variables

1. To display a function of two variables, $z = f(x,y)$, generate `X` and `Y` matrices consisting of repeated rows and columns, respectively, over the domain of the function. You will use these matrices to evaluate and graph the function.

2. The `meshgrid` function transforms the domain specified by two vectors, `x` and `y`, into matrices `X` and `Y`. You then use these matrices to evaluate functions of two variables: The rows of `X` are copies of the vector `x` and the columns of `Y` are copies of the vector `y`.

Example: Illustrating the Use of meshgrid

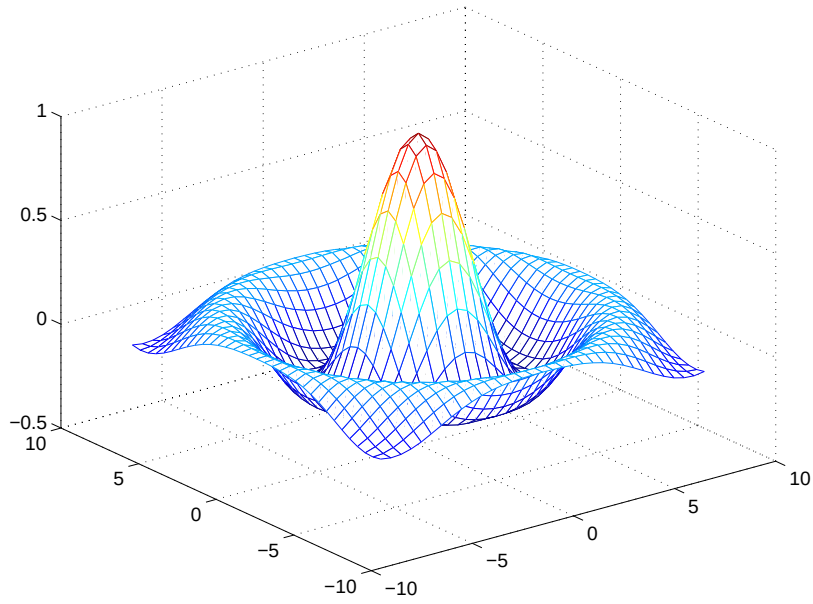
To illustrate the use of `meshgrid`, consider the `sin(r)/r` or `sinc` function. To evaluate this function between -8 and 8 in both x and y, you need pass only one vector argument to `meshgrid`, which is then used in both directions.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
```

The matrix `R` contains the distance from the center of the matrix, which is the origin. Adding `eps` prevents the divide by zero (in the next step) that produces `Inf` values in the data.

Forming the `sinc` function and plotting `Z` with `mesh` results in the 3-D surface.

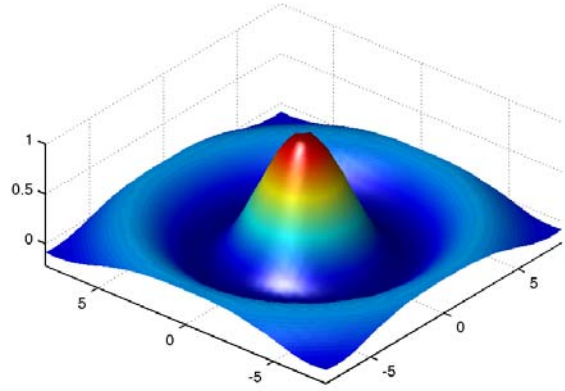
```
Z = sin(R)./R;
figure
mesh(X,Y,Z)
```



Emphasizing Surface Shape

MATLAB provides a number of techniques that can enhance the information content of your graphs. For example, this graph of the `sinc` function uses the same data as the previous graph, but employs lighting and view adjustment to emphasize the shape of the graphed function (`daspect`, `axis`, `view`, `camlight`).

```
figure
surf(X,Y,Z,'FaceColor','interp',...
   'EdgeColor','none',...
   'FaceLighting','phong')
daspect([5 5 1])
axis tight
view(-50,30)
camlight left
```



See the `surf` function for more information on surface plots.

Surface Plots of Nonuniformly Sampled Data

You can use `meshgrid` to create a grid of uniformly sampled data points at which to evaluate and graph the `sinc` function. MATLAB then constructs the surface plot by connecting neighboring matrix elements to form a mesh of quadrilaterals.

To produce a surface plot from nonuniformly sampled data, use `scatteredInterpolant` to interpolate the values at uniformly spaced points, and then use `mesh` and `surf` in the usual way.

Example – Displaying Nonuniform Data on a Surface

This example evaluates the `sinc` function at random points within a specific range and then generates uniformly sampled data for display as a surface plot. The process involves these tasks:

- Use `linspace` to generate evenly spaced values over the range of your unevenly sampled data.
- Use `meshgrid` to generate the plotting grid with the output of `linspace`.

- Use `scatteredInterpolant` to interpolate the irregularly sampled data to the regularly spaced grid returned by `meshgrid`.
- Use a plotting function to display the data.

1 Generate unevenly sampled data within the range [-8, 8] and use it to evaluate the function:

```
x = rand(100,1)*16 - 8;
y = rand(100,1)*16 - 8;
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r)./r;
```

2 The `linspace` function provides a convenient way to create uniformly spaced data with the desired number of elements. The following statements produce vectors over the range of the random data with the same resolution as that generated by the -8:.5:8 statement in the previous `sinc` example:

```
xlin = linspace(min(x),max(x),33);
ylin = linspace(min(y),max(y),33);
```

3 Now use these points to generate a uniformly spaced grid:

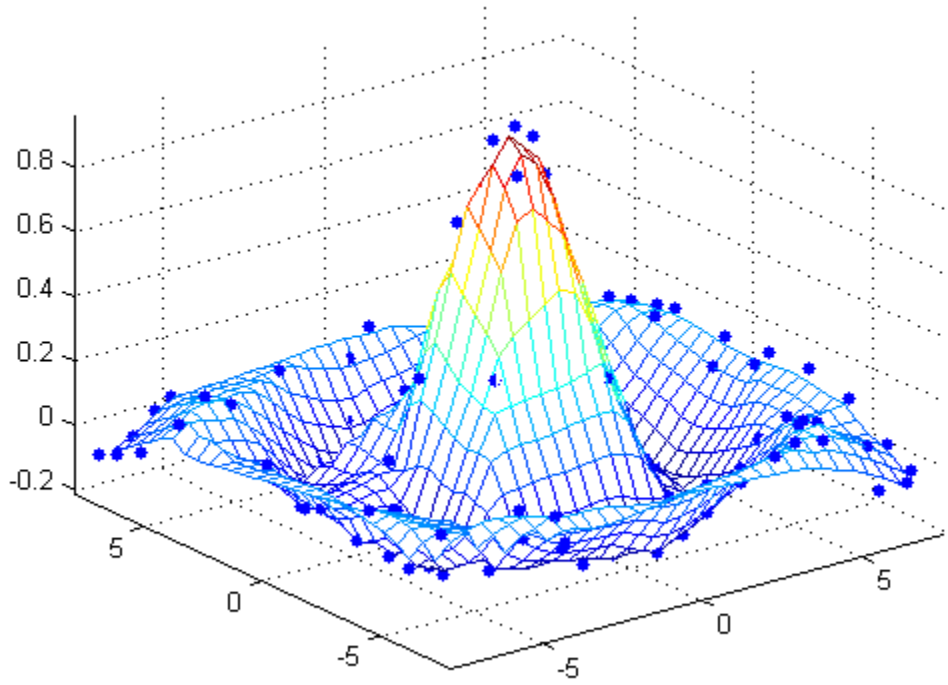
```
[X,Y] = meshgrid(xlin,ylin);
```

4 The key to this process is to use `scatteredInterpolant` to interpolate the values of the function at the uniformly spaced points, based on the values of the function at the original data points (which are random in this example). This statement uses the default linear interpolation to generate the new data:

```
f = scatteredInterpolant(x,y,z);
Z = f(X,Y);
```

5 Plot the interpolated and the nonuniform data to produce:

```
figure
mesh(X,Y,Z) %interpolated
axis tight; hold on
plot3(x,y,z,'.','MarkerSize',15) %nonuniform
```



Reshaping Data

Suppose you have a collection of data with the following (X, Y, Z) triplets:

X	Y	Z
1	1	152
2	1	89
3	1	100
4	1	100
5	1	100

X	Y	Z
1	2	103
2	2	0
3	2	100
4	2	100
5	2	100
1	3	89
2	3	13
3	3	100
4	3	100
5	3	100
1	4	115
2	4	100
3	4	187
4	4	200
5	4	111
1	5	100
2	5	85
3	5	111
4	5	97
5	5	48

You can represent data that is in vector form using various MATLAB graph types, such as `surf`, `contour`, and `stem3`, by first restructuring the data. Use the (X, Y) values to define the coordinates in an x-y plane at which there is a Z value. The `reshape` and `transpose` functions can restructure your data so that the (X, Y, Z) triplets form a rectangular grid:

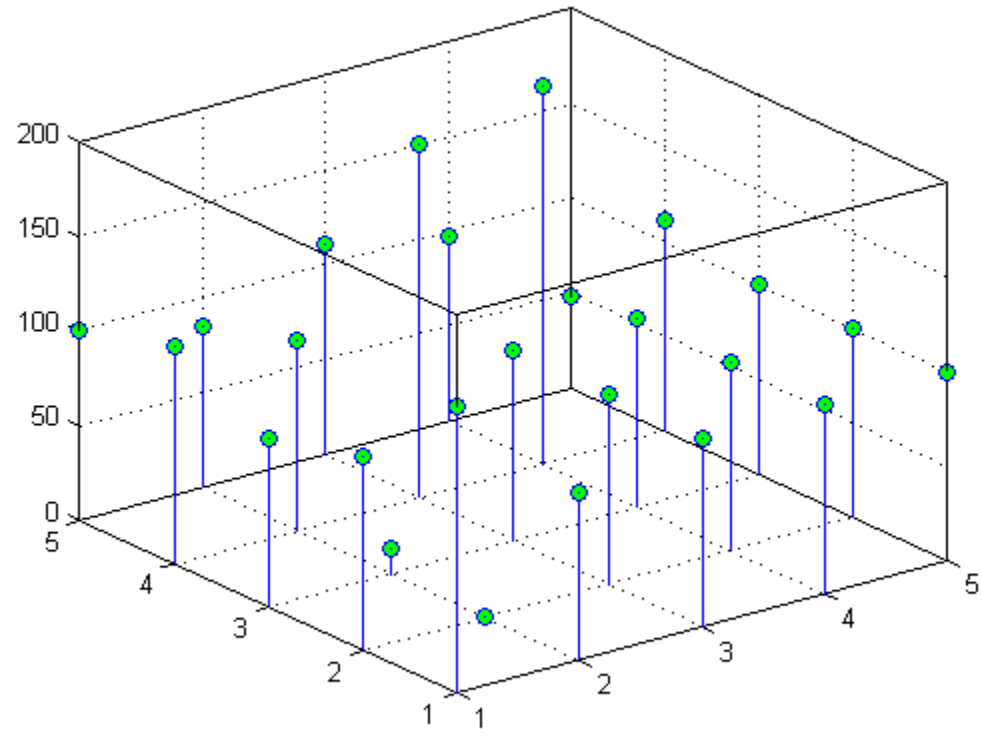
```
x = reshape(X,5,5)';
y = reshape(Y,5,5)';
z = reshape(Z,5,5)';
```

Reshaping results in three 5–by-5 arrays:

```
x =

     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5

y =

     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
     4     4     4     4     4
     5     5     5     5     5

z =

   152    89   100   100   100
   103     0   100   100   100
    89    13   100   100   100
   115   100   187   200   111
   100    85   111    97    48
```

You can now represent the values of Z with respect to X and Y. For example, create a 3–D stem graph:

```
stem3(x,y,z,'MarkerFaceColor','g')
```



Parametric Surfaces

The functions that draw surfaces can take two additional vector or matrix arguments to describe surfaces with specific *x* and *y* data. If `Z` is an *m*-by-*n* matrix, `x` is an *n*-vector, and `y` is an *m*-vector, then

```
mesh(x,y,Z,C)
```

describes a mesh surface with vertices having color `C(i,j)` and located at the points

```
(x(j), y(i), Z(i,j))
```

where `x` corresponds to the columns of `Z` and `y` to its rows.

More generally, if `X`, `Y`, `Z`, and `C` are matrices of the same dimensions, then

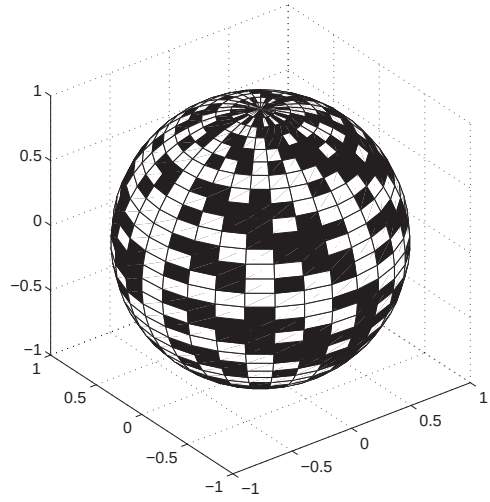
```
mesh(X,Y,Z,C)
```

describes a mesh surface with vertices having color `C(i,j)` and located at the points

```
(X(i,j), Y(i,j), Z(i,j))
```

This example uses spherical coordinates to draw a sphere and color it with the pattern of pluses and minuses in a Hadamard matrix, an orthogonal matrix used in signal processing coding theory. The vectors `theta` and `phi` are in the range $-\pi \le$ `theta` $\le \pi$ and $-\pi/2 \le$ `phi` $\le \pi/2$. Because `theta` is a row vector and `phi` is a column vector, the multiplications that produce the matrices `X`, `Y`, and `Z` are vector outer products.

```
figure
k = 5;
n = 2^k-1;
theta = pi*(-n:2:n)/n;
phi = (pi/2)*(-n:2:n)'/n;
X = cos(phi)*cos(theta);
Y = cos(phi)*sin(theta);
Z = sin(phi)*ones(size(theta));
colormap([0 0 0;1 1 1])
C = hadamard(2^k);
surf(X,Y,Z,C)
axis square
```

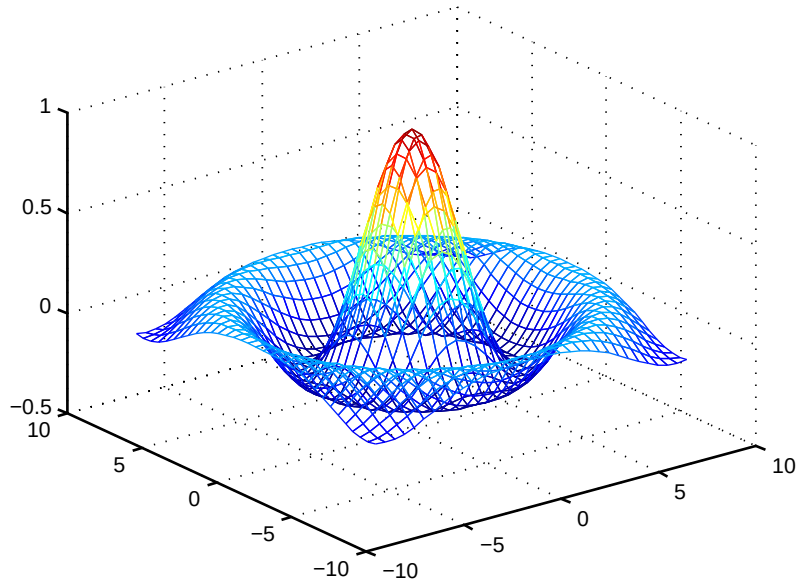


Hidden Line Removal

By default, MATLAB removes lines that are hidden from view in mesh plots, even though the faces of the plot are not colored. You can disable hidden line removal and allow the faces of a mesh plot to be transparent with the command

```
hidden off
```

This is the surface plot with `hidden` set to `off`.


1
0.5
0
−0.5
10
5
0
−5
−10
−10
−5
0
5
10

Coloring Mesh and Surface Plots

In this section...

Coloring Techniques

You can enhance the information content of surface plots by controlling the way MATLAB graphics apply color to these plots. MATLAB can map particular data values to colors specified explicitly or can map the entire range of data to a predefined range of colors called a *colormap*.

You can apply three different coloring techniques:

- "Indexed Color Surfaces — Direct and Scaled Color Mapping" on page 1-23 — MATLAB colors the surface plot by assigning each data point an index into the figure's colormap. The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated).
- "Truecolor Surfaces" on page 1-27 — MATLAB colors the surface plot using the explicitly specified colors (i.e., the RGB triplets). The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated). To be rendered accurately, truecolor requires computers with 24-bit displays; however, MATLAB simulates truecolor on indexed systems. See the `shading` command for information on the types of shading.
- "Texture Mapping" on page 1-29 — Texture mapping displays a 2-D image mapped onto a 3-D surface.

Types of Color Data

The type of color data you specify (i.e., single values or RGB triplets) determines how MATLAB interprets it. When you create a surface plot, you can:

- Provide no explicit color data, in which case MATLAB generates colormap indices from the *z*-data.
- Specify an array of color data that is equal in size to the *z*-data and is used for indexed colors.
- Specify an *m*-by-*n*-by-3 array of color data that defines an RGB triplet for each element in the *m*-by-*n* *z*-data array and is used for truecolor.

Colormaps

Each MATLAB figure window has a colormap associated with it. A colormap is simply a three-column matrix whose length is equal to the number of colors it defines. Each row of the matrix defines a particular color by specifying three values in the range zero to one. These values define the RGB components (i.e., the intensities of the red, green, and blue video components).

The `colormap` function, with no arguments, returns the current figure's colormap. For example, the MATLAB hsv colormap contains 64 colors by default and the 33rd color is cyan.

```
colormap hsv
cm = colormap;
cm(33,:)
ans =
      0   1   1
```

RGB Color Components

This table lists some representative RGB color definitions.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red

Red	Green	Blue	Color
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
1	0	1	Magenta
0	1	1	Cyan
0.5	0.5	0.5	Gray
0.5	0	0	Dark red
1	0.62	0.40	Copper
0.49	1	0.83	Aquamarine

You can create colormaps with MATLAB array operations or you can use any of several functions that generate useful maps, including `hsv`, `hot`, `cool`, `summer`, and `gray`. Each function has an optional parameter that specifies the number of rows in the resulting map.

For example,

```
hot(m)
```

creates an *m*-by-3 matrix whose rows specify the RGB intensities of a map that varies from black, through shades of red, orange, and yellow, to white.

If you do not specify the colormap length, MATLAB creates a colormap the same length as the current colormap. The default colormap is `jet(64)`.

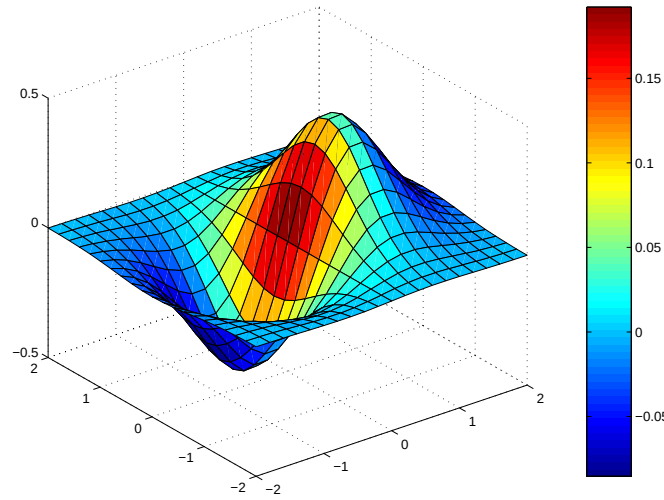
If you use long colormaps (> 64 colors) in each of several figure windows, it might become necessary for the operating system to swap in different color lookup tables as the active focus is moved among the windows.

Displaying Colormaps

The `colorbar` function displays the current colormap, either vertically or horizontally, in the figure window along with your graph. For example, the statements

```
[x,y] = meshgrid([-2:.2:2]);
Z = x.*exp(-x.^2-y.^2);
surf(x,y,Z,gradient(Z))
colorbar
```

produce a surface plot and a vertical strip of color corresponding to the colormap. The colorbar indicates the mapping of data value to color with the axis labels.



Indexed Color Surfaces — Direct and Scaled Color Mapping

MATLAB can use two different methods to map indexed color data to the colormap — direct and scaled.

Direct Mapping

Direct mapping uses the color data directly as indices into the colormap. For example, a value of 1 points to the first color in the colormap, a value of 2 points to the second color, and so on. If the color data is noninteger, MATLAB rounds it toward zero. Values greater than the number of colors in

the colormap are set equal to the last color in the colormap (i.e., the number `length(colormap)`). Values less than one are set to one.

Scaled Mapping

Scaled mapping uses a two-element vector `[cmin cmax]` (specified with the `caxis` command) to control the mapping of color data to the figure colormap. `cmin` specifies the data value to map to the first color in the colormap and `cmax` specifies the data value to map to the last color in the colormap. Data values in between are linearly transformed from the second to the next-to-last color, using the expression

```
colormap_index = fix((color_data-cmin)/(cmax-cmin)*cm_length)+1
```

`cm_length` is the length of the colormap.

By default, MATLAB sets `cmin` and `cmax` to span the range of the color data of all graphics objects within the axes. However, you can set these limits to any range of values. This enables you to display multiple axes within a single figure window and use different portions of the figure's colormap for each one. See "Calculating Color Limits" in the MATLAB Graphics documentation for an example that uses color limits.

By default, MATLAB uses scaled mapping. To use direct mapping, you must turn off scaling when you create the plot. For example:

```
surf(Z,C,'CDataMapping','direct')
```

See `surface` for more information on specifying color data.

Specifying Indexed Colors

When creating a surface plot with a single matrix argument, `surf(Z)` for example, the argument `Z` specifies both the height and the color of the surface. MATLAB transforms `Z` to obtain indices into the current colormap.

With two matrix arguments, the statement

```
surf(Z,C)
```

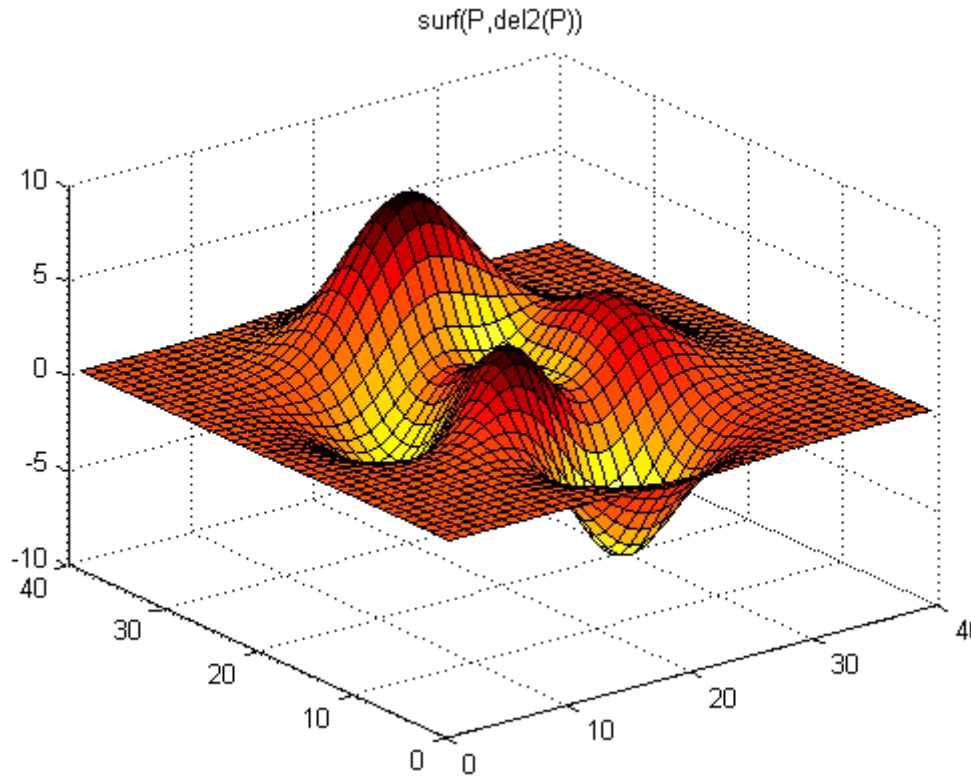
independently specifies the color using the second argument.

Example — Mapping Surface Curvature to Color

The Laplacian of a surface plot is related to its curvature; it is positive for functions shaped like `i^2 + j^2` and negative for functions shaped like `-(i^2 + j^2)`. The function `del2` computes the discrete Laplacian of any matrix. For example, use `del2` to determine the color for the data returned by `peaks`.

```
P = peaks(40);
C = del2(P);
surf(P,C)
colormap hot
```

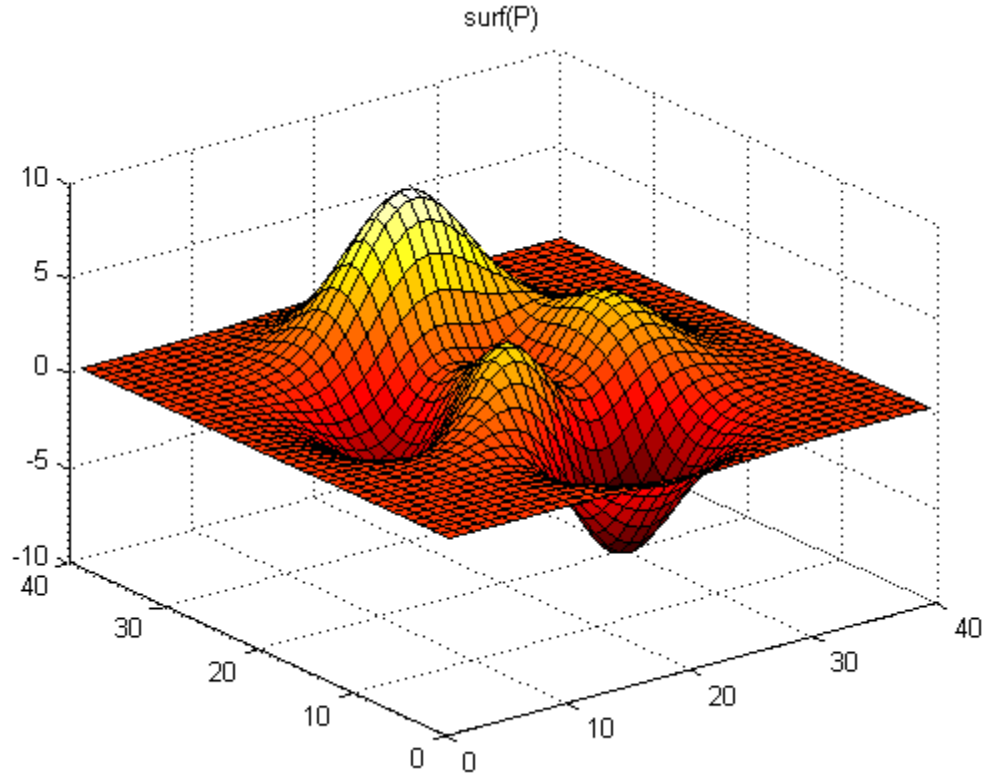
Creating a color array by applying the Laplacian to the data is useful because it causes regions with similar curvature to be drawn in the same color.



Compare this surface coloring with that produced by the statements

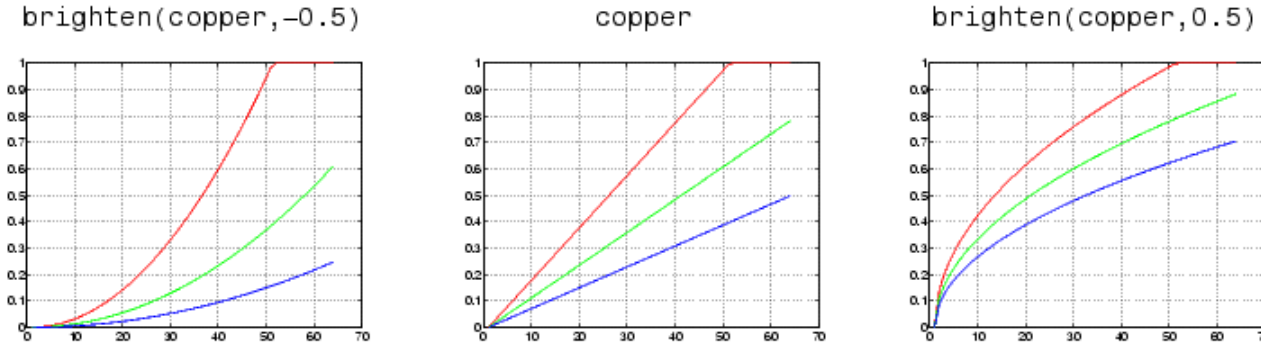
```
surf(P)
colormap hot
```

which use the same colormap, but map regions with similar *z* value (height above the *x*-*y* plane) to the same color.



Altering Colormaps

Because colormaps are matrices, you can manipulate them like other arrays. The `brighten` function takes advantage of this fact to increase or decrease the intensity of the colors. Plotting the values of the R, G, and B components of a colormap using `rgbplot` illustrates the effects of `brighten`.



NTSC Color Encoding

The brightness component of television signals uses the National Television System Committee (NTSC) color encoding scheme.

```
b = .30*red + .59*green + .11*blue
  = sum(diag([.30 .59 .11])*map')';
```

Using the nonlinear grayscale map,

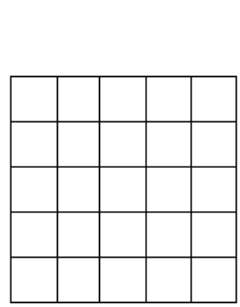
```
colormap([b b b])
```

effectively converts a color image to its NTSC black-and-white equivalent.

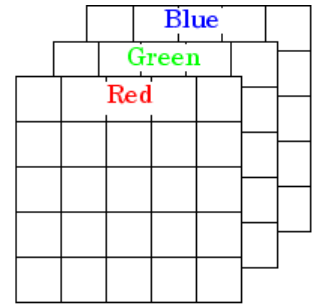
Truecolor Surfaces

Computer systems with 24-bit displays are capable of displaying over 16 million (2^{24}) colors, as opposed to the 256 colors available on 8-bit displays. You can take advantage of this capability by defining color data directly as RGB values and eliminating the step of mapping numerical values to locations in a colormap.

Specify truecolor using an *m*-by-*n*-by-3 array, where the size of `Z` is *m*-by-*n*.



m-by-*n* matrix defining surface plot

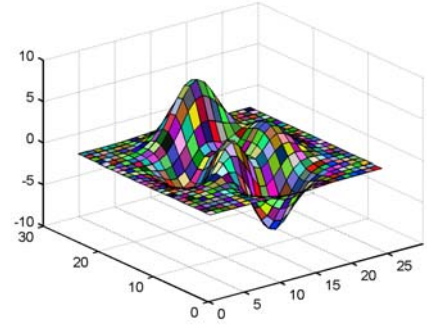


Corresponding *m*-by-*n*-by-3 matrix specifying truecolor for the surface plot

For example, the statements

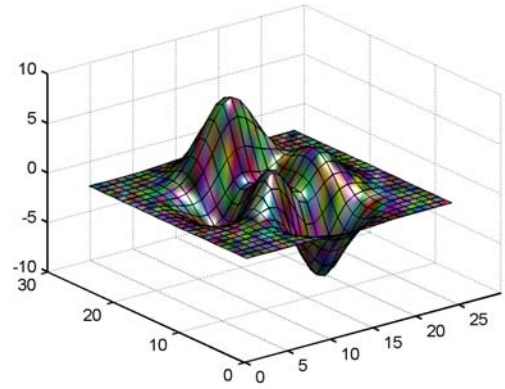
```
Z = peaks(25);
C(:,:,1) = rand(25);
C(:,:,2) = rand(25);
C(:,:,3) = rand(25);
surf(Z,C)
```

create a plot of the `peaks` matrix with random coloring.



You can set surface properties as with indexed color.

