MATLAB®

3-D Visualization

MATLAB®

R2025b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

MATLAB® 3-D Visualization

© COPYRIGHT 1984–2025 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 MATLAB 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online only	Revised for MATLAB 8.6 (Release 2015b)
March 2016	Online only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)
September 2021	Online only	Revised for MATLAB 9.11 (Release 2021b)
March 2022	Online only	Revised for MATLAB 9.12 (Release 2022a)
September 2022	Online only	Revised for MATLAB 9.13 (Release 2022b)
March 2023	Online only	Revised for MATLAB 9.14 (Release 2023a)
September 2023	Online only	Revised for Version 23.2 (R2023b)
March 2024	Online only	Revised for Version 24.1 (R2024a)
September 2024	Online only	Revised for Version 24.2 (R2024b)
March 2025	Online only	Revised for Version 25.1 (R2025a)
September 2025	Online only	Rereleased for Version 25.2 (R2025b)

Contents

1 Surface and Mesh Plots

Creating 3-D Plots 1-2

Changing Surface Properties 1-7

Creating the MATLAB Logo 1-14

Representing Data as a Surface 1-20
- Functions for Plotting Data Grids 1-20
- Functions for Gridding and Interpolating Data 1-20
- Mesh and Surface Plots 1-21
- Visualizing Functions of Two Variables 1-21
- Surface Plots of Nonuniformly Sampled Data 1-24
- Reshaping Data 1-26
- Parametric Surfaces 1-28

2 Polygons

Introduction to Patch Objects 2-2
- What Are Patch Objects? 2-2
- Behavior of the patch Function 2-2
- Creating a Single Polygon 2-3

Multifaceted Patches 2-6
- Example — Defining a Cube 2-6

3 Volume Visualization

Overview of Volume Visualization 3-2
- Examples of Volume Data 3-2
- Selecting Visualization Techniques 3-2
- Steps to Create a Volume Visualization 3-3
- Volume Visualization Functions 3-3

Techniques for Visualizing Scalar Volume Data 3-5
- What Is Scalar Volume Data? 3-5
- Ways to Display MRI Data 3-5

Exploring Volumes with Slice Planes **3-12**
Slicing Fluid Flow Data **3-12**
Modify the Color Mapping **3-15**

Connecting Equal Values with Isosurfaces **3-18**
Isosurfaces in Fluid Flow Data **3-18**

Isocaps Add Context to Visualizations **3-20**
What Are Isocaps? **3-20**
Other Isocap Applications **3-21**
Defining Isocaps **3-21**
Adding Isocaps to an Isosurface **3-22**

Visualizing Vector Volume Data **3-24**
Lines, Particles, Ribbons, Streams, Tubes, and Cones **3-24**
Using Scalar Techniques with Vector Data **3-24**
Specifying Starting Points for Stream Plots **3-24**
Accessing Subregions of Volume Data **3-27**

Display Streamlines Using Vector Data **3-29**
Wind Mapping Data **3-29**
1. Determine the Range of the Coordinates **3-29**
2. Add Slice Planes for Visual Context **3-29**
3. Add Contour Lines to the Slice Planes **3-29**
4. Define the Starting Points for Stream Lines **3-30**
5. Define the View **3-30**

Displaying Curl with Stream Ribbons **3-31**
What Stream Ribbons Can Show **3-31**
1. Select a Subset of Data to Plot **3-31**
2. Calculate Curl Angular Velocity and Wind Speed **3-31**
3. Create the Stream Ribbons **3-31**
4. Define the View and Add Lighting **3-32**

Displaying Divergence with Stream Tubes **3-33**
What Stream Tubes Can Show **3-33**
1. Load Data and Calculate Required Values **3-33**
2. Draw the Slice Planes **3-33**
3. Add Contour Lines to Slice Planes **3-34**
4. Create the Stream Tubes **3-34**
5. Define the View **3-34**

Create Stream Particle Animations **3-36**
Projectile Path Over Time **3-36**
What Particle Animations Can Show **3-37**
1. Specify Starting Points of the Data Range **3-37**
2. Create Stream Lines to Indicate Particle Paths **3-37**
3. Define the View **3-37**
4. Calculate the Stream Particle Vertices **3-38**

Vector Field Displayed with Cone Plots **3-40**
What Cone Plots Can Show **3-40**
1. Create an Isosurface **3-40**
2. Add Isocaps to the Isosurface **3-40**
3. Create the First Set of Cones **3-41**

4. Create Second Set of Cones **3-41**
5. Define the View **3-41**
6. Add Lighting **3-42**

Visualizing Volume Data **3-43**

Visualizing Four-Dimensional Data **3-50**

Displaying Complex Three-Dimensional Objects **3-55**

Displaying Topographic Data **3-62**

Surface and Mesh Plots

- “Creating 3-D Plots” on page 1-2
- “Changing Surface Properties” on page 1-7
- “Creating the MATLAB Logo” on page 1-14
- “Representing Data as a Surface” on page 1-20

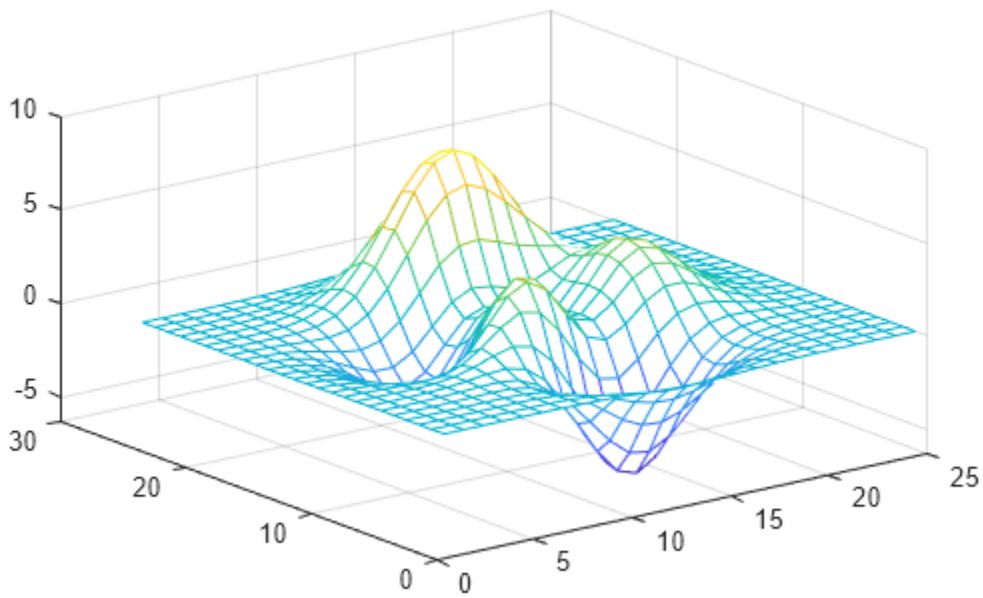
Creating 3-D Plots

This example shows how to create a variety of 3-D plots in MATLAB®.

Mesh Plot

The `mesh` function creates a wireframe mesh. By default, the color of the mesh is proportional to the surface height.

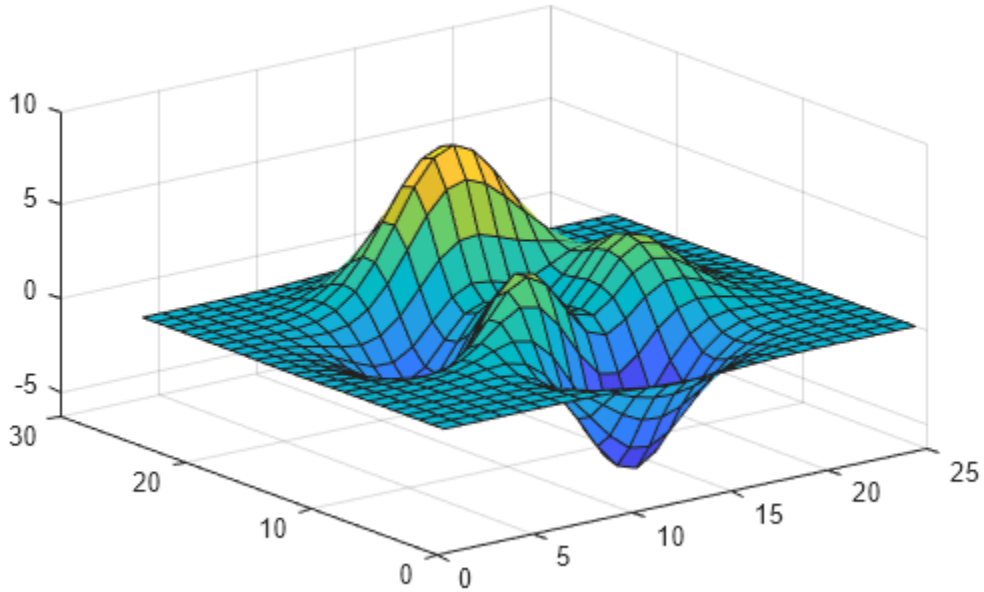
```
z = peaks(25);

figure
mesh(z)
```



Surface Plot

The `surf` function is used to create a 3-D surface plot.

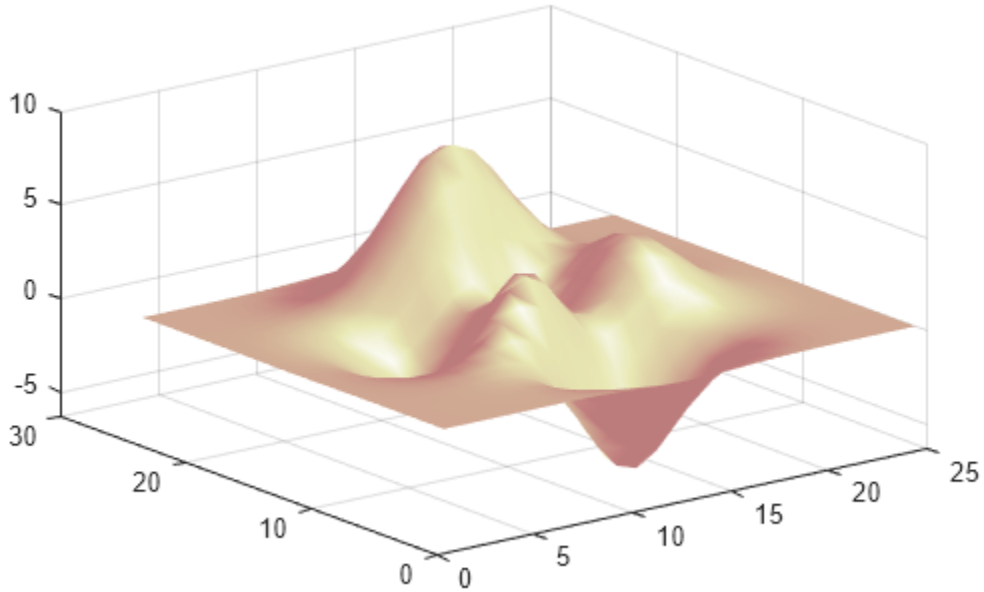
```
surf(z)
```



Surface Plot (with Shading)

The `surfl` function creates a surface plot with colormap-based lighting. For smoother color transitions, use a colormap with linear intensity variation such as `pink`.

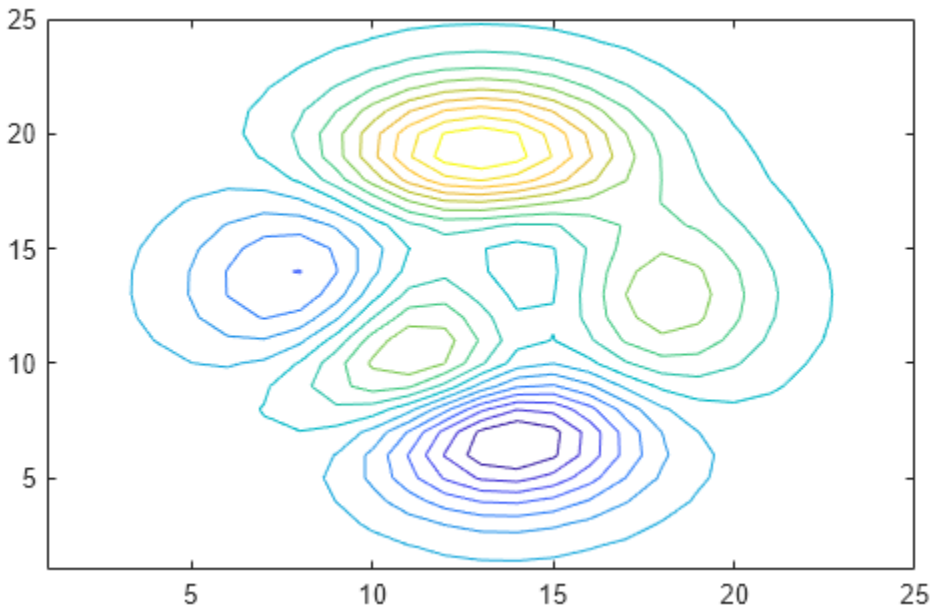
```
surfl(z)
colormap(pink)     % change color map
shading interp     % interpolate colors across lines and faces
```



Contour Plot

The `contour` function is used to create a plot with contour lines of constant value.

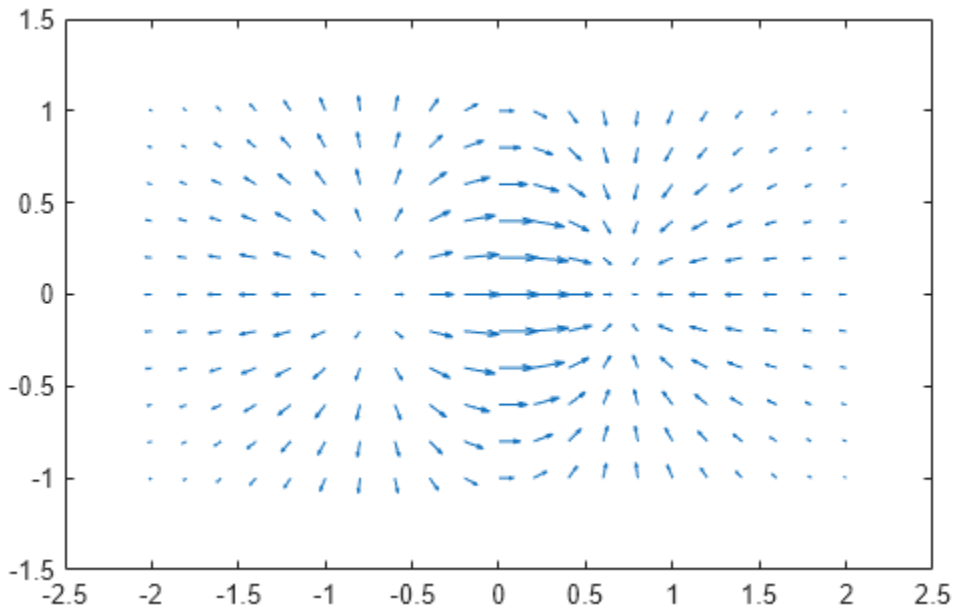
```
contour(z,16)
colormap default    % change color map
```



Quiver Plot

The `quiver` function plots 2-D vectors as arrows.

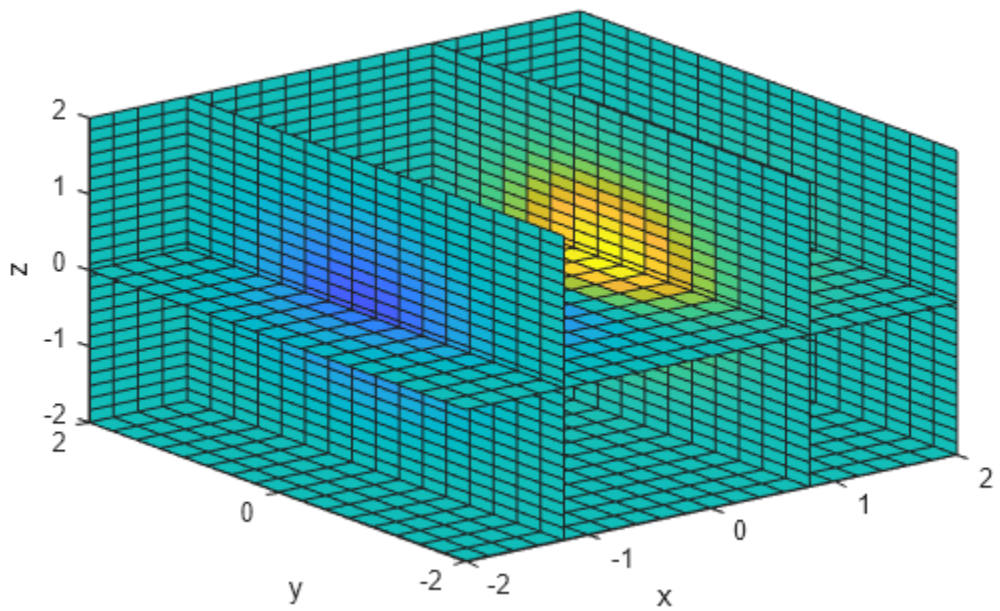
```
x = -2:.2:2;
y = -1:.2:1;

[xx,yy] = meshgrid(x,y);
zz = xx.*exp(-xx.^2-yy.^2);
[px,py] = gradient(zz,.2,.2);

quiver(x,y,px,py)
xlim([-2.5 2.5])    % set limits of x axis
```



Slices through 3-D Volumes

The `slice` function displays data at planes that slice through volumetric data.

```
x = -2:.2:2;
y = -2:.25:2;
z = -2:.16:2;

[x,y,z] = meshgrid(x,y,z);
v = x.*exp(-x.^2-y.^2-z.^2);

xslice = [-1.2,.8,2];    % location of y-z planes
yslice = 2;              % location of x-z plane
zslice = [-2,0];         % location of x-y planes

slice(x,y,z,v,xslice,yslice,zslice)
xlabel('x')
ylabel('y')
zlabel('z')
```


2
1
0
-1
-2
z
2
0
-2
y
-2
-1
0
1
2
x

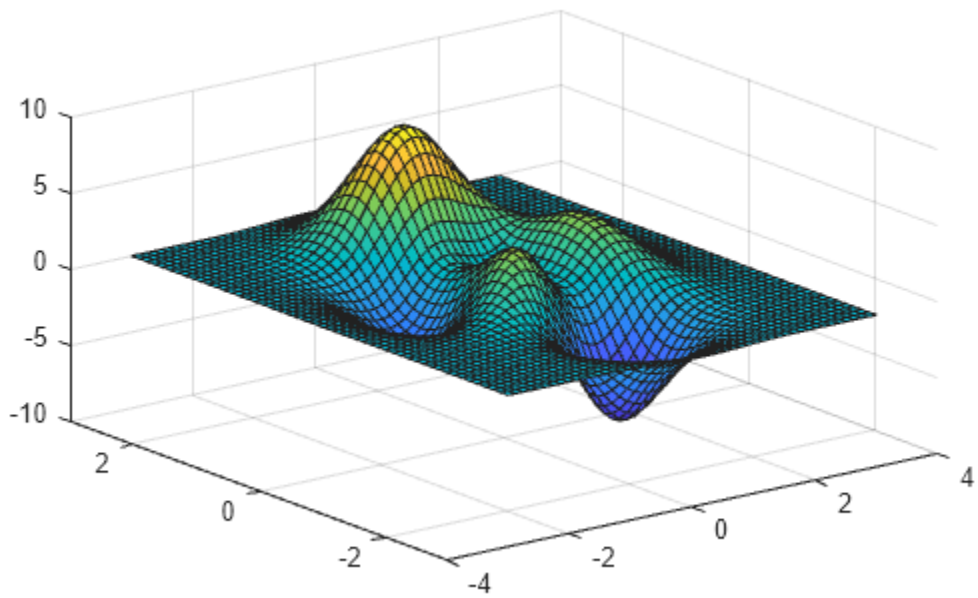
Changing Surface Properties

This example shows how to get properties of a surface plot in MATLAB® and change the property values to customize your plot.