```
surf(Z,C,'FaceColor','interp','FaceLighting','phong')
camlight right
```



Rendering Methods for Truecolor

MATLAB always uses either OpenGL® or the Z-buffer rendering method when displaying truecolor. If the figure `RendererMode` property is set to `auto`, MATLAB automatically switches the value of the `Renderer` property to `zbuffer` whenever you specify truecolor data.

If you explicitly set `Renderer` to `painters` (this sets `RendererMode` to `manual`) and attempt to define an image, patch, or surface object using truecolor, MATLAB returns a warning and does not render the object.

See the `image`, `patch`, and `surface` functions for information on defining truecolor for these objects.

Texture Mapping

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

Texture mapping allows the dimensions of the color data array to be different from the data defining the surface plot. You can apply an image of arbitrary

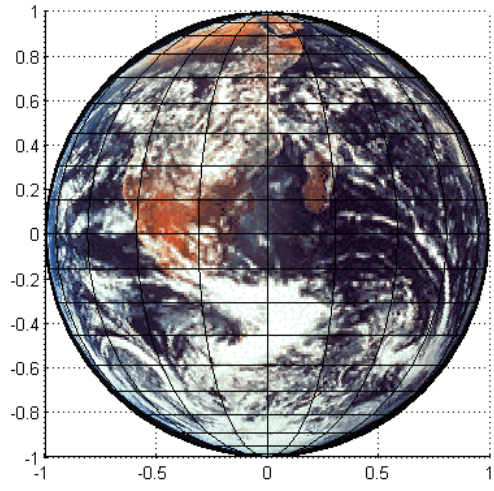
size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

Example — Texture Mapping a Surface

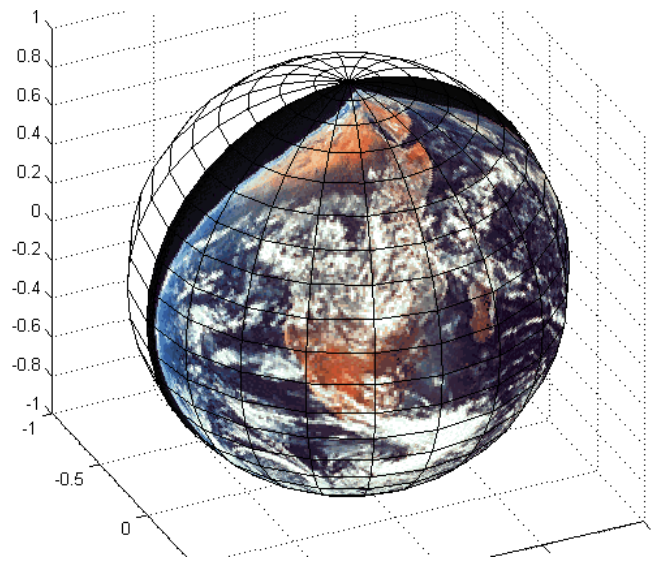
This example creates a spherical surface using the `sphere` function and texture maps it with an image of the earth taken from space. Because the earth image is a view of earth from one side, this example maps the image to only one side of the sphere, padding the image data with 1s. In this case, the image data is a 257-by-250 matrix, so it is padded equally on each side with two 257-by-125 matrices of 1s by concatenating the three matrices.

To use texture mapping, set the `FaceColor` to `texturemap` and assign the image to the surface's `CData`.

```
load earth % Load image data, X, and colormap, map
sphere; h = findobj('Type','surface');
hemisphere = [ones(257,125),...
              X,...
              ones(257,125)];
set(h,'CData',flipud(hemisphere),'FaceColor','texturemap')
colormap(map)
axis equal
view([90 0])
```




```
set(gca,'CameraViewAngleMode','manual')
view([65 30])
```



2

Defining the View

View Overview

In this section...

Viewing 3-D Graphs and Scenes

The *view* is the particular orientation you select to display your graph or graphical scene. The term *viewing* refers to the process of displaying a graphical scene from various directions, zooming in or out, changing the perspective and aspect ratio, flying by, and so on.

This section describes how to define the various viewing parameters to obtain the view you want. Generally, viewing is applied to 3-D graphs or models, although you might want to adjust the aspect ratio of 2-D views to achieve specific proportions or make a graph fit in a particular shape.

MATLAB viewing is composed of two basic areas:

- Positioning the viewpoint to orient the scene
- Setting the aspect ratio and relative axis scaling to control the shape of the objects being displayed

Positioning the Viewpoint

- Setting the Viewpoint -- Discusses how to specify the point from which you view a graph in terms of azimuth and elevation. This is conceptually simple, but does have limitations.
- Defining Scenes with Camera Graphics, View Control with the Camera Toolbar, and Camera Graphics Functions — How to compose complex scenes using the MATLAB camera viewing model.
- Dollying the Camera and Moving the Camera Through a Scene -- Programming techniques for moving the view around and through scenes.

- Low-Level Camera Properties — The graphics properties that control the camera and illustrates the effects they cause.

Setting the Aspect Ratio

- View Projection Types -- Describes orthographic and perspective projection types and illustrates their use.
- Understanding Axes Aspect Ratio and Axes Aspect Ratio Properties — How MATLAB sets the aspect ratio of the axes and how you can select the most appropriate setting for your graphs.

Default Views

MATLAB automatically sets the view when you create a graph. The actual view that MATLAB selects depends on whether you are creating a 2- or 3-D graph. See “Default Viewpoint Selection” on page 2-30 and “Default Aspect Ratio Selection” on page 2-47 for a description of how MATLAB defines the standard view.

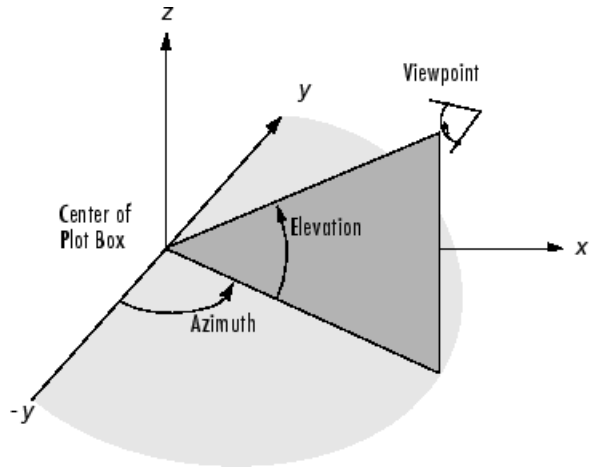
Setting the Viewpoint with Azimuth and Elevation

Azimuth and Elevation

You can control the orientation of the graphics displayed in an axes using MATLAB graphics functions. You can specify the viewpoint, view target, orientation, and extent of the view displayed in a figure window. These viewing characteristics are controlled by a set of graphics properties. You can specify values for these properties directly or you can use the `view` command and rely on MATLAB automatic property selection to define a reasonable view.

The `view` command specifies the viewpoint by defining azimuth and elevation with respect to the axis origin. Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



Default 2-D and 3-D Views

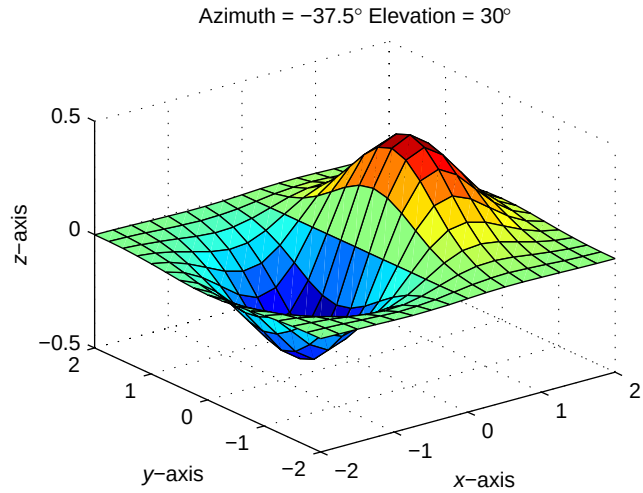
MATLAB automatically selects a viewpoint that is determined by whether the plot is 2-D or 3-D:

- For 2-D plots, the default is azimuth = 0° and elevation = 90°.
- For 3-D plots, the default is azimuth = -37.5° and elevation = 30°.

Examples of Views Specified with Azimuth and Elevation

For example, these statements create a 3-D surface plot and display it in the default 3-D view.

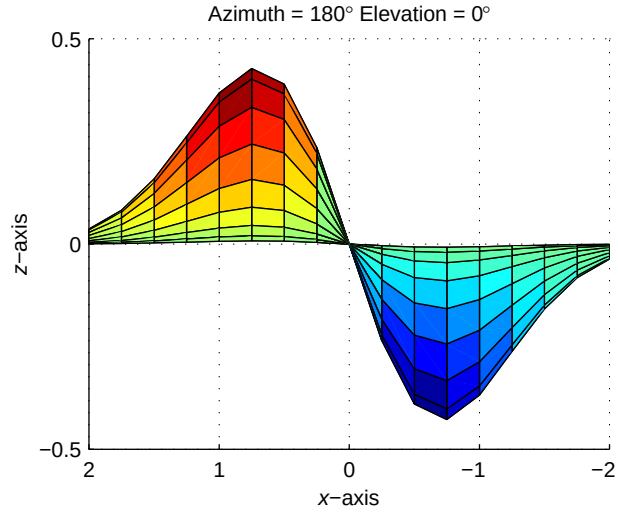
```
[X,Y] = meshgrid([-2:.25:2]);
Z = X.*exp(-X.^2 -Y.^2);
surf(X,Y,Z)
```



The statement

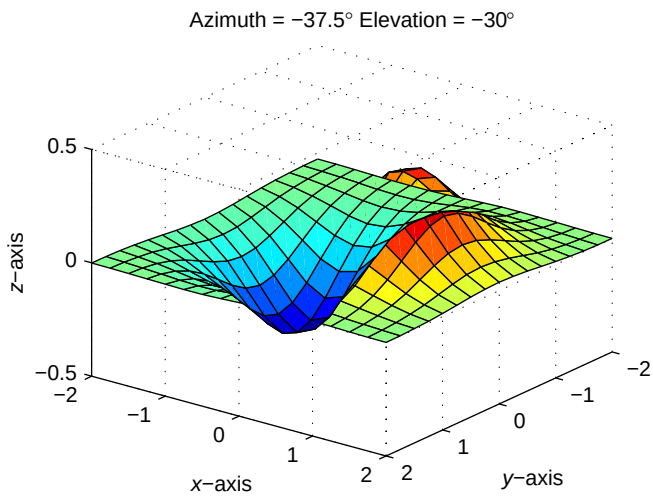
```
view([180 0])
```

sets the viewpoint so you are looking in the negative *y*-direction with your eye at the $z = 0$ elevation.



You can move the viewpoint to a location below the axis origin using a negative elevation.

```
view([-37.5 -30])
```



Limitations of Azimuth and Elevation

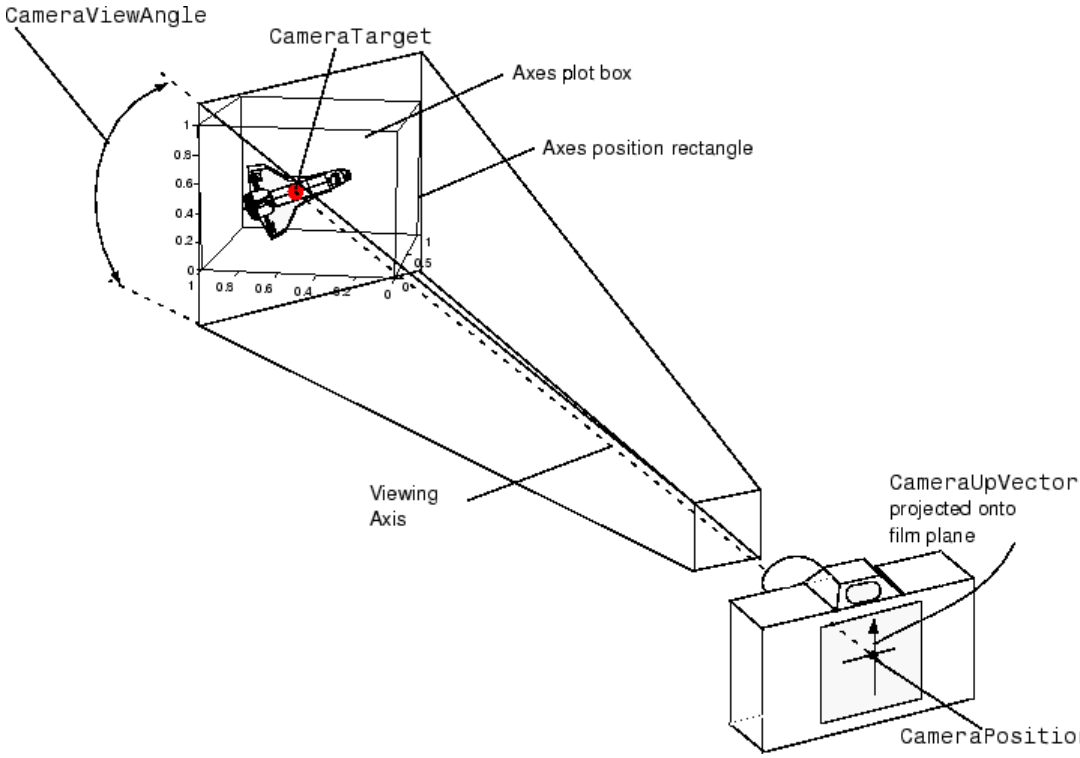
Specifying the viewpoint in terms of azimuth and elevation is conceptually simple, but it has limitations. It does not allow you to specify the actual position of the viewpoint, just its direction, and the *z*-axis is always pointing up. It does not allow you to zoom in and out on the scene or perform arbitrary rotations and translations.

MATLAB camera graphics provides greater control than the simple adjustments allowed with azimuth and elevation. The following sections discuss how to use camera properties to control the view.

Defining Scenes with Cameras

When you look at the graphics objects displayed in an axes, you are viewing a scene from a particular location in space that has a particular orientation with regard to the scene. MATLAB Graphics provides functionality, analogous to that of a camera with a zoom lens, that enables you to control the view of the scene created by MATLAB.

This picture illustrates how the camera is defined in terms of properties of the axes.



View Control with the Camera Toolbar

In this section...

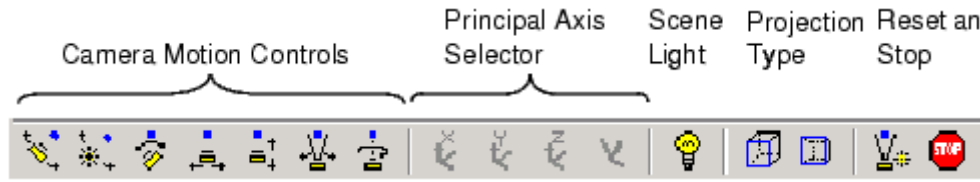
Camera Toolbar

The Camera toolbar enables you to perform a number of viewing operations interactively. To use the Camera toolbar,

- Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu or by typing `CameraToolbar` in the Command Window.
- Select the type of camera motion control you want to use by either clicking on the buttons or changing the `CameraToolbar` mode in the Command Window.
- Position the cursor over the figure window and click, hold down the right mouse button, then move the cursor in the desired direction.

The display updates immediately as you move the mouse.

The toolbar contains the following parts:



- Camera Motion Controls — These tools select which camera motion function to enable. You can also access the camera motion controls from the **Tools** menu.
- Principal Axis Selector — Some camera controls operate with respect to a particular axis. These selectors enable you to select the principal axis or to select nonaxis constrained motion. The selectors are grayed out when not applicable to the currently selected function. You can also access the principal axis selector from the **Tools** menu.
- Scene Light — The scene light button toggles a light source on or off in the scene (one light per axes).
- Projection Type — You can select orthographic or perspective projection types.
- Reset and Stop — Reset returns the scene to the standard 3-D view. Stop causes the camera to stop moving (this can be useful if you apply too much cursor movement). You can also access an expanded set of reset functions from the **Tools** menu.

Principal Axes

The principal axis of a scene defines the direction that is oriented upward on the screen. For example, a MATLAB surface plot aligns the up direction along the positive *z*-axis.

Principal axes constrain camera-tool motion along axes that are (on the screen) parallel and perpendicular to the principal axis that you select. Specifying a principal axis is useful if your data is defined with respect to a specific axis. Z is the default principal axis, because this matches the MATLAB default 3-D view.

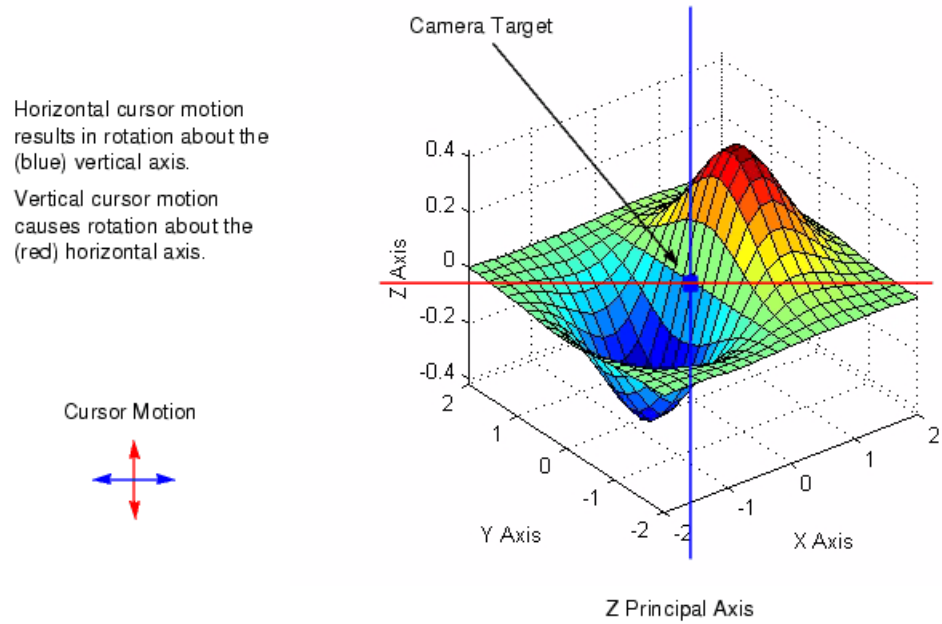
Two of the camera tools (Orbit and Pan/Tilt) allow you to select a principal axis as well as axis-free motion. On the screen, the axes of rotation are

determined by a vertical and a horizontal line, both of which pass through the point defined by the `CameraTarget` property and are parallel and perpendicular to the principal axis.

For example, when the principal axis is z, movement occurs about

- A vertical line that passes through the camera target and is parallel to the *z*-axis
- A horizontal line that passes through the camera target and is perpendicular to the *z*-axis

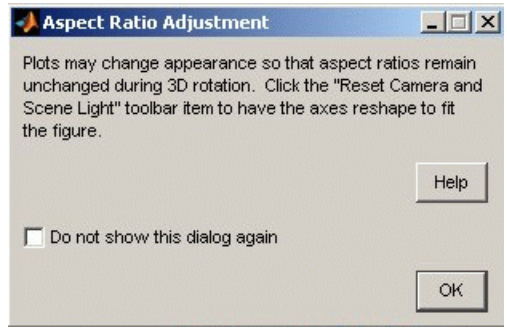
This means the scene (or camera, as the case may be) moves in an arc whose center is at the camera target. The following picture illustrates the rotation axes for a *z* principal axis.



The axes of rotation always pass through the camera target.

Optimizing for 3-D Camera Motion

When you create a plot, MATLAB displays it with an aspect ratio that fits the figure window. This behavior might not create an optimum situation for the manipulation of 3-D graphics, as it can lead to distortion as you move the camera around the scene. To avoid possible distortion, it is best to switch to a 3-D visualization mode (enabled from the command line with the command `axis vis3d`). When using the Camera toolbar, MATLAB automatically switches to the 3-D visualization mode, but warns you first with the following dialog box.



This dialog box appears only once per MATLAB session.

For more information about the underlying effects of related camera properties, see “Understanding Axes Aspect Ratio” on page 2-41. The next section, “Camera Motion Controls” on page 2-12, discusses how to use each tool.

Camera Motion Controls

This section discusses the individual camera motion functions selectable from the toolbar.

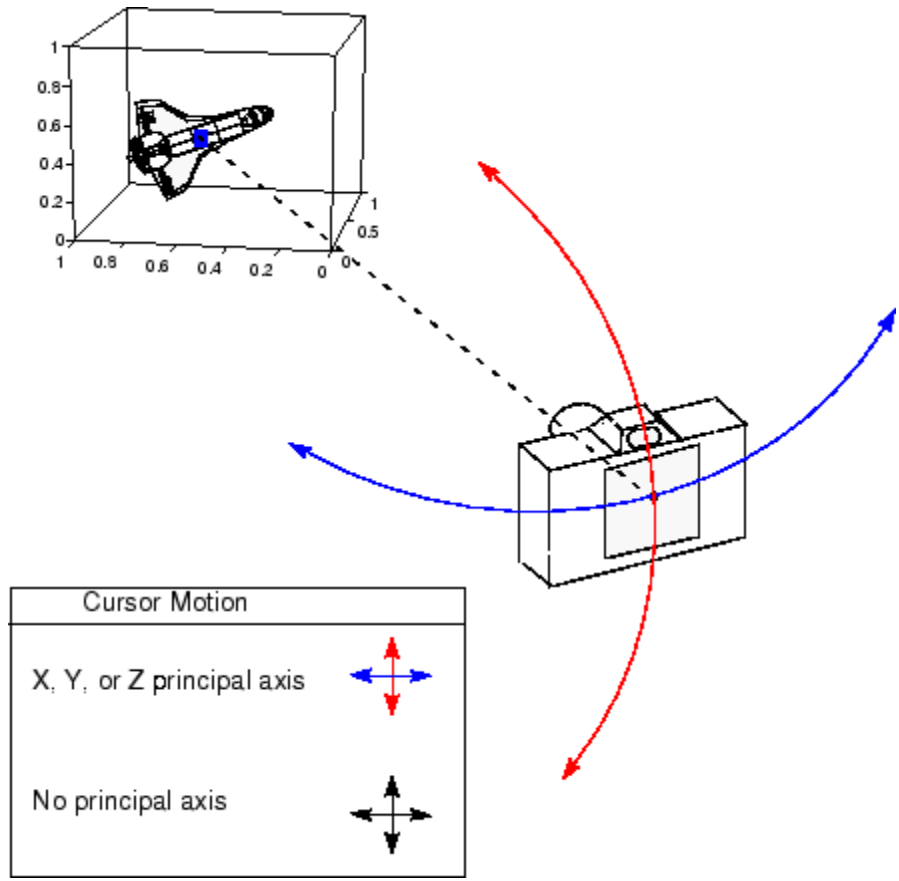
Note When interpreting the following diagrams, keep in mind that the camera always points towards the camera target. See “Defining Scenes with Cameras” on page 2-8 for an illustration of the graphics properties involved in camera motion.

Orbit Camera

Orbit Camera rotates the camera about the *z*-axis (by default). You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axis Selectors. When using no principal axis, you can rotate about an arbitrary axis.

Graphics Properties

Orbit Camera changes the `CameraPosition` property while keeping the `CameraTarget` fixed.



Orbit Scene Light

The scene light is a light source that is placed with respect to the camera position. By default, the scene light is positioned to the right of the camera (i.e., `camlight right`). Orbit Scene Light changes the light's offset from the camera position. There is only one scene light; however, you can add other lights using the `light` command.

Toggle the scene light on and off by clicking the yellow light bulb icon.

Graphics Properties

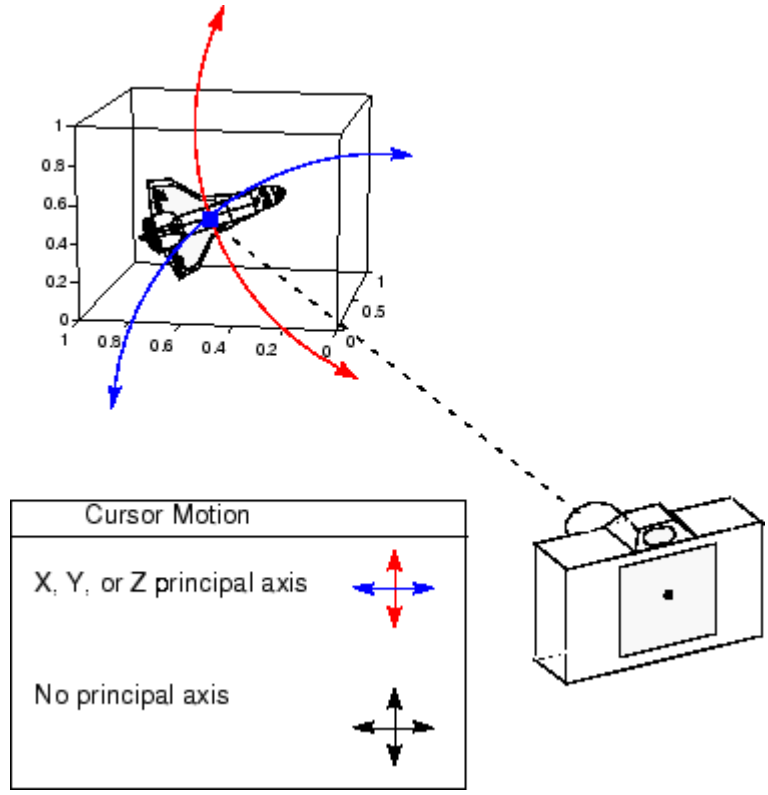
Orbit Scene Light moves the scene light by changing the light's `Position` property.

Pan/Tilt Camera

Pan/Tilt Camera moves the point in the scene that the camera points to while keeping the camera fixed. The movement occurs in an arc about the *z*-axis by default. You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axes Selectors.

Graphics Properties

Pan/Tilt Camera moves the point in the scene that the camera is pointing to by changing the `CameraTarget` property.

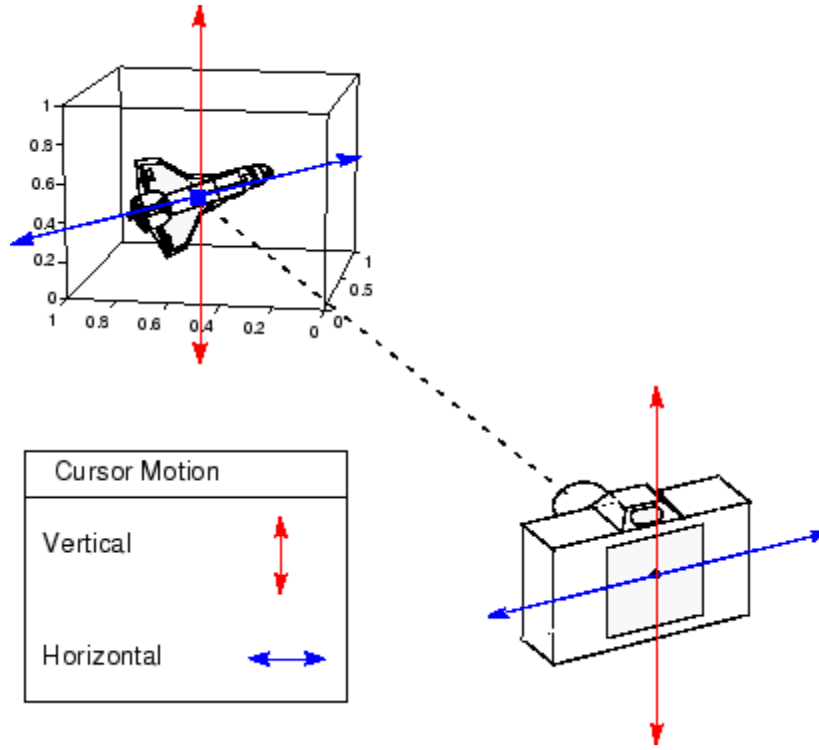


Move Camera Horizontally/Vertically

Moving the cursor horizontally or vertically (or any combination of the two) moves the scene in the same direction.

Graphics Properties

The horizontal and vertical movement is achieved by moving the `CameraPosition` and the `CameraTarget` in unison along parallel lines.

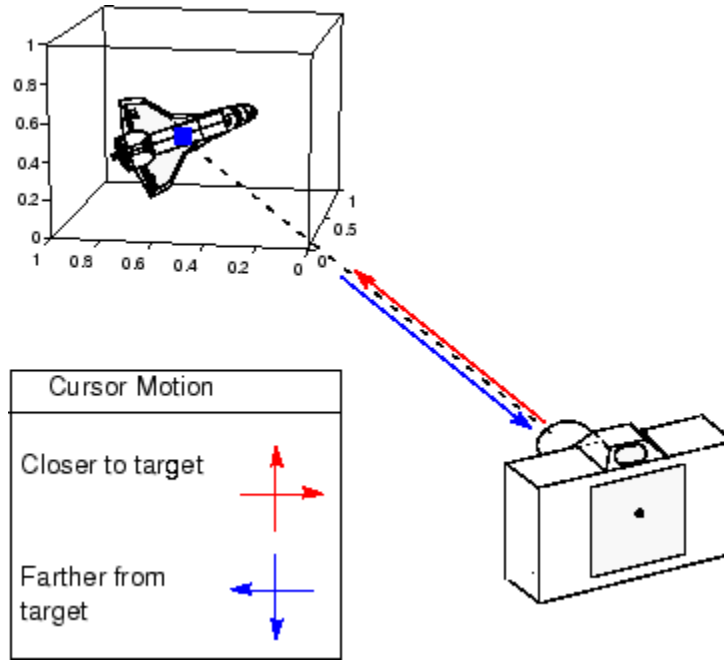


Move Camera Forward and Backward

Moving the cursor up or to the right moves the camera toward the scene. Moving the cursor down or to the left moves the camera away from the scene. It is possible to move the camera through objects in the scene and to the other side of the camera target.

Graphics Properties

This function moves the `CameraPosition` along the line connecting the camera position and the camera target.

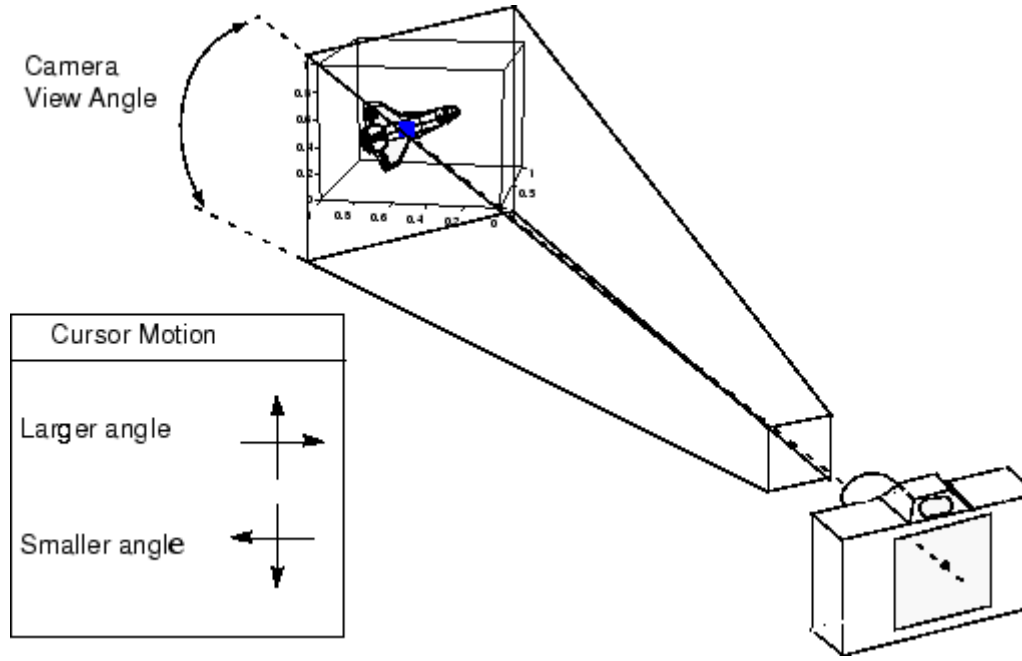


Zoom Camera

Zoom Camera makes the scene larger as you move the cursor up or to the right and smaller as you move the cursor down or to the left. Zooming does not move the camera and therefore cannot move the viewpoint through objects in the scene.

Graphics Properties

Zoom is implemented by changing the `CameraViewAngle`. The larger the angle, the smaller the scene appears, and vice versa.

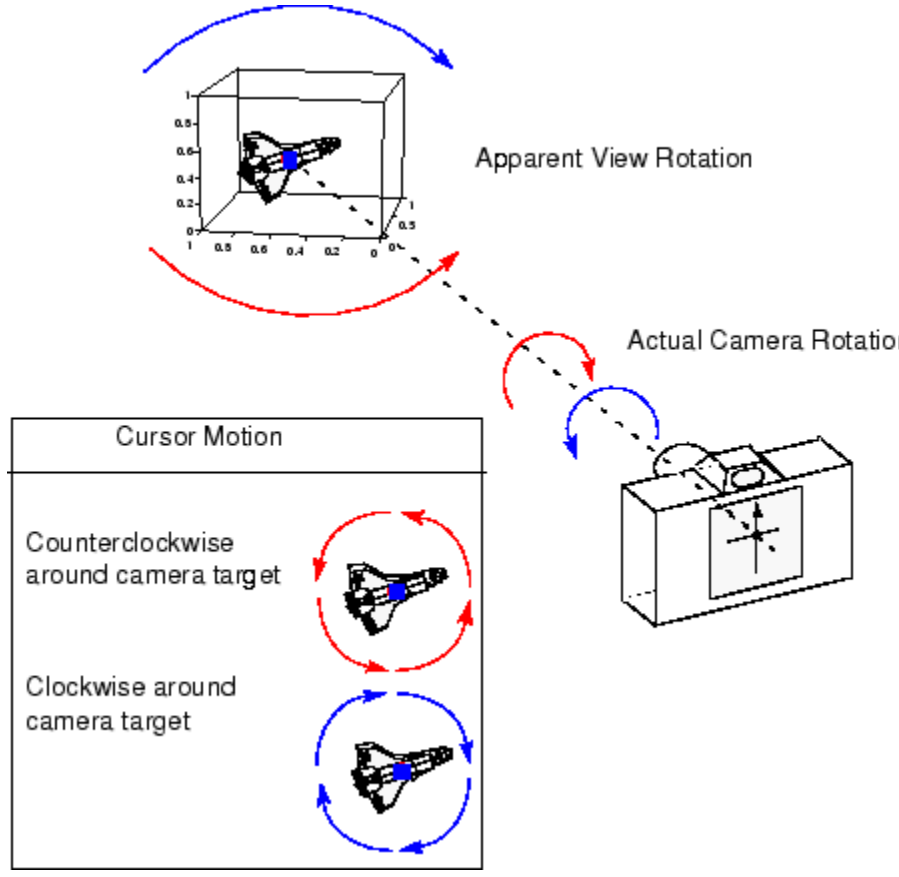


Camera Roll

Camera Roll rotates the camera about the viewing axis, thereby rotating the view on the screen.

Graphics Properties

Camera Roll changes the `CameraUpVector`.


Apparent View Rotation
Actual Camera Rotation
Cursor Motion
Counterclockwise around camera target
Clockwise around camera target

Camera Graphics Functions

The following table lists MATLAB functions that enable you to perform a number of useful camera maneuvers. The individual command descriptions provide information on using each one.

Function	Purpose
`camdolly`	Move camera position and target
`camlookat`	View specific objects
`camorbit`	Orbit the camera about the camera target
`campan`	Rotate the camera target about the camera position
`campos`	Set or get the camera position
`camproj`	Set or get the projection type (orthographic or perspective)
`camroll`	Rotate the camera about the viewing axis
`camtarget`	Set or get the camera target location
`camup`	Set or get the value of the camera up vector
`camva`	Set or get the value of the camera view angle
`camzoom`	Zoom the camera in or out on the scene

Dollying the Camera

In this section...

Summary of Techniques

In the camera metaphor, a dolly is a stage that enables movement of the camera from side to side with respect to the scene. The `camdolly` command implements similar behavior by moving both the position of the camera and the position of the camera target in unison (or just the camera position if you so desire).

This example illustrates how to use `camdolly` to explore different regions of an image. It shows how to use the following functions:

- `ginput` to obtain the coordinates of locations on the image
- The `camdolly` `data` coordinates option to move the camera and target to the new position based on coordinates obtained from `ginput`
- `camva` to zoom in and to fix the camera view angle, which is otherwise under automatic control

Implementation

First load the Cape Cod image and zoom in by setting the camera view angle (using `camva`).

```
load cape
image(X)
colormap(map)
axis image
camva(camva/2.5)
```

Then use `ginput` to select the *x*- and *y*-coordinates of the camera target and camera position.

```
while 1
```

```
  [x,y] = ginput(1);
  if ~strcmp(get(gcf,'SelectionType'),'normal')
    break
  end
  ct = camtarget;
  dx = x - ct(1);
  dy = y - ct(2);
  camdolly(dx,dy,ct(3),'movetarget','data')
  drawnow
end
```

Moving the Camera Through a Scene

In this section...

Summary of Techniques

A fly-through is an effect created by moving the camera through three-dimensional space, giving the impression that you are flying along with the camera as if in an aircraft. You can fly through regions of a scene that might be otherwise obscured by objects in the scene or you can fly by a scene by keeping the camera focused on a particular point.

To accomplish these effects you move the camera along a particular path, the *x*-axis for example, in a series of steps. To produce a fly-through, move both the camera position and the camera target at the same time.

The following example makes use of the fly-though effect to view the interior of an isosurface drawn within a volume defined by a vector field of wind velocities. This data represents air currents over North America.

This example employs a number of visualization techniques. It uses

- Isosurfaces and cone plots to illustrate the flow through the volume
- Lighting to illuminate the isosurface and cones in the volume
- Stream lines to define a path for the camera through the volume
- Coordinated motion of the camera position, camera target, and light

See `coneplot` for a fixed visualization of the same data.

Graphing the Volume Data

The first step is to draw the isosurface and plot the air flow using cone plots.

See `isosurface`, `isonormals`, `reducepatch`, and `coneplot` for information on using these commands.

Setting the data aspect ratio (`daspect`) to `[1,1,1]` before drawing the cone plot enables MATLAB software to calculate the size of the cones correctly for the final view.

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);

hpatch = patch(isosurface(x,y,z,wind_speed,35));
isonormals(x,y,z,wind_speed,hpatch)
set(hpatch,'FaceColor','red','EdgeColor','none');

[f vt] = reducepatch(isosurface(x,y,z,wind_speed,45),0.05);
daspect([1,1,1]);
hcone = coneplot(x,y,z,u,v,w,vt(:,1),vt(:,2),vt(:,3),2);
set(hcone,'FaceColor','blue','EdgeColor','none');
```

Setting Up the View

You need to define viewing parameters to ensure the scene is displayed correctly:

- Selecting a perspective projection provides the perception of depth as the camera passes through the interior of the isosurface (`camproj`).
- Setting the camera view angle to a fixed value prevents MATLAB from automatically adjusting the angle to encompass the entire scene as well as zooming in the desired amount (`camva`).

  ```
  camproj perspective
  camva(25)
  ```

Specifying the Light Source

Positioning the light source at the camera location and modifying the reflectance characteristics of the isosurface and cones enhances the realism of the scene:

- Creating a light source at the camera position provides a "headlight" that moves along with the camera through the isosurface interior (`camlight`).
- Setting the reflection properties of the isosurface gives the appearance of a dark interior (`AmbientStrength` set to 0.1) with highly reflective material (`SpecularStrength` and `DiffuseStrength` set to 1).
- Setting the `SpecularStrength` of the cones to 1 makes them highly reflective.

```
hlight = camlight('headlight');
set(hpatch,'AmbientStrength',.1,...
      'SpecularStrength',1,...
      'DiffuseStrength',1);
set(hcone,'SpecularStrength',1);
set(gcf,'Color','k')
```

Selecting a Renderer

Because this example uses lighting, MATLAB must use either `zbuffer` or, if available, `OpenGL` renderer settings. The `OpenGL` renderer is likely to be much faster displaying the animation; however, you need to use `gouraud` lighting with OpenGL, which is not as smooth as Phong lighting, which you can use with the `zbuffer` renderer. The two choices are

```
lighting gouraud
set(gcf,'Renderer','OpenGL')
```

or for `zbuffer`

```
lighting phong
set(gcf,'Renderer','zbuffer')
```

Defining the Camera Path as a Stream Line

Stream lines indicate the direction of flow in the vector field. This example uses the *x*-, *y*-, and *z*-coordinate data of a single stream line to map a path

through the volume. The camera is then moved along this path. The tasks include

- Create a stream line starting at the point x = 80, y = 30, z = 11.
- Get the *x*-, *y*-, and *z*-coordinate data of the stream line.
- Delete the stream line (you could also use `stream3` to calculate the stream line data without actually drawing the stream line).

```
hsline = streamline(x,y,z,u,v,w,80,30,11);
xd = get(hsline,'XData');
yd = get(hsline,'YData');
zd = get(hsline,'ZData');
delete(hsline)
```

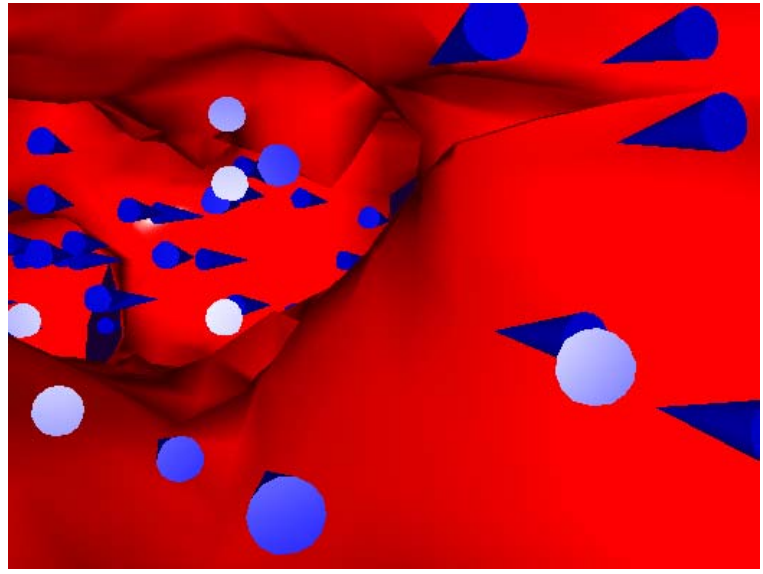
Implementing the Fly-Through

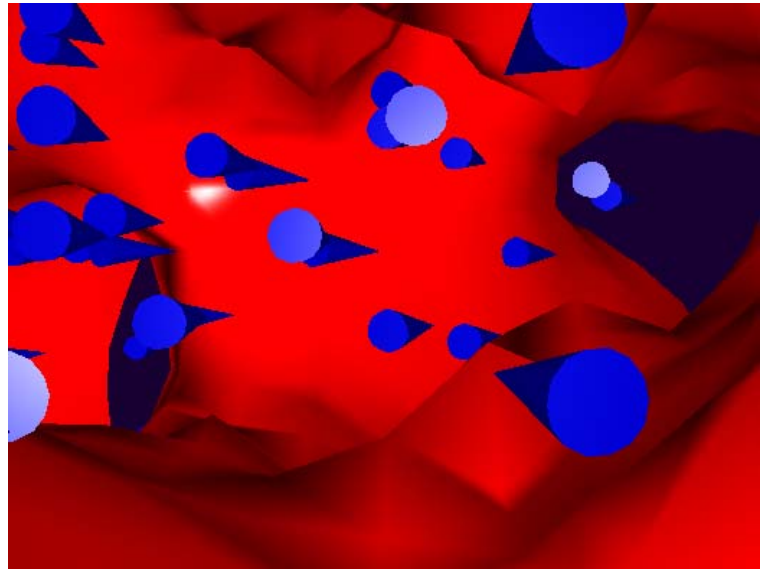
To create a fly-through, move the camera position and camera target along the same path. In this example, the camera target is placed five elements further along the *x*-axis than the camera. A small value is added to the camera target x position to prevent the position of the camera and target from becoming the same point if the condition `xd(n) = xd(n+5)` should occur:

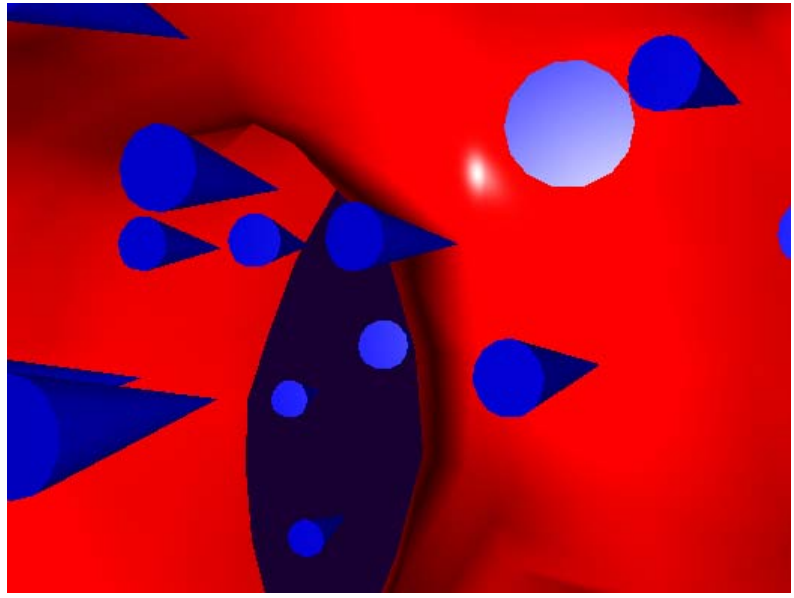
- Update the camera position and camera target so that they both move along the coordinates of the stream line.
- Move the light along with the camera.
- Call `drawnow` to display the results of each move.

```
for i=1:length(xd)-50
  campos([xd(i),yd(i),zd(i)])
  camtarget([xd(i+5)+min(xd)/100,yd(i),zd(i)])
  camlight(hlight,'headlight')
  drawnow
end
```

These snapshots illustrate the view at values of `i` equal to `10`, `110`, and `185`.







Low-Level Camera Properties

In this section...

Camera Properties You Can Set

Camera graphics is based on a group of axes properties that control the position and orientation of the camera. In general, the camera commands make it unnecessary to access these properties directly.

Property	Description
`CameraPosition`	Specifies the location of the viewpoint in axes units.
`CameraPositionMode`	In `automatic` mode, the scene determines the position. In `manual` mode, you specify the viewpoint location.
`CameraTarget`	Specifies the location in the axes pointed to by the camera. Together with the `CameraPosition`, it defines the viewing axis.
`CameraTargetMode`	In `automatic` mode, MATLAB specifies the `CameraTarget` as the center of the axes plot box. In `manual` mode, you specify the location.
`CameraUpVector`	The rotation of the camera around the viewing axis is defined by a vector indicating the direction taken as up.
`CameraUpVectorMode`	In `automatic` mode, MATLAB orients the up vector along the positive *y*-axis for 2-D views and along the positive *z*-axis for 3-D views. In `manual` mode, you specify the direction.

Property	Description
CameraViewAngle	Specifies the field of view of the "lens." If you specify a value for CameraViewAngle, MATLAB overrides stretch-to-fill behavior (see “Understanding Axes Aspect Ratio” on page 2-41).
CameraViewAngleMode	In automatic mode, MATLAB adjusts the view angle to the smallest angle that captures the entire scene. In manual mode, you specify the angle. Setting CameraViewAngleMode to manual overrides stretch-to-fill behavior.
Projection	Selects either an orthographic or perspective projection.

Default Viewpoint Selection

When all the camera mode properties are set to auto (the default), MATLAB automatically controls the view, selecting appropriate values based on the assumption that you want the scene to fill the position rectangle (which is defined by the width and height components of the axes Position property).

By default, MATLAB

- Sets the CameraPosition so the orientation of the scene is the standard MATLAB 2-D or 3-D view (see the view command)
- Sets the CameraTarget to the center of the plot box
- Sets the CameraUpVector so the *y*-direction is up for 2-D views and the *z*-direction is up for 3-D views
- Sets the CameraViewAngle to the minimum angle that makes the scene fill the position rectangle (the rectangle defined by the axes Position property)
- Uses orthographic projection

This default behavior generally produces desirable results. However, you can change these properties to produce useful effects.

Moving In and Out on the Scene

You can move the camera anywhere in the 3-D space defined by the axes. The camera continues to point towards the target regardless of its position. When the camera moves, MATLAB varies the camera view angle to ensure the scene fills the position rectangle.

Moving Through a Scene

You can create a fly-by effect by moving the camera through the scene. To do this, continually change `CameraPosition` property, moving it toward the target. Because the camera is moving through space, it turns as it moves past the camera target. Override the MATLAB automatic resizing of the scene each time you move the camera by setting the `CameraViewAngleMode` to `manual`.

If you update the `CameraPosition` and the `CameraTarget`, the effect is to pass through the scene while continually facing the direction of movement.

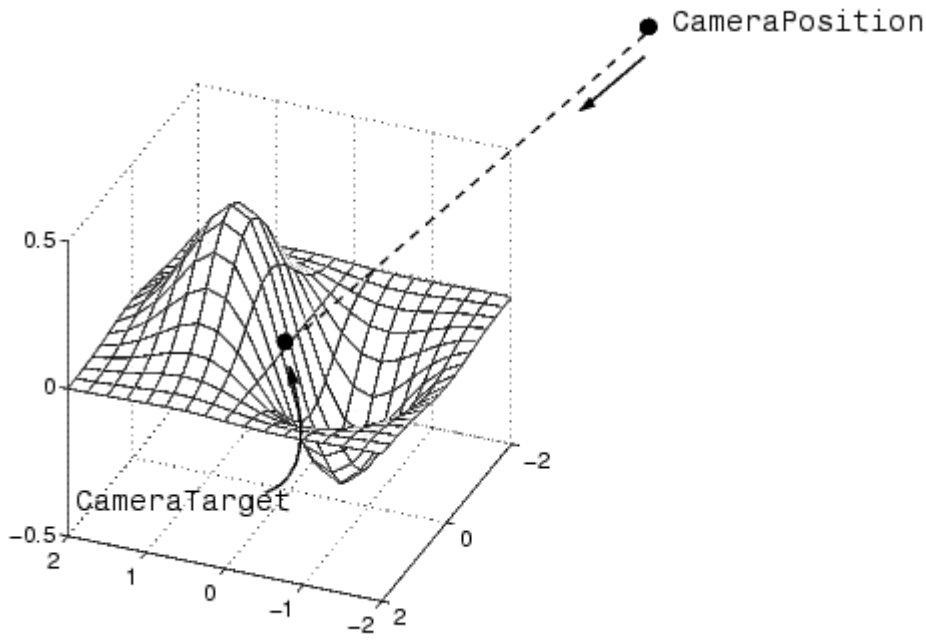
If the `Projection` is set to `perspective`, the amount of perspective distortion increases as the camera gets closer to the target and decreases as it gets farther away.

Example — Moving Toward or Away from the Target

To move the camera along the viewing axis, you need to calculate new coordinates for the `CameraPosition` property. This is accomplished by subtracting (to move closer to the target) or adding (to move away from the target) some fraction of the total distance between the camera position and the camera target.

The function `movecamera` calculates a new `CameraPosition` that moves in on the scene if the argument `dist` is positive and moves out if `dist` is negative.

```
function movecamera(dist) %dist in the range [-1 1]
set(gca,'CameraViewAngleMode','manual')
newcp = cpos - dist * (cpos - ctarg);
set(gca,'CameraPosition',newcp)
function out = cpos
out = get(gca,'CameraPosition');
function out = ctarg
out = get(gca,'CameraTarget');
```



Setting the `CameraViewAngleMode` to `manual` overrides MATLAB stretch-to-fill behavior and can cause an abrupt change in the aspect ratio. See "Understanding Axes Aspect Ratio" on page 2-41 for more information on stretch-to-fill.

Making the Scene Larger or Smaller

Adjusting the `CameraViewAngle` property makes the view of the scene larger or smaller. Larger angles cause the view to encompass a larger area, thereby making the objects in the scene appear smaller. Similarly, smaller angles make the objects appear larger.

Changing `CameraViewAngle` makes the scene larger or smaller without affecting the position of the camera. This is desirable if you want to zoom in without moving the viewpoint past objects that will then no longer be in the scene (as could happen if you changed the camera position). Also, changing

the CameraViewAngle does not affect the amount of perspective applied to the scene, as changing CameraPosition does when the figure Projection property is set to perspective.

Revolving Around the Scene

You can use the view command to revolve the viewpoint about the *z*-axis by varying the azimuth, and about the azimuth by varying the elevation. This has the effect of moving the camera around the scene along the surface of a sphere whose radius is the length of the viewing axis. You could create the same effect by changing the CameraPosition, but doing so requires you to perform calculations that MATLAB performs for you when you call view.

For example, the function orbit moves the camera around the scene.

```
function orbit(deg)
[az el] = view;
rotvec = 0:deg/10:deg;
for i = 1:length(rotvec)
    view([az+rotvec(i) el])
    drawnow
end
```

Rotation Without Resizing

When CameraViewAngleMode is auto, MATLAB calculates the CameraViewAngle so that the scene is as large as can fit in the axes position rectangle. This causes an apparent size change during rotation of the scene. To prevent resizing during rotation, you need to set the CameraViewAngleMode to manual (which happens automatically when you specify a value for the CameraViewAngle property). To do this in the orbit function, add the statement

```
set(gca,'CameraViewAngleMode','manual')
```

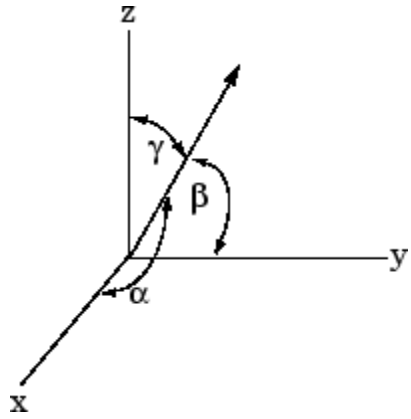
Rotation About the Viewing Axis

You can change the orientation of the scene by specifying the direction defined as up. By default, MATLAB defines *up* as the *y*-axis in 2-D views (the CameraUpVector is [0 1 0]) and the *z*-axis for 3-D views (the

`CameraUpVector` is `[0 0 1]`). However, you can specify up as any arbitrary direction.

The vector defined by the `CameraUpVector` property forms one axis of the camera's coordinate system. Internally, MATLAB determines the actual orientation of the camera up vector by projecting the specified vector onto the plane that is normal to the camera direction (i.e., the viewing axis). This simplifies the specification of the `CameraUpVector` property, because it need not lie in this plane.

In many cases, you might find it convenient to visualize the desired up vector in terms of angles with respect to the axes *x*-, *y*-, and *z*-axis. You can then use *direction cosines* to convert from angles to vector components. For a unit vector, the expression simplifies to



where the angles α, ß, and γ are specified in degrees.

XComponent = cos(α(pi/180));*

YComponent = cos(β(pi/180));*

ZComponent = cos(γ(pi/180));*

Consult a mathematics book on vector analysis for a more detailed explanation of direction cosines.

Calculating a Camera Up Vector

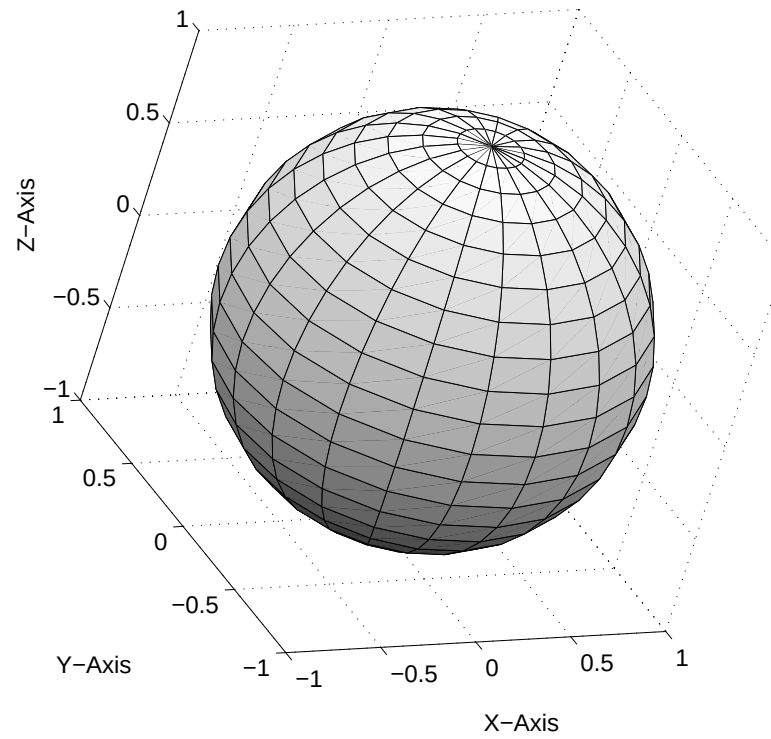
To specify an up vector that makes an angle of 30° with the *z*-axis and lies in the *y*-*z* plane, use the expression

```
upvec = [cos(90*(pi/180)),cos(60*(pi/180)),cos(30*(pi/180))];
```

and then set the `CameraUpVector` property.

```
set(gca,'CameraUpVector',upvec)
```

Drawing a sphere with this orientation produces



Understanding View Projections

In this section...

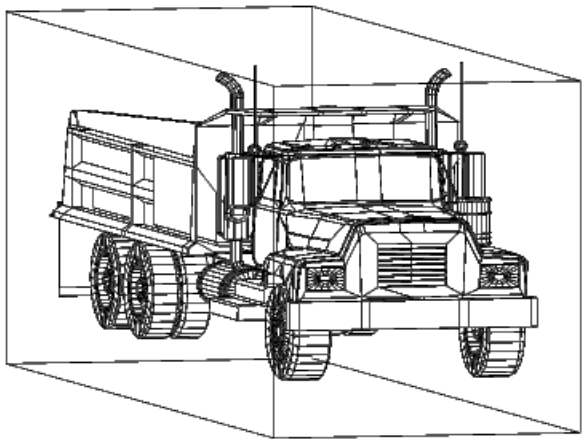
Two Types of Projections

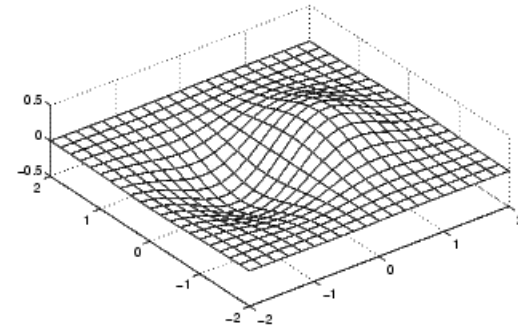
MATLAB Graphics supports both orthographic and perspective projection types for displaying 3-D graphics. The one you select depends on the type of graphics you are displaying:

- `orthographic` projects the viewing volume as a rectangular parallelepiped (i.e., a box whose opposite sides are parallel). Relative distance from the camera does not affect the size of objects. This projection type is useful when it is important to maintain the actual size of objects and the angles between objects.
- `perspective` projects the viewing volume as the frustum of a pyramid (a pyramid whose apex has been cut off parallel to the base). Distance causes foreshortening; objects further from the camera appear smaller. This projection type is useful when you want to display realistic views of real objects.

By default, MATLAB displays objects using orthographic projection. You can set the projection type using the `camproj` command.

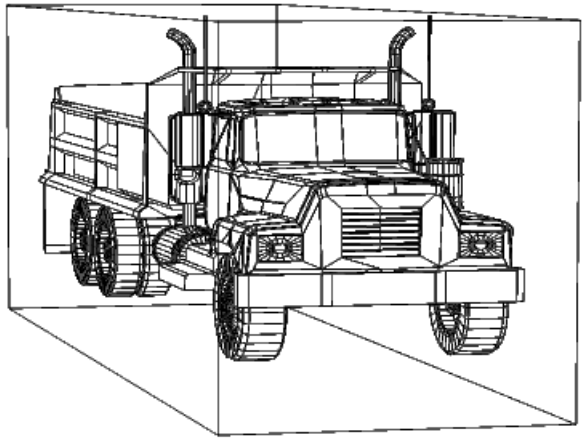
These pictures show a drawing of a dump truck (created with `patch`) and a surface plot of a mathematical function, both using orthographic projection.

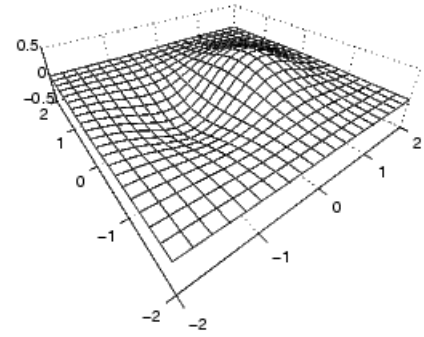




If you measure the width of the front and rear faces of the box enclosing the dump truck, you'll see they are the same size. This picture looks unnatural because it lacks the apparent perspective you see when looking at real objects with depth. On the other hand, the surface plot accurately indicates the values of the function within rectangular space.

Now look at the same graphics objects with perspective added. The dump truck looks more natural because portions of the truck that are farther from the viewer appear smaller. This projection mimics the way human vision works. The surface plot, on the other hand, looks distorted.