Surface Objects

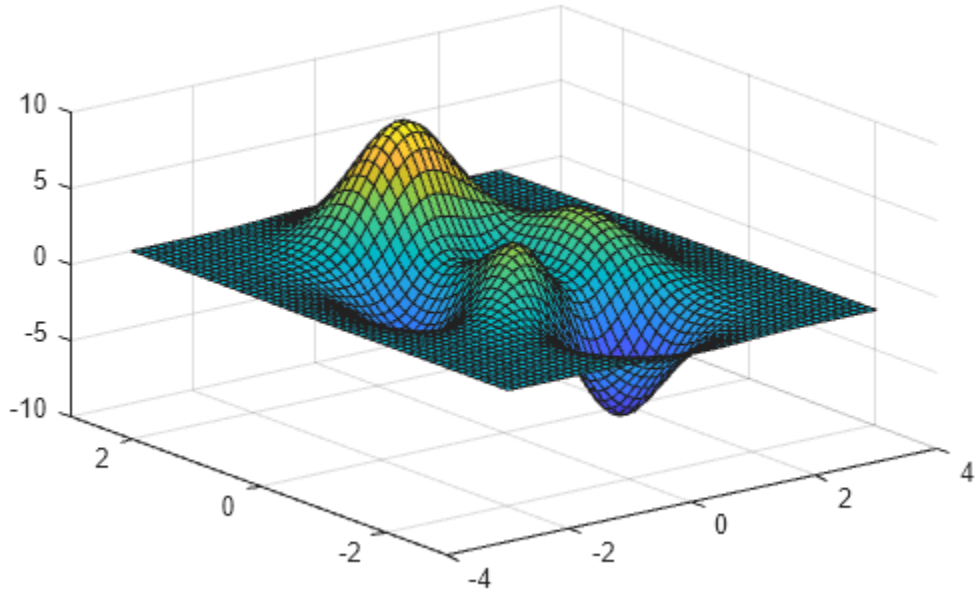
There are several ways to create a surface object in MATLAB. One way is to use `surf`.

```
[X,Y,Z] = peaks(50);

figure
surf(X,Y,Z)
```



Like all graphics objects, surfaces have properties that you can view and modify. These properties have default values. The display of the surface object, `s`, shows the most commonly used surface properties, such as `EdgeColor`, `LineStyle`, `FaceColor`, and `FaceLighting`.

```
s = surf(X,Y,Z)
```




```
s =
  Surface with properties:

        EdgeColor: [0.1294 0.1294 0.1294]
        LineStyle: '-'
        FaceColor: 'flat'
      FaceLighting: 'flat'
        FaceAlpha: 1
            XData: [50×50 double]
            YData: [50×50 double]
            ZData: [50×50 double]
            CData: [50×50 double]

  Show all properties
```

Get Individual Surface Properties

To access individual properties, use dot notation syntax `object.PropertyName`. For example, return the `FaceColor` property of the surface.

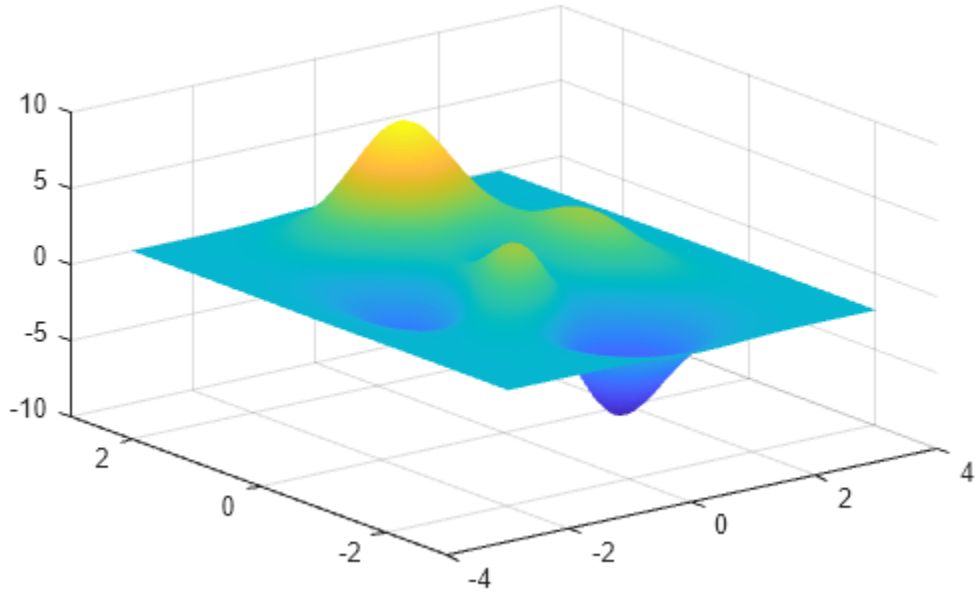
```
s.FaceColor
```

```
ans =
'flat'
```

Change Commonly Used Surface Properties

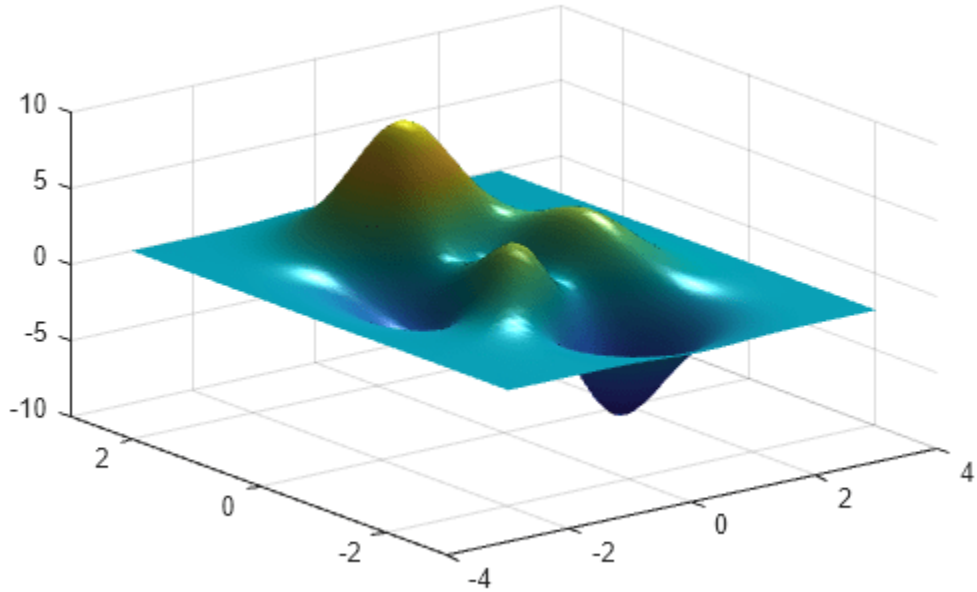
Several functions are available to change surface properties. For example, use the `shading` function to control the shading of your surface.

```
shading interp    % interpolate the colormap across the surface face
```



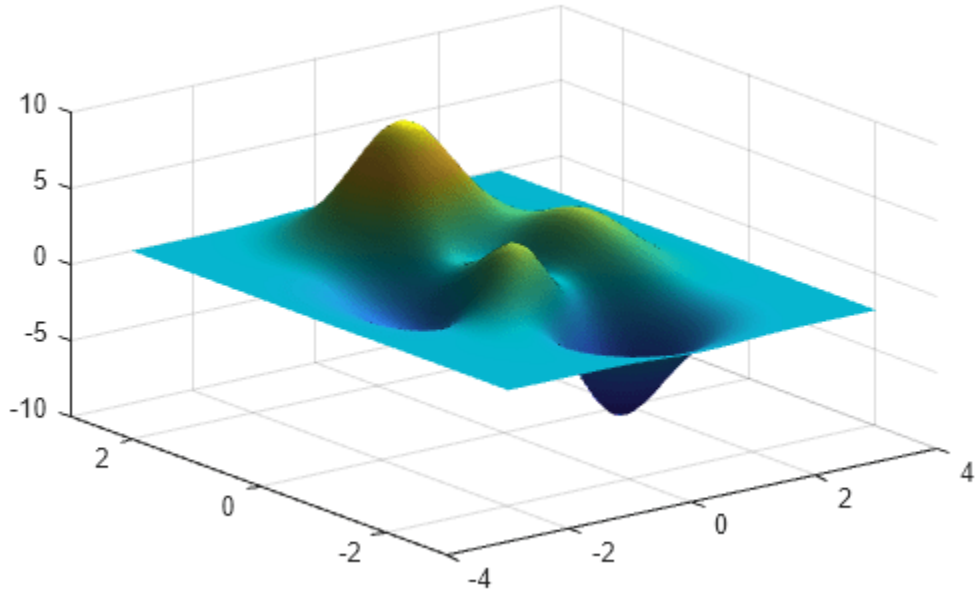
Use the `lighting` function to adjust the lighting characteristics of your surface. In order for `lighting` to have any affect, you must light your surface by creating a light object.

```
light                % create a light
lighting gouraud     % preferred method for lighting curved surfaces
```



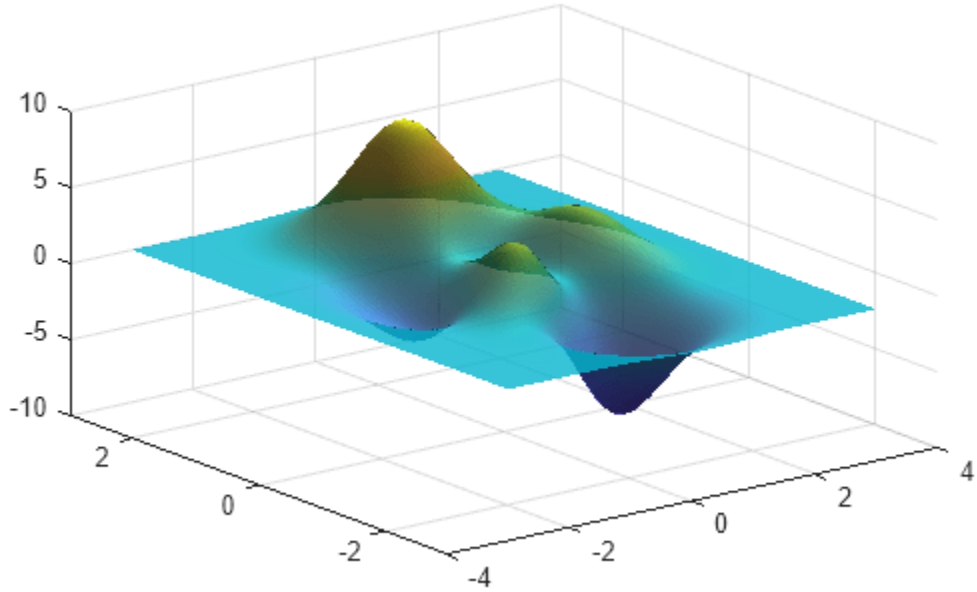
To change the reflectance property of your surface, use the `material` function.

```
material dull     % set material to be dull, no specular highlights
```



To set the transparency for all objects in the current axes, use the `alpha` function. This function sets the transparency to any value between 1 and 0, where 1 means fully opaque and 0 means completely transparent.

```
alpha(0.8)    % set transparency to 0.8
```

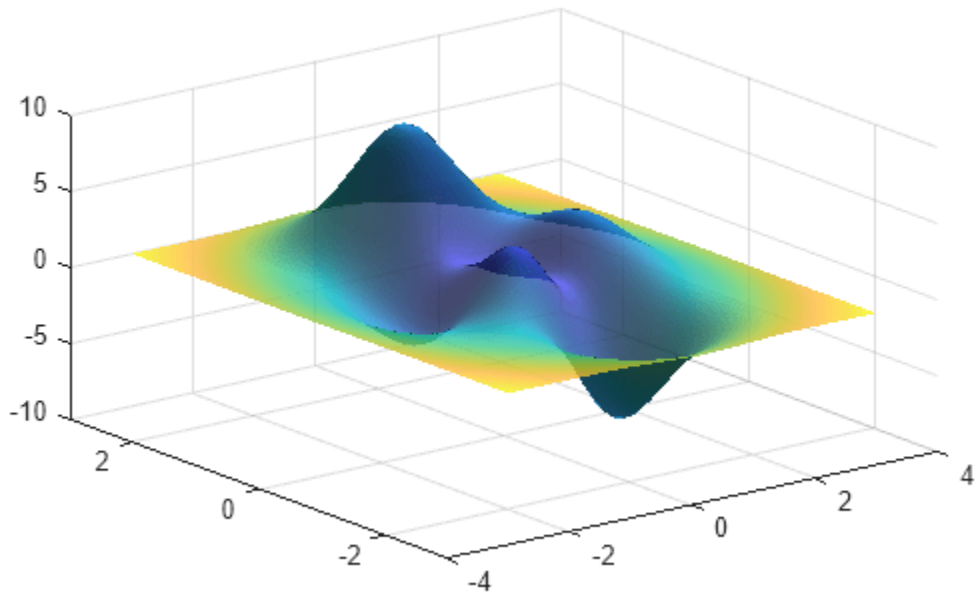


Change Other Surface Properties

To customize the look of your surface, change property values using dot notation.

`CData` defines the colors for the vertices of the surface. The `FaceColor` property indicates how the colors of the surface faces are determined from the vertex colors.

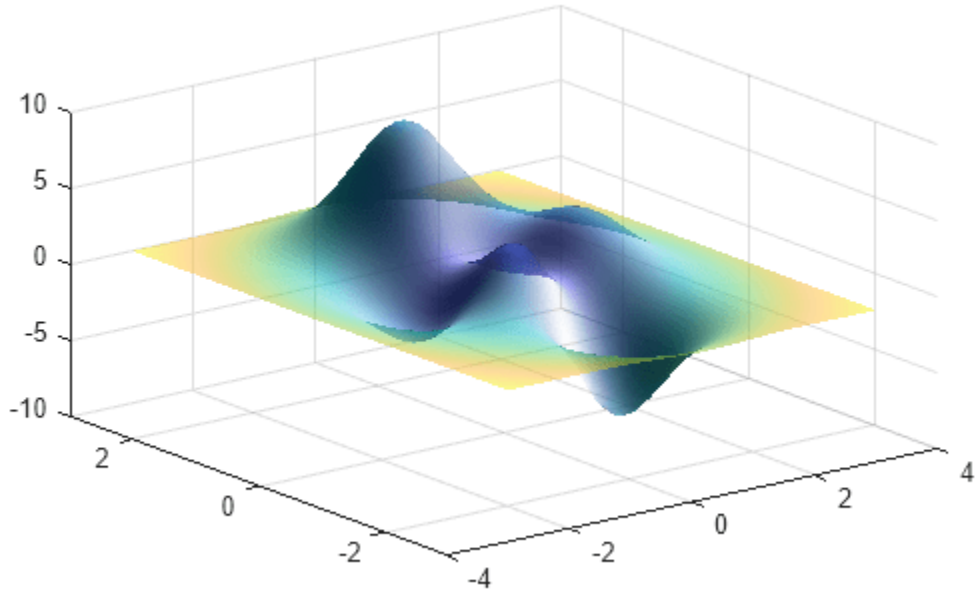
```
s.CData = hypot(X,Y);      % set color data
```




```
s.FaceColor = 'interp';    % interpolate to get face colors
```

`AlphaData` defines the transparency for each vertex of the surface. The `FaceAlpha` property indicates how the transparency of the surface faces are determined from vertex transparency.

```
s.AlphaData = gradient(Z);    % set vertex transparencies
s.FaceAlpha = 'interp';       % interpolate to get face transparencies
```



Get All Surface Properties

Graphics objects in MATLAB have many properties. To see all the properties of a surface, use the `get` command.

```
get(s)
```

```
          AlignVertexCenters: off
                   AlphaData: [50×50 double]
             AlphaDataMapping: 'scaled'
              AmbientStrength: 0.3000
                   Annotation: [1×1 matlab.graphics.eventdata.Annotation]
             BackFaceLighting: 'reverselit'
                 BeingDeleted: off
                   BusyAction: 'queue'
              ButtonDownFcn: ''
                       CData: [50×50 double]
                CDataMapping: 'scaled'
                   CDataMode: 'manual'
                 CDataSource: ''
                    Children: [0×0 GraphicsPlaceholder]
                    Clipping: on
                 ContextMenu: [0×0 GraphicsPlaceholder]
                   CreateFcn: ''
              DataTipTemplate: [1×1 matlab.graphics.datatip.DataTipTemplate]
                   DeleteFcn: ''
              DiffuseStrength: 0.8000
                 DisplayName: ''
                   EdgeAlpha: 1
                   EdgeColor: 'none'
               EdgeLighting: 'none'
                   FaceAlpha: 'interp'
                   FaceColor: 'interp'
```

```
             FaceLighting: 'gouraud'
              FaceNormals: [49×49×3 double]
          FaceNormalsMode: 'auto'
         HandleVisibility: 'on'
                  HitTest: on
            Interruptible: on
                LineStyle: '-'
                LineWidth: 0.5000
                   Marker: 'none'
          MarkerEdgeColor: 'auto'
          MarkerFaceColor: 'none'
               MarkerSize: 6
                MeshStyle: 'both'
                   Parent: [1×1 Axes]
            PickableParts: 'visible'
                 Selected: off
      SelectionHighlight: on
SpecularColorReflectance: 1
        SpecularExponent: 10
        SpecularStrength: 0
                     Tag: ''
                    Type: 'surface'
                UserData: []
            VertexNormals: [50×50×3 double]
        VertexNormalsMode: 'auto'
                 Visible: on
                   XData: [50×50 double]
                XDataMode: 'manual'
              XDataSource: ''
                   YData: [50×50 double]
                YDataMode: 'manual'
              YDataSource: ''
                   ZData: [50×50 double]
              ZDataSource: ''
```

Creating the MATLAB Logo

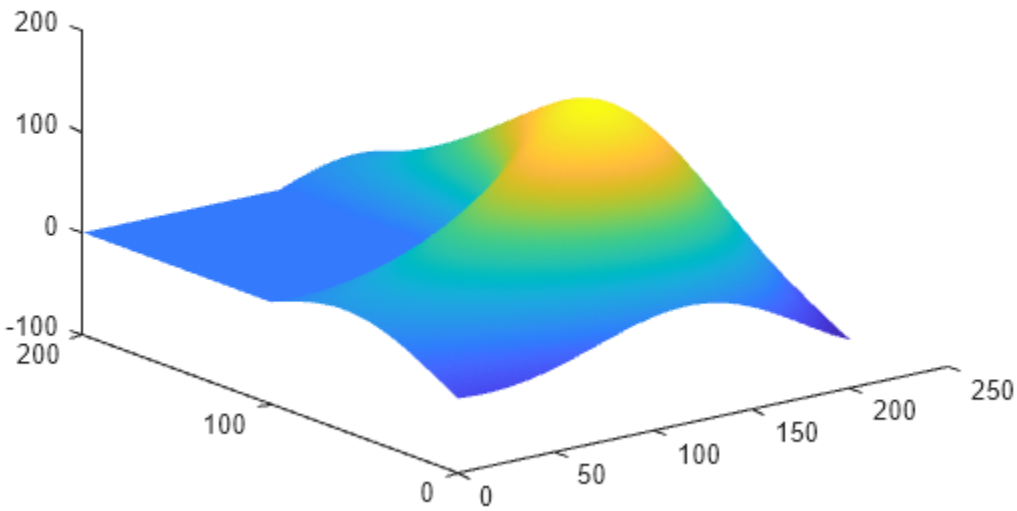
This example shows how to create and display the MATLAB® logo.

Use the `membrane` command to generate the surface data for the logo.

```
L = 160*membrane(1,100);
```

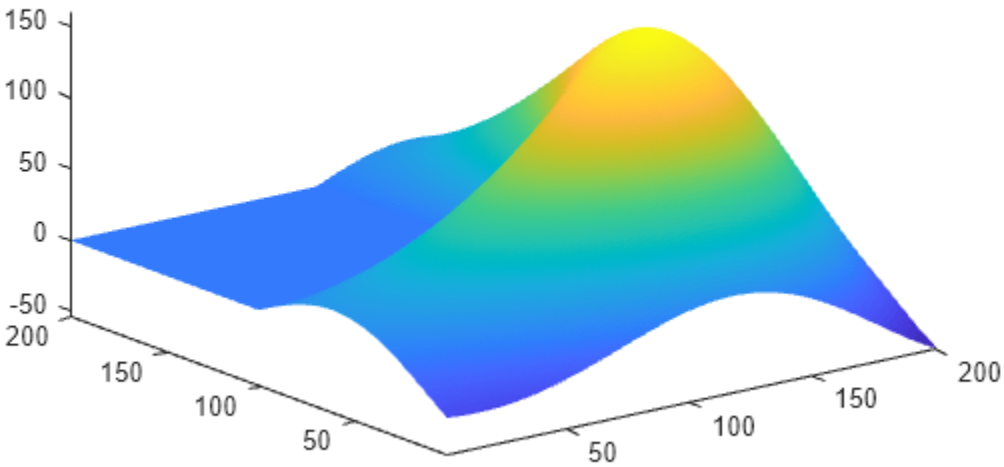
Create a figure and an axes to display the logo. Then, create a surface for the logo using the points from the `membrane` command. Turn off the lines in the surface.

```
f = figure;
ax = axes;

s = surface(L);
s.EdgeColor = 'none';
view(3)
```



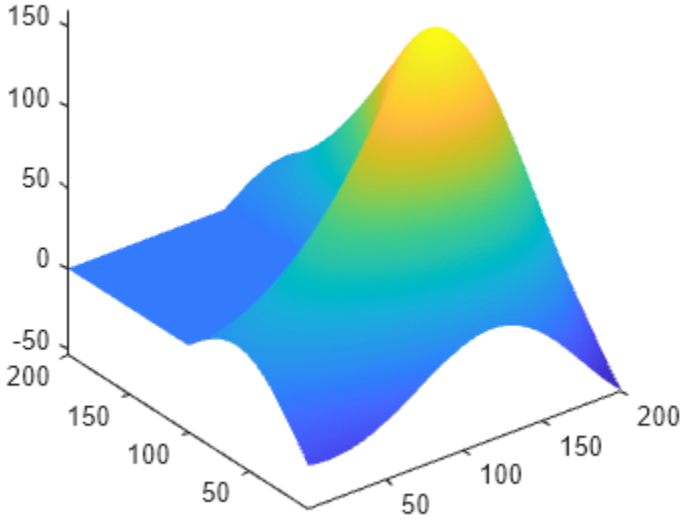
Adjust the axes limits so that the axes are tight around the logo.

```
ax.XLim = [1 201];
ax.YLim = [1 201];
ax.ZLim = [-53.4 160];
```



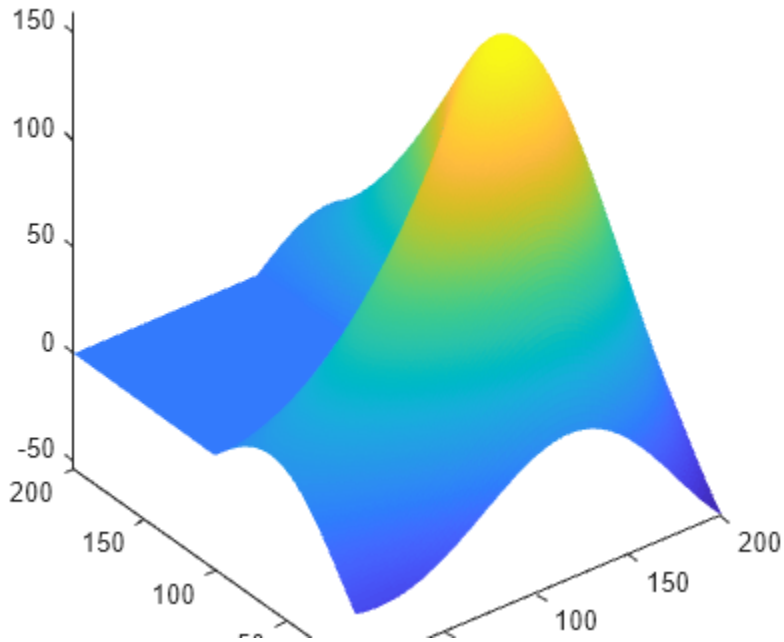
Adjust the view of the logo using the camera properties of the axes. Camera properties control the view of a three dimensional scene like a camera with a zoom lens.

```
ax.CameraPosition = [-145.5 -229.7 283.6];
ax.CameraTarget = [77.4 60.2 63.9];
ax.CameraUpVector = [0 0 1];
ax.CameraViewAngle = 36.7;
```



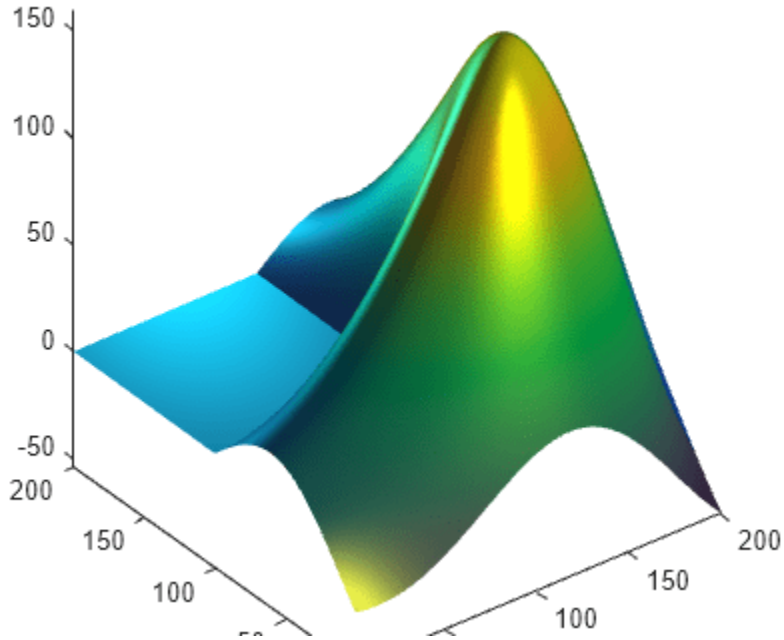
Change the position of the axes and the *x*, *y*, and *z* aspect ratio to fill the extra space in the figure window.

```
ax.Position = [0 0 1 1];
ax.DataAspectRatio = [1 1 .9];
```



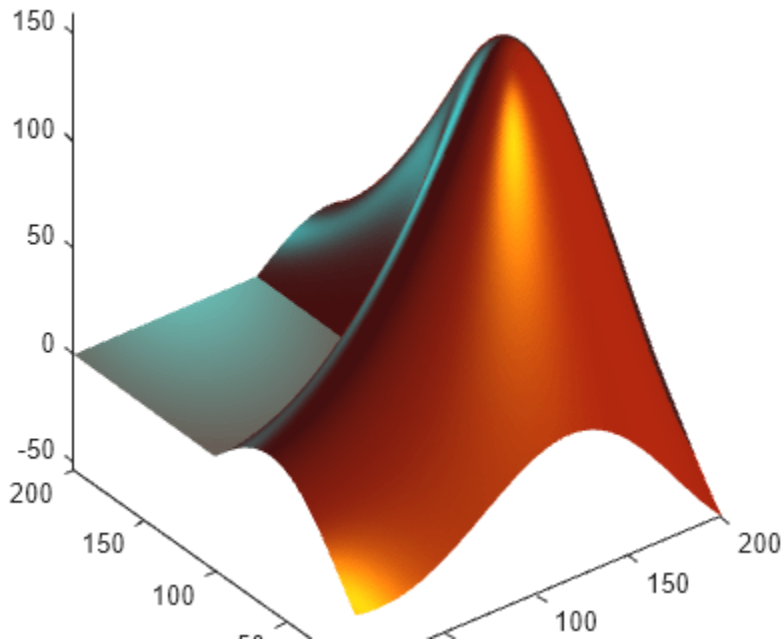
Create lights to illuminate the logo. The light itself is not visible but its properties can be set to change the appearance of any patch or surface object in the axes.

```
l1 = light;
l1.Position = [160 400 80];
l1.Style = 'local';
l1.Color = [0 0.8 0.8];

l2 = light;
l2.Position = [.5 -1 .4];
l2.Color = [0.8 0.8 0];
```



Change the color of the logo.

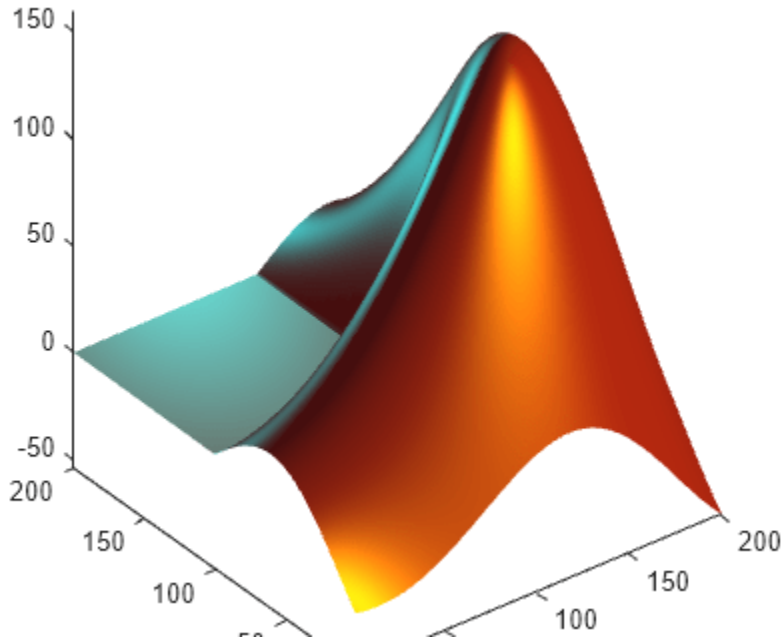
```
s.FaceColor = [0.9 0.2 0.2];
```



Use the lighting and specular (reflectance) properties of the surface to control the lighting effects.

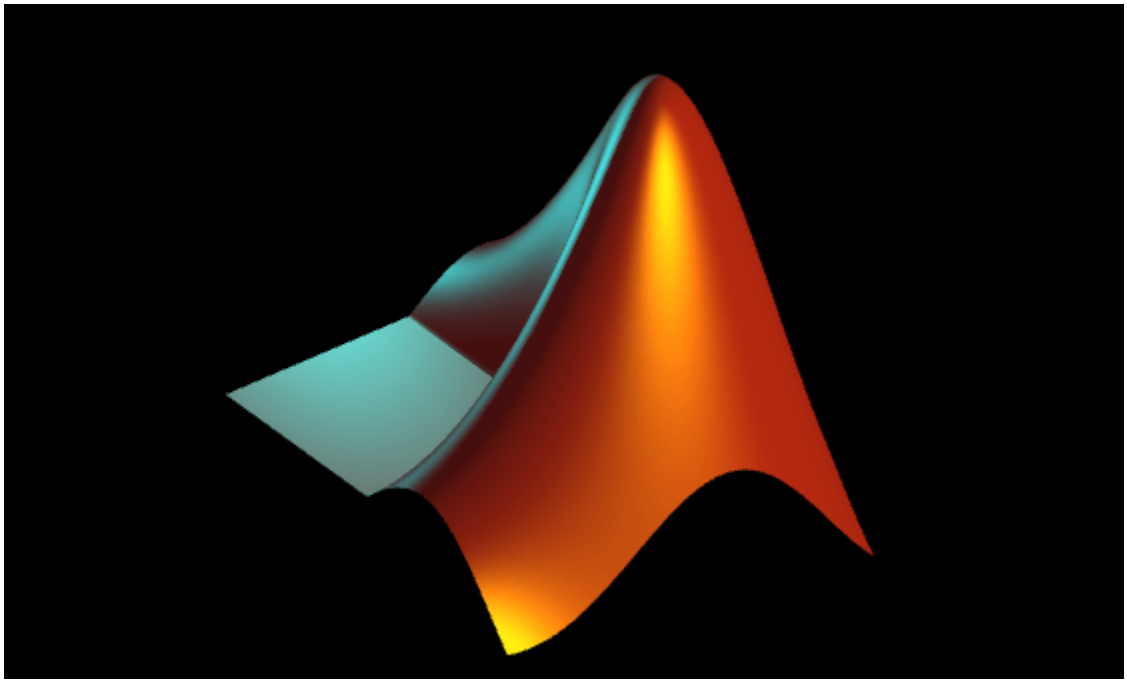
```
s.FaceLighting = 'gouraud';
s.AmbientStrength = 0.3;
s.DiffuseStrength = 0.6;
```

```
s.BackFaceLighting = 'lit';

s.SpecularStrength = 1;
s.SpecularColorReflectance = 1;
s.SpecularExponent = 7;
```



Turn the axis off to see the final result.

```
axis off
f.Color = 'black';
```



Representing Data as a Surface

In this section...
“Functions for Plotting Data Grids” on page 1-20 “Functions for Gridding and Interpolating Data” on page 1-20 “Mesh and Surface Plots” on page 1-21 “Visualizing Functions of Two Variables” on page 1-21 “Surface Plots of Nonuniformly Sampled Data” on page 1-24 “Reshaping Data” on page 1-26 “Parametric Surfaces” on page 1-28

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 1.1. Example: Illustrating the Use of meshgrid

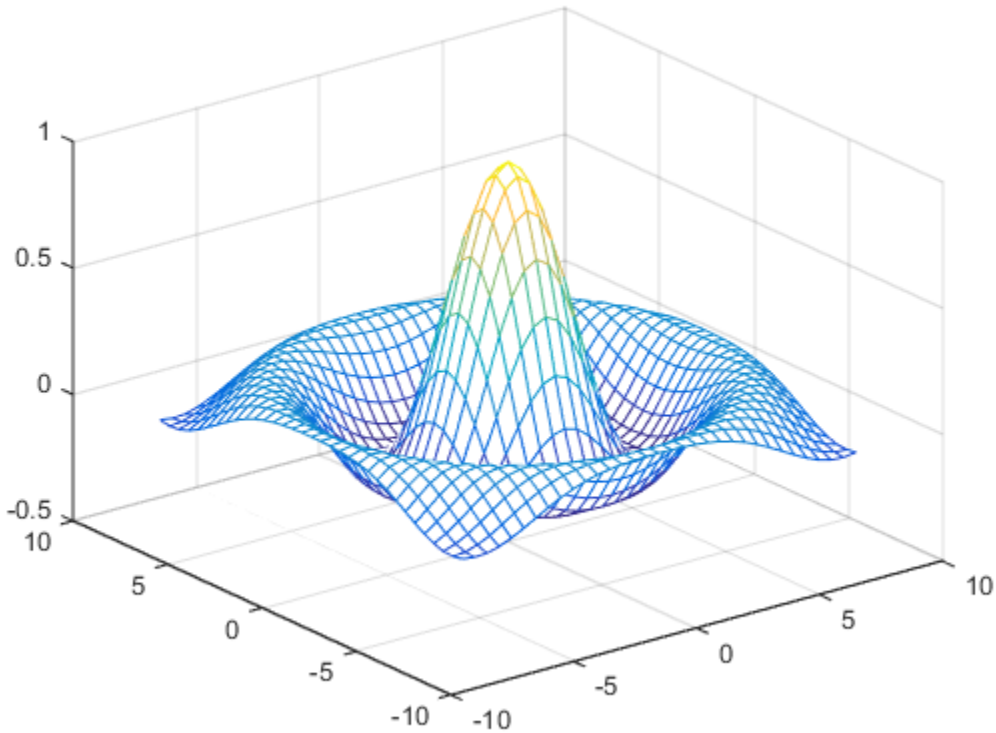
To illustrate the use of `meshgrid`, consider the `sin(r)/r` or `sinc` function. To evaluate this function between -8 and 8 in both *x* and *y*, you need pass only one vector argument to `meshgrid`, which is then used in both directions.

```
[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.

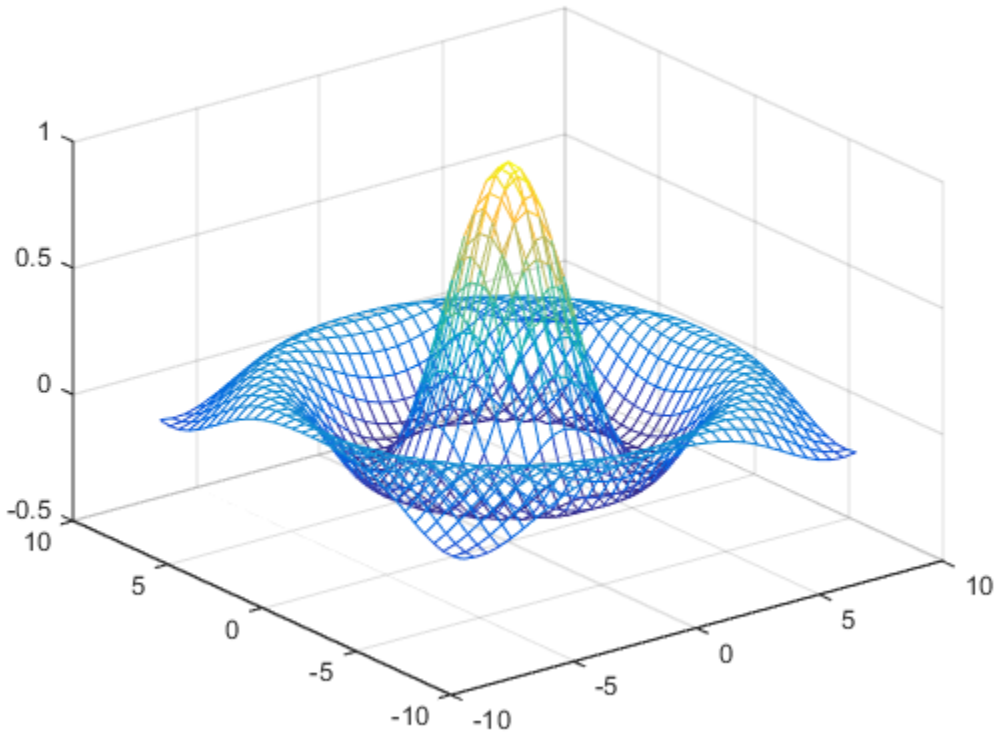
```
Z = sin(R)./R;
figure
mesh(X,Y,Z)
```



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 filled. You can disable hidden line removal and allow the faces of a mesh plot to be transparent with the `hidden` command:

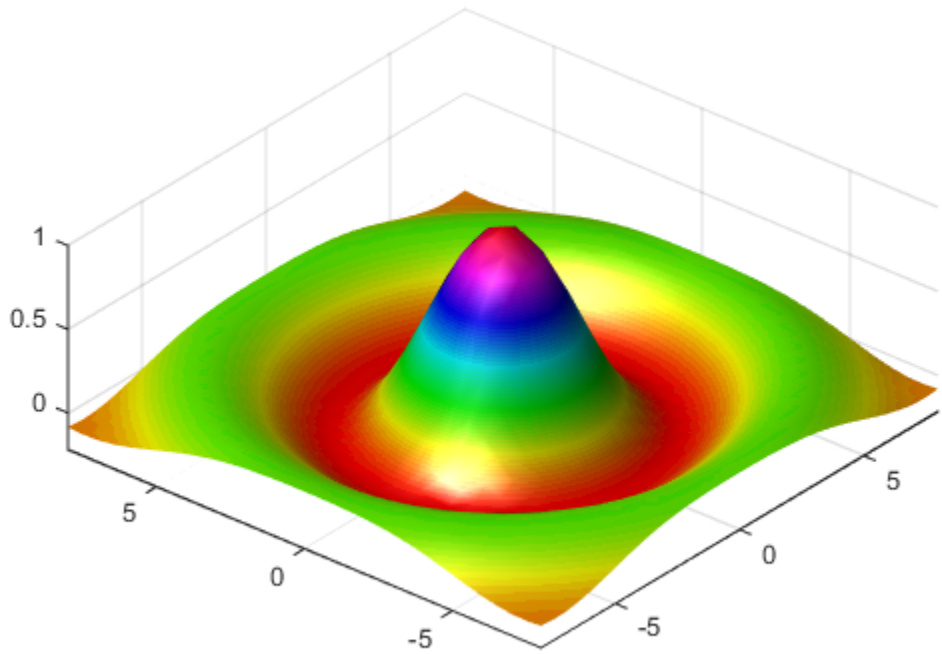
```
hidden off
```



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, view adjustments, and a different colormap to emphasize the shape of the graphed function (`daspect`, `axis`, `view`, `camlight`).

```
figure
colormap hsv
surf(X,Y,Z,'FaceColor','interp',...
   'EdgeColor','none',...
   'FaceLighting','gouraud')
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:

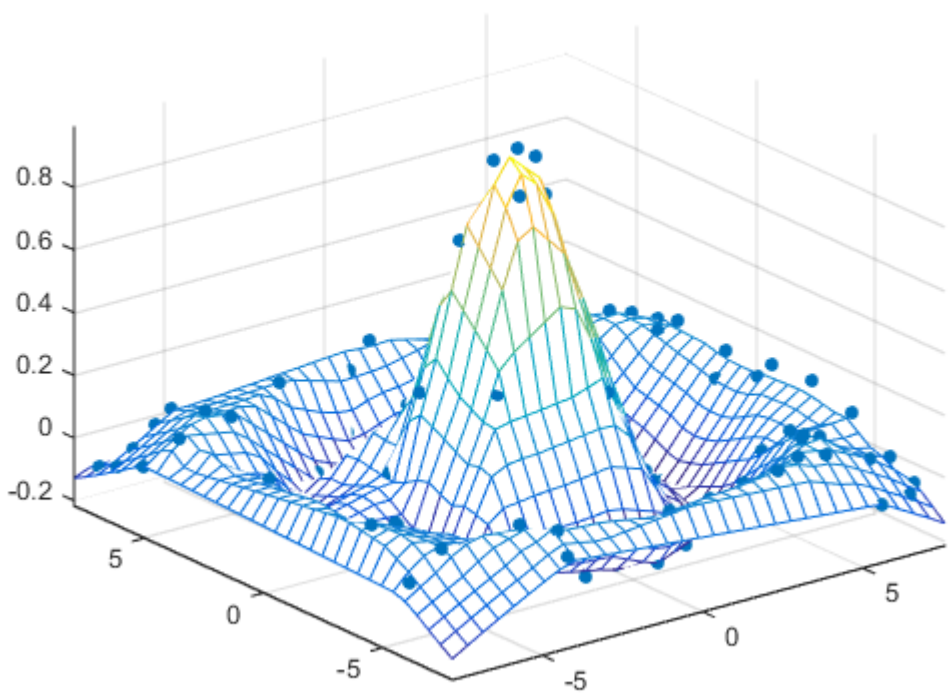
```
[X,Y] = meshgrid(xlin,ylin);
```

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
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)';
```

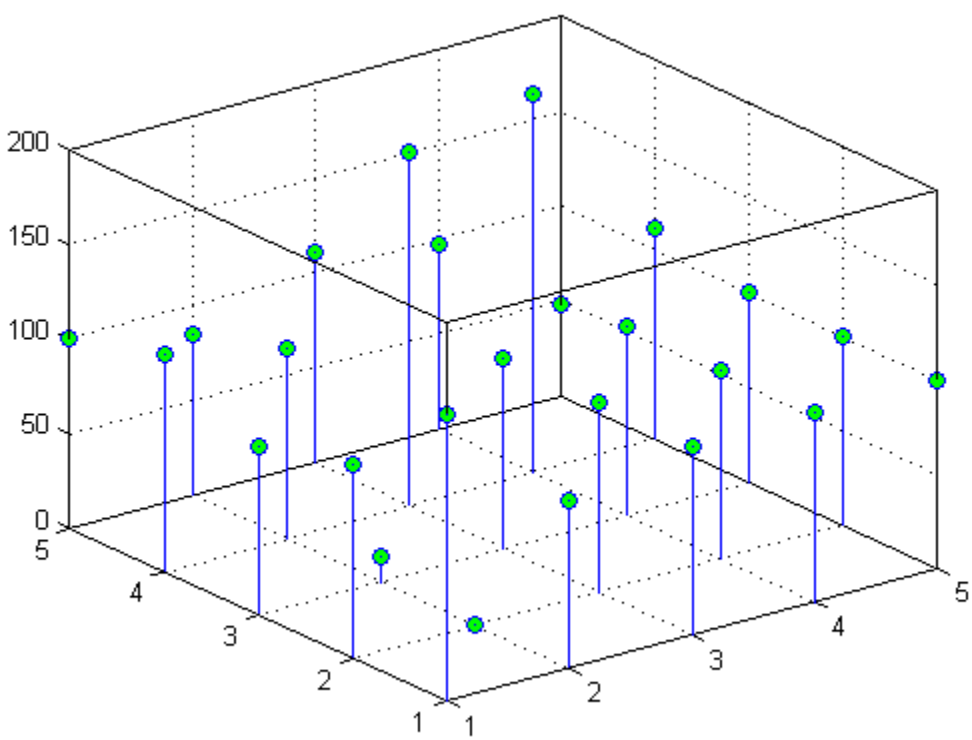
Reshaping results in three 5–by-5 arrays:

```
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

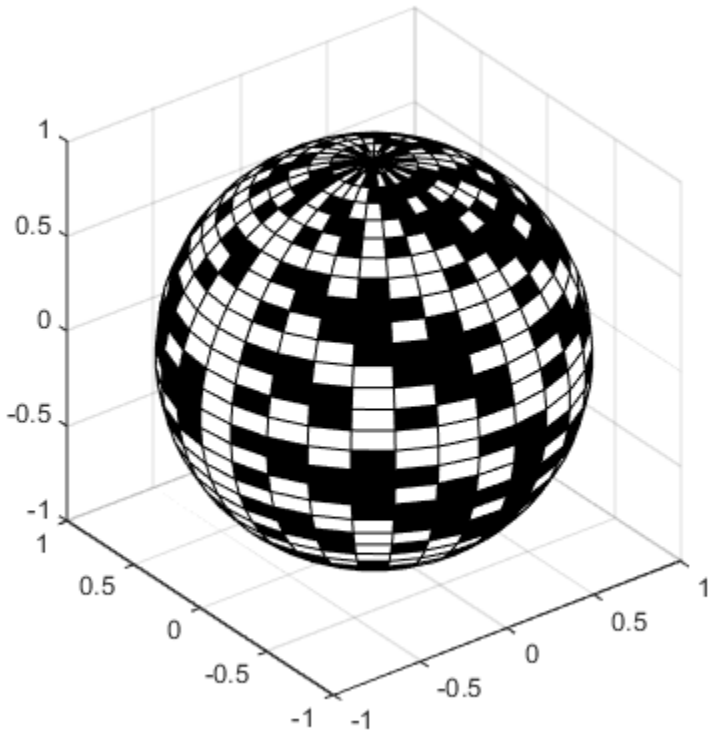
```
mesh(X,Y,Z,C)
```

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 `-π ≤ theta ≤ π` and `-π/2 ≤ phi ≤ π/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
```


1
0.5
0
-0.5
-1
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

2

Polygons

- “Introduction to Patch Objects” on page 2-2
- “Multifaceted Patches” on page 2-6

Introduction to Patch Objects

In this section...

"What Are Patch Objects?" on page 2-2
"Behavior of the patch Function" on page 2-2
"Creating a Single Polygon" on page 2-3

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:

- Specify the coordinates of the vertices of each polygon, which are connected to form the patch.
- Specify the coordinates of each *unique* vertex and a matrix that specifies how to connect these vertices to form the faces. This 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. For more information, see "Multifaceted Patches" on page 2-6.

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.

```
x = [0 1 1 0];
y = [0 0 1 1];
```

```
patch(x,y)
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',x,'YData',y)
```

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

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

See the list of Patch 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 high-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,

```
patch(x,y,1:length(x))
```

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

```
patch(x,y,1:length(x),'y')
```

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

“How Patch Data Relates to a Colormap” provides more information on options for coloring patches.

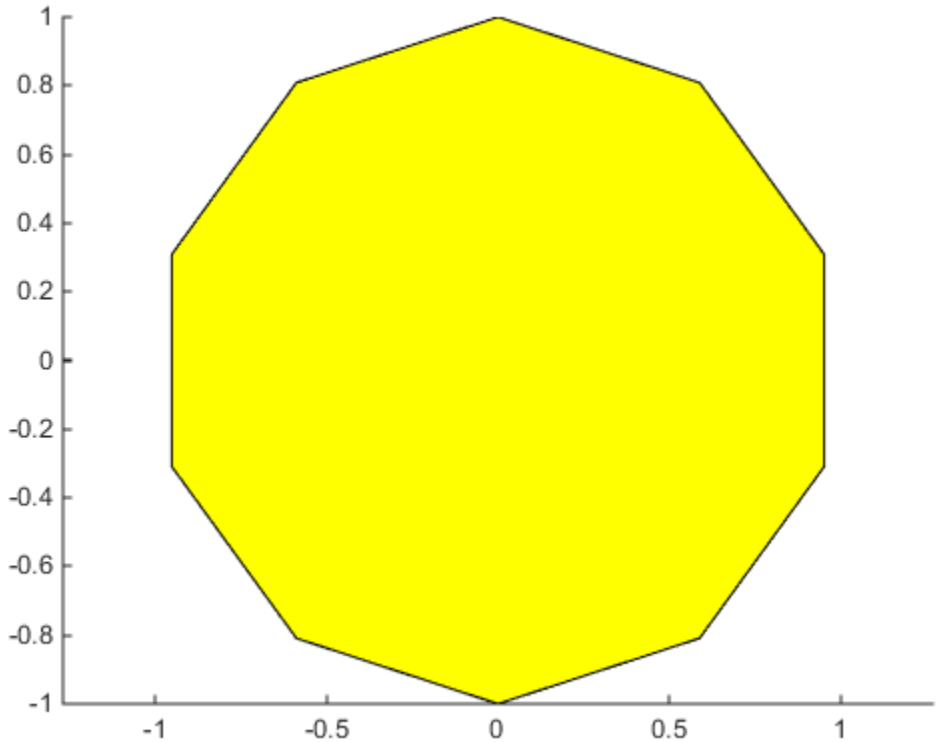
Creating a Single Polygon

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
```

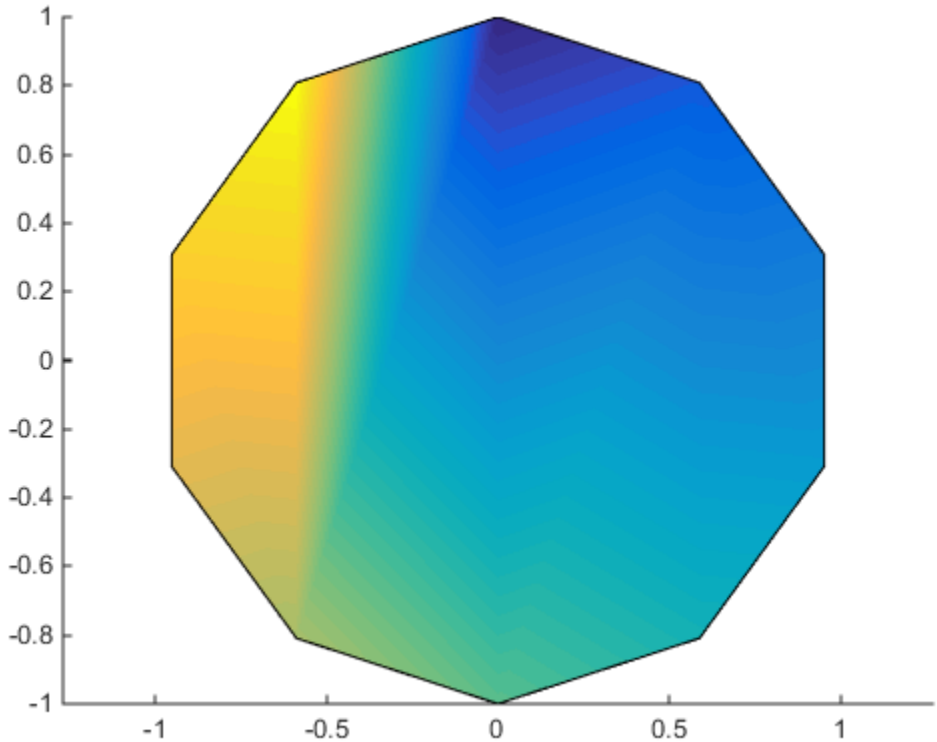


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

Interpolated Face Colors

You can control many aspects of the patch coloring. For example, instead of specifying a single color, 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(a),cos(a),1:length(a),'EdgeColor','interp')
```

“How Patch Data Relates to a Colormap” provides more information on options for coloring patches.

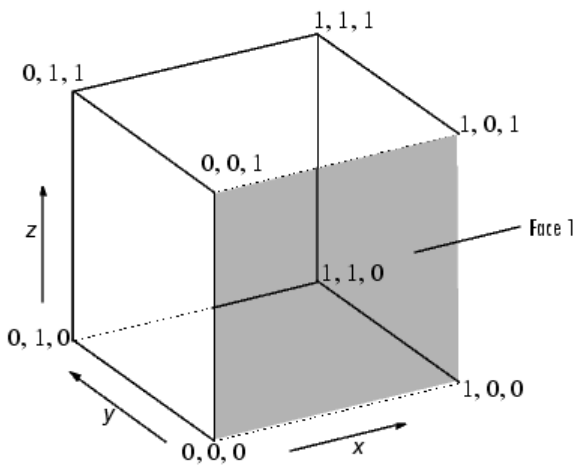
Multifaceted Patches

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.

x-coordinates

0	1	1	0	0	0
1	1	0	0	1	1
1	1	0	0	1	1
0	1	1	0	0	0

y-coordinates

0	0	1	1	0	0
0	1	1	0	0	0
0	1	1	0	1	1
0	0	1	1	1	1

z-coordinates

0	0	0	0	0	1
0	0	0	0	0	1
1	1	1	1	0	1
1	1	1	1	0	1

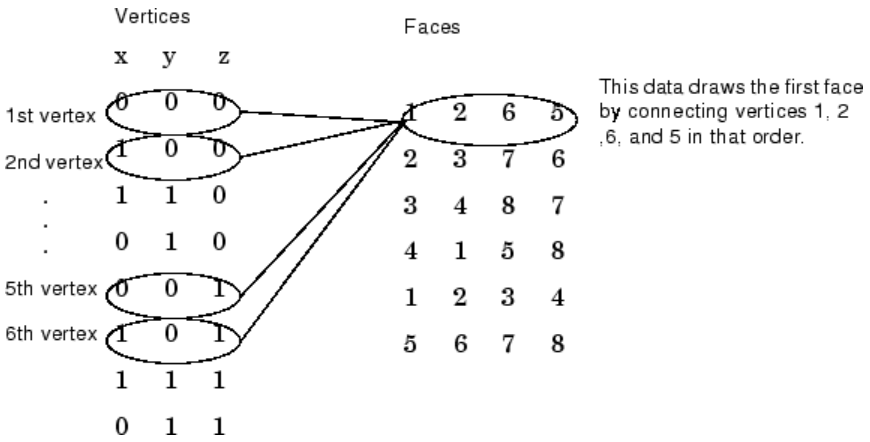
Face 1

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

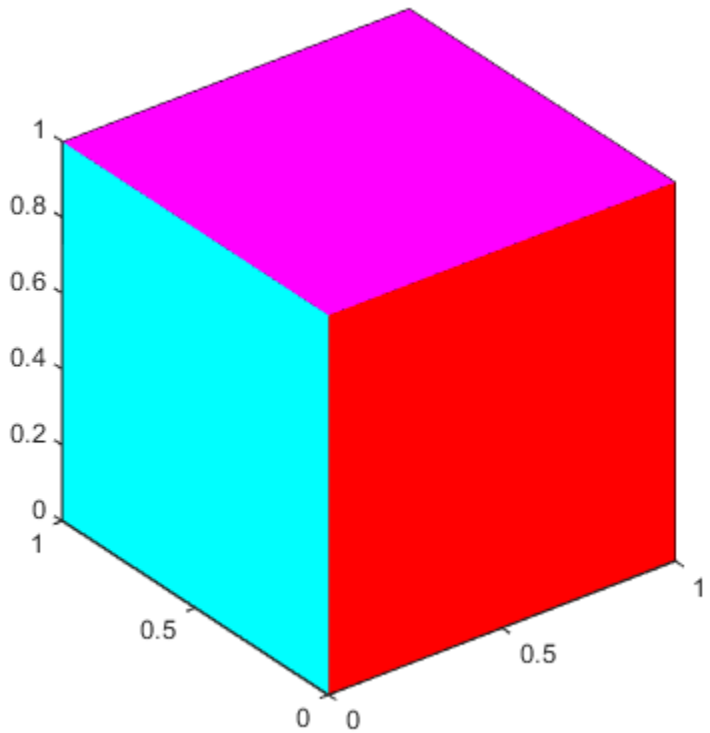
Flat face color is the result of specifying one color per face. For example, using the vertices/faces technique and the `FaceVertexCData` property to define color, this statement specifies one color per face and sets the `FaceColor` property to `flat`.

```
vert = [0 0 0;1 0 0;1 1 0;0 1 0;0 0 1;1 0 1;1 1 1;0 1 1];
fac = [1 2 6 5;2 3 7 6;3 4 8 7;4 1 5 8;1 2 3 4;5 6 7 8];
patch('Vertices',vert,'Faces',fac,...
      'FaceVertexCData',hsv(6),'FaceColor','flat')
```

Adjust the axes:

```
view(3)
axis vis3d
```

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.

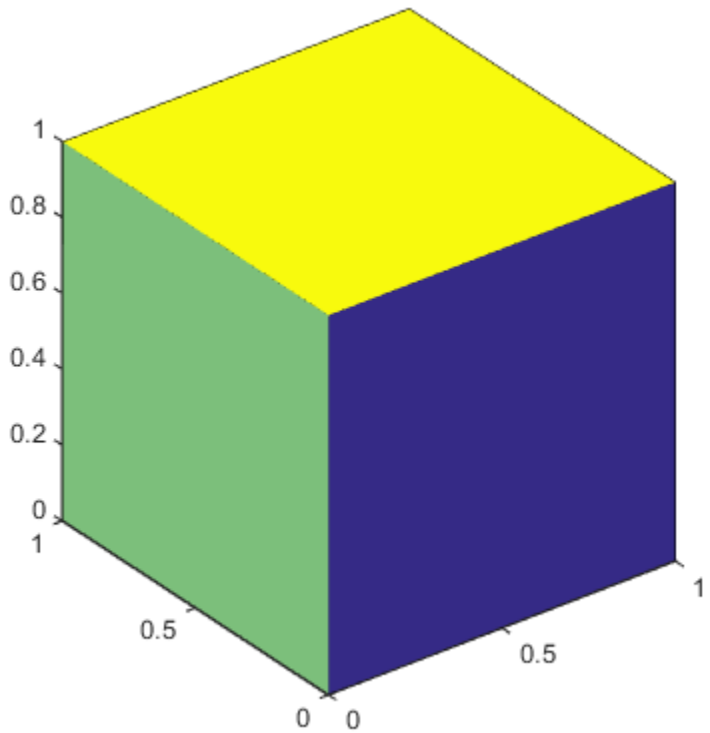


To map face colors to the current colormap, assign an n-by-1 array to the `FaceVertexCData` property:

```
patch('Vertices',vert,'Faces',fac,...
      'FaceVertexCData',(1:6)','FaceColor','flat')
```

Adjust the axes:

```
view(3)
axis vis3d
```



Interpolated Face Color

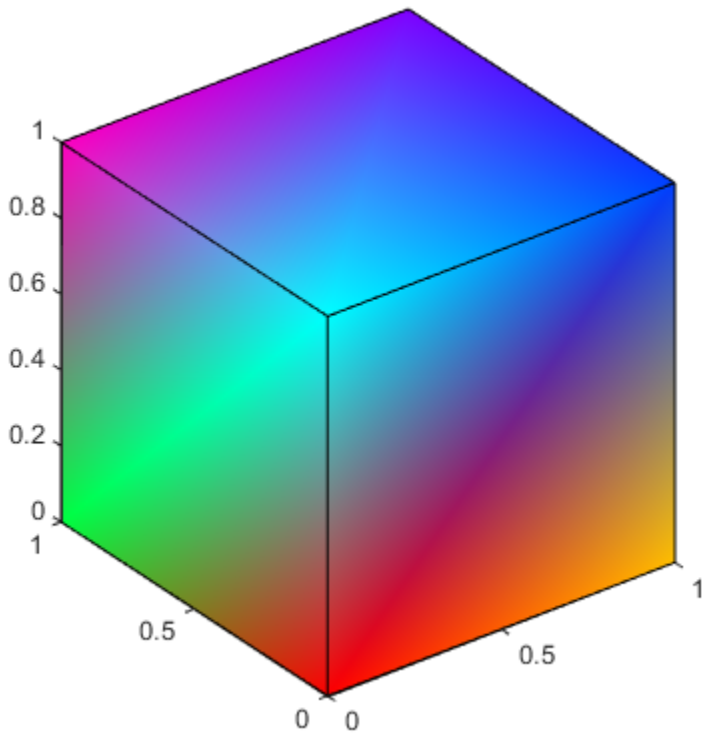
Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`.

```
patch('Vertices',vert,'Faces',fac,...
      'FaceVertexCData',hsv(8),'FaceColor','interp')
```

Adjust the axes:

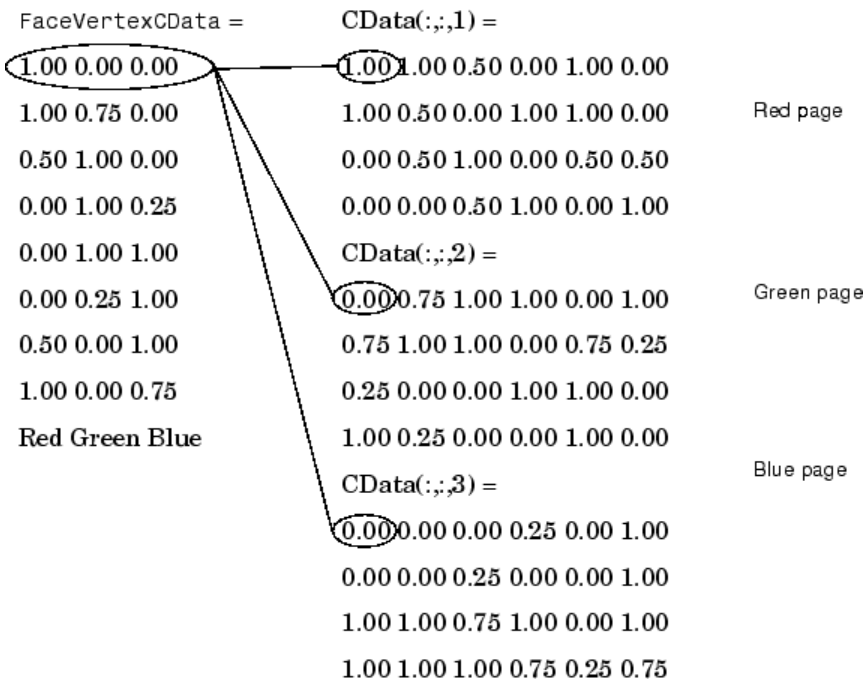
```
view(3)
axis vis3d
```

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.



“How Patch Data Relates to a Colormap” discusses coloring techniques in more detail.

Volume Visualization

- “Overview of Volume Visualization” on page 3-2
- “Techniques for Visualizing Scalar Volume Data” on page 3-5
- “Exploring Volumes with Slice Planes” on page 3-12
- “Connecting Equal Values with Isosurfaces” on page 3-18
- “Isocaps Add Context to Visualizations” on page 3-20
- “Visualizing Vector Volume Data” on page 3-24
- “Display Streamlines Using Vector Data” on page 3-29
- “Displaying Curl with Stream Ribbons” on page 3-31
- “Displaying Divergence with Stream Tubes” on page 3-33
- “Create Stream Particle Animations” on page 3-36
- “Vector Field Displayed with Cone Plots” on page 3-40
- “Visualizing Volume Data” on page 3-43
- “Visualizing Four-Dimensional Data” on page 3-50
- “Displaying Complex Three-Dimensional Objects” on page 3-55
- “Displaying Topographic Data” on page 3-62

Overview of Volume Visualization

In this section...

"Examples of Volume Data" on page 3-2
"Selecting Visualization Techniques" on page 3-2
"Steps to Create a Volume Visualization" on page 3-3
"Volume Visualization Functions" on page 3-3

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 your 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

Function	Purpose
`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?” on page 3-5 “Ways to Display MRI Data” on page 3-5

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 on page 3-12 example, and slice planes used to show context. on page 3-20) 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

- “Changing the Data Format” on page 3-6
- “Displaying Images of MRI Data” on page 3-6
- “Displaying a 2-D Contour Slice” on page 3-7
- “Displaying 3-D Contour Slices” on page 3-8
- “Applying an Isosurface to the MRI Data” on page 3-9
- “Adding Isocaps Show Cut-Away Surface” on page 3-9
- “Defining the View” on page 3-10
- “Add Lighting” on page 3-10

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 on page 3-6 representing slices through the head

- 2-D on page 3-7 and 3-D on page 3-8 contour slices taken at arbitrary locations within the data
- An isosurface with isocaps on page 3-9 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.

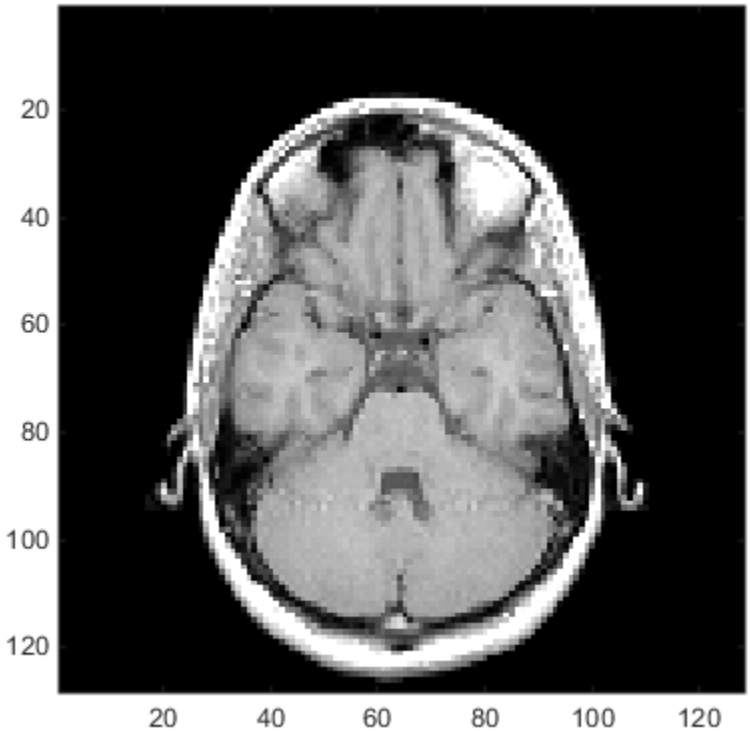
```
load mri
D = squeeze(D);
```

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;
```