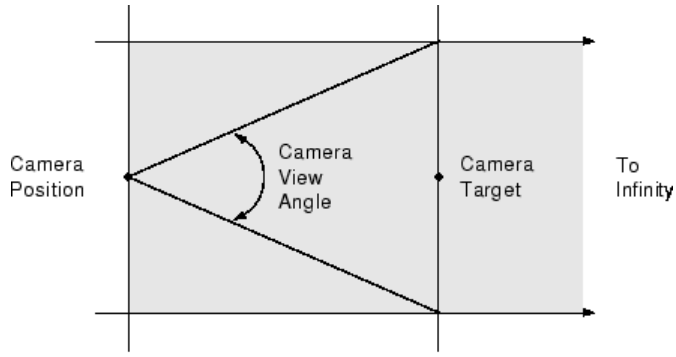


Projection Types and Camera Location

By default, MATLAB adjusts the `CameraPosition`, `CameraTarget`, and `CameraViewAngle` properties to point the camera at the center of the scene and to include all graphics objects in the axes. If you position the camera so that there are graphics objects behind the camera, the scene displayed can be affected by both the axes `Projection` property and the figure `Renderer` property. The following summarizes the interactions between projection type and rendering method.

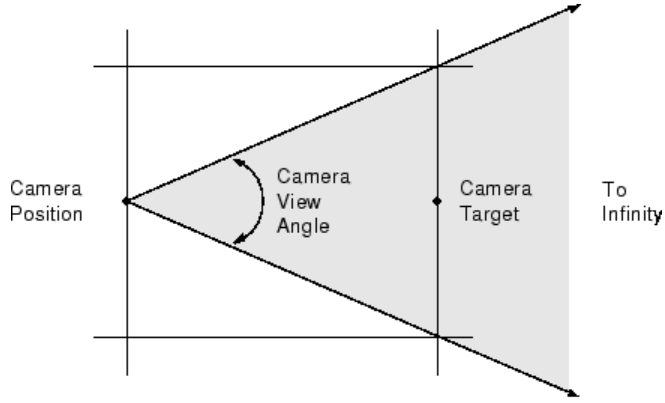
	Orthographic	**Perspective**
Z-buffer	`CameraViewAngle` determines extent of scene at `CameraTarget`.	`CameraViewAngle` determines extent of scene from `CameraPosition` to infinity.
Painters	All objects are displayed regardless of `CameraPosition`.	Not recommended if graphics objects are behind the `CameraPosition`.

This diagram illustrates what you see (gray area) when using orthographic projection and Z-buffer. Anything in front of the camera is visible.



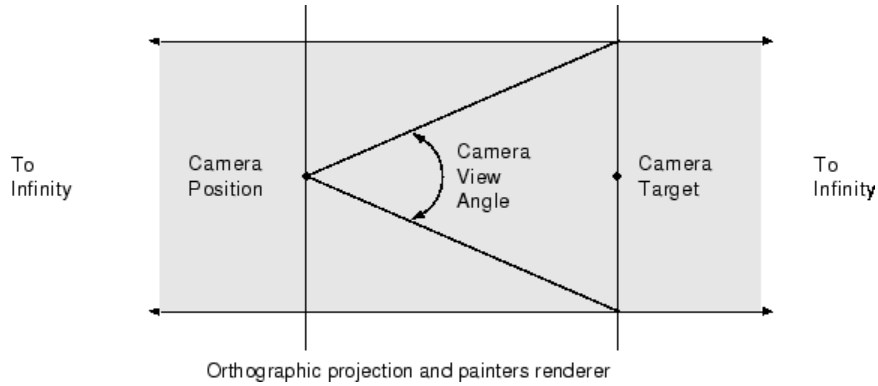
Orthographic projection and Z-buffer renderer

In perspective projection, you see only what is visible in the cone of the camera view angle.



Perspective projection and Z-buffer renderer

Painters rendering method is less suited to moving the camera in 3-D space because MATLAB does not clip along the viewing axis. Orthographic projection in painters method results in all objects contained in the scene being visible regardless of the camera position.



Orthographic projection and painters renderer

Printing 3-D Scenes

The same effects described in the previous section occur in hardcopy output. However, because of the differences in the process of rendering to the screen and to a printing format, MATLAB might render using Z-buffer and generate printed output using painters. You might need to specify Z-buffer printing explicitly to obtain the results displayed on the screen (use the `-zbuffer` option with the `print` command).

Additional Information

See Basic Printing and Exporting and Selecting a Renderer in Figure Properties in the Using MATLAB Graphics documentation for information on printing and rendering methods.

Understanding Axes Aspect Ratio

In this section...

Stretch-to-Fill

Axes shape graphics objects by setting the scaling and limits of each axis. When you create a graph, the values or size of the plotted data automatically determines axis scaling , and then draws the axes to fit the space available for display. Axes aspect ratio properties control how MATLAB performs the scaling required to create a graph.

By default, the size of the axes MATLAB creates for plotting is normalized to the size of the figure window (but is slightly smaller to allow for borders). If you resize the figure, the size and possibly the aspect ratio (the ratio of width to height) of the axes changes proportionally. This enables the axes to always fill the available space in the window. MATLAB also sets the *x*-, *y*-, and *z*-axis limits to provide the greatest resolution in each direction, again optimizing the use of available space.

This stretch-to-fill behavior is generally desirable; however, you might want to control this process to produce specific results. For example, images need to be displayed in correct proportions regardless of the aspect ratio of the figure window, or you might want graphs always to be a particular size on a printed page.

Axis Scaling

The `axis` command enables you to adjust the scaling of graphs. By default, MATLAB finds the maxima and minima of the plotted data and chooses appropriate axes ranges. You can override the defaults by setting axis limits.

```
axis([xmin xmax ymin ymax zmin zmax])
```

You can control how MATLAB scales the axes with predefined `axis` options:

- `axis auto` returns the axis scaling to its default, automatic mode. `v = axis` saves the scaling of the axes of the current plot in vector `v`. For subsequent graphics commands to have these same axis limits, follow them with `axis(v)`.
- `axis manual` freezes the scaling at the current limits. If you then set `hold on`, subsequent plots use the current limits. Specifying values for axis limits also sets axis scaling to manual.
- `axis tight` sets the axis limits to the range of the data.
- `axis ij` places MATLAB into its "matrix" axes mode. The coordinate system origin is at the upper left corner. The *i*-axis is vertical and is numbered from top to bottom. The *j*-axis is horizontal and is numbered from left to right.
- `axis xy` places MATLAB into its default Cartesian axes mode. The coordinate system origin is at the lower left corner. The *x*-axis is horizontal and is numbered from left to right. The *y*-axis is vertical and is numbered from bottom to top.

Aspect Ratio

The `axis` command enables you to adjust the aspect ratio of graphs. Normally MATLAB stretches the axes to fill the window. In many cases, it is more useful to specify the aspect ratio of the axes based on a particular characteristic such as the relative length or scaling of each axis. The `axis` command provides a number of useful options for adjusting the aspect ratio:

- `axis equal` changes the current axes scaling so that equal tick mark increments on the *x*-, *y*-, and *z*-axis are equal in length. This makes the surface displayed by `sphere` look like a sphere instead of an ellipsoid. `axis equal` overrides stretch-to-fill behavior.
- `axis square` makes each axis the same length and overrides stretch-to-fill behavior.

- `axis vis3d` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill. Use this option after other `axis` options to keep settings from changing while you rotate the scene.
- `axis image` makes the aspect ratio of the axes the same as the image.
- axis auto returns the *x*-, *y*-, and *z*-axis limits to automatic selection mode.
- `axis normal` restores the current axis box to full size and removes any restrictions on the scaling of the units. It undoes the effects of `axis square`. Used in conjunction with `axis auto`, it undoes the effects of `axis equal`.

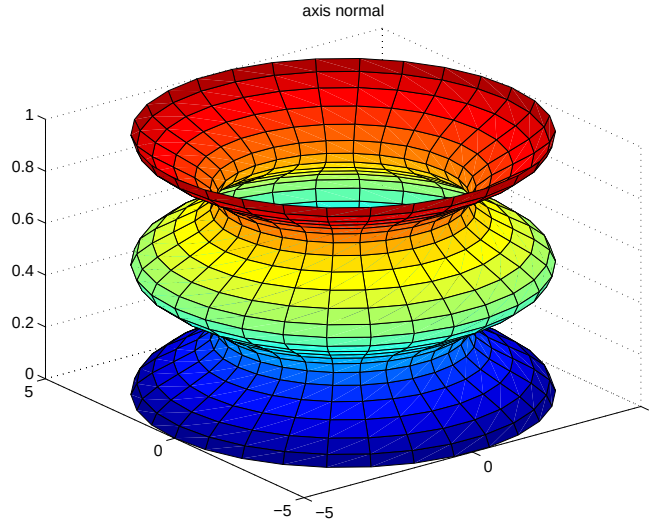
The `axis` command works by manipulating axes graphics object properties.

axis Command Options

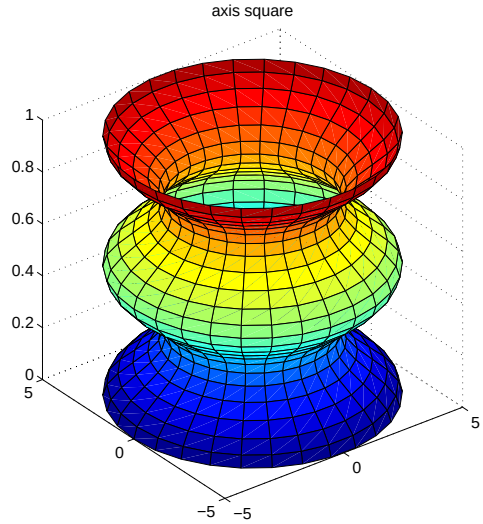
The following three pictures illustrate the effects of three `axis` options on a cylindrical surface created with the statements

```
t = 0:pi/6:4*pi;
[x,y,z] = cylinder(4+cos(t),30);
surf(x,y,z)
```

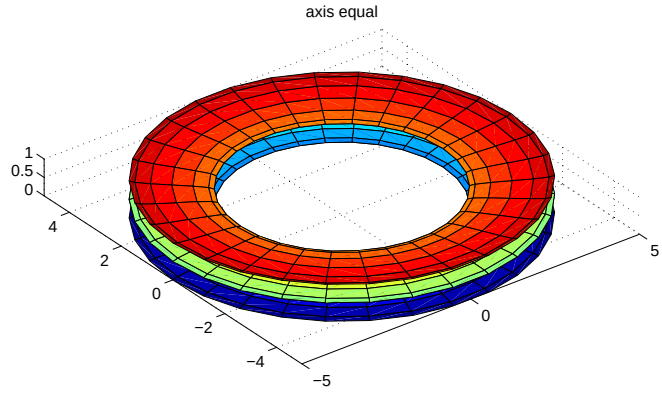
`axis normal` is the default behavior. MATLAB automatically sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window.



`axis square` creates an axes that is square regardless of the shape of the figure window. The cylindrical surface is no longer distorted because it is not warped to fit the window. However, the size of one data unit is not equal along all axes (the *z*-axis spans only one unit while the *x*-axes and *y*-axes span 10 units each).



`axis equal` makes the length of one data unit equal along each axis while maintaining a nearly square plot box. It also prevents warping of the axis to fill the window's shape.



Additional Commands for Setting Aspect Ratio

You can control the aspect ratio of your graph in three ways:

- Specifying the relative scales of the *x*-, *y*-, and *z*-axes (data aspect ratio)
- Specifying the shape of the space defined by the axes (plot box aspect ratio)
- Specifying the axis limits

The following commands enable you to set these values.

Command	Purpose
`daspect`	Set or query the data aspect ratio
`pbaspect`	Set or query the plot box aspect ratio
`xlim`	Set or query *x*-axis limits
`ylim`	Set or query *y*-axis limits
`zlim`	Set or query *z*-axis limits

See "Manipulating Axes Aspect Ratio" on page 2-46 for a list of the axes properties that control aspect ratio.

Manipulating Axes Aspect Ratio

In this section...

Axes Aspect Ratio Properties

The `axis` command works by setting various axes object properties. You can set these properties directly to achieve precisely the effect you want.

Property	Description
`DataAspectRatio`	Sets the relative scaling of the individual axis data values. Set `DataAspectRatio` to `[1 1 1]` to display real-world objects in correct proportions. Specifying a value for DataAspectRatio overrides stretch-to-fill behavior.
`DataAspectRatioMode`	In `auto`, MATLAB software selects axis scales that provide the highest resolution in the space available.
`PlotBoxAspectRatio`	Sets the proportions of the axes plot box (set `box` to `on` to see the box). Specifying a value for PlotBoxAspectRatio overrides stretch-to-fill behavior.
`PlotBoxAspectRatioMode`	In `auto`, MATLAB sets the `PlotBoxAspectRatio` to `[1 1 1]` unless you explicitly set the `DataAspectRatio` and/or the axis limits.
`Position`	Defines the location and size of the axes with a four-element vector: [*left offset*, *bottom offset*, *width*, *height*].
`XLim, YLim, ZLim`	Sets the minimum and maximum limits of the respective axes.
`XLimMode, YLimMode, ZLimMode`	In `auto`, MATLAB selects the axis limits.

By default, MATLAB automatically determines values for all of these properties (i.e., all the modes are auto) and then applies stretch-to-fill. You can override any property's automatic operation by specifying a value for the property or setting its mode to manual. The value you select for a particular property depends primarily on what type of data you want to display.

Much of the data visualized with MATLAB is either

- Numerical data displayed as line or mesh plots
- Representations of real-world objects (e.g., a dump truck or a section of the earth's topography)

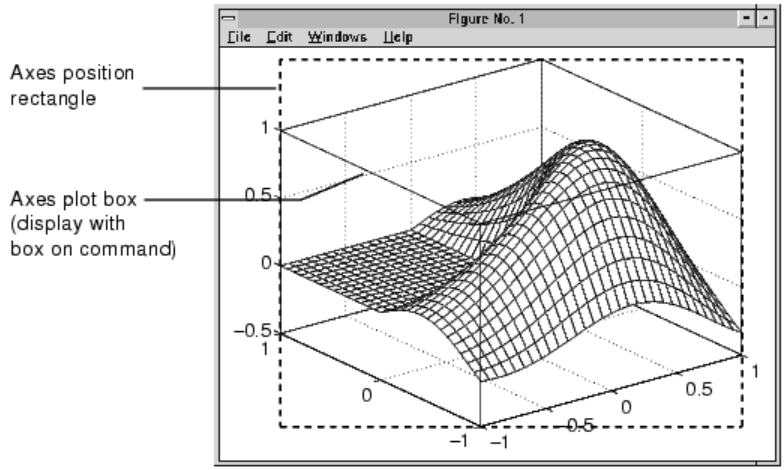
In the first case, it is generally desirable to select axis limits that provide good resolution in each axis direction and to fill the available space. Real-world objects, on the other hand, need to be represented accurately in proportion, regardless of the angle of view.

Default Aspect Ratio Selection

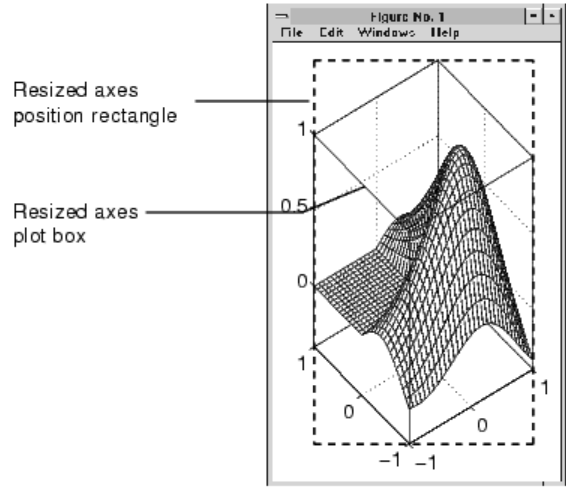
There are two key elements to the default behavior — normalizing the axes size to the window size and stretch-to-fill.

The axes `Position` property specifies the location and dimensions of the axes. The third and fourth elements of the `Position` vector (`width` and `height`) define a rectangle in which MATLAB draws the axes (indicated by the dotted line in the following pictures). MATLAB stretches the axes to fill this rectangle.

The default value for the axes `Units` property is `normalized` to the parent figure dimensions. This means the shape of the figure window determines the shape of the position rectangle. As you change the size of the window, MATLAB reshapes the position rectangle to fit it.



The view is the 2-D projection of the plot box onto the screen.



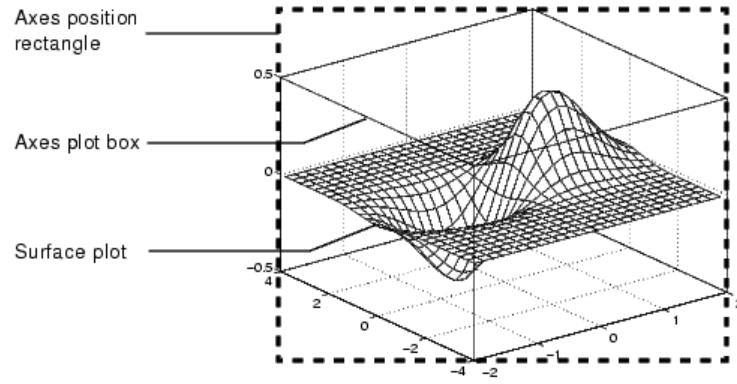
As you can see, reshaping the axes to fit into the figure window can change the aspect ratio of the graph. MATLAB applies stretch-to-fill so the axes fill the position rectangle and in the process can distort the shape. This is

generally desirable for graphs of numeric data, but not for displaying objects realistically.

MATLAB Defaults

MATLAB surface plots are well suited for visualizing mathematical functions of two variables. For example, to display a mesh plot of the function $z=xe^{(-x^2-y^2)}$ evaluated over the range $-2 \le x \le 2$, $-4 \le y \le 4$, use the statements

```
[X,Y] = meshgrid([-2:.15:2],[-4:.3:4]);
Z = X.*exp(-X.^2 - Y.^2);
mesh(X,Y,Z)
```



The MATLAB default property values are designed to

- Select axis limits to span the range of the data (`XLimMode`, `YLimMode`, and `ZLimMode` are set to `auto`).
- Provide the highest resolution in the available space by setting the scale of each axis independently (`DataAspectRatioMode` and the `PlotBoxAspectRatioMode` are set to `auto`).
- Draw axes that fit the position rectangle by adjusting the `CameraViewAngle` and then stretch-to-fill the axes if necessary.

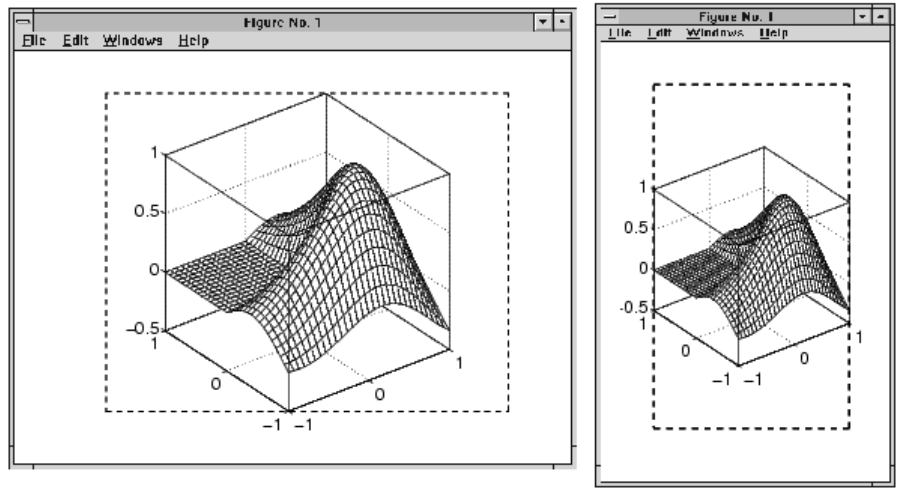
Overriding Stretch-to-Fill

To maintain a particular shape, you can specify the size of the axes in absolute units such as inches, which are independent of the figure window size. However, this is not a good approach if you are writing a MATLAB file that you want to work with a figure window of any size. A better approach is to specify the aspect ratio of the axes and override automatic stretch-to-fill.

In cases where you want a specific aspect ratio, you can override stretching by specifying a value for these axes properties:

- `DataAspectRatio` or `DataAspectRatioMode`
- `PlotBoxAspectRatio` or `PlotBoxAspectRatioMode`
- `CameraViewAngle` or `CameraViewAngleMode`

The first two sets of properties affect the aspect ratio directly. Setting either of the mode properties to `manual` simply disables stretch-to-fill while maintaining all current property values. In this case, MATLAB enlarges the axes until one dimension of the position rectangle constrains it.



Setting the `CameraViewAngle` property disables stretch-to-fill, and also prevents MATLAB from readjusting the size of the axes if you change the view.

Aspect Ratio Properties

It is important to understand how properties interact with each other, in order to obtain the results you want. The `DataAspectRatio`, `PlotBoxAspectRatio`, and the *x*-, *y*-, and *z*-axis limits (`XLim`, `YLim`, and `ZLim` properties) all place constraints on the shape of the axes.

Data Aspect Ratio

The `DataAspectRatio` property controls the ratio of the axis scales. For a mesh plot of the function $z=xe^{(-x_2-y_2)}$ evaluated over the range $-2 \le x \le 2$, $-4 \le y \le 4$

```
[X,Y] = meshgrid([-2:.15:2],[-4:.3:4]);
Z = X.*exp(-X.^2 - Y.^2);
mesh(X,Y,Z)
```

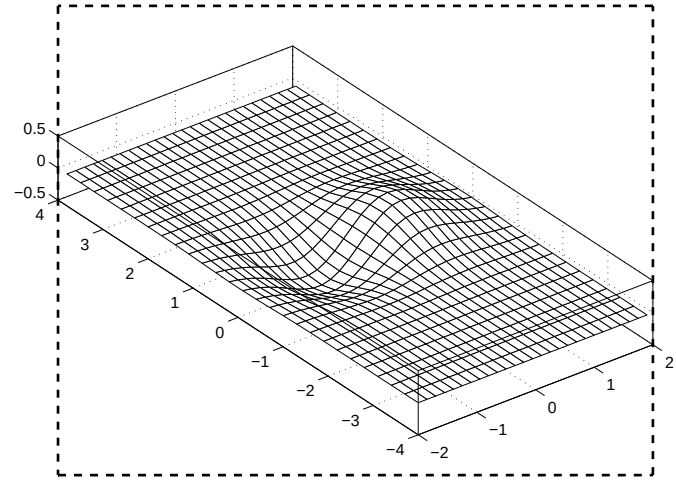
the values are

```
get(gca,'DataAspectRatio')
ans =
     4 8 1
```

This means that four units in length along the *x*-axis cover the same data values as eight units in length along the *y*-axis and one unit in length along the *z*-axis. The axes fill the plot box, which has an aspect ratio of [1 1 1] by default.

If you want to view the mesh plot so that the relative magnitudes along each axis are equal with respect to each other, you can set the `DataAspectRatio` to `[1 1 1]`.

```
set(gca,'DataAspectRatio',[1 1 1])
```



Setting the value of the `DataAspectRatio` property also sets the `DataAspectRatioMode` to `manual` and overrides stretch-to-fill so the specified aspect ratio is achieved.

Plot Box Aspect Ratio

Looking at the value of the `PlotBoxAspectRatio` for the graph in the previous section shows that it has now taken on the former value of the `DataAspectRatio`.

```
get(gca,'PlotBoxAspectRatio')
ans =
     4 8 1
```

MATLAB has rescaled the plot box to accommodate the graph using the specified `DataAspectRatio`.

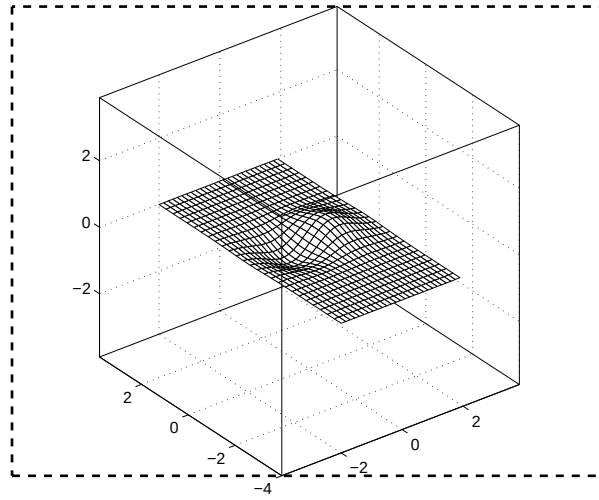
The `PlotBoxAspectRatio` property controls the shape of the axes plot box. MATLAB sets this property to `[1 1 1]` by default and adjusts the `DataAspectRatio` property so that graphs fill the plot box if stretching is on, or until reaching a constraint if stretch-to-fill has been overridden.

When you set the value of the `DataAspectRatio` and thereby prevent it from changing, MATLAB varies the `PlotBoxAspectRatio` instead. If you specify both the `DataAspectRatio` and the `PlotBoxAspectRatio`, MATLAB is forced to change the axis limits to obey the two constraints you have already defined.

Continuing with the mesh example, if you set both properties,

```
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1])
```

MATLAB changes the axis limits to satisfy the two constraints placed on the axes.



Adjusting Axis Limits

MATLAB enables you to set the axis limits to the values you want. However, specifying a value for `DataAspectRatio`, `PlotBoxAspectRatio`, and the axis limits overconstrains the axes definition. For example, it is not possible for MATLAB to draw the axes if you set these values:

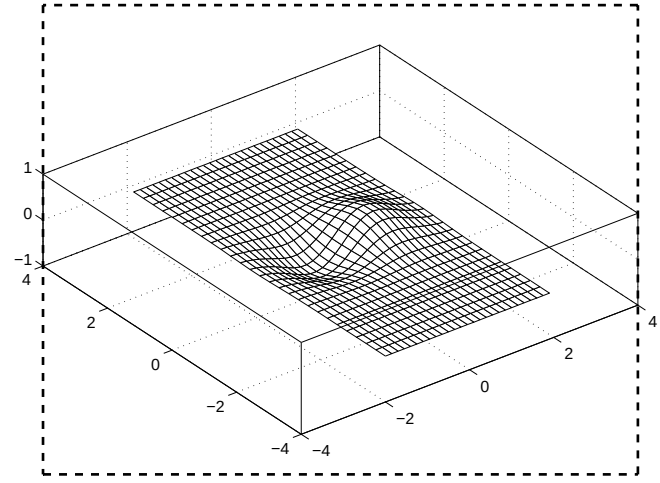
```
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],...
```

```
        'XLim',[-4 4],...
        'YLim',[-4 4],...
        'ZLim',[-1 1])
```

In this case, MATLAB ignores the setting of the `PlotBoxAspectRatio` and determines its value automatically. These particular values cause the `PlotBoxAspectRatio` to return to its calculated value.

```
get(gca,'PlotBoxAspectRatio')
ans =
      4 8 1
```

MATLAB can now draw the axes using the specified `DataAspectRatio` and axis limits.



Displaying Cross-Sections of Surfaces

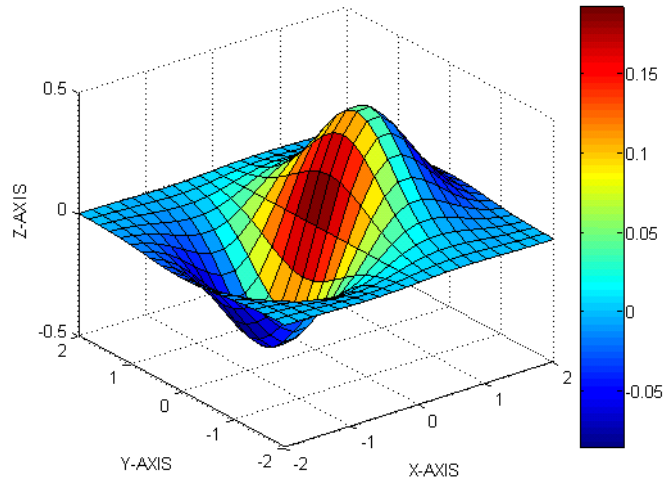
Sometimes projecting a 3-D surface onto an *x*-, *y*-, or *z*-axis can aid visualization. To do this, you might change the aspect ratio, in order to make space for the projection. The following example illustrates how to do this:

1 Create an *x*-*y* grid and *z* value for it:

```
[x,y] = meshgrid([-2:.2:2]);
Z = x.*exp(-x.^2-y.^2);
```

2 Plot the surface in 3-D; annotate with a colorbar and axis labels:

```
surf(x,y,Z,gradient(Z))
colorbar
xlabel('X-AXIS')
ylabel('Y-AXIS')
zlabel('Z-AXIS')
```



3 Use `axis` to change the `Ymax` value in to 3, stretching the plot in one direction:

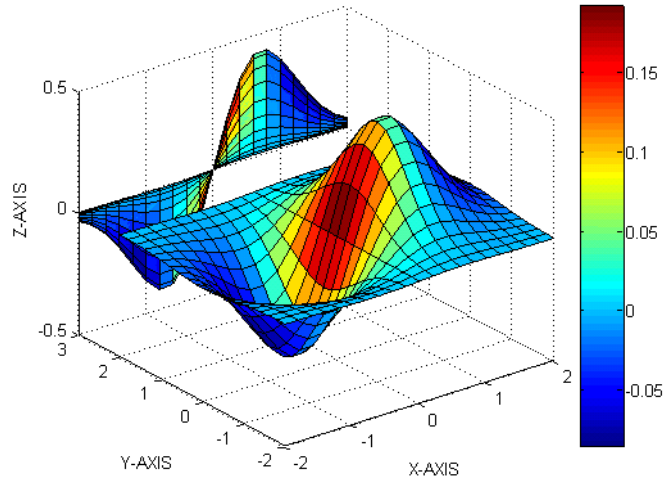
```
axis([-2 2 -2 3 -0.5 0.5]) %
```

4 Regrid the surface, setting all Y value equal to 3:

```
y = 3*ones(21);
```

5 Plaster a plot of the surface onto the Y-axis:

```
hold on
surf(x,y,Z,gradient(Z))
```



Displaying Real Objects

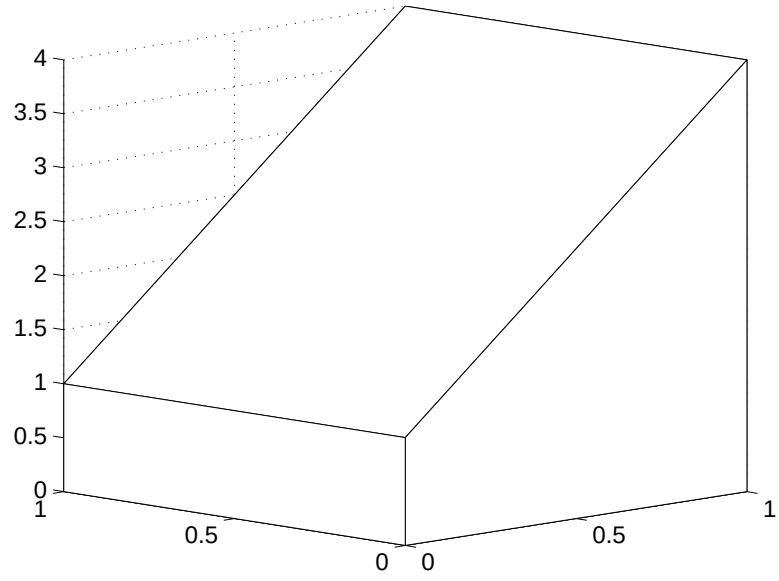
If you want to display an object so that it looks realistic, you need to change MATLAB defaults. For example, this data defines a wedge-shaped patch object.

```
vertex_list =

     0     0     0
     0     1     0
     1     1     0
     1     0     0
     0     0     1
     0     1     1
     1     1     4
     1     0     4
```

```
vertex_connection =

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

```
patch('Vertices',vertex_list,'Faces',vertex_connection,...
```

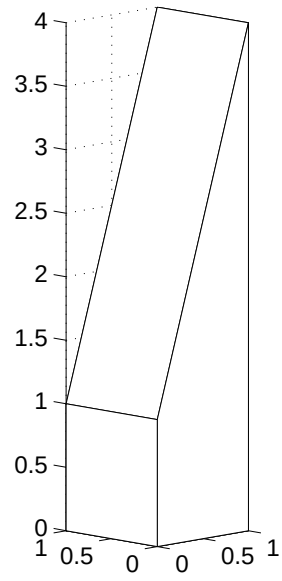
```
 'FaceColor','w','EdgeColor','k')
view(3)
```



However, this axes distorts the actual shape of the solid object defined by the data. To display it in correct proportions, set the `DataAspectRatio`.