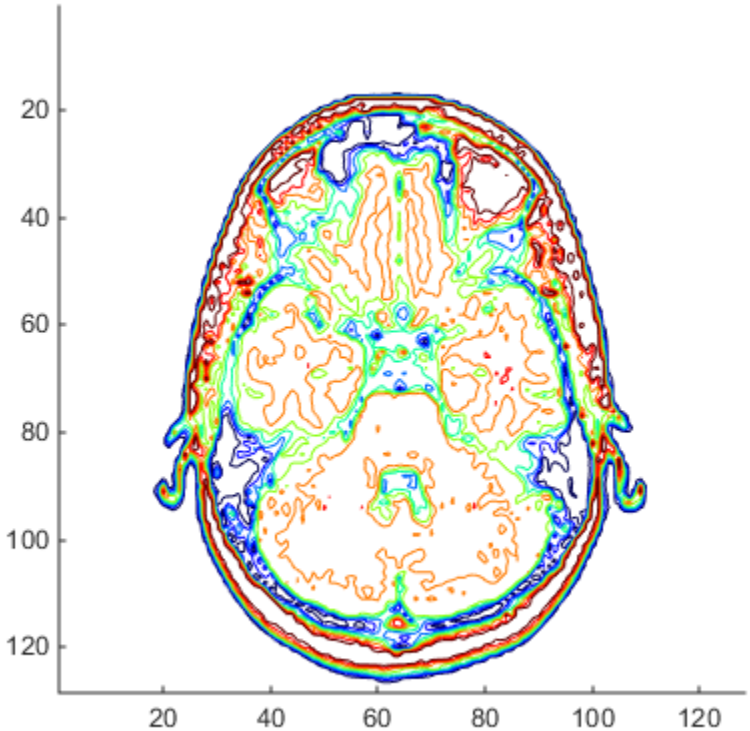
Displaying a 2-D Contour Slice

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.

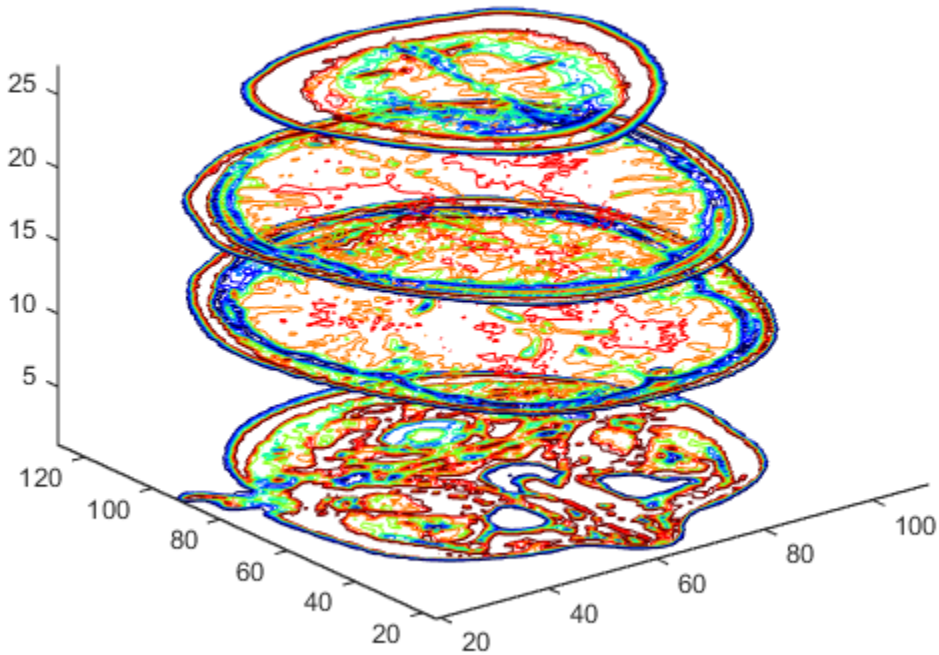
```
cm = brighten(jet(length(map)),-.5);
figure
colormap(cm)
contourslice(D,[],[],image_num)
axis ij
xlim(x)
ylim(y)
daspect([1,1,1])
```



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`).

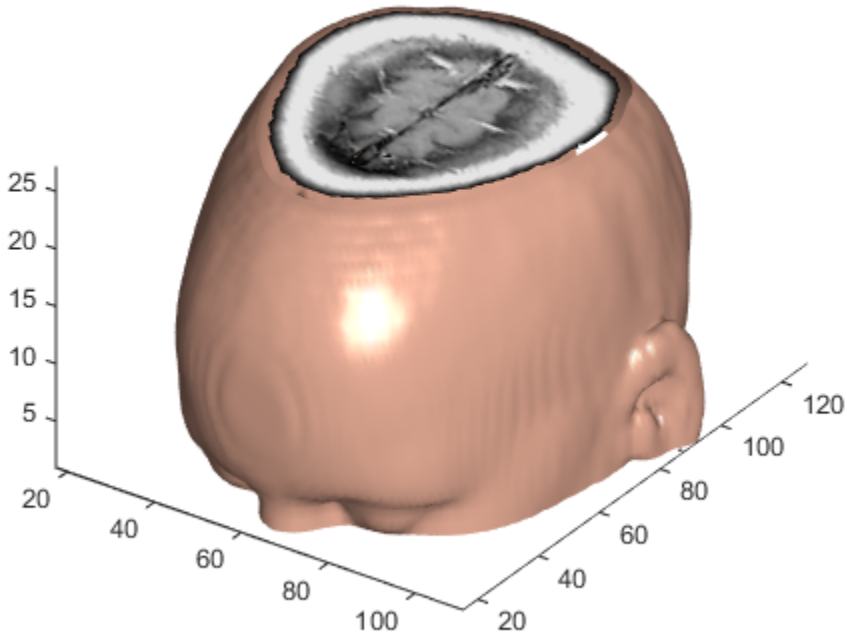
```
view(35,30)
axis tight
daspect([1,1,.4])
```

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);
lighting gouraud
hcap.AmbientStrength = 0.6;
hiso.SpecularColorReflectance = 0;
hiso.SpecularExponent = 50;
```

Here is a visualization of the MRI data that combines an isocap with an isosurface.



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" on page 3-12
"Modify the Color Mapping" on page 3-15

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 "Display Streamlines Using Vector Data" on page 3-29 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));
```

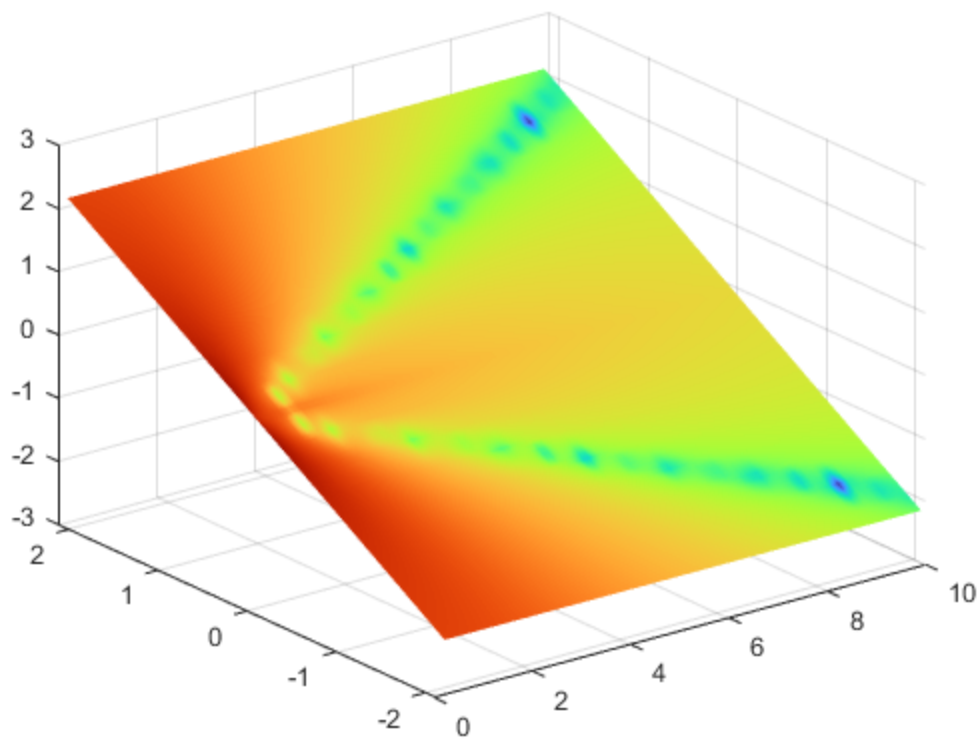
Rotate the surface by -45 degrees about the *x*-axis and save the surface `XData`, `YData`, and `ZData` to define the slice plane; then delete the surface.

```
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.

```
colormap(turbo)
h = slice(x,y,z,v,xd,yd,zd);
h.FaceColor = 'interp';
h.EdgeColor = 'none';
h.DiffuseStrength = 0.8;
```



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,[],[]);
hx.FaceColor = 'interp';
hx.EdgeColor = 'none';

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

hz = slice(x,y,z,v,[],[],zmin);
hz.FaceColor = 'interp';
hz.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`). The orientation of the axes can be selected initially using `rotate3d` to determine the best `view`.

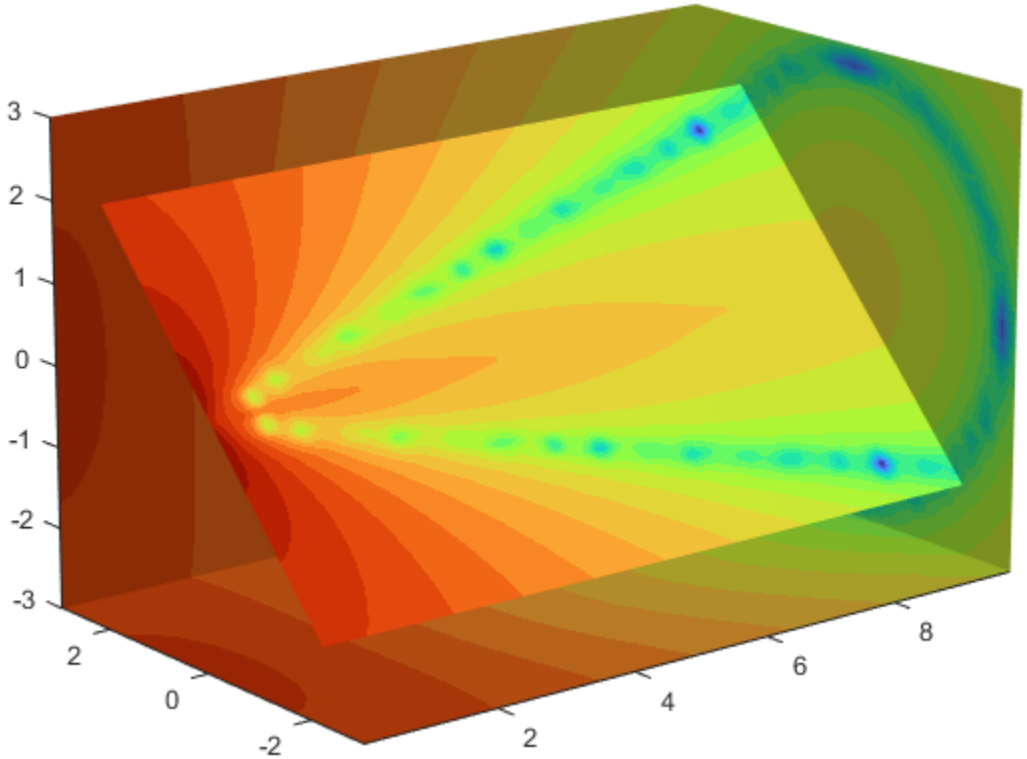
Zooming in on the scene provides a larger view of the volume (`camzoom`). Selecting a projection type of `perspective` gives the rectangular solid more natural proportions than the default orthographic projection (`camproj`).

```
daspect([1,1,1])
axis tight
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(turbo(24))
```



“Modify the Color Mapping” on page 3-15 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:

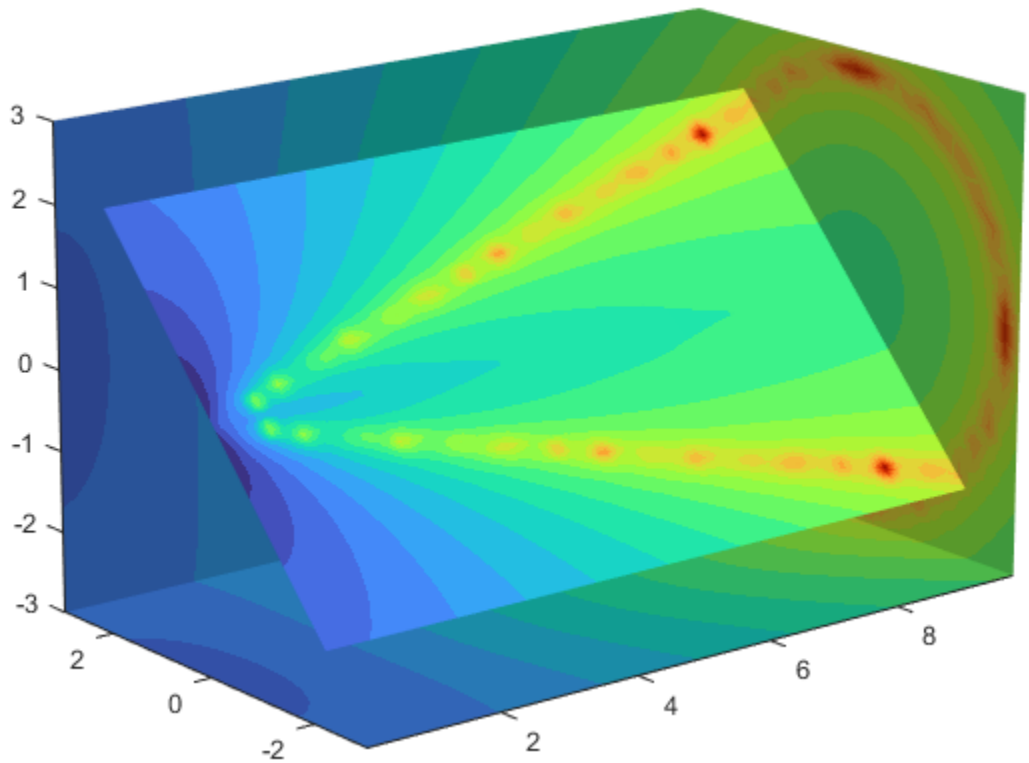
- Changing the colormap
- Changing the mapping of data value to color

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 `turbo` colormap).

1. Customize the Colormap

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

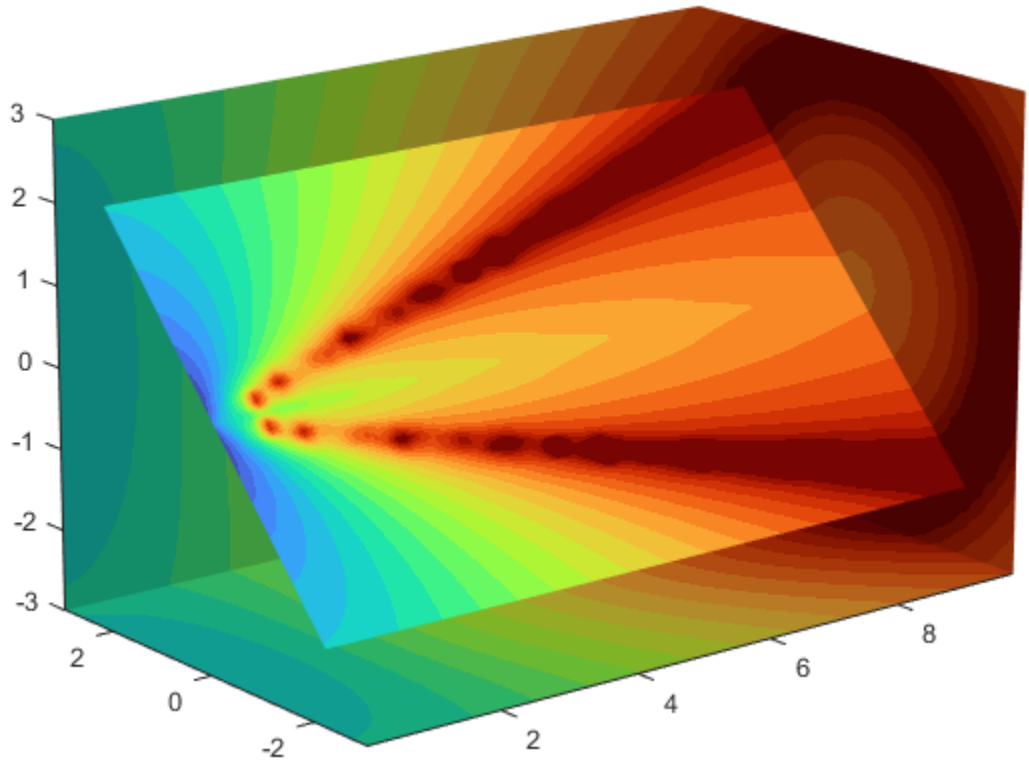
```
colormap(flipud(turbo(24)))
```



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.

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

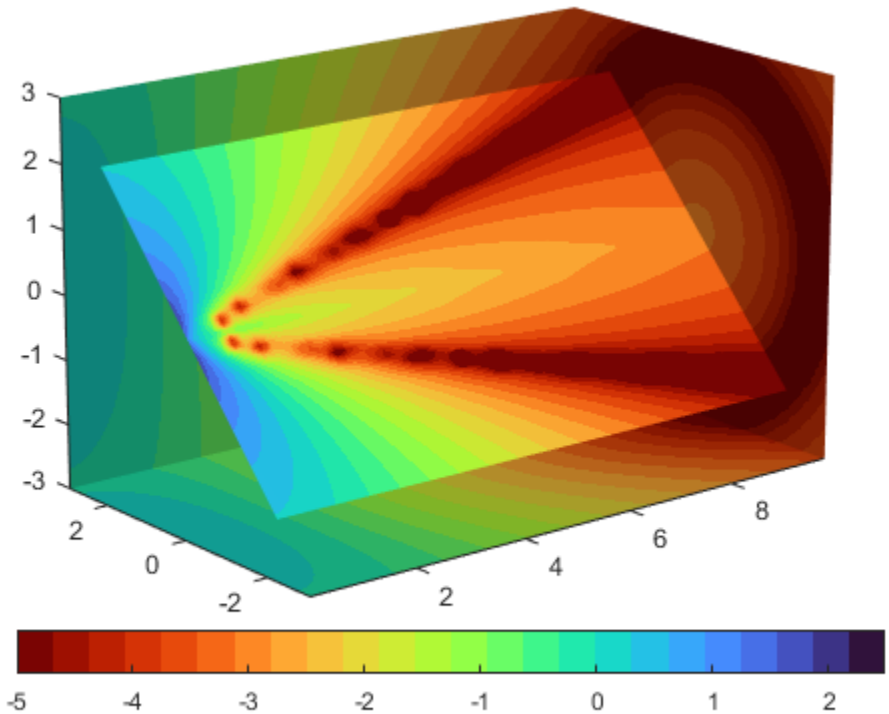


Adjust the color limits using `caxis`, which has the same syntaxes and arguments as `clim`.

3. Add a Color Bar

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

```
colorbar('southoutside')
```


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;
```

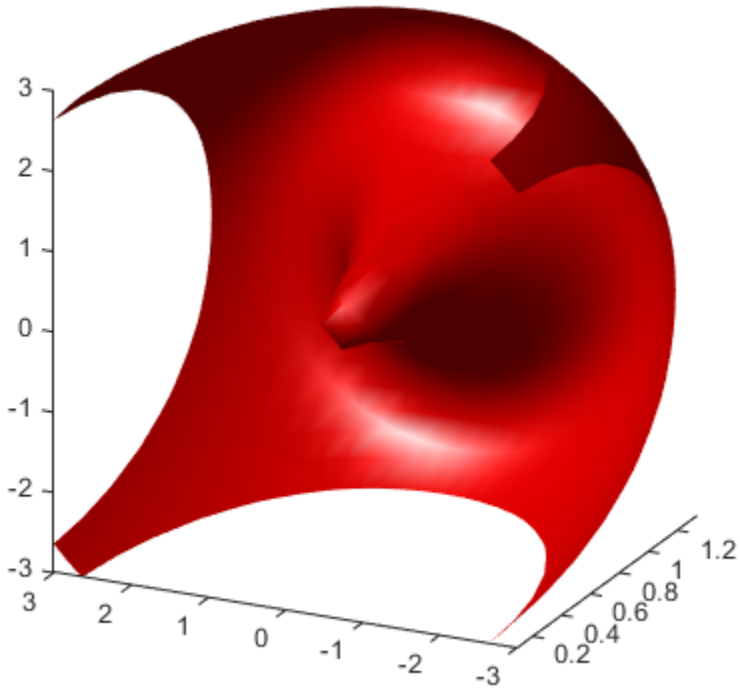
To select the isovalue, determine the range of values in the volume data.

```
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)
hpatch.FaceColor = 'red';
hpatch.EdgeColor = 'none';
daspect([1,4,4])
view([-65,20])
axis tight
camlight left;
lighting gouraud
```


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

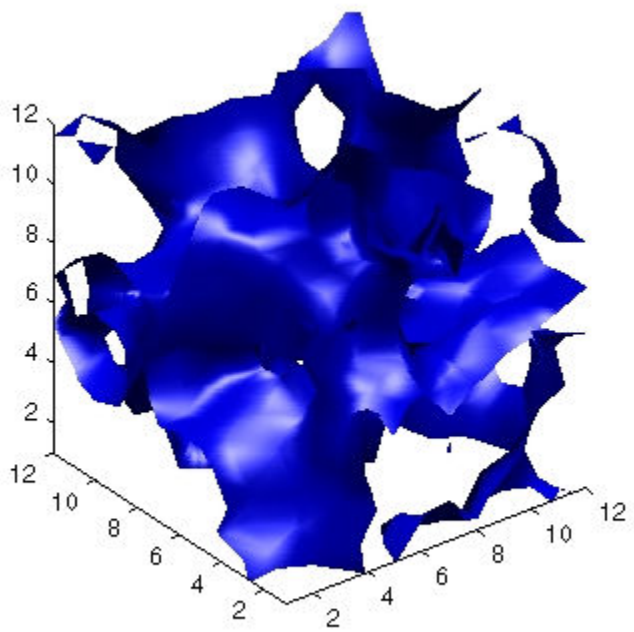
In this section...