```
set(gca,'DataAspectRatio',[1 1 1])
```

The units are now equal in the *x*-, *y*-, and *z*-directions and the axes is not being stretched to fill the position rectangle, revealing the true shape of the object.


4
3.5
3
2.5
2
1.5
1
0.5
0
1
0.5
0
0
0.5
1

3

Lighting as a Visualization Tool

Lighting Overview

In this section...

Lighting Commands

The MATLAB graphics environment provides commands that enable you to position light sources and adjust the characteristics of the objects that are reflecting the lights. These commands include the following.

Command	Purpose
`camlight`	Create or move a light with respect to the camera position
`lightangle`	Create or position a light in spherical coordinates
`light`	Create a light object
`lighting`	Select a lighting method
`material`	Set the reflectance properties of lit objects

You might find it useful to set light or lit-object properties directly to achieve specific results. In addition to the material in this topic area, you can explore the following lighting examples as an introduction to lighting for visualization.

Light Objects

You create a light object using the `light` function. Three important light object properties are

- `Color` — Color of the light cast by the light object
- `Style` — Either infinitely far away (the default) or local

- `Position` — Direction (for infinite light sources) or the location (for local light sources)

The `Color` property determines the color of the directional light from the light source. The color of an object in a scene is determined by the color of the object and the light source.

The `Style` property determines whether the light source is a point source (`Style` set to `local`), which radiates from the specified position in all directions, or a light source placed at infinity (`Style` set to `infinite`), which shines from the direction of the specified position with parallel rays.

The `Position` property specifies the location of the light source in axes data units. In the case of a light source at infinity, `Position` specifies the direction to the light source.

Lights affect surface and patch objects that are in the same axes as the light. These objects have a number of properties that alter the way they look when illuminated by lights.

Properties That Affect Lighting

You cannot see light objects themselves, but you can see their effects on any patch and surface objects present in the axes containing the light. A number of functions create these objects, including `surf`, `mesh`, `pcolor`, `fill`, and `fill3` as well as the `surface` and `patch` functions.

You control lighting effects by setting various axes, light, patch, and surface object properties. All properties have default values that generally produce desirable results. However, you can achieve the specific effect you want by adjusting the values of these properties.

Property	Effect
`AmbientLightColor`	An axes property that specifies the color of the background light in the scene, which has no direction and affects all objects uniformly. Ambient light effects occur only when there is a visible light object in the axes.
`AmbientStrength`	A patch and surface property that determines the intensity of the ambient component of the light reflected from the object.

Property	Effect
`DiffuseStrength`	A patch and surface property that determines the intensity of the diffuse component of the light reflected from the object.
`SpecularStrength`	A patch and surface property that determines the intensity of the specular component of the light reflected from the object.
`SpecularExponent`	A patch and surface property that determines the size of the specular highlight.
`SpecularColorReflectance`	A patch and surface property that determines the degree to which the specularly reflected light is colored by the object color or the light source color.
`FaceLighting`	A patch and surface property that determines the method used to calculate the effect of the light on the faces of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
`EdgeLighting`	A patch and surface property that determines the method used to calculate the effect of the light on the edges of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
`BackFaceLighting`	A patch and surface property that determines how faces are lit when their vertex normals point away from the camera. This property is useful for discriminating between the internal and external surfaces of an object.
`FaceColor`	A patch and surface property that specifies the color of the object faces.
`EdgeColor`	A patch and surface property that specifies the color of the object edges.

Property	Effect
`VertexNormals`	A patch and surface property that contains normal vectors for each vertex of the object. MATLAB uses vertex normal vectors to perform lighting calculations. While MATLAB automatically generates this data, you can also specify your own vertex normals.
`NormalMode`	A patch and surface property that determines whether MATLAB recalculates vertex normals if you change object data (`auto`) or uses the current values of the `VertexNormals` property (`manual`). If you specify values for `VertexNormals`, MATLAB sets this property to `manual`.

For more information, see descriptions of `axes`, `surface`, and `patch` object properties.

Examples of Lighting Control

Lighting is a technique for adding realism to a graphical scene. It does this by simulating the highlights and dark areas that occur on objects under natural lighting (e.g., the directional light that comes from the sun). To create lighting effects, MATLAB defines a graphics object called a light. MATLAB applies lighting to surface and patch objects.

These examples illustrate the use of lighting in a visualization context.

- Tracing a stream line through a volume — Sets properties of surfaces, patches, and lights (MATLAB Graphics documentation).
- Using slice planes and cone plots — Sets lighting characteristics of objects in a scene independently to achieve a desired result (MATLAB `coneplot` function).
- Lighting multiple slice planes independently to visualize fluid flow (MATLAB Graphics documentation).
- Combining single-color lit surfaces with interpolated coloring. See "Example — Visualizing MRI Data" (3–D Visualization documentation).

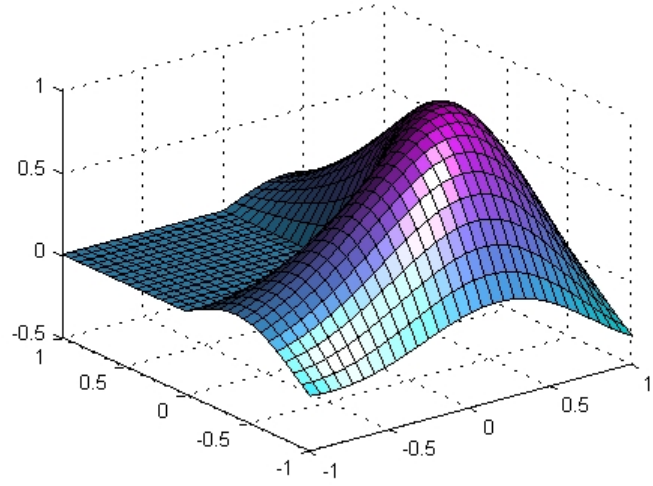
- Employing lighting to reveal surface shape. The fluid flow isosurface example and the surface plot of the `sinc` function examples illustrate this technique (3–D Visualization documentation).

Example — Adding Lights to a Scene

This example displays the membrane surface and illuminates it with a light source emanating from the direction defined by the position vector `[0 -2 1]`. This vector defines a direction from the axes origin passing through the point with the coordinates 0, -2, 1. The light shines from this direction toward the axes origin.

```
membrane
light('Position',[0 -2 1])
```

Creating a light activates a number of lighting-related properties controlling characteristics such as the ambient light and reflectance properties of objects. It also switches to Z-buffer renderer if not already in that mode.



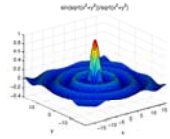
Example — Illuminating Mathematical Functions

Lighting can enhance surface graphs of mathematical functions. For example, use the `ezsurf` command to evaluate the expression

$$\sin\left(\sqrt{x^2+y^2}\right)\div\left(\sqrt{x^2+y^2}\right)$$

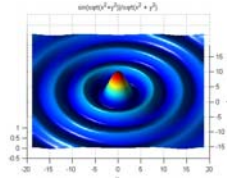
over the region -6π to 6π.

```
ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)',[-6*pi,6*pi])
```



Now add lighting using the `lightangle` command, which accepts the light position in terms of azimuth and elevation.

```
view(0,75)
shading interp
lightangle(-45,30)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'type','surface'),...
    'FaceLighting','phong',...
    'AmbientStrength',.3,'DiffuseStrength',.8,...
    'SpecularStrength',.9,'SpecularExponent',25,...
    'BackFaceLighting','unlit')
```



After obtaining the surface object's handle using `findobj`, you can set properties that affect how the light reflects from the surface. See "Properties That Affect Lighting" on page 3-3 for more detailed descriptions of these properties.

Selecting a Lighting Method

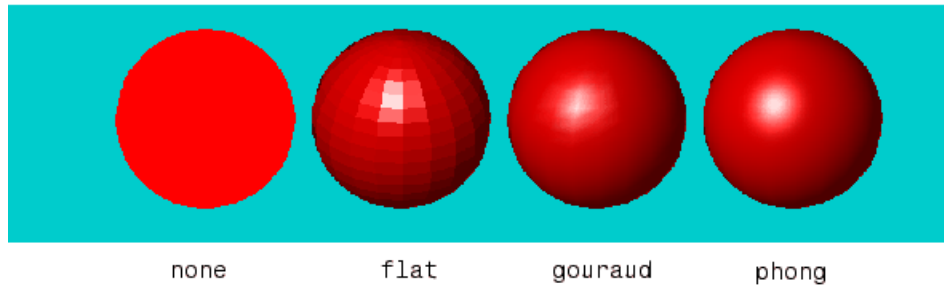
Face and Edge Lighting Methods

When you add lights to an axes, MATLAB rendering software determines the effects these lights have on the patch and surface objects that are displayed in that axes. There are different methods used to calculate the face and edge coloring of lit objects, and the one you select depends on the results you want to obtain.

MATLAB supports three different algorithms for lighting calculations, selected by setting the `FaceLighting` and `EdgeLighting` properties of each patch and surface object in the scene. Each algorithm produces somewhat different results:

- Flat lighting — Produces uniform color across each of the faces of the object. Select this method to view faceted objects.
- Gouraud lighting — Calculates the colors at the vertices and then interpolates colors across the faces. Select this method to view curved surfaces.
- Phong lighting — Interpolates the vertex normals across each face and calculates 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.

This illustration shows how a red sphere looks using each of the lighting methods with one white light source.



The `lighting` command (as opposed to the `light` function) provides a convenient way to set the lighting method.

Reflectance Characteristics of Graphics Objects

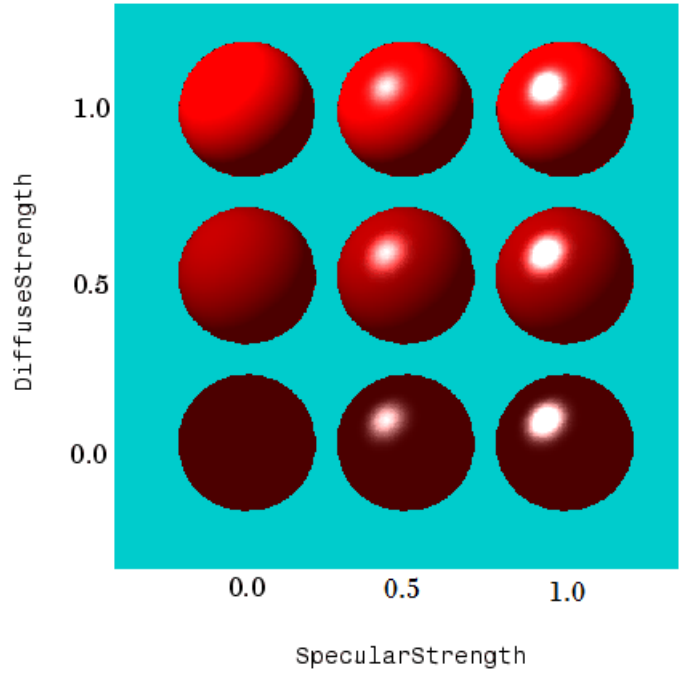
In this section...

Specular and Diffuse Reflection

You can specify the reflectance characteristics of patch and surface objects and thereby affect the way they look when lights are applied to the scene. It is likely you will adjust these characteristics in combination to produce particular results.

Also see the `material` command for a convenient way to produce certain lighting effects.

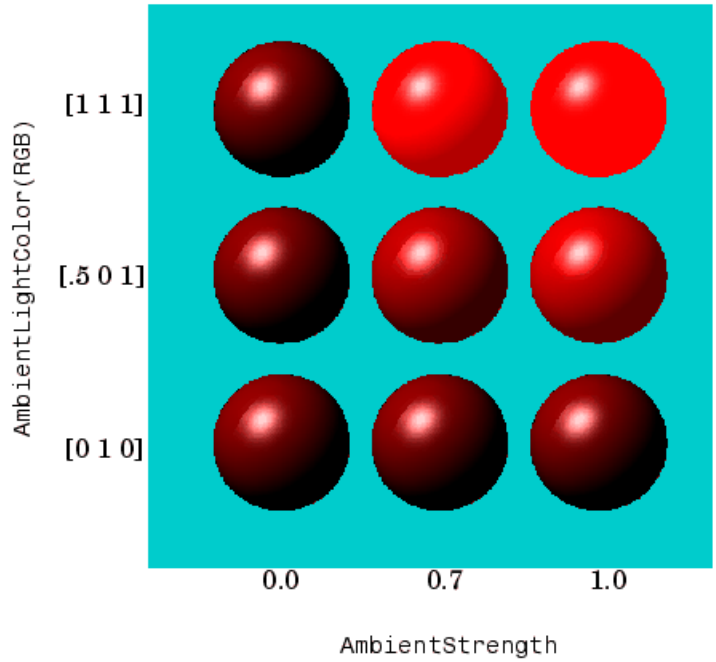
You can control the amount of specular and diffuse reflection from the surface of an object by setting the `SpecularStrength` and `DiffuseStrength` properties. This picture illustrates various settings.



Ambient Light

Ambient light is a directionless light that shines uniformly on all objects in the scene. Ambient light is visible only when there are light objects in the axes. There are two properties that control ambient light — `AmbientLightColor` is an axes property that sets the color, and `AmbientStrength` is a property of patch and surface objects that determines the intensity of the ambient light on the particular object.

This illustration shows three different ambient light colors at various intensities. The sphere is red and there is a white light object present.

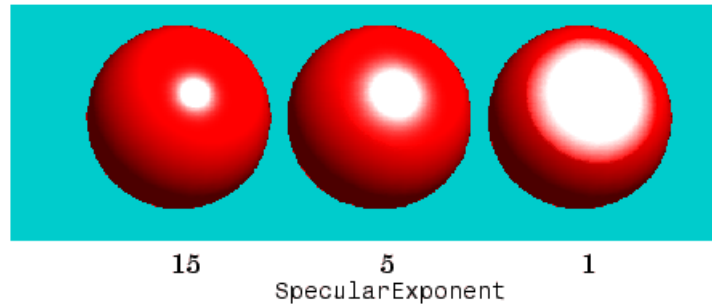


The green [0 1 0] ambient light does not affect the scene because there is no red component in green light. However, the color defined by the RGB values [.5 0 1] does have a red component, so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

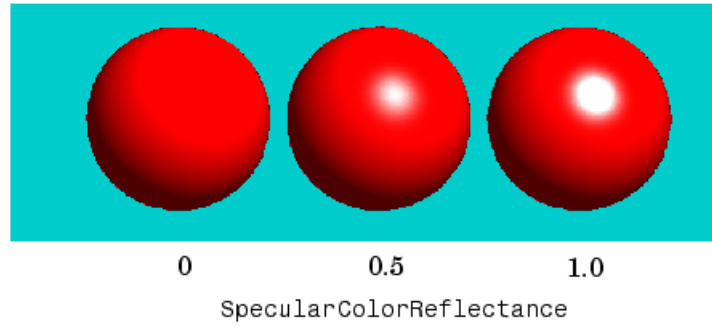
The size of the specular highlight spot depends on the value of the patch and surface object's `SpecularExponent` property. Typical values for this property range from 1 to 500, with normal objects having values in the range 5 to 20.

This illustration shows a red sphere illuminated by a white light with three different values for the `SpecularExponent` property.



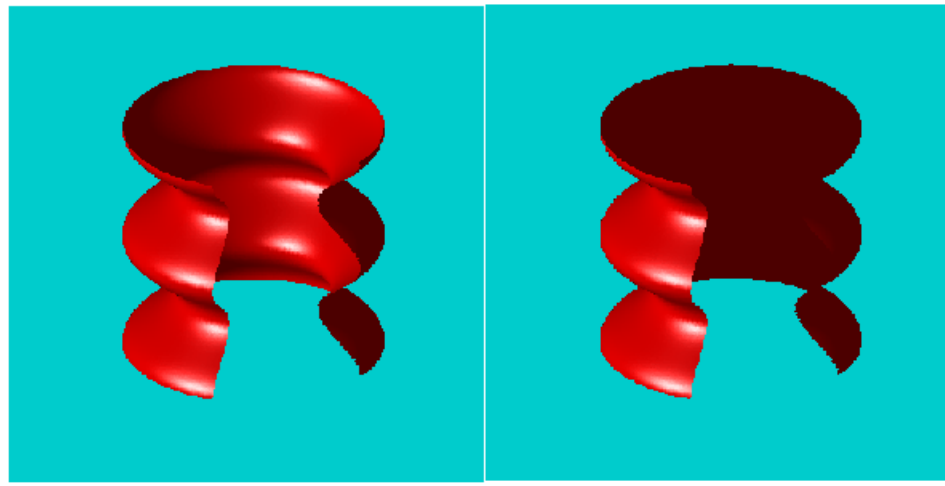
Specular Color Reflectance

The color of the specularly reflected light can range from a combination of the color of the object and the color of the light source to the color of the light source only. The patch and surface `SpecularColorReflectance` property controls this color. This illustration shows a red sphere illuminated by a white light. The values of the `SpecularColorReflectance` property range from 0 (object and light color) to 1 (light color).



Back Face Lighting

Back face lighting is useful for showing the difference between internal and external faces. These pictures of cut-away cylindrical surfaces illustrate the effects of back face lighting.

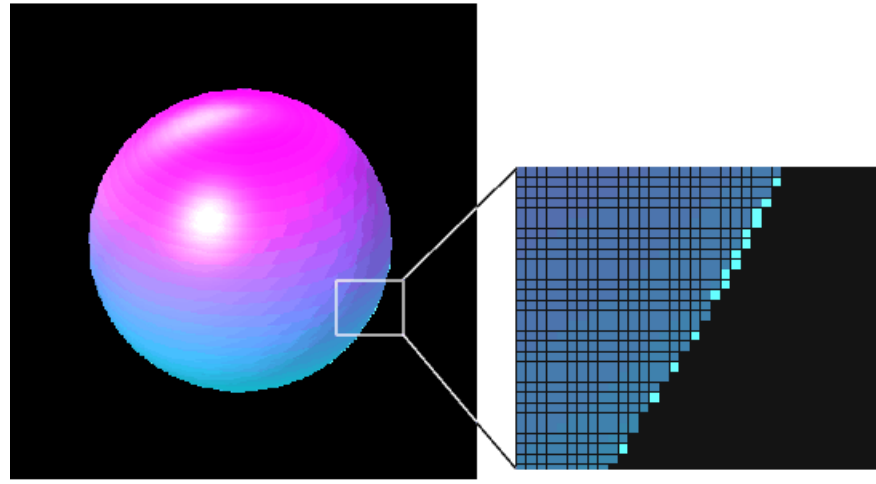


BackFaceLighting = reverselit BackFaceLighting = unlit

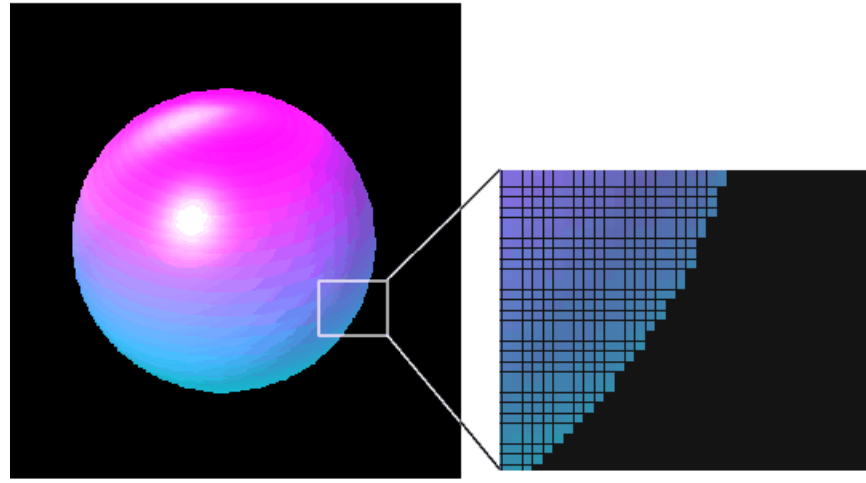
The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point away from the camera.

You can also use `BackFaceLighting` to remove edge effects for closed objects. These effects occur when `BackFaceLighting` is set to `reverselit` and pixels along the edge of a closed object are lit as if their vertex normals faced the camera. This produces an improperly lit pixel because the pixel is visible but is really facing away from the camera.

To illustrate this effect, the next picture shows a blowup of the edge of a lit sphere. Setting `BackFaceLighting` to `lit` prevents the improper lighting of pixels.



BackFaceLighting = reverselit



BackFaceLighting = lit

Positioning Lights in Data Space

This example creates a sphere and a cube to illustrate the effects of various properties on lighting. The variables `vert` and `fac` define the cube using the `patch` function.

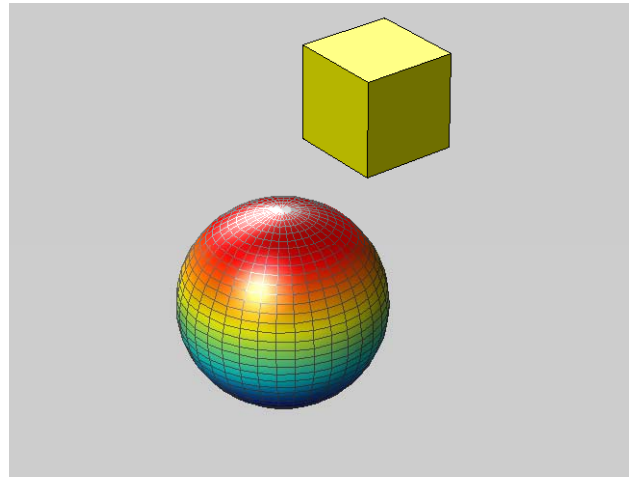
```
vert =                          fac =

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

```
sphere(36);
h = findobj('Type','surface');
set(h,'FaceLighting','phong',...
      'FaceColor','interp',...
      'EdgeColor',[.4 .4 .4],...
      'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material shiny
axis vis3d off
hold off
```

All faces of the cube have `FaceColor` set to yellow. The `sphere` function creates a spherical surface and the handle of this surface is obtained using `findobj` to search for the object whose `Type` property is `surface`. The `light` functions define two white (the default color) light objects located at infinity in the direction specified by the `Position` vectors. These vectors are defined in axes coordinates [*x*, *y*, *z*].

The patch uses `flat FaceLighting` (the default) to enhance the visibility of each side. The surface uses `phong FaceLighting` because it produces the smoothest interpolation of lighting effects. The `material shiny` command affects the reflectance properties of both the cube and sphere (although its effects are noticeable only on the sphere because of the cube's flat shading).

Because the sphere is closed, the `BackFaceLighting` property is changed from its default setting, which reverses the direction of vertex normals that face away from the camera, to normal lighting, which removes undesirable edge effects.



Examining the code in the `lighting` and `material` files can help you understand how various properties affect lighting.

4

Manipulating Transparency

Making Objects Transparent

In this section...

About Transparency

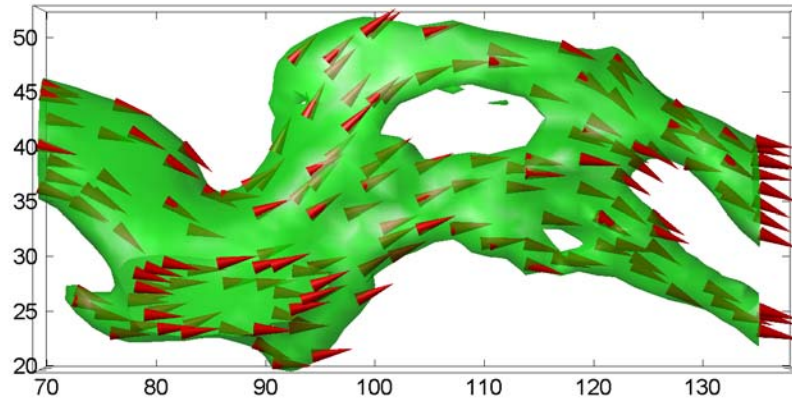
Making graphics objects semitransparent is a useful technique in 3-D visualization to make it possible to see an object, while at the same time, see what information the object would obscure if it was completely opaque. You can also use transparency as another dimension for displaying data, much the way color is used in MATLAB Graphics.

The transparency of a graphics object determines the degree to which you can see through the object. You can specify a continuous range of transparency varying from completely transparent (i.e., invisible) to completely opaque (i.e., no transparency).

Objects that support transparency are

- Image
- Patch
- Surface

The following picture illustrates the effect of transparency. The green isosurface (patch object) reveals the cone plot that lies in the interior.



Note You must have OpenGL available on your system to use transparency. When rendering transparency MATLAB automatically uses OpenGL if it is available. If it is not available, transparency does not display. See the figure property `RendererMode` for more information.

Specifying Transparency

Transparency values, which range from [0 1], are referred to as *alpha* values. An alpha value of 0 means completely transparent (i.e., invisible); an alpha value of 1 means completely opaque (i.e., no transparency).

MATLAB treats transparency in a way that is analogous to how it treats color for the respective objects:

- Patches and surfaces can define a single face and edge alpha value or use flat or interpolated transparency based on values in the figure's alphamap.
- Images, patches, and surfaces can define alpha data that is used as indices into the alphamap or directly as alpha values.
- Axes define alpha limits that control the mapping of object data to alpha values.
- Figures contain alphamaps, which are m-by-1 arrays of alpha values.

See the following sections for more information on color:

- "Specifying Patch Coloring" on page 5-15 in Creating 3-D Models with Patches in the Using MATLAB Graphics documentation
- "Coloring Mesh and Surface Plots" on page 1-20 in Creating 3-D Graphs in the Using MATLAB Graphics documentation

Transparency Properties

The following table summarizes the object properties that control transparency.

Property	Purpose
`AlphaData`	Transparency data for image and surface objects
`AlphaDataMapping`	Transparency data mapping method
`FaceAlpha`	Transparency of the faces (patch and surface only)
`EdgeAlpha`	Transparency of the edges (patch and surface only)
`FaceVertexAlphaData`	Patch only alpha data property
`ALim`	Alpha axis limits
`ALimMode`	Alpha axis limits mode
`Alphamap`	Figure alphamap

Transparency Functions

There are three functions that simplify the process of setting alpha properties.

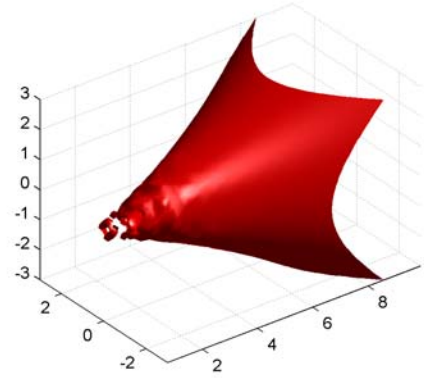
Function	Purpose
`alpha`	Set or query transparency properties for objects in current axes
`alphamap`	Specify the figure alphamap
`alim`	Set or query the axes alpha limits

Example — A Transparent Isosurface

Specifying a single transparency value for graphics objects is useful when you want to reveal structure that is obscured with opaque objects. For patches and surfaces, use the `FaceAlpha` and `EdgeAlpha` properties to specify the transparency of faces and edges. The following example illustrates this.

This example uses the `flow` function to generate data for the speed profile of a submerged jet within an infinite tank. One way to visualize this data is by creating an isosurface illustrating where the rate of flow is equal to a specified value.

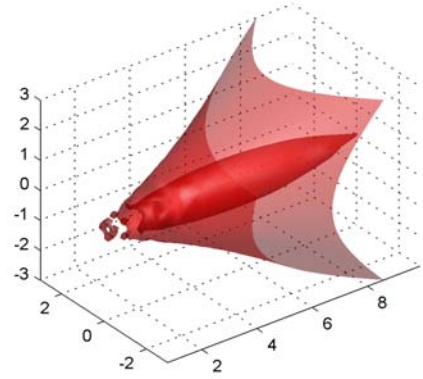
```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p);
set(p,'facecolor','red','edgecolor','none');
daspect([1 1 1]);
view(3); axis tight; grid on;
camlight; lighting gouraud;
```



Adding transparency to the isosurface reveals that there is greater complexity in the fluid flow than is visible using the opaque surface. The statement

```
alpha(.5)
```

sets the `FaceAlpha` value for the isosurface face to .5.



Setting a Single Transparency Value for Images

For images, the statement

```
alpha(.5)
```

sets `AlphaData` to .5. When the `AlphaDataMapping` property is set to `none` (the default), setting `AlphaData` on an image causes the entire image to be rendered with the specified alpha value.

Mapping Data to Transparency — Alpha Data

In this section...

What Is Alpha Data?

Alpha data is analogous to color data (e.g., the `CData` property of surfaces). When you create a surface, MATLAB rendering software maps each element in the color data array to a color in the colormap. Similarly, each element in the alpha data maps to a transparency value in the alphamap.

Specify surface and image alpha data with the `AlphaData` property. For patch objects, use the `FaceVertexAlphaData` property.

You can control how MATLAB interprets alpha data with the following properties:

- `FaceAlpha` and `EdgeAlpha` — Enable you to select flat or interpolated transparency rendering. If set to a single transparency value, MATLAB applies this value to all faces or edges and does not use the alpha data.
- `AlphaDataMapping` and `ALim` — Determine how MATLAB maps the alpha data to the alphamap. By default, MATLAB scales the alpha data to be within the range [0 1].
- `Alphamap` — Contains the actual transparency values to which the data is to be mapped.

There are differences between the default values of equivalent color and alpha properties because, in contrast to color, transparency is not displayed by default. The following table highlights these differences.

Color Property	Default	Alpha Property	Default
`FaceColor`	Flat	`FaceAlpha`	1 (opaque)
`CData`	Equal to `ZData`	`AlphaData` and `FaceVertexAlphaData`	1 (scalar)

By default, objects have single-valued alpha data. Therefore you cannot specify `flat` or `interp` `FaceAlpha` or `EdgeAlpha` without first setting `AlphaData` to an array of the appropriate size.

The sections that follow illustrate how to use these properties to display object data as degrees of transparency.