"What Are Isocaps?" on page 3-20
"Other Isocap Applications" on page 3-21
"Defining Isocaps" on page 3-21
"Adding Isocaps to an Isosurface" on page 3-22

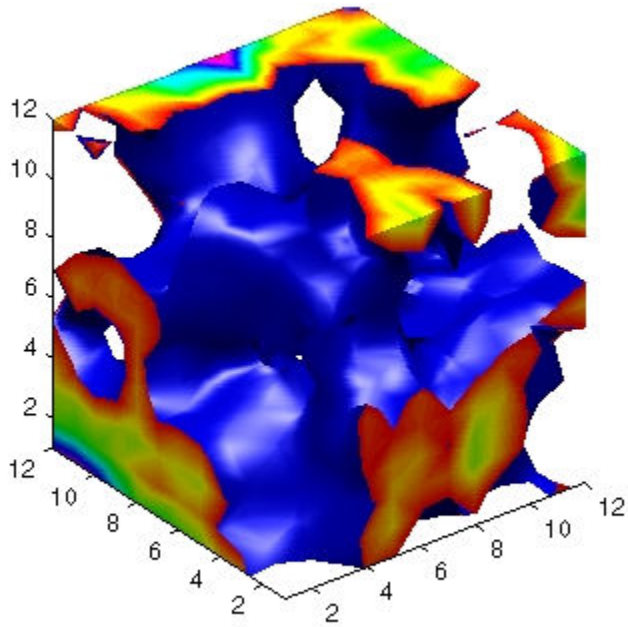
What Are Isocaps?

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. on page 3-9
- Isocaps enhance the visibility of the isosurface limits. on page 3-20

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. on page 3-22
2. Create the isosurface and isocaps and set patch properties to control the coloring and lighting. on page 3-22
3. Create the isocaps and set properties. on page 3-22
4. Specify the view. on page 3-22
5. Add lights to the scene. on page 3-14

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

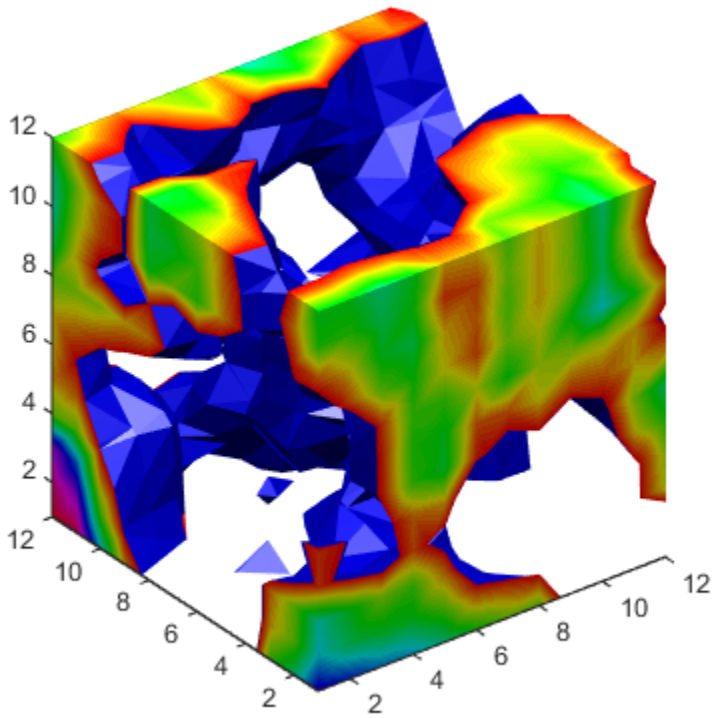
Set the data aspect ratio to `[1,1,1]` so that the display is in correct proportions (`daspect`). Eliminate white space within the axes and set the view to 3-D (`axis tight`, `view`).

```
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 Gouraud lighting to produce the smoothest variation of color (`lighting`).

```
camlight right
camlight left
```



Visualizing Vector Volume Data

In this section...

"Lines, Particles, Ribbons, Streams, Tubes, and Cones" on page 3-24
"Using Scalar Techniques with Vector Data" on page 3-24
"Specifying Starting Points for Stream Plots" on page 3-24
"Accessing Subregions of Volume Data" on page 3-27

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

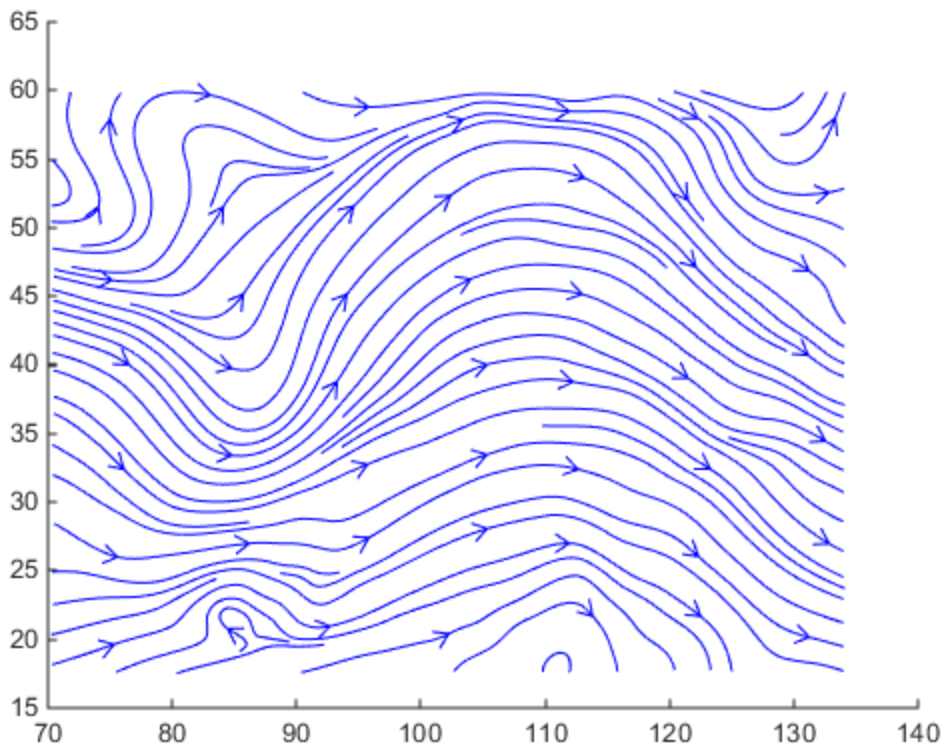
Stream plots (stream lines, ribbons, tubes, and cones or arrows) illustrate the flow of a 3-D vector field. The MATLAB stream-plotting functions (`streamline`, `streamribbon`, `streamtube`, `coneplot`, `stream2`, `stream3`) all require you to specify the point at which you want to begin each stream trace.

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

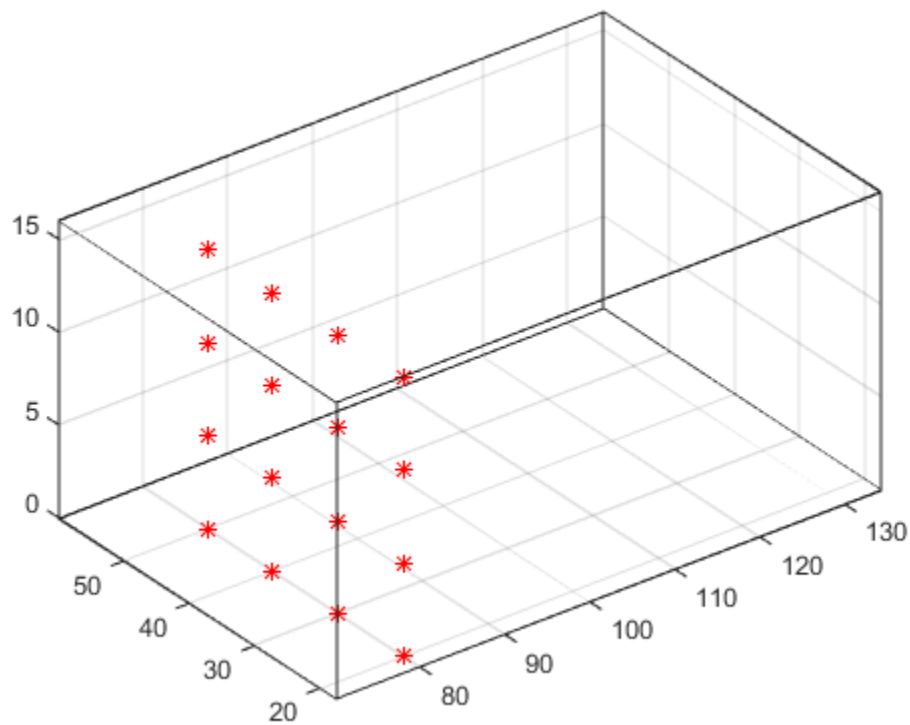
To specify the starting point for one stream line, you need the *x*-, *y*-, and *z*-coordinates of the point. The `meshgrid` command provides a convenient way to create arrays of starting points. For example,

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 on
set(gca,'BoxStyle','full','Box','on')
daspect([2 2 1])
```



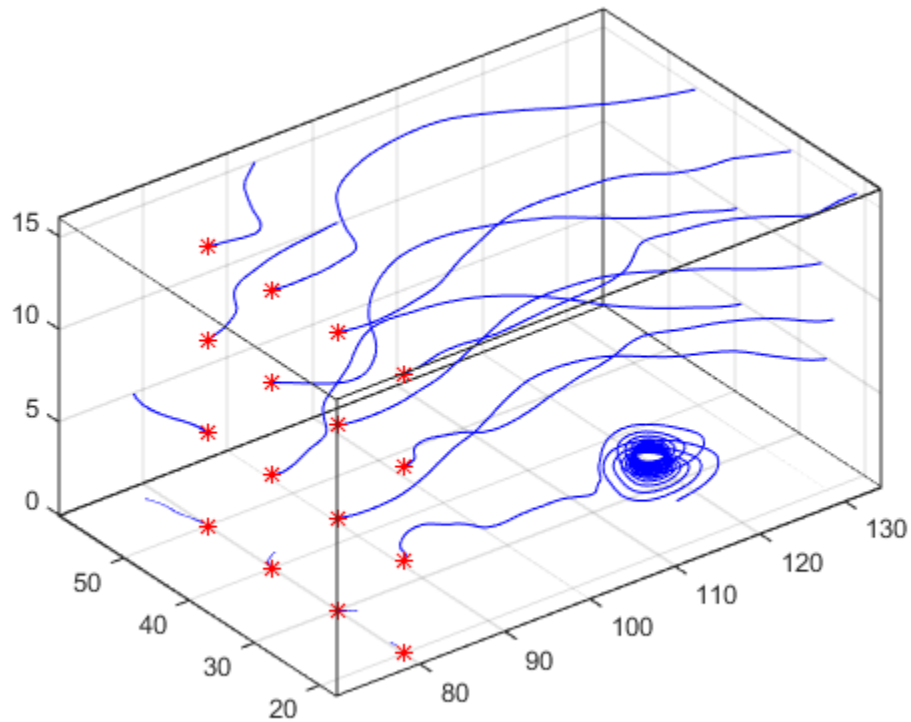
You do not need to use 3-D arrays, such as those returned by `meshgrid`, but the size of each array must be the same, and `meshgrid` provides a convenient way to generate arrays when you do not have an equal number of unique values in each coordinate. You can also define starting-point arrays as column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

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

  Name      Size             Bytes  Class     Attributes

  sx        4x1x4              128  double
  sy        4x1x4              128  double
  sz        4x1x4              128  double
```

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 to the starting points 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.

Display Streamlines Using Vector Data

In this section...

"Wind Mapping Data" on page 3-29
"1. Determine the Range of the Coordinates" on page 3-29
"2. Add Slice Planes for Visual Context" on page 3-29
"3. Add Contour Lines to the Slice Planes" on page 3-29
"4. Define the Starting Points for Stream Lines" on page 3-30
"5. Define the View" on page 3-30

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')
colormap turbo
```

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',[0.7 0.7 0.7],'LineWidth',0.5)
```

4. Define the Starting Points for Stream Lines

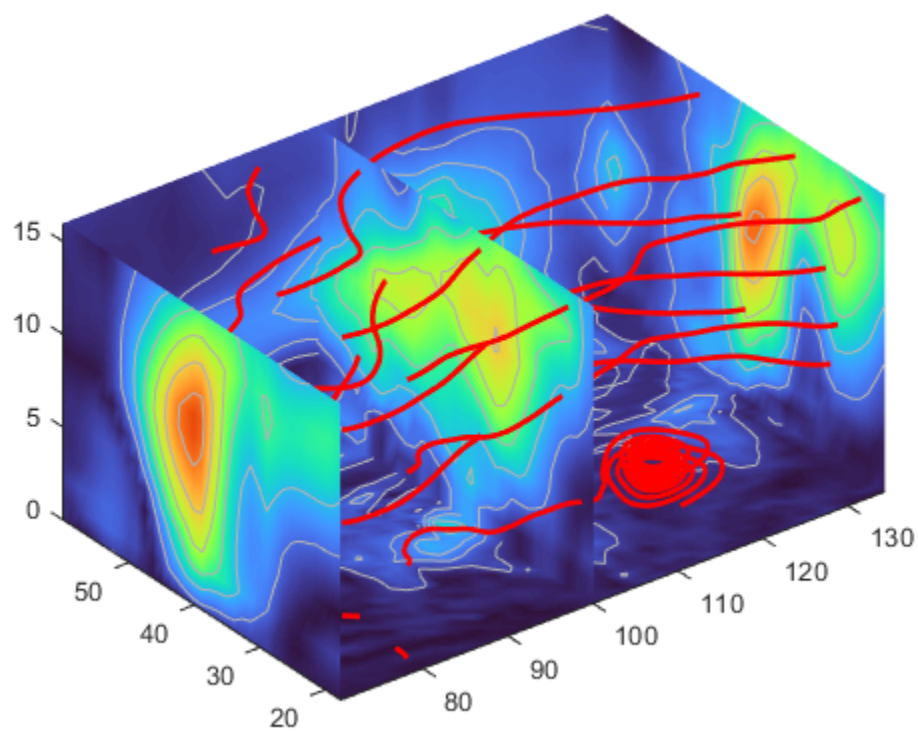
In this example, all stream lines start at an *x*-axis value of 80 and span the range 20 to 50 in the *y*-direction and 0 to 15 in the *z*-direction. Save the handles of the stream lines and set the line width and color (`meshgrid`, `streamline`, `LineWidth`, `Color`).

```
[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" on page 3-31
"1. Select a Subset of Data to Plot" on page 3-31
"2. Calculate Curl Angular Velocity and Wind Speed" on page 3-31
"3. Create the Stream Ribbons" on page 3-31
"4. Define the View and Add Lighting" on page 3-32

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 3-24 for 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`).

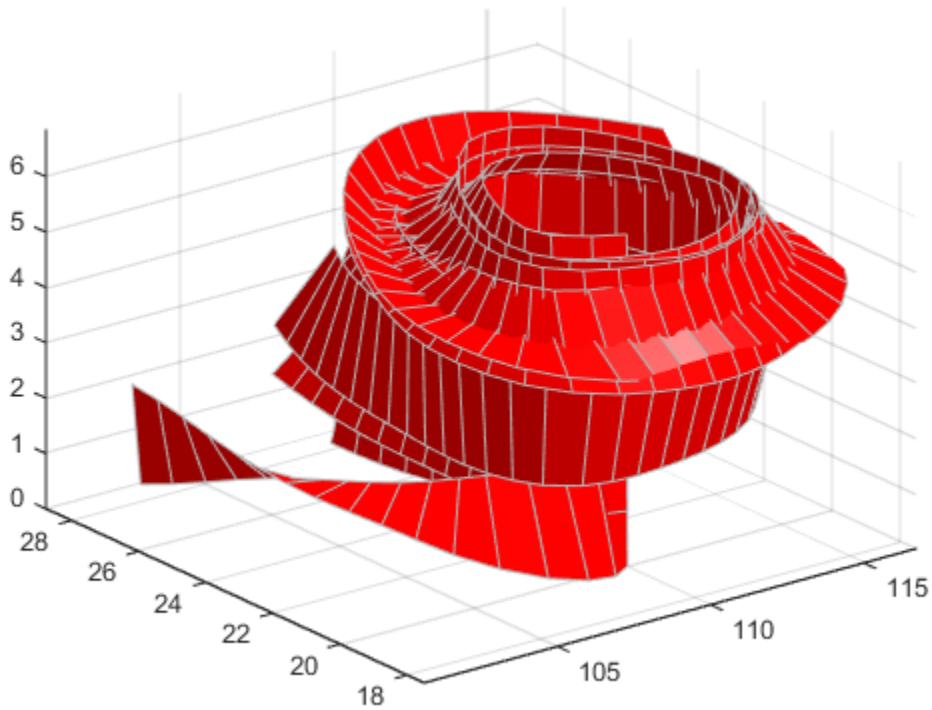
  ```
  [sx sy sz] = meshgrid(110,20:5:30,1:5);
  verts = stream3(x,y,z,u,v,w,sx,sy,sz,.5);
  ```

```
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 Gouraud.

```
axis(volumebounds(x,y,z,wind_speed))
grid on
view(3)
camlight right;
```



Displaying Divergence with Stream Tubes

In this section...

"What Stream Tubes Can Show" on page 3-33
"1. Load Data and Calculate Required Values" on page 3-33
"2. Draw the Slice Planes" on page 3-33
"3. Add Contour Lines to Slice Planes" on page 3-34
"4. Create the Stream Tubes" on page 3-34
"5. Define the View" on page 3-34

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 `clim` to get the current color limits.

  Use `caxis`, which has the same syntaxes and arguments as `clim`.
- Set the interpolation method used by `contourslice` to `linear` to match the default used by `slice`.

```
color_lim = clim;
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`.