Size of the Alpha Data Array

In order to use nonscalar alpha data, you need to specify the alpha data as an array equal in size to

- `CData` of images and surfaces
- The number of faces (flat) or the number of vertices (interpolated) defined in the `FaceVertexAlphaData` property of patches

Once you have specified an alpha data array of the proper size, you can select the face and edge rendering you want to use. Flat uses one transparency value per face, while interpolated performs bilinear interpolation of the values at each vertex.

Mapping Alpha Data to the Alphamap

You can control how MATLAB maps the alpha data to the alphamap using the `AlphaDataMapping` property. There are three possible mappings:

- `none` — Interpret the values in alpha data as transparency values (data values must be between 0 and 1, or will be clamped to 0 or 1).
- `scaled` — Transform the alpha data to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values. This is the same way color data is mapped to the colormap.

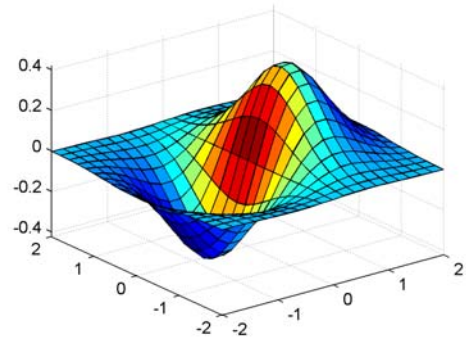
- `direct` — Use the alpha data directly as indices into the figure alphamap.

By default, objects have scalar alpha data (`AlphaData` and `FaceVertexAlphaData`) set to the value 1.

Example — Mapping Data to Color or Transparency

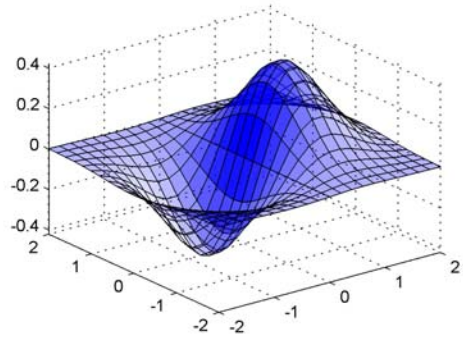
This example displays a surface plot of a function of two variables. The color is mapped to the gradient of the `z` data.

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z,gradient(z)); axis tight
```



You can map transparency to the gradient of `z` in a similar way.

```
surf(x,y,z,'FaceAlpha','flat',...
   'AlphaDataMapping','scaled',...
   'AlphaData',gradient(z),...
   'FaceColor','blue');
axis tight
```


0.4
0.2
0
-0.2
-0.4
2
1
0
-1
-2
-2
-1
0
1
2

Selecting an Alphamap

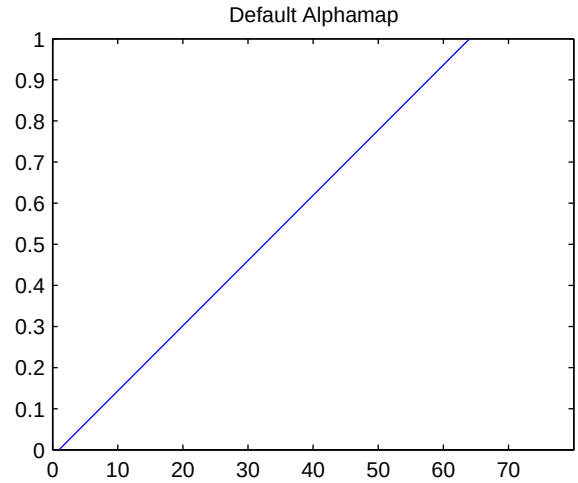
In this section...

What Is an Alphamap?

An alphamap is simply an array of values ranging from 0 to 1. The size of the array can be either m-by-1 or 1-by-m.

The default alphamap contains 64 values ranging linearly from 0 to 1, as you can see in the following plot.

```
plot(get(gcf,'Alphamap'))
```

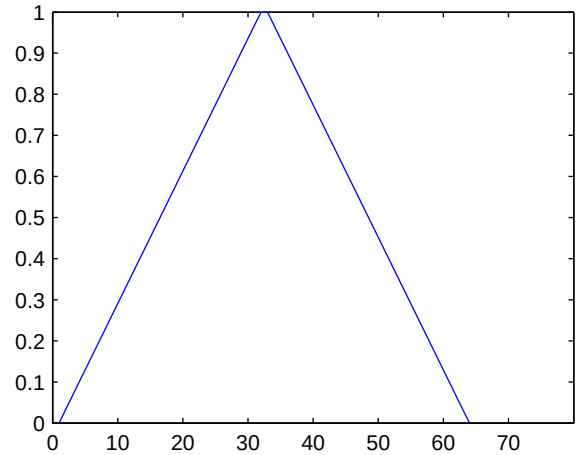


This alphamap displays the lowest alpha data values as completely transparent and the highest alpha data values as opaque.

The `alphamap` function creates some useful predefined alphamaps and also enables you to modify existing maps. For example,

```
plot(alphamap('vup'))
```

produces the following alphamap.

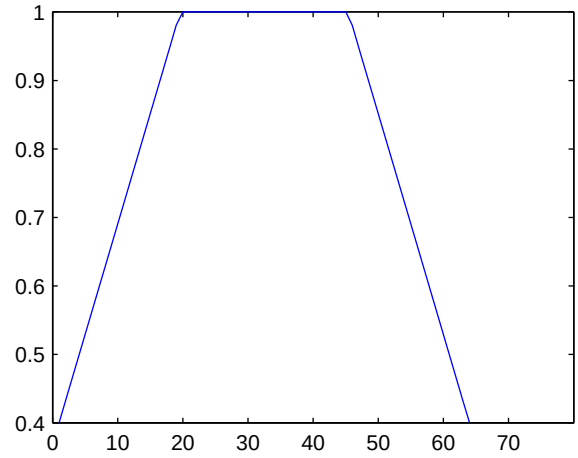


You can shift the values using the `increase` or `decrease` options. For example,

```
alphamap('increase',.4)
```

adds the value .4 to all values in the current figure's alphamap. Replotting the `'vup'` alphamap illustrates the change. The values are clamped to the range [0 1].

```
plot(get(gcf,'Alphamap'))
```



Example — Modifying the Alphamap

This example uses slice planes to examine volume data. The slice planes use the color data for alpha data and employ a rampdown alphamap (the values range from 1 to 0):

1. Create the volume data by evaluating a function of three variables.

```
[x,y,z] = meshgrid(-1.25:.1:-.25,-2:.2:2,-2:.1:2);
v = x.*exp(-x.^2-y.^2-z.^2);
```

2. Create the slice planes, set the alpha data equal to the color data, and specify interpolated `FaceAlpha`.

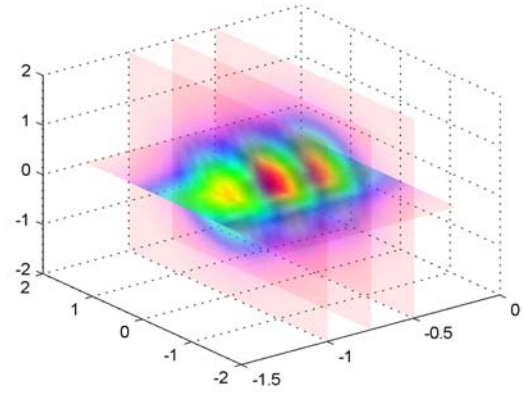
```
h = slice(x,y,z,v,[-1 -.75 -.5],[],[0]);
alpha('color')
set(h,'EdgeColor','none','FaceColor','interp',...
 'FaceAlpha','interp')
```

3. Install the rampdown alphamap and increase each value in the alphamap by .1 to achieve the desired degree of transparency. Specify the `hsv` colormap.

```
alphamap('rampdown')
```

```
alphamap('increase',.1)
colormap(hsv)
```

This alphamap causes the smallest values of the function (around zero) to be displayed with the least transparency and the greatest values to display with the most transparency. This enables you to see through the slice planes, while at the same time preserving the data around zero.



5

Creating 3-D Models with Patches

Introduction to Patch Objects

In this section...

What Are Patch Objects?

A patch graphics object is composed of one or more polygons that may or may not be connected. Patches are useful for modeling real-world objects such as airplanes or automobiles, and for drawing 2- or 3-D polygons of arbitrary shape.

In contrast, surface objects are rectangular grids of quadrilaterals and are better suited for displaying planar topographies such as the values of mathematical functions of two variables, the contours of data in a rectangular plane, or parameterized surfaces such as spheres.

A number of MATLAB functions create patch objects — `fill`, `fill3`, `isosurface`, `isocaps`, some of the `contour` functions, and `patch`. This section concentrates on use of the `patch` function.

You define a patch by specifying the coordinates of its vertices and some form of color data. Patches support a variety of coloring options that are useful for visualizing data superimposed on geometric shapes.

There are two ways to specify a patch:

- By specifying the coordinates of the vertices of each polygon, which are connected to form the patch
- By specifying the coordinates of each *unique* vertex and a matrix that specifies how to connect these vertices to form the faces

The second technique is preferred for multifaceted patches because it generally requires less data to define the patch; vertices shared by more than

one face need be defined only once. This section provides examples of both techniques.

Behavior of the patch Function

There are two forms of the `patch` function -- high-level syntax and low-level syntax. The behavior of the `patch` function differs somewhat depending on which syntax you use.

High-Level Syntax

When you use the high-level syntax, MATLAB automatically determines how to color each face based on the color data you specify. The high-level syntax enables you to omit the property names for the *x*-, *y*-, and *z*-coordinates and the color data, as long as you specify these arguments in the correct order.

```
patch(x-coordinates,y-coordinates,z-coordinates,colordata)
```

However, you must specify color data so MATLAB can determine what type of coloring to use. If you do not specify color data, MATLAB returns an error.

```
patch(sin(t),cos(t))
??? Error using ==> patch
Not enough input arguments.
```

Low-Level Syntax

The low-level syntax accepts only property name/property value pairs as arguments and does not automatically color the faces unless you also change the value of the `FaceColor` property. For example, the statement

```
patch('XData',sin(t),'YData',cos(t)) % Low-level syntax
```

draws a patch with white face color because the factory default value for the `FaceColor` property is the color white.

```
get(0,'FactoryPatchFaceColor')
ans =
      1     1     1
```

See the list of patch properties in the MATLAB Function Reference and the `get` command for information on how to obtain the factory and user default values for properties.

Interpreting the Color Argument

When you use the low-level syntax, MATLAB interprets the third (or fourth if there are *z*-coordinates) argument as color data. If you intend to define a patch with *x*-, *y*-, and *z*-coordinates, but leave out the color, MATLAB interprets the *z*-coordinates as color data, and then draws a 2-D patch. For example,

```
h = patch(sin(t),cos(t),1:length(t))
```

draws a patch with all vertices at z = 0, colored by interpolating the vertex colors (since there is one color for each vertex), whereas

```
h = patch(sin(t),cos(t),1:length(t),'y')
```

draws a patch with vertices at increasing values of *z*, colored yellow.

"Specifying Patch Coloring" on page 5-15 provides more information on options for coloring patches.

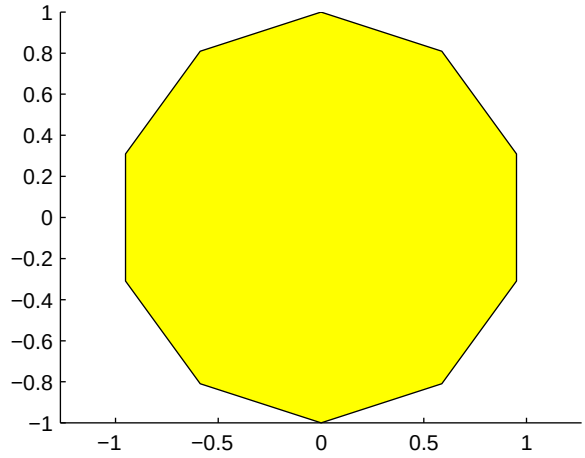
Creating a Single Polygon

A polygon is simply a patch with one face. To create a polygon, specify the coordinates of the vertices and color data with a statement of the form

```
patch(x-coordinates,y-coordinates,[z-coordinates],colordata)
```

For example, these statements display a 10-sided polygon with a yellow face enclosed by a black edge. The `axis equal` command produces a correctly proportioned polygon.

```
t = 0:pi/5:2*pi;
figure
patch(sin(t),cos(t),'y')
axis equal
```

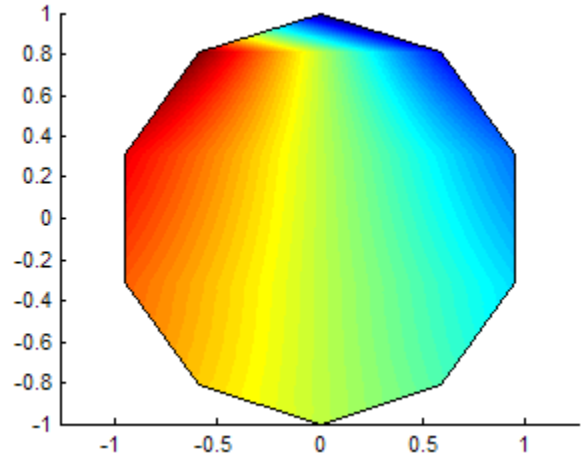


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

Interpolated Face Colors

You can control many aspects of the patch coloring. For example, instead of specifying a single color, you can provide a range of numerical values that map the color at each vertex to a color in the figure colormap.

```
a = t(1:length(t)-1); %remove redundant vertex definition
figure
patch(sin(a),cos(a),1:length(a),'FaceColor','interp')
axis equal
```



MATLAB now interpolates the colors across the face of the patch. You can color the edges of the patch the same way, by setting the edge colors to be interpolated. The command is

```
patch(sin(t),cos(t),1:length(t),'EdgeColor','interp')
```

"Interpolating in Indexed Color Versus Truecolor" on page 5-23 provides information on how patches interpolate face colors.

"Specifying Patch Coloring" on page 5-15 provides more information on options for coloring patches.

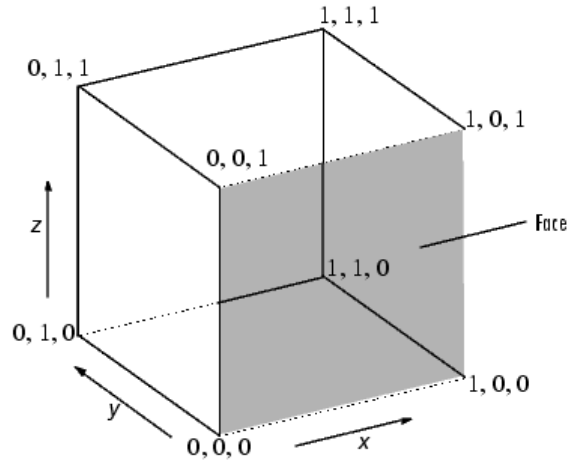
Multifaceted Patches

Example — Defining a Cube

A cube is defined by eight vertices that form six sides. This illustration shows the *x*-, *y*-, and *z*-coordinates of the vertices defining a cube in which the sides are one unit in length.

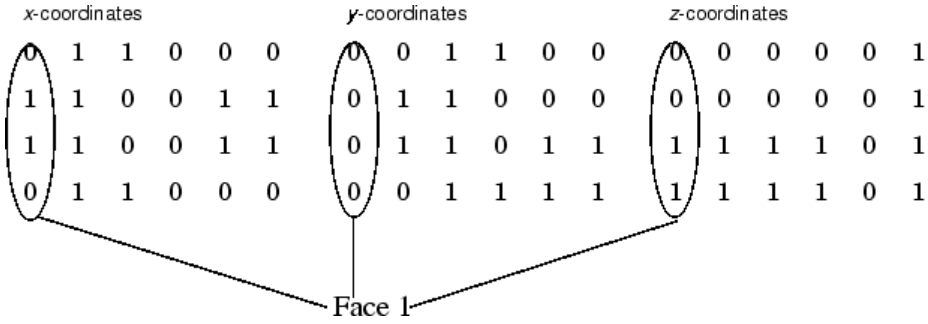
If you specify the *x*-, *y*-, and *z*-coordinate arguments as vectors, they render as a single polygon with points connected in sequence. If the arguments are matrices, MATLAB draws one polygon per column, producing a single patch with multiple faces. These faces need not be connected and can be self-intersecting.

Alternatively, you can specify the coordinates of each unique vertex and the order in which to connect them to form the faces. The examples in this section illustrate both techniques.



Specifying X, Y, and Z Coordinates

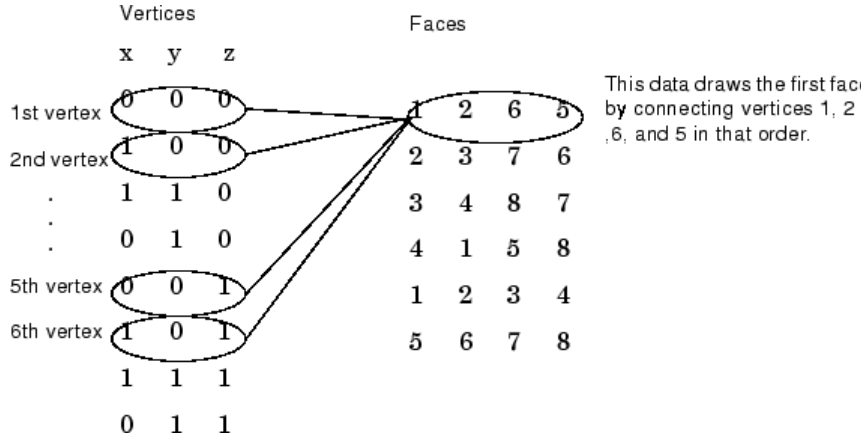
Each of the six faces has four vertices. Because you do not need to close each polygon (i.e., the first and last vertices do not need to be the same), you can define this cube using a 4-by-6 matrix for each of the *x*-, *y*-, and *z*-coordinates.



Each column of the matrices specifies a different face. While there are only eight vertices, you must specify 24 vertices to define all six faces. Since each face shares vertices with four other faces, you can define the patch more efficiently by defining each vertex only once and then specifying the order in which to connect these vertices to form each face. The patch `Vertices` and `Faces` properties define patches in just this way.

Specifying Faces and Vertices

These matrices specify the cube using `Vertices` and `Faces`.



Using the vertices/faces technique can save a considerable amount of computer memory when patches contain a large number of faces. This technique requires the formal `patch` function syntax, which entails assigning values to the `Vertices` and `Faces` properties explicitly. For example,

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix)
```

Because the high-level syntax does not automatically assign face or edge colors, you must set the appropriate properties to produce patches with colors other than the default white face color and black edge color.

Flat Face Color

Flat face color is the result of specifying one color per face. For example, using the vertices/faces technique and the `FaceVertexCData` property to define color, this statement specifies one color per face and sets the `FaceColor` property to `flat`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(6),'FaceColor','flat')
```

Because truecolor specified with the `FaceVertexCData` property has the same format as a MATLAB colormap (i.e., an *n*-by-3 array of RGB values), this example uses the `hsv` colormap to generate the six colors required for flat shading.

Interpolated Face Color

Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(8),'FaceColor','interp')
```

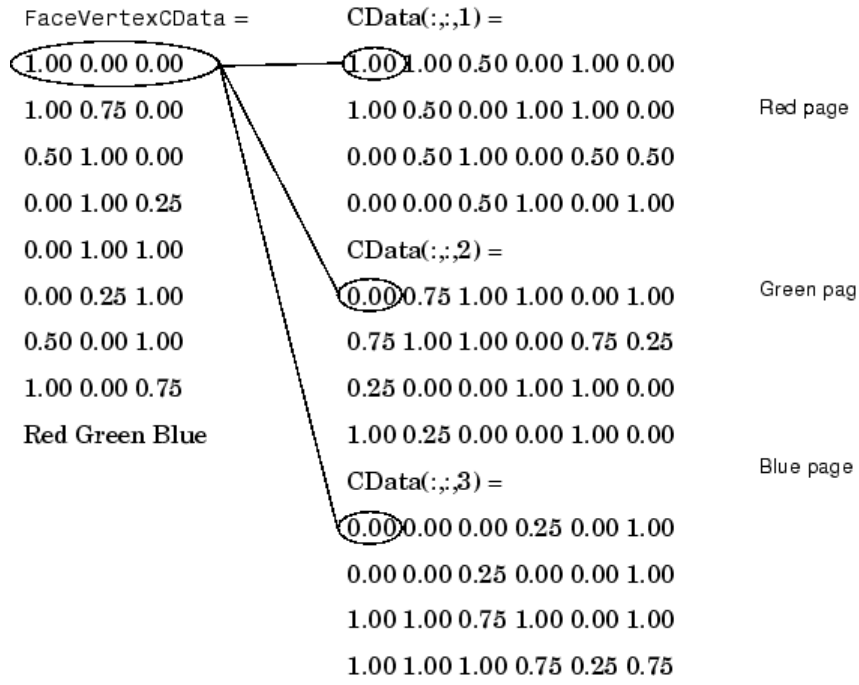
Changing to the standard 3-D view and making the axis square,

```
view(3); axis square
```

produces a cube with each face colored by interpolating the vertex colors.

To specify the same coloring using the x, y, z, c technique, c must be an *m*-by-*n*-by-3 array, where the dimensions of x, y, and z are *m*-by-*n*.

This diagram shows the correspondence between the `FaceVertexCData` and `CData` properties.



"Specifying Patch Coloring" on page 5-15 discusses coloring techniques in more detail.

Modifying Data on Existing Patch Objects

In this section...

Specifying Patch Data

In general, if you define a patch with `Faces` and `Vertices` data and then want to modify its data, you should continue to use these same properties. Do not switch modes and modify the `XData`, `YData`, `ZData`, or `CData` properties.

Handling Mixed Data Specification

If you specify a patch with `Faces` and `Vertices` data, MATLAB constructs arrays of data for the `XData`, `YData`, `ZData` and `CData` properties when you query these properties. However, these arrays contain only enough data to define the same number of vertices as there are referred to in the `Faces` property.

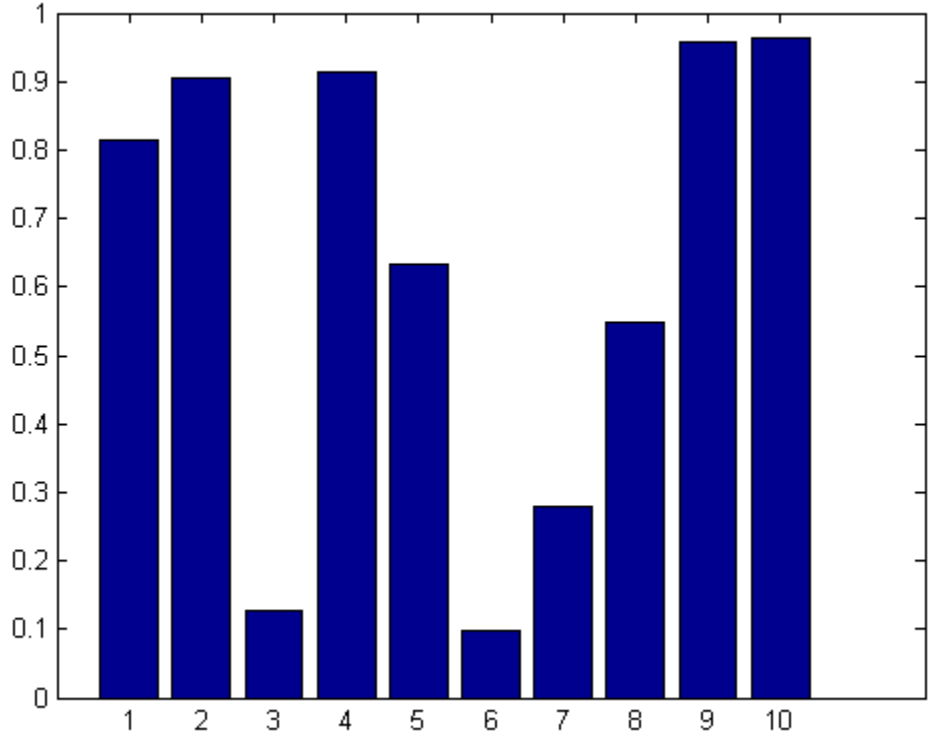
If the number of vertices in the `Vertices` property is greater than the number of vertices used by the `Faces` property, then MATLAB plotting functions cannot generate complete *x*, *y*, and *z* data from the faces and vertex data.

While you should not use mixed data specification when defining patch objects directly, you might need to modify patch data when using functions that create patch objects. For example, the `bar` function creates patch objects to implement the bars in a graph.

Note The barseries `YData` property enables you to modify the bar graph without the need to use the following steps. See the `bar` function for more information on working with bar graphs.

The `bar` function uses y-data values to determine the height of each bar, but creates each bar as the face of a patch specified by faces and vertices. For example,

```
rng(0,'twister')
h = bar(rand(10,1));   % y data for each bar
p = get(h,'children'); % get the handle of the patch
cl = get(gca,'CLim');
```



Before you can change the patch `YData` property, you must switch the patch to x, y, and z data as follows:

```
xd = get(p,'XData');
yd = get(p,'YData');
zd = get(p,'ZData');
cd = get(p,'CData');
set(p,'XData',xd,'YData',yd,'ZData',zd,'CData',cd);
```

```
set(gca,'CLim',cl)
```

Setting the `XData`, `YData`, `ZData` and `CData` properties causes the `patch` function to match the faces and vertex data with x, y, and z data. Because there is a change in the patch data, the color limits change, so you must use the original values for the axes `CLim` property.

You can now modify the y data values to change your graph. For example, the value of bar at x = 6 is 0.0975:

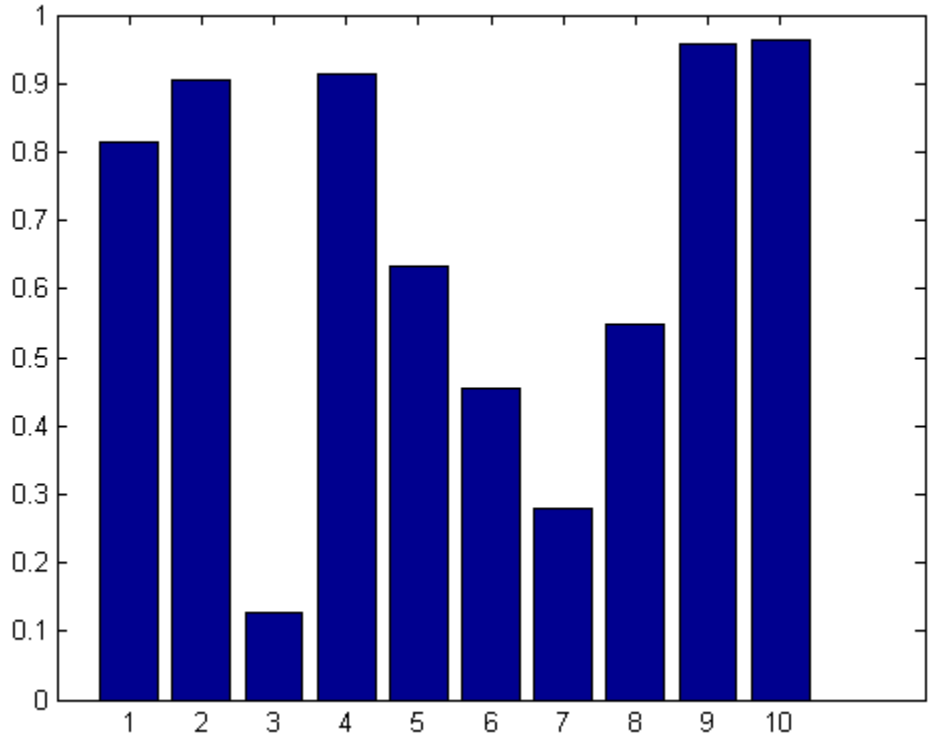
```
yd(:,6) % vertices of patch defining sixth bar
ans =
         0
    0.0975
    0.0975
         0
```

You can change this bar by changes rows 2 and 3 in the y data for column 6. For example, set the height of the bar equal to the average of bar 5 and bar7:

```
% Use value from row 2 to calculate mean
yd(2:3,6) =  mean([yd(2,5),yd(2,7)]);
```

Now reset the patch `YData` property:

```
set(p,'YData',yd)
```



Another reason you might want to modify face-vertex data for bar graphs or other objects is to modify their `CData` to customize how they are colored. Techniques for doing this for 2–D and 3–D bar graphs are explained in "Coloring 2-D Bars According to Height" and "Coloring 3-D Bars According to Height" in the MATLAB Graphics documentation.

Specifying Patch Coloring

In this section...

Patch Color Properties

Patch coloring is defined differently from surface object coloring in that patches do not automatically generate color data based on the value of the *z*-coordinate at each vertex. You must explicitly specify patch coloring if you do not want the default white face color and black edge color.

You can specify patch face coloring by defining

- A single color for all faces
- One color for each face, which is used for flat coloring
- One color for each vertex, which is used for interpolated coloring

Specify the face color using either the `CData` property, if you are using *x*-, *y*-, and *z*-coordinates, or the `FaceVertexCData` property, if you are specifying vertices and faces.

This table summarizes the patch properties that control color (exclusive of those used when light sources are present).

Property	Purpose
`CData`	Specify single, per face, or per vertex colors in conjunction with x, y, and z data
`CDataMapping`	Specifies whether color data is scaled or used directly as indices into the figure colormap
`FaceVertexCData`	Specify single, per face, or per vertex colors in conjunction with faces and vertices data

Property	Purpose
`EdgeColor`	Specifies whether edges are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors
`FaceColor`	Specifies whether faces are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors
`MarkerEdgeColor`	Specifies the color of the marker, or the edge color for filled markers
`MarkerFaceColor`	Specifies the fill color for markers that are closed shapes

Patch Edge Coloring

Each patch face has a bounding edge, which you can color as

- A single color for all edges
- A flat color defined by the color of the vertex that precedes the edge
- Interpolated colors determined by the two vertices that bound the edge

Patch edge colors can be flat or interpolated only when you specify a color for each vertex. Flat edge coloring uses the color of the vertex preceding the edge to determine the color of the edge. The order in which you specify the vertices establishes which vertex colors a particular edge.

The following examples illustrate patch edge coloring:

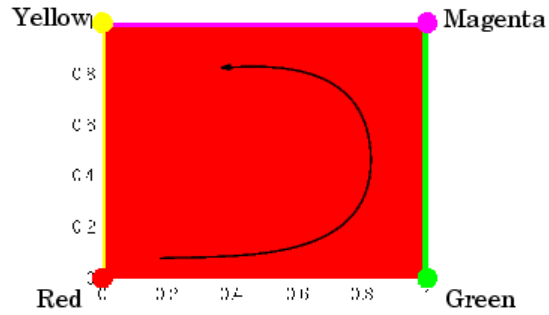
- "Example — Specifying Flat Edge and Face Coloring" on page 5-16
- "Coloring Edges with Shared Vertices" on page 5-18

Example — Specifying Flat Edge and Face Coloring

These statements create a square patch.

```
v  = [0 0 0;1 0 0;1 1 0;0 1 0];
f = [1 2 3 4];
```

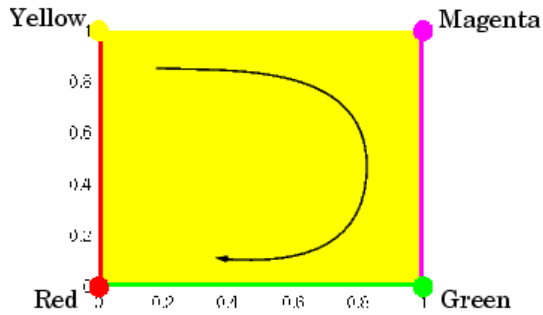
```
fvc = [1 0 0;0 1 0;1 0 1;1 1 0];
patch('Vertices',v,'Faces',f,'FaceVertexCData',fvc,...
      'FaceColor','flat','EdgeColor','flat',...
      'Marker','o','MarkerFaceColor','flat')
```



The `Faces` property value, `[1 2 3 4]`, determines the order in which MATLAB connects the vertices. In this case, the order is red, green, magenta, and yellow. If you change this order, the results can be quite different. For example, specifying the `Faces` property as

```
f = [4 3 2 1];
```

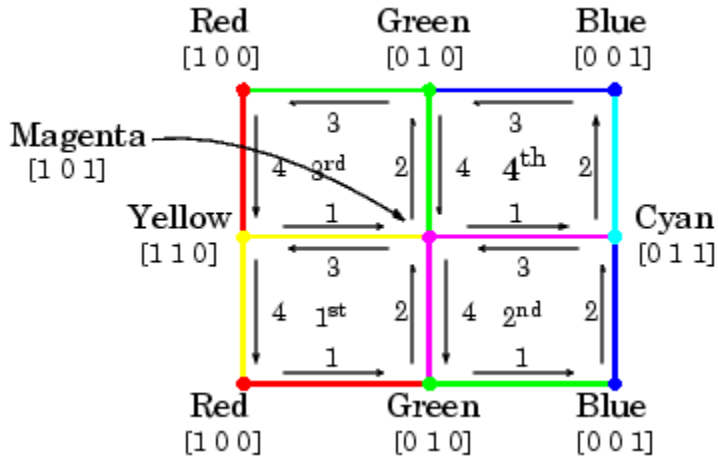
changes the order to yellow, magenta, green, and red. Changing the order not only changes the color of the edges, but also the color of the face, which is the color of the first vertex specified.



Coloring Edges with Shared Vertices

Each patch face is bound by edges, which are line segments that connect the vertices. When patches have multiple faces that share vertices, some of the edges might overlap. In such cases, the edges of the most recently drawn face overlie previously drawn edges.

For example, this illustration shows a patch with four faces and flat colored edges (`FaceColor` set to `none`, `EdgeColor` set to `flat`).



The arrows indicate the order in which each edge is drawn in the first, second, third, and fourth face. The color at each vertex determines the color of the edge that follows it. Notice how the second edge in the first face would be green except that the second face drew its fourth edge from the magenta vertex. You can see similar effects in all shared edges.

For `EdgeColor` set to `interp`, MATLAB interpolates colors between adjacent vertices. In this case, the order in which you specify the vertices does not affect the edge color.

Interpreting Indexed and Truecolor Data

In this section...

Introduction

Patch color data is interpreted in either of two ways:

- Indexed Color Data — Numerical values that are mapped to colors defined in the figure colormap
- Truecolor Data — RGB triples that define colors explicitly and do not make use of the figure colormap

The dimensions of the color data (`CData` or `FaceVertexCData`) determine how such data is interpreted. If you specify only one numeric value per patch, per face, or per vertex, the data is regarded as indexed. If there are three numeric values per patch, face, or vertex, the values are interpreted as RGB triplets.

Indexed Color Data

Indexed color data can either be interpreted as values to scale before mapping to the colormap, or directly as indices into the colormap. You control the interpretation by setting the `CDataMapping` property. The default is to scale the data.

Scaled Color

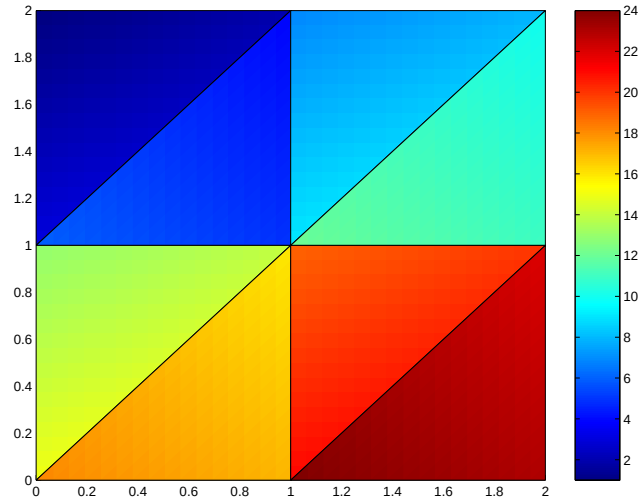
By default, the color data is scaled so that the minimum value maps to the first color in the colormap, the maximum value maps to the last color in the colormap, and values in between are linearly transformed to span the colormap. This enables you to use colormaps of different sizes without changing your data and to use data in any range of values without changing the colormap.

For example, the following patch has eight triangular faces with a total of 24 (nonunique) vertices. The color data are integers that range from one to 24, but could be any values.

The variable `c` contains the color data. It is a 3-by-8 matrix, with each column specifying the colors for the three vertices of each face.

```
c =
     1     4     7    10    13    16    19    22
     2     5     8    11    14    17    20    23
     3     6     9    12    15    18    21    24
```

The color bar (`colorbar`) on the right side of the patch illustrates the colormap used and indicates with the vertical axis which color is mapped to the respective data value.



You can alter the mapping of color data to colormap entry using the `caxis` command. This command uses a two-element vector [`cmin cmax`] to specify what data values map to the beginning and end of the colormap, thereby shifting the color mapping.

By default, `cmin` is set to the minimum value and `cmax` to the maximum value of the color data of all graphics objects within the axes. However, you can set these limits to span any range of values and thereby shift the color mapping. See Calculating Color Limits in "Axes Properties" in the Using MATLAB Graphics documentation for more information.

The color data does not need to be a sequential list of integers; it can be any matrix with dimensions matching the coordinate data. For example,

```
patch(x,y,z,rand(size(z)))
```

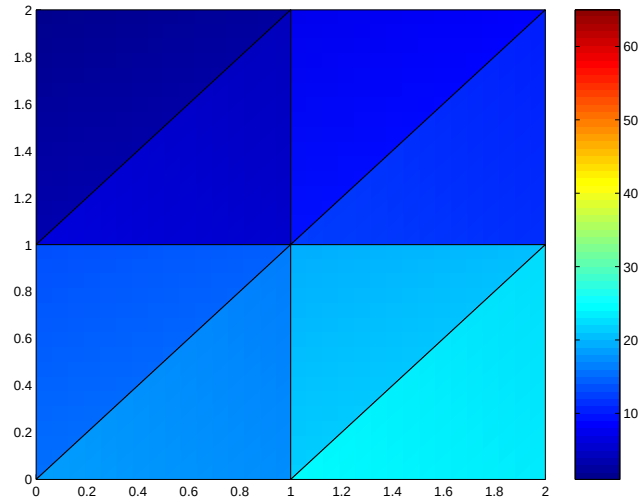
Direct Color

If you set the patch `CDataMapping` property to `direct`,

```
set(patch_handle,'CDataMapping','direct')
```

MATLAB graphic software interprets each color data value as a direct index into the colormap. That is, a value of 1 maps to the first color, a value of 2 maps to the second color, and so on.

The patch from the previous example would then use only the first 24 colors in the colormap.



This example uses integer color data. However, if the values are not integers, they are converted according to these rules:

- If value is < 1, it maps to the first color in the colormap.
- If value is not an integer, it is rounded to the nearest integer toward zero.
- If value > `length(colormap)`, it maps to the last color in the colormap.

Unscaled color data is more commonly used for images where there is typically a colormap associated with a particular image.

Truecolor Patches

Truecolor is a means to specify a color explicitly with RGB values rather than pointing to an entry in the figure colormap. Truecolor generally provides a greater range of colors than can be defined in a colormap.

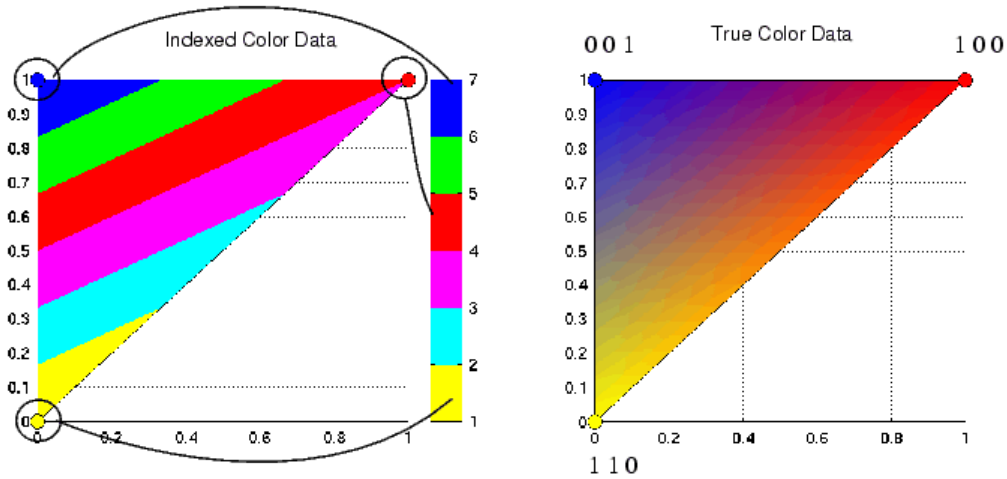
Using truecolor eliminates the mapping of data to colormap entries. On the other hand, you cannot change the coloring of the patch without redefining the color data (as opposed to just changing the colormap).

Interpolating in Indexed Color Versus Truecolor

When you specify interpolated face coloring, the color of each face is determined by interpolating the vertex colors. The method of interpolation depends on whether you specified truecolor data or indexed color data.

With truecolor data, the numeric RGB values defined for the vertices are interpolated. This generally produces a smooth variation of color across the face. In contrast, indexed color interpolation uses only colors that are defined in the colormap. With certain colormaps, the results can be quite different.

To illustrate this difference, these two patches are defined with the same vertex colors. Circular markers indicate the yellow, red, and blue vertex colors.



The patch on the left uses indexed colors obtained from the six-element colormap shown next to it. The color data maps the vertex colors to the colormap elements indicated in the picture. With this colormap, interpolating from the cyan vertex to the blue vertex can include only the colors green, red, yellow, and magenta, hence the banding.

Interpolation in RGB space makes no use of the colormap. It is simply the gradual transition from one numeric value to another. For example,

interpolating from the cyan vertex to the blue vertex follows a progression similar to these values.

```
0 1 1, 0 0.9 1, 0 0.8 1, ... 0 0.2 1, 0 0.1 1, 0 0 1
```

In reality each pixel would be a different color so the incremental change would be much smaller than illustrated here.

6

Volume Visualization Techniques

Overview of Volume Visualization

In this section...

Examples of Volume Data

Volume visualization is the creation of graphical representations of data sets that are defined on three-dimensional grids. Volume data sets are characterized by multidimensional arrays of scalar or vector data. These data are typically defined on lattice structures representing values sampled in 3-D space. There are two basic types of volume data:

- *Scalar volume data* contains single values for each point.
- *Vector volume data* contains two or three values for each point, defining the components of a vector.

An example of scalar volume data is that produced by `flow`. The flow data represents the speed profile of a submerged jet within an infinite tank. Typing

```
[x,y,z,v] = flow;
```

produces four 3-D arrays. The `x`, `y`, and `z` arrays specify the coordinates of the scalar values in the array `v`.

The `wind` data set is an example of vector volume data that represents air currents over North America. You can load this data in the MATLAB workspace with the command:

```
load wind
```

This data set comprises six 3-D arrays: `x`, `y`, and `z` are the coordinate data for the arrays `u`, `v`, and `w`, which are the vector components for each point in the volume.

Selecting Visualization Techniques

The techniques you select to visualize volume data depend on what type of data you have and what you want to learn. In general:

- Scalar data is best viewed with isosurfaces, slice planes, and contour slices.
- Vector data represents both a magnitude and direction at each point, which is best displayed by stream lines (particles, ribbons, and tubes), cone plots, and arrow plots. Most visualizations, however, employ a combination of techniques to best reveal the content of the data.

The material in these sections describes how to apply a variety of techniques to typical volume data.

Interpolating and Gridding Data

MATLAB provides functions that enable you to interpolate and restructure you data in preparation for visualization. See these sections for more information:

- "Interpolating Gridded Data"
- "Interpolating Scattered Data"

Steps to Create a Volume Visualization

Creating an effective visualization requires a number of steps to compose the final scene. These steps fall into four basic categories:

1. Determine the characteristics of your data. Graphing volume data usually requires knowledge of the range of both the coordinates and the data values.
2. Select an appropriate plotting routine. The information in this section helps you select the right methods.
3. Define the view. The information conveyed by a complex three-dimensional graph can be greatly enhanced through careful composition of the scene. Viewing techniques include adjusting camera position, specifying aspect ratio and project type, zooming in or out, and so on.
4. Add lighting and specify coloring. Lighting is an effective means to enhance the visibility of surface shape and to provide a three-dimensional

perspective to volume graphs. Color can convey data values, both constant and varying.

Volume Visualization Functions

MATLAB functions enable you to apply a variety of volume visualization techniques. The following tables group these functions into two categories based on the type of data (scalar or vector) that each is designed to work with. The reference page for each function provides examples of the intended use.

Functions for Scalar Data

Function	Purpose
`contourslice`	Draw contours in volume slice planes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute the colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`patch`	Create a patch (multipolygon) graphics object
`reducepatch`	Reduce the number of patch faces
`reducevolume`	Reduce the number of elements in a volume data set
`shrinkfaces`	Reduce the size of each patch face
`slice`	Draw slice planes in volume
`smooth3`	Smooth 3-D data
`surf2patch`	Convert surface data to patch data
`subvolume`	Extract subset of volume data set

Functions for Vector Data

Function	Purpose
`coneplot`	Plot velocity vectors as cones in 3-D vector fields
`curl`	Compute the curl and angular velocity of a 3-D vector field
`divergence`	Compute the divergence of a 3-D vector field
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`streamline`	Draw stream lines from 2-D or 3-D vector data
`streamparticles`	Draw stream particles from vector volume data
`streamribbon`	Draw stream ribbons from vector volume data
`streamslice`	Draw well-spaced stream lines from vector volume data
`streamtube`	Draw stream tubes from vector volume data
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

Techniques for Visualizing Scalar Volume Data

In this section...

What Is Scalar Volume Data?

Typical scalar volume data is composed of a 3-D array of data and three coordinate arrays of the same dimensions. The coordinate arrays specify the *x*-, *y*-, and *z*-coordinates for each data point.

The units of the coordinates depend on the type of data. For example, flow data might have coordinate units of inches and data units of psi.

A number of MATLAB functions are useful for visualizing scalar data:

- Slice planes provide a way to explore the distribution of data values within the volume by mapping values to colors. You can orient slice planes at arbitrary angles, as well as use nonplanar slices. (For illustrations of how to use slice planes, see `slice`, a volume slicing example, and slice planes used to show context.) You can specify the data used to color isosurfaces, enabling you to display different information in color and surface shape (see `isocolors`).
- Contour slices are contour plots drawn at specific coordinates within the volume. Contour plots enable you to see where in a given plane the data values are equal. See `contourslice` for an example.
- Isosurfaces are surfaces constructed by using points of equal value as the vertices of `patch` graphics objects.

Ways to Display MRI Data

An example of scalar data includes magnetic resonance imaging (MRI) data. This data typically contains a number of slice planes taken through a volume, such as the human body. MATLAB includes an MRI data set that contains 27 image slices of a human head. This example illustrate the following techniques applied to MRI data:

- A series of 2-D images representing slices through the head
- 2-D and 3-D contour slices taken at arbitrary locations within the data
- An isosurface with isocaps showing a cross section of the interior

Changing the Data Format

The MRI data, `D`, is stored as a 128-by-128-by-1-by-27 array. The third array dimension is used typically for the image color data. However, since these are indexed images (a colormap, `map`, is also loaded) there is no information in the third dimension, which you can remove using the `squeeze` command. The result is a 128-by-128-by-27 array.

The first step is to load the data and transform the data array from 4-D to 3-D.

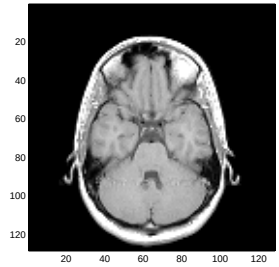
```
load mri
D = squeeze(D);
```

Displaying Images of MRI Data

To display one of the MRI images, use the `image` command:

- Create a new `figure` that uses the MRI colormap, which is loaded with the data:
- Index into the data array to obtain the data for the eighth image.
- Adjust `axis` scaling.

```
figure('Colormap',map)
image_num = 8;
image(D(:,:,image_num))
axis image
```



Save the *x*- and *y*-axis limits for use in the next part of the example:

```
x = xlim;
y = ylim;
```

Displaying a 2-D Contour Slice

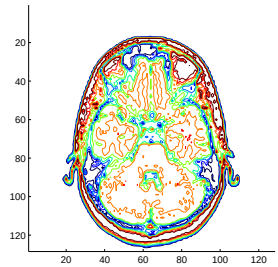
Visualize MRI data as a volume data because it is a collection of slices taken progressively through the 3-D object. Use `contourslice` to display a contour plot of a volume slice. Create a contour plot with the same orientation and size as the image created in the first part of this example:

- Adjust the *y*-axis direction (`axis`).
- Set the limits (`xlim`, `ylim`).
- Set the data aspect ratio (`daspect`).

To improve the visibility of details, this contour plot uses the `jet` colormap. The `brighten` function reduces the brightness of the color values.

```
cm = brighten(jet(length(map)),-.5);
figure('Colormap',cm)
contourslice(D,[],[],image_num)
axis ij
xlim(x)
```

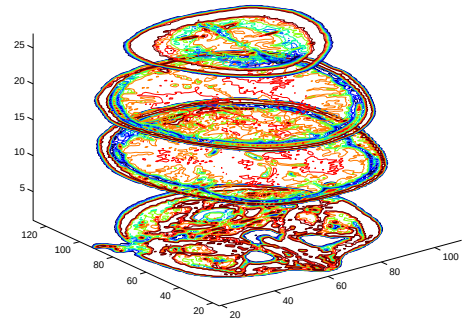
```
ylim(y)
daspect([1,1,1])
```



Displaying 3-D Contour Slices

Unlike images, which are 2-D objects, contour slices are 3-D objects that you can display in any orientation. For example, you can display four contour slices in a 3-D view.

```
figure('Colormap',cm)
contourslice(D,[],[],[1,12,19,27],8);
view(3); axis tight
```



Applying an Isosurface to the MRI Data

You can use isosurfaces to display the overall structure of a volume. When combined with isocaps, this technique can reveal information about data on the interior of the isosurface.

First, smooth the data with `smooth3`; then use `isosurface` to calculate the isodata. Use `patch` to display this data in a figure that uses the original gray scale color map for the isocaps.

```
figure('Colormap',map)
Ds = smooth3(D);
hiso = patch(isosurface(Ds,5),...
 'FaceColor',[1,.75,.65],...
 'EdgeColor','none');
 isonormals(Ds,hiso)
```

The `isonormals` function to renders the isosurface using vertex normals obtained from the smoothed data, improving the quality of the isosurface. The isosurface uses a single color to represent its isovalue.

Adding Isocaps Show Cut-Away Surface

Use `isocaps` to calculate the data for another patch that is displayed at the same isovalue (`5`) as the isosurface. Use the unsmoothed data (`D`) to show details of the interior. You can see this as the sliced-away top of the head. The lower isocap is not visible in the final view.

```
hcap = patch(isocaps(D,5),...
 'FaceColor','interp',...
 'EdgeColor','none');
```

Defining the View

Define the view and set the aspect ratio (`view`, `axis`, `daspect`).

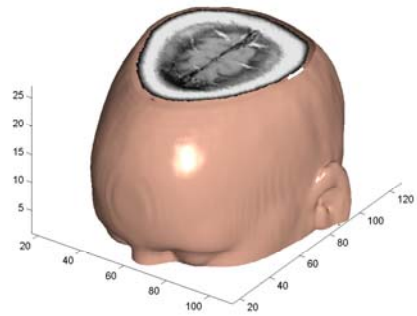
```
view(35,30)
axis tight
daspect([1,1,.4])
```

Add Lighting

Add lighting and recalculate the surface normals based on the gradient of the volume data, which produces smoother lighting (`camlight`, `lighting`, `isonormals`). Increase the `AmbientStrength` property of the isocap to brighten the coloring without affecting the isosurface. Set the `SpecularColorReflectance` of the isosurface to make the color of the specular reflected light closer to the color of the isosurface; then set the `SpecularExponent` to reduce the size of the specular spot.

```
lightangle(45,30);
set(gcf,'Renderer','zbuffer'); lighting phong
set(hcap,'AmbientStrength',.6)
set(hiso,'SpecularColorReflectance',0,'SpecularExponent',50)
```

An Isocap combined with an isosurface to visualize MRI data.



The isocaps use interpolated face coloring, which means the figure colormap determines the coloring of the patch. This example uses the colormap supplied with the data.

To display isocaps at other data values, try changing the isosurface value or use the `subvolume` command. See the `isocaps` and `subvolume` reference pages for examples.

Exploring Volumes with Slice Planes

In this section...

Slicing Fluid Flow Data

A slice plane (which does not have to be planar) is a surface that takes on coloring based on the values of the volume data in the region where the slice is positioned. Slice planes are useful for probing volume data sets to discover where interesting regions exist, which you can then visualize with other types of graphs (see the `slice` example). Slice planes are also useful for adding a visual context to the bound of the volume when other graphing methods are also used (see `coneplot` and "Stream Line Plots of Vector Data" on page 6-32 for examples).

Use the `slice` function to create slice planes. This example slices through a volume generated by `flow`.

1. Investigate the Data

Generate the volume data with the command:

```
[x,y,z,v] = flow;
```

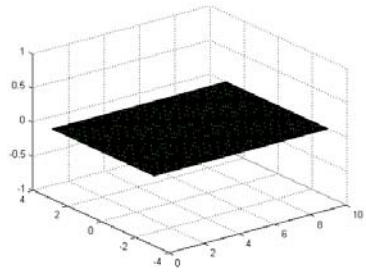
Determine the range of the volume by finding the minimum and maximum of the coordinate data.

```
xmin = min(x(:));
ymin = min(y(:));
zmin = min(z(:));

xmax = max(x(:));
ymax = max(y(:));
zmax = max(z(:));
```

2. Slice Plane at an Angle to the X-Axes

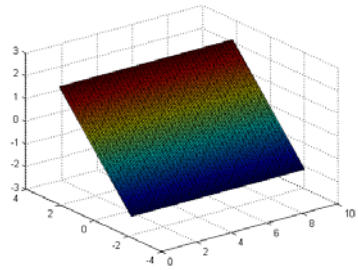
To create a slice plane that does not lie in an axes plane, first define a surface and rotate it to the desired orientation. This example uses a surface that has the same *x*- and *y*-coordinates as the volume.

```
hslice = surf(linspace(xmin,xmax,100),...
 linspace(ymin,ymax,100),...
 zeros(100));
```

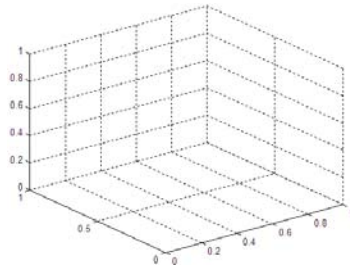


Rotate the surface by -45 degrees about the *x*-axis and save the surface `XData`, `YData`, and `ZData` to define the slice plane; then delete the surface.

```
rotate(hslice,[-1,0,0],-45)
xd = get(hslice,'XData');
yd = get(hslice,'YData');
zd = get(hslice,'ZData');
```



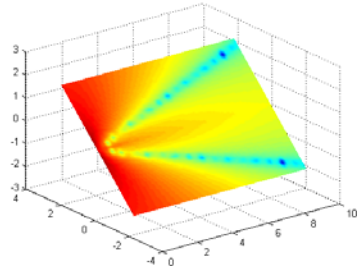
```
delete(hslice)
```



3. Draw the Slice Planes

Draw the rotated slice plane, setting the `FaceColor` to `interp` so that it is colored by the figure colormap, and set the `EdgeColor` to `none`. Increase the `DiffuseStrength` to `.8` to make this plane shine more brightly after adding a light source.

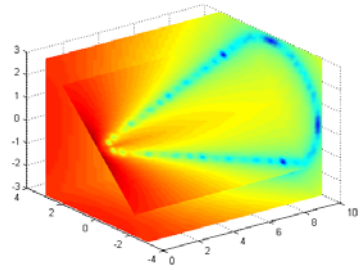
```
h = slice(x,y,z,v,xd,yd,zd);
set(h,'FaceColor','interp',...
 'EdgeColor','none',...
 'DiffuseStrength',.8)
```



Set `hold` to `on` and add three more orthogonal slice planes at `xmax`, `ymax`, and `zmin` to provide a context for the first plane, which slices through the volume at an angle.

```
hold on
hx = slice(x,y,z,v,xmax,[],[]);
set(hx,'FaceColor','interp','EdgeColor','none')

hy = slice(x,y,z,v,[],ymax,[]);
set(hy,'FaceColor','interp','EdgeColor','none')
```

```
hz = slice(x,y,z,v,[],[],zmin);
set(hz,'FaceColor','interp','EdgeColor','none')
```

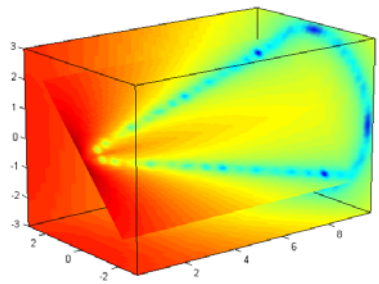


4. Define the View

To display the volume in correct proportions, set the data aspect ratio to `[1,1,1]` (`daspect`). Adjust the axis to fit tightly around the volume (`axis`) and turn on the `box` to provide a sense of a 3-D object. The orientation of the axes can be selected initially using `rotate3d` to determine the best `view`.

Zooming in on the scene provides a larger view of the volume (`camzoom`). Selecting a projection type of `perspective` gives the rectangular solid more natural proportions than the default orthographic projection (`camproj`).

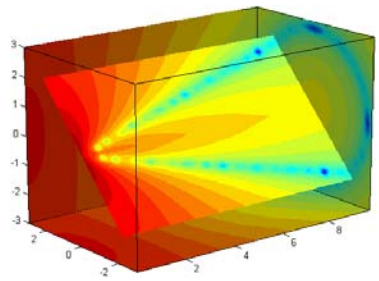
```
daspect([1,1,1])
axis tight
box on
view(-38.5,16)
camzoom(1.4)
camproj perspective
```



5. Add Lighting and Specify Colors

Adding a light to the scene makes the boundaries between the four slice planes more obvious because each plane forms a different angle with the light source (`lightangle`). Selecting a colormap with only 24 colors (the default is 64) creates visible gradations that help indicate the variation within the volume.

```
lightangle(-45,45)
colormap (jet(24))
set(gcf,'Renderer','zbuffer')
```



"Modify the Color Mapping" on page 6-16 shows how to modify how the data is mapped to color.

Modify the Color Mapping

The current colormap determines the coloring of the slice planes. This enables you to change the slice plane coloring by:

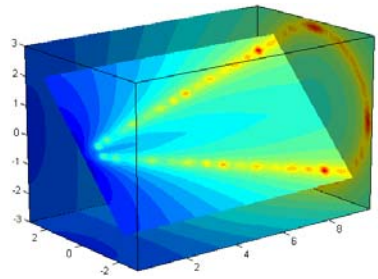
- Changing the colormap
- Changing the mapping of data value to color

Suppose, for example, you are interested in data values only between -5 and 2.5 and would like to use a colormap that mapped lower values to reds and higher values to blues (that is, the opposite of the default `jet` colormap).

1. Customize the Colormap

Flip the colormap using `colormap` and `flipud`:

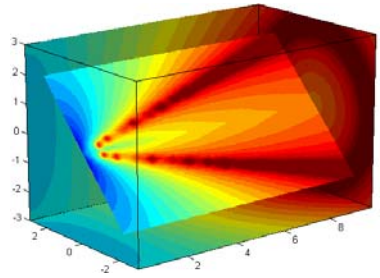
```
colormap (flipud(jet(24)))
```



2. Adjust the Color Limits

Adjust the color limits to emphasize any particular data range of interest. Adjust the color limits to range from -5 to 2.4832 to map any value lower than the value -5 (the original data ranged from -11.5417 to 2.4832) into the same color. (See `caxis` and Axis Color Limits - The CLim Property in Axes Properties in the MATLAB documentation for an explanation of color mapping.)

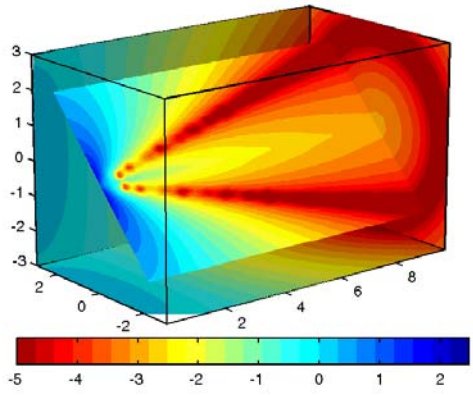
```
caxis([-5,2.4832])
```



3. Add a Color Bar

Add a color bar to provide a key for the data-to-color mapping.

```
colorbar('horiz')
```


3
2
1
0
-1
-2
-3
2
0
-2
2
4
6
8
-5
-4
-3
-2
-1
0
1
2

Connecting Equal Values with Isosurfaces

Isosurfaces in Fluid Flow Data

Create isosurfaces with the `isosurface` and `patch` commands.

This example creates isosurfaces in a volume generated by `flow`. Generate the volume data with the command

```
[x,y,z,v] = flow;
```

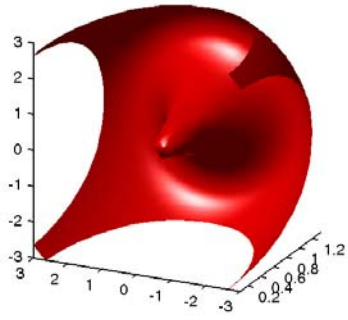
To select the isovalue, determine the range of values in the volume data.

```
min(v(:))
ans =
 -11.5417
max(v(:))
ans =
 2.4832
```

Through exploration, you can select isovalues that reveal useful information about the data. Once selected, use the isovalue to create the isosurface:

- Use `isosurface` to generate data that you can pass directly to `patch`.
- Recalculate the surface normals from the gradient of the volume data to produce better lighting characteristics (`isonormals`).
- Set the patch `FaceColor` to `red` and the `EdgeColor` to `none` to produce a smoothly lit surface.
- Adjust the view and add lighting (`daspect`, `view`, `camlight`, `lighting`).

```
hpatch = patch(isosurface(x,y,z,v,0));
isonormals(x,y,z,v,hpatch)
set(hpatch,'FaceColor','red','EdgeColor','none')
daspect([1,4,4])
view([-65,20])
axis tight
camlight left;
set(gcf,'Renderer','zbuffer'); lighting phong
```


3
2
1
0
-1
-2
-3
3
2
1
0
-1
-2
-3
0.2
0.4
0.6
0.8
1
1.2

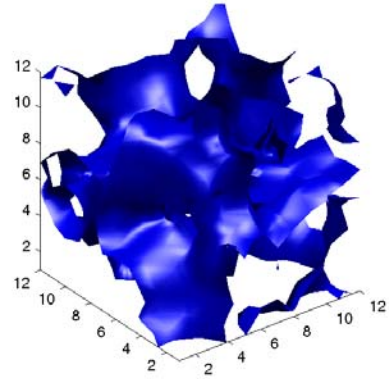
Isocaps Add Context to Visualizations

In this section...