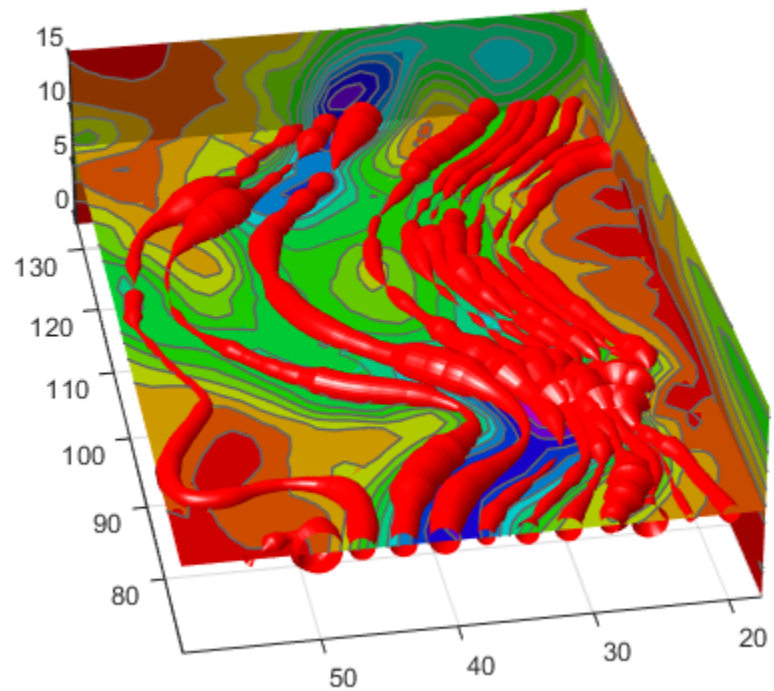
Stream tubes are surface objects, therefore you can control their appearance by setting surface properties. This example sets surface properties to give a brightly lit, red surface.

```
[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

Create Stream Particle Animations

In this section...

"Projectile Path Over Time" on page 3-36
"What Particle Animations Can Show" on page 3-37
"1. Specify Starting Points of the Data Range" on page 3-37
"2. Create Stream Lines to Indicate Particle Paths" on page 3-37
"3. Define the View" on page 3-37
"4. Calculate the Stream Particle Vertices" on page 3-38

Projectile Path Over Time

This example shows how to display the path of a projectile as a function of time using a three-dimensional quiver plot.

Show the path of the following projectile using constants for velocity and acceleration, `vz` and `a`. Calculate `z` as the height as time varies from 0 to 1.

$$z(t) = v_z t + \frac{at^2}{2}$$

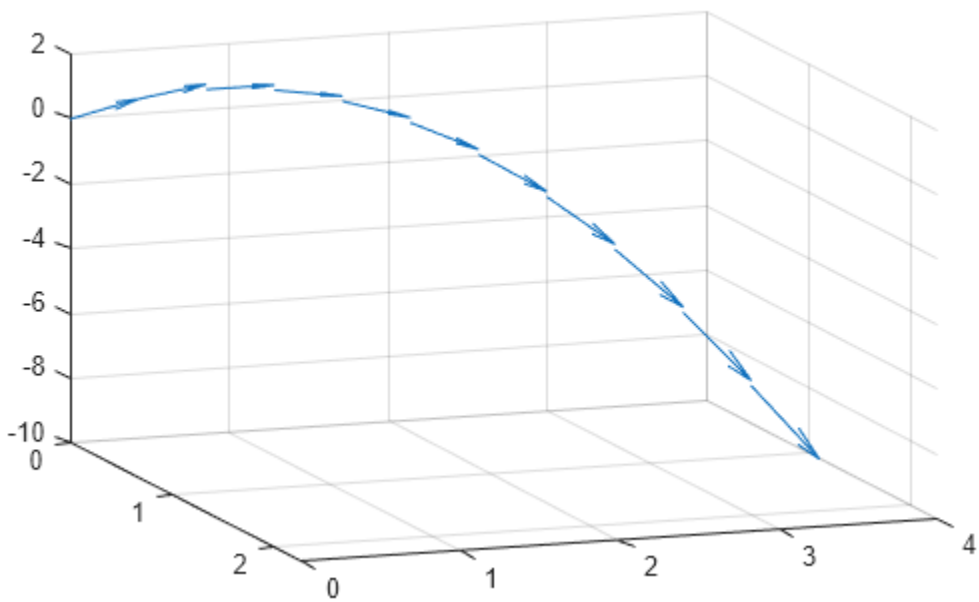
```
vz = 10; % velocity constant
a = -32; % acceleration constant
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
```

Calculate the position in the *x*-direction and *y*-direction.

```
vx = 2;
x = vx*t;

vy = 3;
y = vy*t;
```

Compute the components of the velocity vectors and display the vectors using a 3-D quiver plot. Change the viewpoint of the axes to `[70,18]`.

```
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;

figure
quiver3(x,y,z,u,v,w,scale)
view([70,18])
```



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;
set(gca,'BoxStyle','full','Box','on')
```

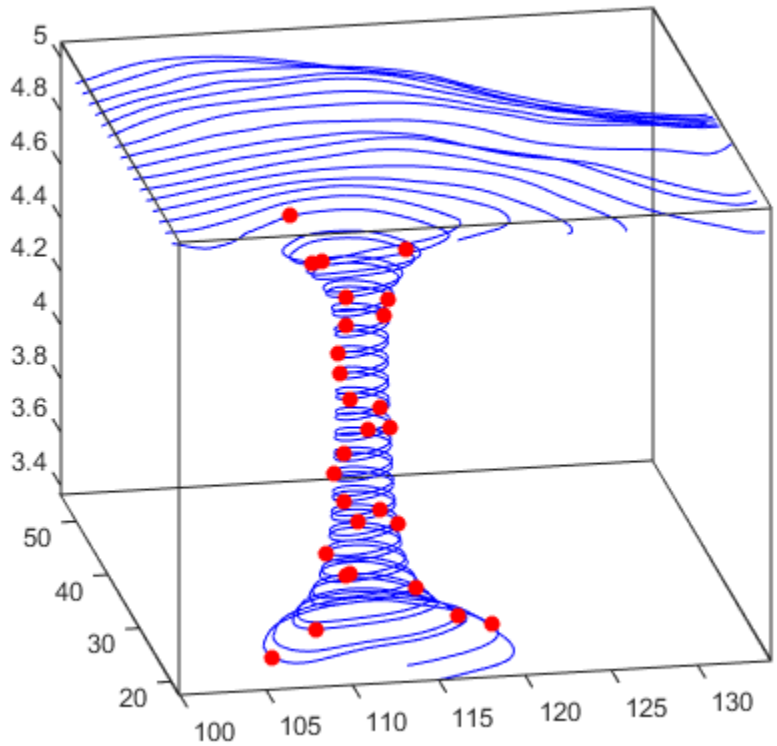
4. Calculate the Stream Particle Vertices

Determine the vertices along the stream line where a particle will 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.

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,0.01);
streamparticles(iverts,15,...
    'Animate',10,...
    'ParticleAlignment','on');
```


5
4.8
4.6
4.4
4.2
4
3.8
3.6
3.4
50
40
30
20
100
105
110
115
120
125
130

Vector Field Displayed with Cone Plots

In this section...

"What Cone Plots Can Show" on page 3-40
"1. Create an Isosurface" on page 3-40
"2. Add Isocaps to the Isosurface" on page 3-40
"3. Create the First Set of Cones" on page 3-41
"4. Create Second Set of Cones" on page 3-41
"5. Define the View" on page 3-41
"6. Add Lighting" on page 3-42

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)
hiso.FaceColor = 'red';
hiso.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);
h1.FaceColor = 'blue';
h1.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);
h2.FaceColor = 'green';
h2.EdgeColor = 'none';
```

5. Define the View

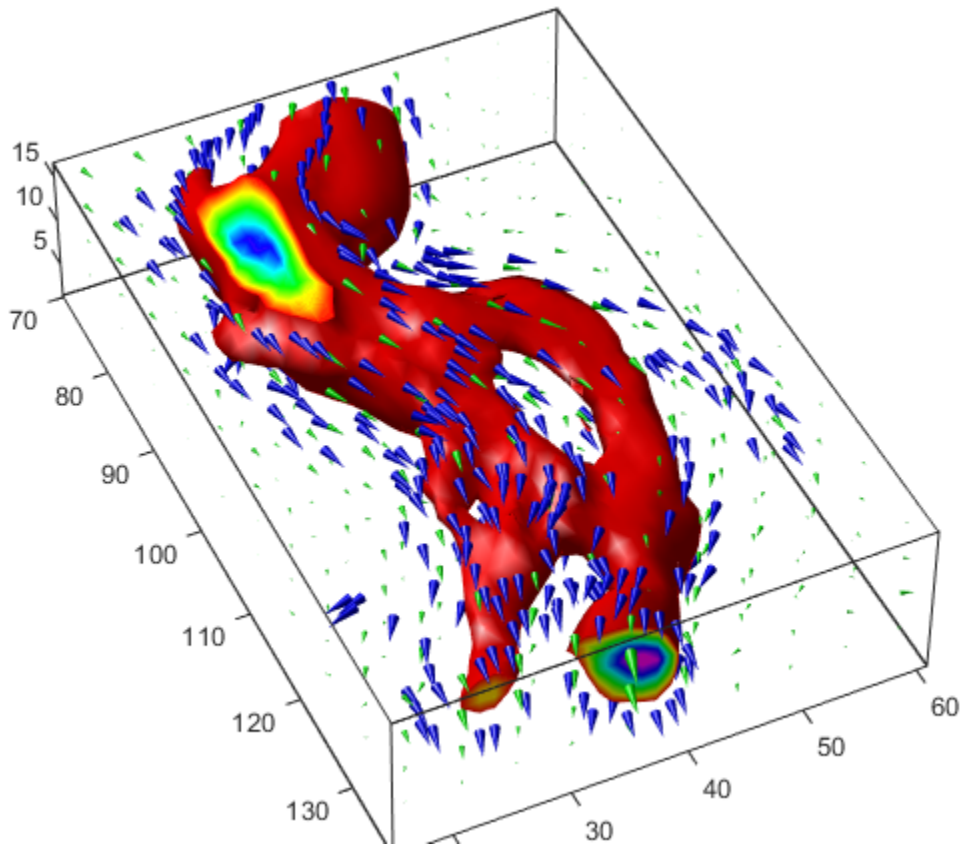
1. Use the `axis` command to set the axis limits equal to the minimum and maximum values of the data and enclose the graph in a box to improve the sense of a volume (`box`).
2. Set the projection type to perspective to create a more natural view of the volume. Set the viewpoint and zoom in to make the scene larger (`camproj`, `camzoom`, `view`).

```
axis tight
set(gca,'BoxStyle','full','Box','on')
camproj perspective
camzoom(1.25)
view(65,45)
```

6. Add Lighting

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

```
camlight(-45,45)
hcap.AmbientStrength = 0.6;
lighting gouraud
```



Visualizing Volume Data

This example shows several methods for visualizing volume data in MATLAB®.

Display Isosurface

An *isosurface* is a surface where all the points within a volume of space have a constant value. Use the `isosurface` function to generate the faces and vertices for the outside of the surface and the `isocaps` function to generate the faces and vertices for the end caps of the volume. Use the `patch` command to draw the volume and its end caps.

```
load mri D
D = squeeze(D);
limits = [NaN NaN NaN NaN NaN 10]
```

```
limits = 1×6

   NaN   NaN   NaN   NaN   NaN    10
```

```
[x, y, z, D] = subvolume(D, limits);
[fo,vo] = isosurface(x,y,z,D,5);
[fe,ve,ce] = isocaps(x,y,z,D,5);
figure
p1 = patch('Faces', fo, 'Vertices', vo);
p1.FaceColor = 'red'
```

```
p1 = 
  Patch with properties:

    FaceColor: [1 0 0]
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [23351×3 double]
     Vertices: [12406×3 double]

  Show all properties
```

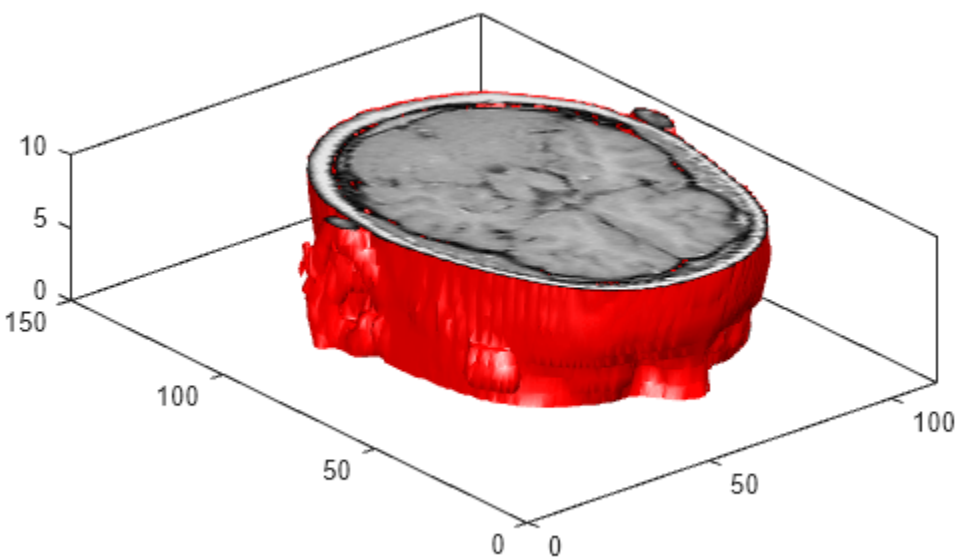
```
p1.EdgeColor = 'none'
```

```
p1 = 
  Patch with properties:

    FaceColor: [1 0 0]
    FaceAlpha: 1
    EdgeColor: 'none'
    LineStyle: '-'
        Faces: [23351×3 double]
     Vertices: [12406×3 double]

  Show all properties
```

```
p2 = patch('Faces', fe, 'Vertices', ve, ...
   'FaceVertexCData', ce)
```

```
p2 =
  Patch with properties:

    FaceColor: [0.1294 0.1294 0.1294]
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [27265×3 double]
     Vertices: [14250×3 double]

  Show all properties
```

```
p2.FaceColor = 'interp';
p2.EdgeColor = 'none';

view(-40,24)
daspect([1 1 0.3])
colormap(gray(100))
box on

camlight(40,40)
camlight(-20,-10)
lighting gouraud
```

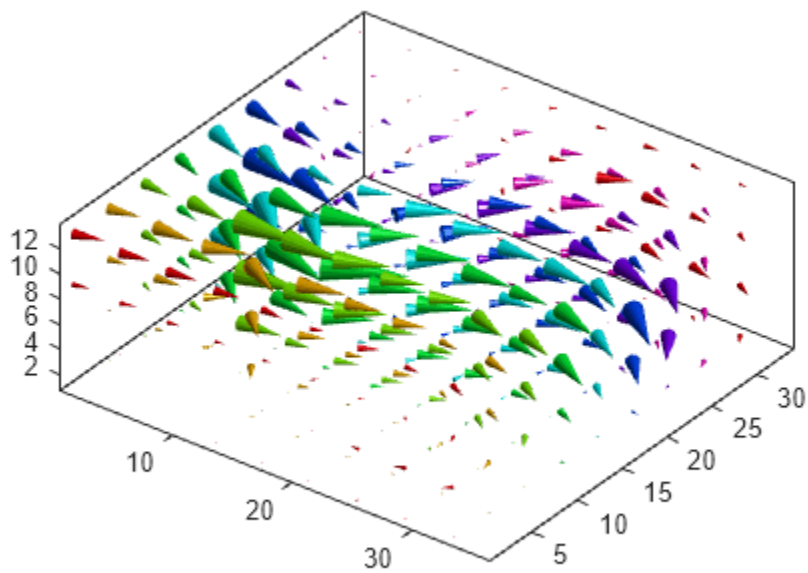


Create Cone Plot

The `coneplot` command plots velocity vectors as cones at *x*, *y*, *z* points in a volume. The cones represent the magnitude and direction of the vector field at each point.

```
cla
load wind u v w x y z
[m,n,p] = size(u)
```

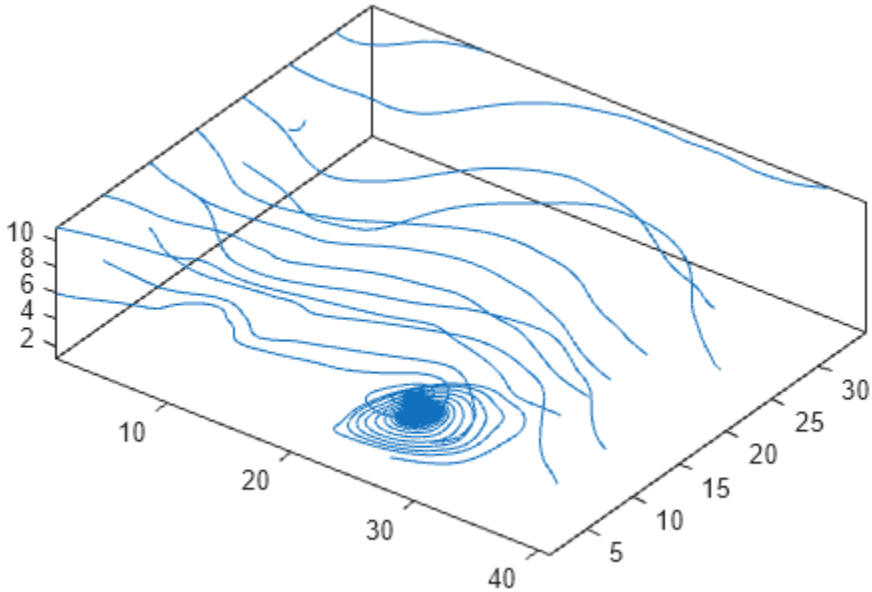
```
m =
35

n =
41

p =
15

[Cx, Cy, Cz] = meshgrid(1:4:m,1:4:n,1:4:p);
h = coneplot(u,v,w,Cx,Cy,Cz,y,4);
set(h,'EdgeColor', 'none')

axis tight equal
view(37,32)
box on
colormap(hsv)
light
```



Plot Streamlines

The `streamline` function plots streamlines for a velocity vector at *x*, *y*, *z* points in a volume to illustrate the flow of a 3-D vector field.

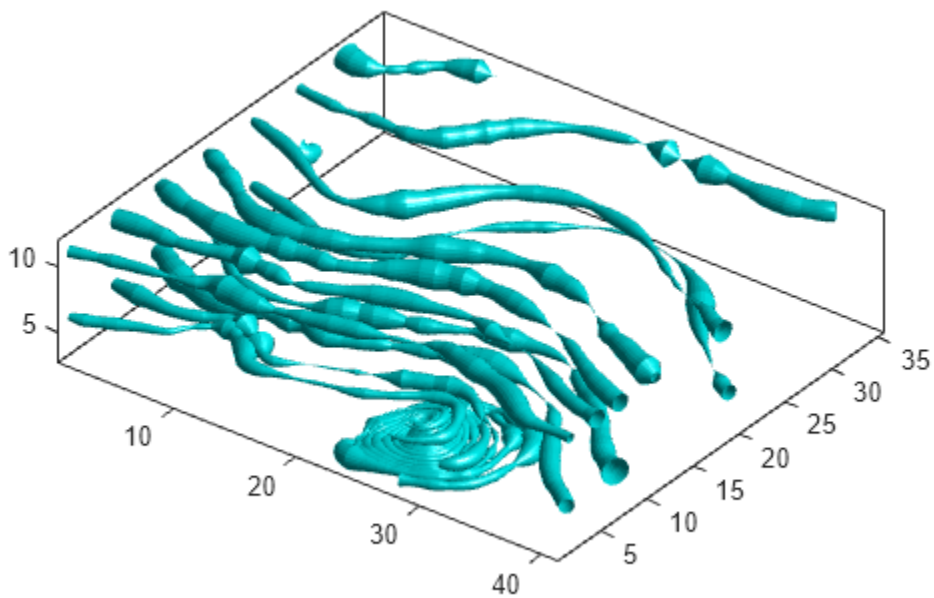
```
cla

[m,n,p] = size(u);
[Sx, Sy, Sz] = meshgrid(1,1:5:n,1:5:p);
streamline(u,v,w,Sx,Sy,Sz)

axis tight equal
view(37,32)
box on
```



Plot Streamtubes

The `streamtube` function plots streamtubes for a velocity vector at *x, y, z* points in a volume. The width of the tube is proportional to the normalized divergence of the vector field at each point.

```
cla

[~,n,p] = size(u);
[Sx, Sy, Sz] = meshgrid(1,1:5:n,1:5:p);
h = streamtube(u,v,w,Sx,Sy,Sz);
set(h, 'FaceColor', 'cyan')
set(h, 'EdgeColor', 'none')

axis tight equal
view(37,32)
box on
light
```



Combine Volume Visualizations

Combine volume visualization in a single plot to get a more comprehensive picture of a velocity field within a volume.

```
cla
spd = sqrt(u.*u + v.*v + w.*w);
[fo,vo] = isosurface(x,y,z,spd,40);
[fe,ve,ce] = isocaps(x,y,z,spd,40);
p1 = patch('Faces', fo, 'Vertices', vo);
p1.FaceColor = 'red'
```

```
p1 =
  Patch with properties:

    FaceColor: [1 0 0]
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [5340×3 double]
     Vertices: [2727×3 double]

  Show all properties
```

```
p1.EdgeColor = 'none'
```

```
p1 =
  Patch with properties:

    FaceColor: [1 0 0]
    FaceAlpha: 1
    EdgeColor: 'none'
    LineStyle: '-'
```

```
        Faces: [5340×3 double]
     Vertices: [2727×3 double]

  Show all properties
```

```
p2 = patch('Faces', fe, 'Vertices', ve, ...
   'FaceVertexCData', ce)
```

```
p2 =
  Patch with properties:

    FaceColor: [0.1294 0.1294 0.1294]
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [464×3 double]
     Vertices: [301×3 double]

  Show all properties
```

```
p2.FaceColor = 'interp'
```