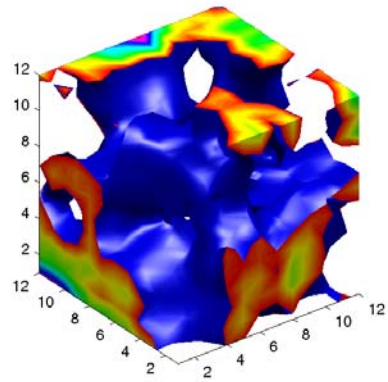
What Are Isocaps?

Isocaps are planes that are fitted to the limits of an isosurface to provide a visual context for the isosurface. Isocaps show a cross-sectional view of the interior of the isosurface for which the isocap provides an *end cap*.

The following two pictures illustrate the use of isocaps. The first is an isosurface without isocaps.



The second picture shows the effect of adding isocaps to the same isosurface.



Other Isocap Applications

Some additional applications of isocaps are shown in the following examples:

- Isocaps show the interior of a cut-away volume.
- Isocaps cap the end of a volume that would otherwise appear empty.
- Isocaps enhance the visibility of the isosurface limits.

Defining Isocaps

Isocaps, like isosurfaces, are created as `patch` graphics objects. Use the `isocaps` command to generate the data to pass to `patch`. For example:

```
patch(isocaps(voldata,isoval),...
 'FaceColor','interp',...
 'EdgeColor','none')
```

creates isocaps for the scalar volume data `voldata` at the value `isoval`. You should create the isosurface using the same volume data and isovalue to ensure that the edges of the isocaps *fit* the isosurface.

Setting the patch `FaceColor` property to `interp` results in a coloring that maps the data values spanned by the isocap to colormap entries. You can also set other patch properties to control the effects of lighting and coloring on the isocaps.

Adding Isocaps to an Isosurface

This example illustrates how to set coloring and lighting characteristics when working with isocaps. There are five basic steps:

1. Generate and process your volume data.
2. Create the isosurface and isocaps and set patch properties to control the coloring and lighting.
3. Create the isocaps and set properties.
4. Specify the view.
5. Add lights to the scene.

1. Prepare the Data

This example uses a 3-D array of random (`rand`) data to define the volume data. The data is then smoothed (`smooth3`).

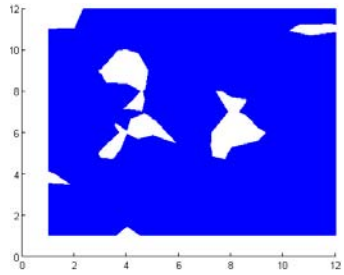
```
data = rand(12,12,12);
data = smooth3(data,'box',5);
```

2. Create the Isosurface and Set Properties

Use `isosurface` and `patch` to create the isosurface and set coloring and lighting properties. Reduce the `AmbientStrength`, `SpecularStrength`, and `DiffuseStrength` of the reflected light to compensate for the brightness of the two light sources used to provide more uniform lighting.

Recalculate the vertex normals of the isosurface to produce smoother lighting (`isonormals`).

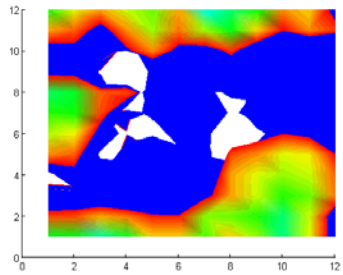
```
isoval = .5;
h = patch(isosurface(data,isoval),...
 'FaceColor','blue',...
 'EdgeColor','none',...
 'AmbientStrength',.2,...
 'SpecularStrength',.7,...
 'DiffuseStrength',.4);
isonormals(data,h)
```



3. Create the Isocaps and Set Properties

Define the `isocaps` using the same data and isovalue as the isosurface. Specify interpolated coloring and select a colormap that provides better contrasting colors with the blue isosurface than those in the default colormap (`colormap`).

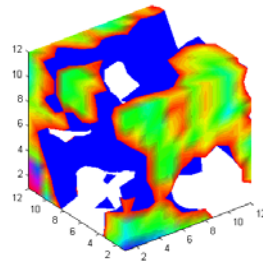
```
patch(isocaps(data,isoval),...
 'FaceColor','interp',...
 'EdgeColor','none')
colormap hsv
```



4. Define the View

Set the data aspect ratio to `[1,1,1]` so that the display is in correct proportions (`daspect`). Eliminate white space within the axes and set the view to 3-D (`axis tight`, `view`).

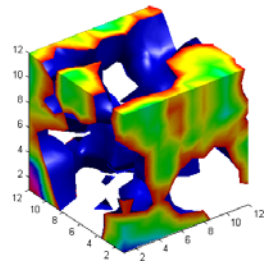
```
daspect([1,1,1])
axis tight
view(3)
```



5. Add Lighting

To add fairly uniform lighting, but still take advantage of the ability of light sources to make visible subtle variations in shape, this example uses two lights, one to the left and one to the right of the camera (`camlight`). Use Phong lighting to produce the smoothest variation of color (`lighting`). Phong lighting requires the `zbuffer` renderer.

```
camlight right
camlight left
set(gcf,'Renderer','zbuffer');
lighting phong
```



Visualizing Vector Volume Data

In this section...

Lines, Particles, Ribbons, Streams, Tubes, and Cones

Vector volume data contains more information than scalar data because each coordinate point in the data set has three values associated with it. These values define a vector that represents both a magnitude and a direction. The velocity of fluid flow is an example of vector data.

A number of techniques are useful for visualizing vector data:

- Stream lines trace the path that a massless particle immersed in the vector field would follow.
- Stream particles are markers that trace stream lines and are useful for creating stream line animations.
- Stream ribbons are similar to stream lines, except that the width of the ribbons enables them to indicate twist. Stream ribbons are useful to indicate curl angular velocity.
- Stream tubes are similar to stream lines, but you can also control the width of the tube. Stream tubes are useful for displaying the divergence of a vector field.
- Cone plots represent the magnitude and direction of the data at each point by displaying a conical arrowhead or an arrow.

It is typically the case that these functions best elucidate the data when used in conjunction with other visualization techniques, such as contours, slice planes, and isosurfaces. The examples in this section illustrate some of these techniques.

Using Scalar Techniques with Vector Data

Visualization techniques such as contour slices, slice planes, and isosurfaces require scalar volume data. You can use these techniques with vector data by taking the magnitude of the vectors. For example, the `wind` data set returns three coordinate arrays and three vector component arrays, `u`, `v`, `w`. In this case, the magnitude of the velocity vectors equals the wind speed at each corresponding coordinate point in the volume.

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

The array `wind_speed` contains scalar values for the volume data. The usefulness of the information produced by this approach, however, depends on what physical phenomenon is represented by the magnitude of your vector data.

Specifying Starting Points for Stream Plots

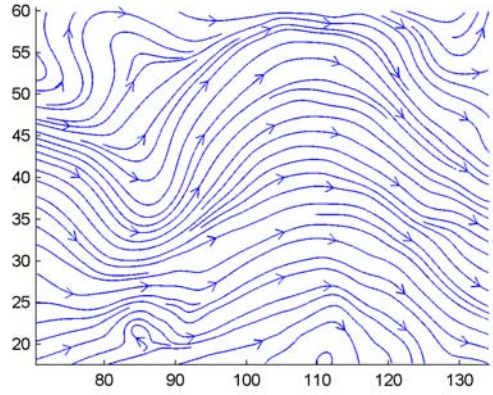
Stream plots (stream lines, ribbons, tubes, and cones or arrows) illustrate the flow of a 3-D vector field. The MATLAB stream-plotting functions (`streamline`, `streamribbon`, `streamtube`, `coneplot`, `stream2`, `stream3`) all require you to specify the point at which you want to begin each stream trace.

Determining the Starting Points

Generally, knowledge of your data's characteristics helps you select the starting points. Information such as the primary direction of flow and the range of the data coordinates helps you decide where to evaluate the data.

The `streamslice` function is useful for exploring your data. For example, these statements draw a slice through the vector field at a *z* value midway in the range.

```
load wind
zmax = max(z(:)); zmin = min(z(:));
streamslice(x,y,z,u,v,w,[],[],(zmax-zmin)/2)
```



This stream slice plot indicates that the flow is in the positive *x*-direction and also enables you to select starting points in both *x* and *y*. You could create similar plots that slice the volume in the *x*-*z* plane or the *y*-*z* plane to gain further insight into your data's range and orientation.

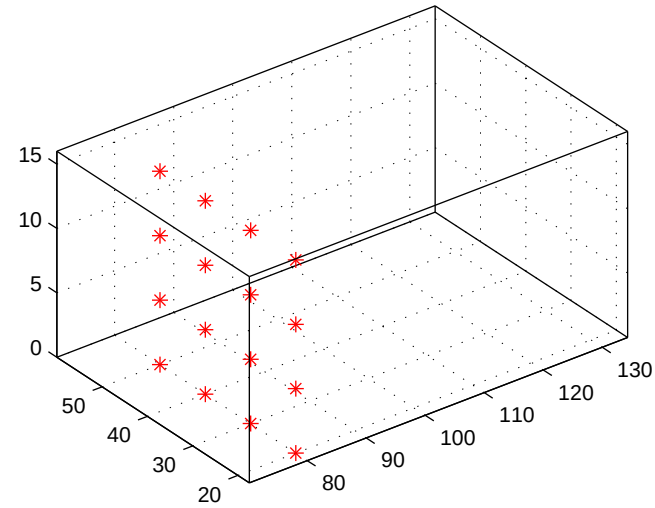
Specifying Arrays of Starting-Point Coordinates

To specify the starting point for one stream line, you need the *x*-, *y*-, and *z*-coordinates of the point. The `meshgrid` command provides a convenient way to create arrays of starting points. For example, you could select the following starting points from the wind data displayed in the previous stream slice.

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
```

This statement defines the starting points as all lying on x = 80, y ranging from 20 to 50, and z ranging from 0 to 15. You can use `plot3` to display the locations.

```
plot3(sx(:),sy(:),sz(:),'*r');
axis(volumebounds(x,y,z,u,v,w))
grid; box; daspect([2 2 1])
```



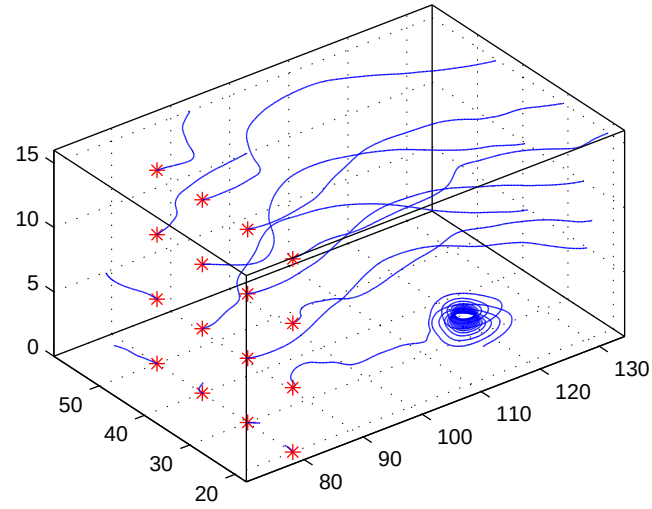
You do not need to use 3-D arrays, such as those returned by `meshgrid`, but the size of each array must be the same, and `meshgrid` provides a convenient way to generate arrays when you do not have an equal number of unique values in each coordinate. You can also define starting-point arrays as column vectors. For example, `meshgrid` returns 3-D arrays:

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
whos
Name      Size          Bytes  Class
sx        4x1x4           128  double array
sy        4x1x4           128  double array
sz        4x1x4           128  double array
```

In addition, you could use 16-by-1 column vectors with the corresponding elements of the three arrays composing the coordinates of each starting point. (This is the equivalent of indexing the values returned by `meshgrid` as `sx(:)`, `sy(:)`, and `sz(:)`.)

For example, adding the stream lines produces:

```
streamline(x,y,z,u,v,w,sx(:),sy(:),sz(:))
```



Accessing Subregions of Volume Data

The `subvolume` function provides a simple way to access subregions of a volume data set. `subvolume` enables you to select regions of interest based on limits rather than using the colon operator to index into the 3-D arrays that define volumes. Consider the following two approaches to creating the data for a subvolume — indexing with the colon operator and using `subvolume`.

Indexing with the Colon Operator

When you index the arrays, you work with values that specify the elements in each dimension of the array.

```
load wind
xsub = x(1:10,20:30,1:7);
ysub = y(1:10,20:30,1:7);
zsub = z(1:10,20:30,1:7);
usub = u(1:10,20:30,1:7);
vsub = v(1:10,20:30,1:7);
wsub = w(1:10,20:30,1:7);
```

Using the subvolume Function

`subvolume` enables you to use coordinate values that you can read from the axes. For example:

```
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[xsub,ysub,zsub,usub,vsub,wsub] = subvolume(x,y,z,u,v,w,lims);
```

You can then use the subvolume data as inputs to any function requiring vector volume data.

Stream Line Plots of Vector Data

In this section...

Wind Mapping Data

The MATLAB vector data set called `wind` represents air currents over North America. This example uses a combination of techniques:

- Stream lines to trace the wind velocity
- Slice planes to show cross-sectional views of the data
- Contours on the slice planes to improve the visibility of slice-plane coloring

1. Determine the Range of the Coordinates

Load the data and determine minimum and maximum values to locate the slice planes and contour plots (`load`, `min`, `max`).

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymax = max(y(:));
zmin = min(z(:));
```

2. Add Slice Planes for Visual Context

Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command. Create slice planes along the *x*-axis at `xmin`, `100`, and `xmax`, along the *y*-axis at `ymax`, and along the *z*-axis at `zmin`. Specify interpolated face coloring so the slice coloring indicates wind speed, and do not draw edges (`sqrt`, `slice`, `FaceColor`, `EdgeColor`).

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hsurfaces = slice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
```

3. Add Contour Lines to the Slice Planes

Draw light gray contour lines on the slice planes to help quantify the color mapping (`contourslice`, `EdgeColor`, `LineWidth`).

```
hcont = ...
contourslice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hcont,'EdgeColor',[.7,.7,.7],'LineWidth',.5)
```

4. Define the Starting Points for Stream Lines

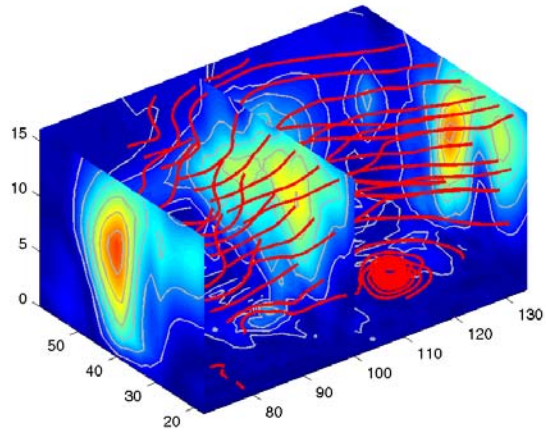
In this example, all stream lines start at an *x*-axis value of 80 and span the range 20 to 50 in the *y*-direction and 0 to 15 in the *z*-direction. Save the handles of the stream lines and set the line width and color (`meshgrid`, `streamline`, `LineWidth`, `Color`).

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
hlines = streamline(x,y,z,u,v,w,sx,sy,sz);
set(hlines,'LineWidth',2,'Color','r')
```

5. Define the View

Set up the view, expanding the *z*-axis to make it easier to read the graph (`view`, `daspect`, `axis`).

```
view(3)
daspect([2,2,1])
axis tight
```



See `coneplot` for an example of the same data plotted with cones.

Displaying Curl with Stream Ribbons

In this section...

What Stream Ribbons Can Show

Stream ribbons illustrate direction of flow, similar to stream lines, but can also show rotation about the flow axis by twisting the ribbon-shaped flow line. The `streamribbon` function enables you to specify a twist angle (in radians) for each vertex in the stream ribbons.

When used in conjunction with the `curl` function, `streamribbon` is useful for displaying the curl angular velocity of a vector field. The following example illustrates this technique.

1. Select a Subset of Data to Plot

Load and select a region of interest in the `wind` data set using `subvolume`. Plotting the full data set first can help you select a region of interest.

```
load wind
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[x,y,z,u,v,w] = subvolume(x,y,z,u,v,w,lims);
```

2. Calculate Curl Angular Velocity and Wind Speed

Calculate the curl angular velocity and the wind speed.

```
cav = curl(x,y,z,u,v,w);
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

3. Create the Stream Ribbons

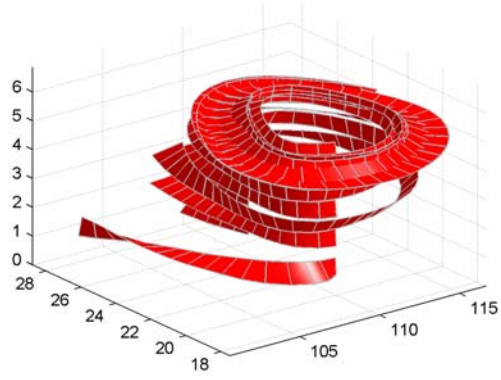
- Use `meshgrid` to create arrays of starting points for the stream ribbons. See "Specifying Starting Points for Stream Plots" on page 6-27for information on specifying the arrays of starting points.
- `stream3` calculates the stream line vertices with a step size of `.5`.
- `streamribbon` scales the width of the ribbon by a factor of `2` to enhance the visibility of the twisting (which indicates curl angular velocity).
- `streamribbon` returns the handles of the surface objects it creates, which are then used to set the color to red (`FaceColor`), the color of the surface edges to light gray (`EdgeColor`), and slightly increase the brightness of the ambient light reflected when lighting is applied (`AmbientStrength`).

```
[sx sy sz] = meshgrid(110,20:5:30,1:5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz,.5);
h = streamribbon(verts,x,y,z,cav,wind_speed,2);
set(h,'FaceColor','r',...
 'EdgeColor',[.7 .7 .7],...
 'AmbientStrength',.6)
```

4. Define the View and Add Lighting

- The `volumebounds` command provides a convenient way to set `axis` and color limits.
- Add a `grid` and set the `view` for 3-D (`streamribbon` does not change the current view).
- `camlight` creates a light positioned to the right of the viewpoint and `lighting` sets the lighting method to Phong (which requires the Z-buffer renderer).

```
axis(volumebounds(x,y,z,wind_speed))
grid on
view(3)
camlight right;
set(gcf,'Renderer','zbuffer'); lighting phong
```


6
5
4
3
2
1
0
28
26
24
22
20
18
105
110
115

Displaying Divergence with Stream Tubes

In this section...

What Stream Tubes Can Show

Stream tubes are similar to stream lines, except the tubes have width, providing another dimension that you can use to represent information.

By default, MATLAB graphics display the divergence of the vector field by the width of the tube. You can also define widths for each tube vertex and thereby map other data to width.

This example uses the following techniques:

- Stream tubes to indicate flow direction and divergence of the vector field in the `wind` data set
- Slice planes colored to indicate the speed of the wind currents overlaid with contour line to enhance visibility

Inputs include the coordinates of the volume, vector field components, and starting locations for the stream tubes.

1. Load Data and Calculate Required Values

Load the data and calculate values needed to make the plots. These values include:

- The location of the slice planes (maximum `x`, minimum `y`, and a value for the altitude)

- The minimum `x` value for the start of the stream tubes
- The speed of the wind (magnitude of the vector field)

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
alt = 7.356; % z value for slice and streamtube plane
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

2. Draw the Slice Planes

Draw the slice planes (`slice`) and set `surface` properties to create a smoothly colored slice. Use 16 colors from the `hsv colormap`.

```
hslice = slice(x,y,z,wind_speed,xmax,ymin,alt);
set(hslice,'FaceColor','interp','EdgeColor','none')
colormap hsv(16)
```

3. Add Contour Lines to Slice Planes

Add contour lines (`contourslice`) to the slice planes. Adjust the contour interval so the lines match the color boundaries in the slice planes:

- Call `caxis` to get the current color limits.
- Set the interpolation method used by `contourslice` to `linear` to match the default used by `slice`.

```
color_lim = caxis;
cont_intervals = linspace(color_lim(1),color_lim(2),17);
hcont = contourslice(x,y,z,wind_speed,xmax,ymin,...
 alt,cont_intervals,'linear');
set(hcont,'EdgeColor',[.4 .4 .4],'LineWidth',1)
```

4. Create the Stream Tubes

Use `meshgrid` to create arrays for the starting points for the stream tubes, which begin at the minimum `x` value, range from 20 to 50 in `y`, and lie in a single plane in `z` (corresponding to one of the slice planes).

The stream tubes (`streamtube`) are drawn at the specified locations and scaled to be 1.25 times the default width to emphasize the variation in divergence (width). The second element in the vector [1.25 30] specifies the number of points along the circumference of the tube (the default is 20). You might want to increase this value as the tube size increases, to maintain a smooth-looking tube.

Set the data aspect ratio (`daspect`) before calling `streamtube`.

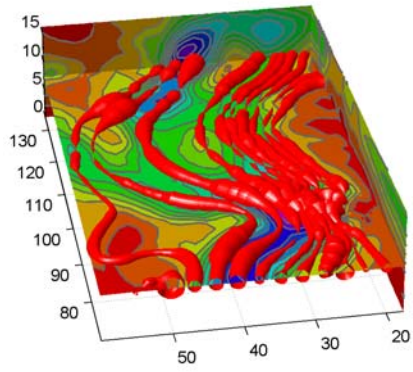
Stream tubes are surface objects, therefore you can control their appearance by setting surface properties. This example sets surface properties to give a brightly lit, red surface.

```
[sx,sy,sz] = meshgrid(xmin,20:3:50,alt);
daspect([1,1,1]) % set DAR before calling streamtube
htubes = streamtube(x,y,z,u,v,w,sx,sy,sz,[1.25 30]);
set(htubes,'EdgeColor','none','FaceColor','r',...
 'AmbientStrength',.5)
```

5. Define the View

Define the view and add lighting (`view`, `axis volumebounds`, `Projection`, `camlight`).

```
view(-100,30)
axis(volumebounds(x,y,z,wind_speed))
set(gca,'Projection','perspective')
camlight left
```


15
10
5
0
130
120
110
100
90
80
50
40
30
20

Creating Stream Particle Animations

In this section...

What Particle Animations Can Show

A stream particle animation is useful for visualizing the flow direction and speed of a vector field. The "particles" (represented by any of the line markers) trace the flow along a particular stream line. The speed of each particle in the animation is proportional to the magnitude of the vector field at any given point along the stream line.

1. Specify Starting Points of the Data Range

This example determines the region of the volume to plot by specifying the appropriate starting points. In this case, the stream plots begin at x = 100 and y spans 20 to 50 in the z = 5 plane, which is not the full volume bounds.

```
load wind
[sx sy sz] = meshgrid(100,20:2:50,5);
```

2. Create Stream Lines to Indicate Particle Paths

This example uses stream lines (`stream3`, `streamline`) to trace the path of the animated particles, which adds a visual context for the animation.

```
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
```

3. Define the View

While all the stream lines start in the z = 5 plane, the values of some spiral down to lower values. The following settings provide a clear view of the animation:

- The viewpoint (`view`) selected shows both the plane containing most stream lines and the spiral.
- Selecting a data aspect ratio (`daspect`) of `[2 2 0.125]` provides greater resolution in the *z*-direction to make the stream particles more easily visible in the spiral.
- Set the axes limits to match the data limits (`axis`) and draw the axis box (`box`).

```
view(-10.5,18)
daspect([2 2 0.125])
axis tight; box on
```

4. Calculate the Stream Particle Vertices

Determine the vertices along the stream line where a particle should be drawn. The `interpstreamspeed` function returns this data based on the stream line vertices and the speed of the vector data. This example scales the velocities by 0.05 to increase the number of interpolated vertices.

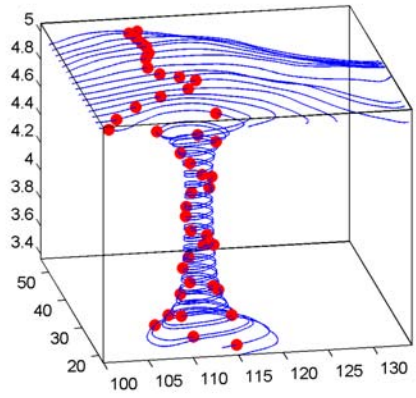
Set the axes `DrawMode` property to `fast` so the animation will run faster.

The `streamparticles` function sets the following properties:

- `Animate` to `10` to run the animation 10 times.
- `ParticleAlignment` to `on` to start all particle traces together.
- `MarkerEdgeColor` to `none` to draw only the face of the circular marker. Animations usually run faster when marker edges are not drawn.
- `MarkerFaceColor` to `red`.
- `Marker` to `o`, which draws a circular marker. You can use other line markers as well.

```
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.05);
set(gca,'drawmode','fast');
streamparticles(iverts,15,...
 'Animate',10,...
 'ParticleAlignment','on',...
 'MarkerEdgeColor','none',...
 'MarkerFaceColor','red',...
```

```
'Marker','o');
```



Vector Field Displayed with Cone Plots

In this section...

What Cone Plots Can Show

This example plots the velocity vector cones for the `wind` data. The graph produced employs a number of visualization techniques:

- An isosurface is used to provide visual context for the cone plots and to provide means to select a specific data value for a set of cones.
- Lighting enables the shape of the isosurface to be clearly visible.
- The use of perspective projection, camera positioning, and view angle adjustments composes the final view.

1. Create an Isosurface

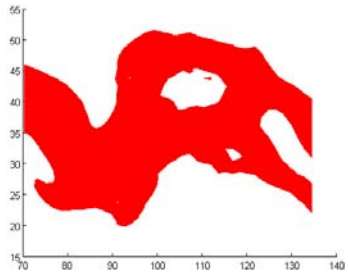
Displaying an isosurface within the rectangular space of the data provides a visual context for the cone plot. Creating the isosurface requires a number of steps:

1. Calculate the magnitude of the vector field, which represents the speed of the wind.
2. Use `isosurface` and `patch` to draw an isosurface illustrating where in the rectangular space the wind speed is equal to a particular value. Regions inside the isosurface have higher wind speeds, regions outside the isosurface have lower wind speeds.

3 Use `isonormals` to compute vertex normals of the isosurface from the volume data rather than calculate the normals from the triangles used to render the isosurface. These normals generally produce more accurate results.

4 Set visual properties of the isosurface, making it red and without drawing edges (`FaceColor`, `EdgeColor`).

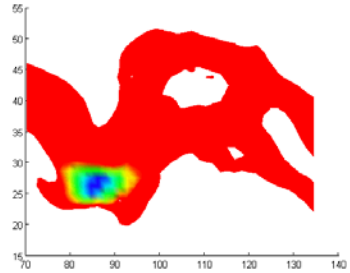
```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hiso = patch(isosurface(x,y,z,wind_speed,40));
isonormals(x,y,z,wind_speed,hiso)
set(hiso,'FaceColor','red','EdgeColor','none');
```



2. Add Isocaps to the Isosurface

Isocaps are similar to slice planes in that they show a cross section of the volume. They are designed to be the end caps of isosurfaces. Using interpolated face color on an isocap causes a mapping of data value to color in the current colormap. To create isocaps for the isosurface, define them at the same isovalue (`isocaps`, `patch`, `colormap`).

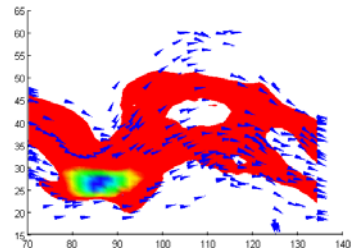
```
hcap = patch(isocaps(x,y,z,wind_speed,40),...
    'FaceColor','interp',...
    'EdgeColor','none');
colormap hsv
```



3. Create the First Set of Cones

- Use `daspect` to set the data aspect ratio of the axes before calling `coneplot` so function can determine the proper size of the cones.
- Determine the points at which to place cones by calculating another isosurface that has a smaller isovalue (so the cones are displayed outside the first isosurface) and use `reducepatch` to reduce the number of faces and vertices (so there are not too many cones on the graph).
- Draw the cones and set the face color to `blue` and the edge color to `none`.

```
daspect([1,1,1]);
[f verts] = reducepatch(isosurface(x,y,z,wind_speed,30),0.07);
h1 = coneplot(x,y,z,u,v,w,verts(:,1),verts(:,2),verts(:,3),3);
set(h1,'FaceColor','blue','EdgeColor','none');
```

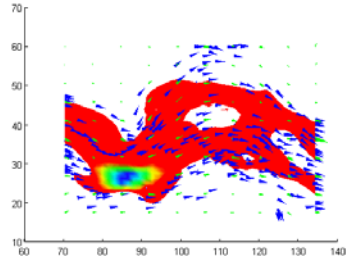


4. Create Second Set of Cones

1 Create a second set of points at values that span the data range (`linspace`, `meshgrid`).

2 Draw a second set of cones and set the face color to green and the edge color to none.

```
xrange = linspace(min(x(:)),max(x(:)),10);
yrange = linspace(min(y(:)),max(y(:)),10);
zrange = 3:4:15;
[cx,cy,cz] = meshgrid(xrange,yrange,zrange);
h2 = coneplot(x,y,z,u,v,w,cx,cy,cz,2);
set(h2,'FaceColor','green','EdgeColor','none');
```

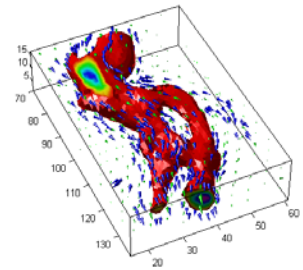


5. Define the View

1 Use the `axis` command to set the axis limits equal to the minimum and maximum values of the data and enclose the graph in a box to improve the sense of a volume (`box`).

2 Set the projection type to perspective to create a more natural view of the volume. Set the viewpoint and zoom in to make the scene larger (`camproj`, `camzoom`, `view`).

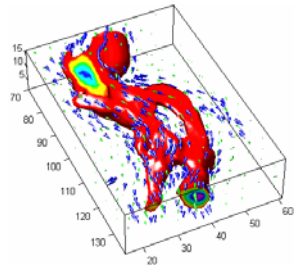
```
axis tight
box on
camproj perspective
camzoom(1.25)
view(65,45)
```



6. Add Lighting

Add a light source and use Phong lighting for the smoothest lighting of the isosurface (Phong lighting requires the Z-buffer renderer). Increase the strength of the background lighting on the isocaps to make them brighter (`camlight`, `lighting`, `AmbientStrength`).

```
camlight(-45,45)
set(gcf,'Renderer','zbuffer');
lighting phong
set(hcap,'AmbientStrength',.6)
```



Index

A

B

C

D

E

N

O

P

R

S

T

V

W

Z