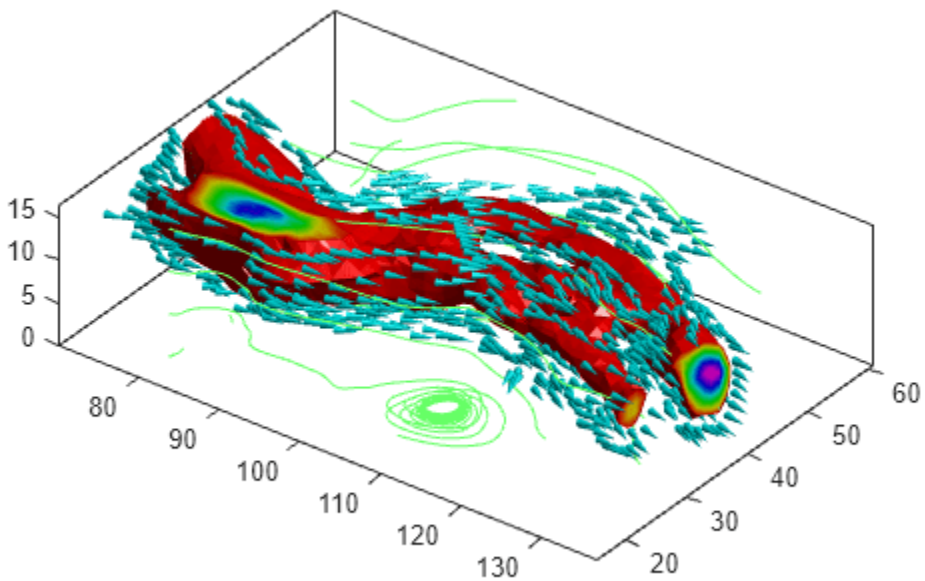
```
p2 =
  Patch with properties:

    FaceColor: 'interp'
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [464×3 double]
     Vertices: [301×3 double]

  Show all properties
```

```
p2.EdgeColor = 'none'
```

```
p2 =
  Patch with properties:

    FaceColor: 'interp'
    FaceAlpha: 1
    EdgeColor: 'none'
    LineStyle: '-'
        Faces: [464×3 double]
     Vertices: [301×3 double]

  Show all properties
```

```
[fc, vc] = isosurface(x, y, z, spd, 30);
[fc, vc] = reducepatch(fc, vc, 0.2);
h1 = coneplot(x,y,z,u,v,w,vc(:,1),vc(:,2),vc(:,3),3);
h1.FaceColor = 'cyan';
h1.EdgeColor = 'none';

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

```
h2 = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h2, 'Color', [.4 1 .4])

axis tight equal
view(37,32)
box on
light
```



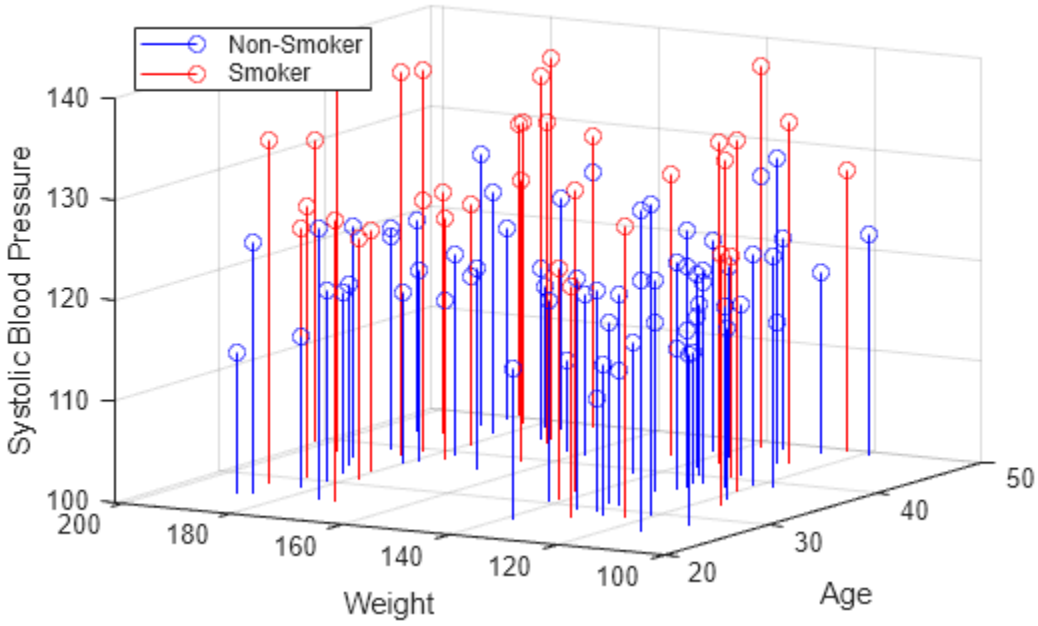
Visualizing Four-Dimensional Data

This example shows several techniques to visualize four dimensional (4-D) data in MATLAB®.

Visualize 4-D Data with One Discrete Variable

Sometimes data has a variable which is discrete with only a few possible values. You can create multiple plots of the same type for data in each discrete group. For example, use the `stem3` function to see the relationship between three variables where the fourth variable divides the population into discrete groups.

```
load patients Smoker Age Weight Systolic                          % load data

nsIdx = Smoker == 0;
smIdx = Smoker == 1;

figure
stem3(Age(nsIdx), Weight(nsIdx), Systolic(nsIdx), 'Color', 'b')    % stem plot for non-smokers
hold on
stem3(Age(smIdx), Weight(smIdx), Systolic(smIdx), 'Color', 'r')    % stem plot for smokers
hold off

view(-60,15)
zlim([100 140])

xlabel('Age')                                                     % add labels and a legend
ylabel('Weight')
zlabel('Systolic Blood Pressure')
legend('Non-Smoker', 'Smoker', 'Location', 'NorthWest')
```

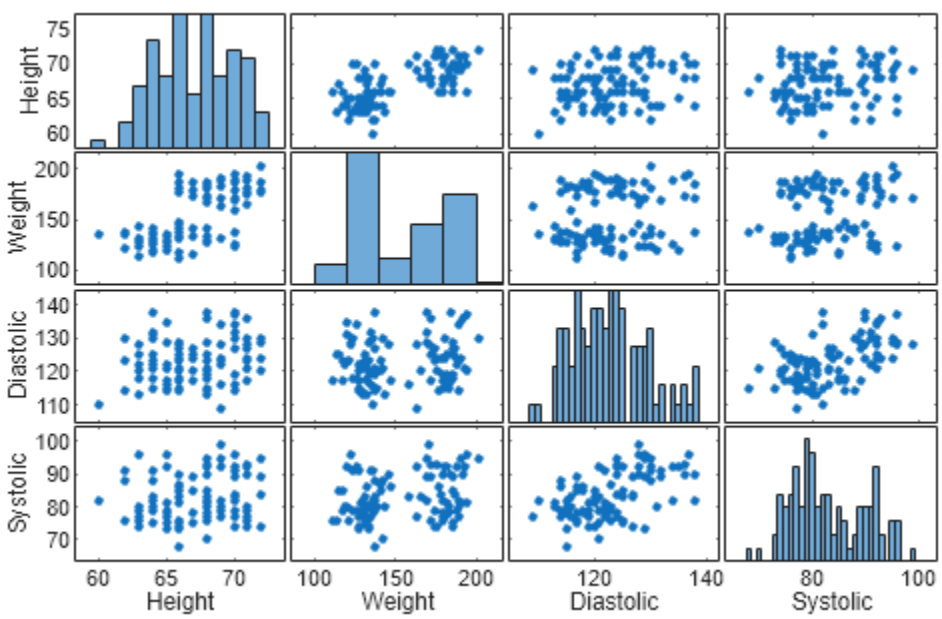


Visualize 4-D Data with Multiple Plots

With a large data set you might want to see if individual variables are correlated. You can use the `plotmatrix` function to create an *n* by *n* matrix of plots to see the pair-wise relationships between the variables. The `plotmatrix` function returns two outputs. The first output is a matrix of the line objects used in the scatter plots. The second is a matrix of the axes objects that are created.

The `plotmatrix` function can also be used for higher order data sets.

```
load patients Height Weight Diastolic Systolic    % load data

labels = {'Height' 'Weight' 'Diastolic' 'Systolic'};
data = [Height Weight Systolic Diastolic];

[h,ax] = plotmatrix(data);                        % create a 4 x 4 matrix of plots
for i = 1:4                                       % label the plots
  xlabel(ax(4,i), labels{i})
  ylabel(ax(i,1), labels{i})
end
```



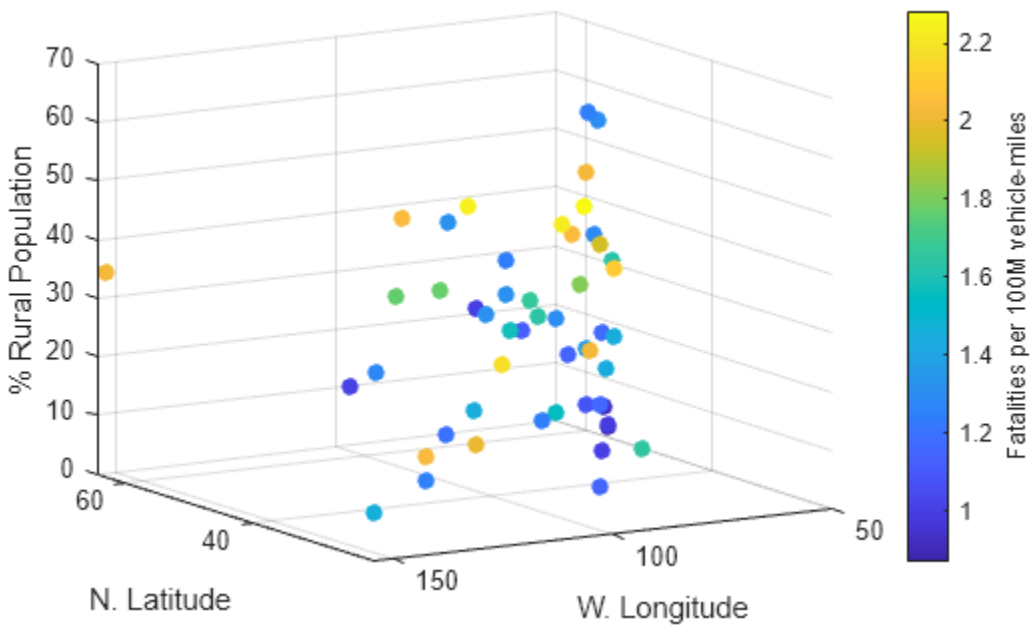
Visualize Function of Three Variables

For many kinds of four dimensional data, you can use color to represent the fourth dimension. This works well if you have a function of three variables.

For example, represent highway deaths in the United States as a function of longitude, latitude, and if the location is rural or urban. The *x*, *y*, and *z* values in the plot represent these three variables. The color represents the number of highway deaths.

```
cla
load accidents hwydata                            % load data

long = -hwydata(:,2);                             % longitude data
```

```
lat = hwydata(:,3);                                  % latitude data
rural = 100 - hwydata(:,17);                         % percent rural data
fatalities = hwydata(:,11);                          % fatalities data

scatter3(long,lat,rural,40,fatalities,'filled')      % draw the scatter plot
ax = gca;
ax.XDir = 'reverse';
view(-31,14)
xlabel('W. Longitude')
ylabel('N. Latitude')
zlabel('% Rural Population')

cb = colorbar;                                       % create and label the colorbar
cb.Label.String = 'Fatalities per 100M vehicle-miles';
```

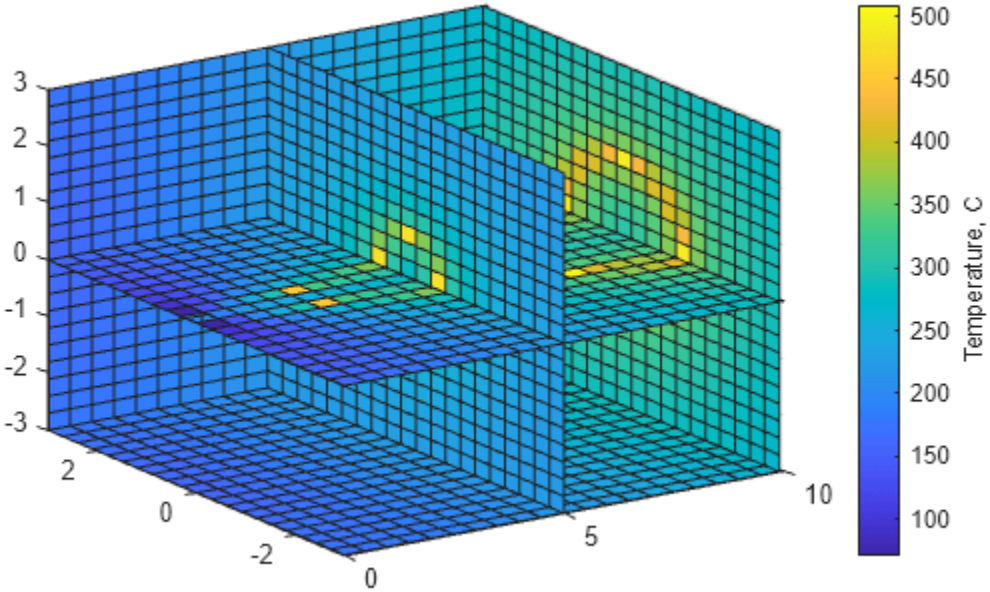


Visualize Data in a Volume

Your data may contain a measured value for a physical object such as temperature in a pipe. In this cases, the physical dimensions can be represented as a volume with color used to represent the magnitude of the measurement. For example, use the `slice` function to show the value of the measured variable at cross-sections within the volume.

```
load fluidtemp x y z temp                        % load data

xslice = [5 9.9];                                % define the cross sections to view
yslice = 3;
zslice = ([-3 0]);

slice(x, y, z, temp, xslice, yslice, zslice)     % display the slices
ylim([-3 3])
view(-34,24)
```

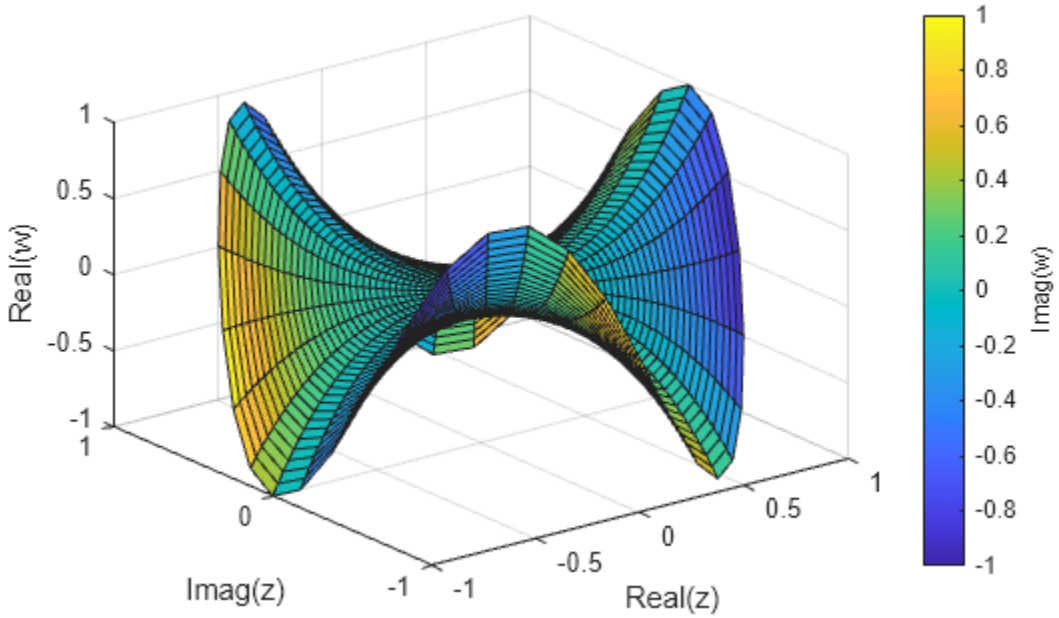
```
cb = colorbar;                                  % create and label the colorbar
cb.Label.String = 'Temperature, C';
```



Plot the Function of a Complex Variable

A complex function has an input with real and imaginary parts and an output with real and imaginary parts. You can use a three dimensional plot with color to represent the complex function. In this case the *x* and *y* axes represent the real and imaginary parts of the input. The *z* axis represents the real part of the output and the color represents the imaginary part of the output.

```
r = (0:0.025:1)';                           % create a matrix of complex inputs
theta = pi*(-1:0.05:1);
z = r*exp(1i*theta);
w = z.^3;                                   % calculate the complex outputs

surf(real(z),imag(z),real(w),imag(w))       % visualize the complex function using surf
xlabel('Real(z)')
ylabel('Imag(z)')
zlabel('Real(w)')
cb = colorbar;
cb.Label.String = 'Imag(w)';
```


Real(w)
1
0.5
0
-0.5
-1
Imag(z)
1
0
-1
Real(z)
-1
-0.5
0
0.5
1
Imag(w)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

Displaying Complex Three-Dimensional Objects

This example shows how to create and display a complex three dimensional object and control its appearance.

Get Geometry of Object

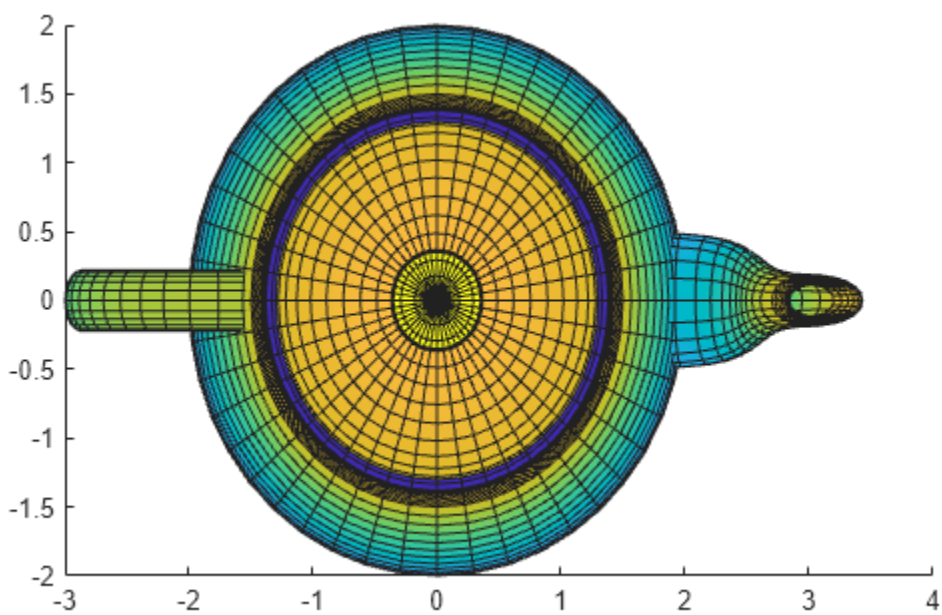
This example uses a graphics object called the Newell teapot. The vertex, face, and color index data for the teapot are calculated by the `teapotData` function. Since the teapot is a complex geometric shape, there are a large number of vertices (4608) and faces (3872) returned by the function.

```
[verts, faces, cindex] = teapotGeometry;
```

Create Teapot Patch Object

Using the geometry data, draw the teapot using the `patch` command. The `patch` command creates a patch object.

```
figure
p = patch('Faces',faces,'Vertices',verts,'FaceVertexCData',cindex,'FaceColor','interp')
```

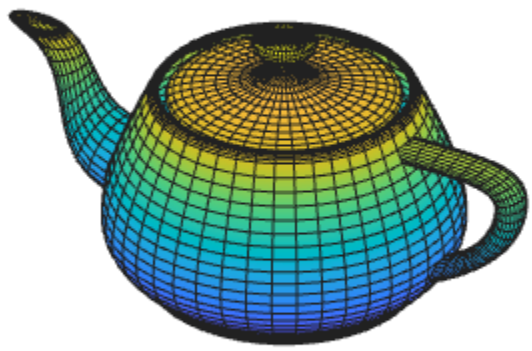



```
p =
  Patch with properties:

    FaceColor: 'interp'
    FaceAlpha: 1
    EdgeColor: [0.1294 0.1294 0.1294]
    LineStyle: '-'
        Faces: [3872×4 double]
     Vertices: [4608×3 double]
```

```
  Show all properties
```

Use the `view` command to change the orientation of the object.

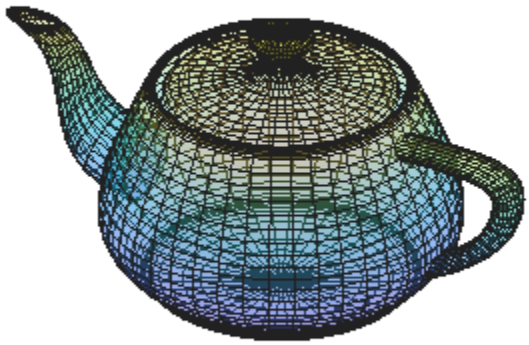
```
view(-151,30)      % change the orientation
axis equal off     % make the axes equal and invisible
```



Adjust Transparency

Make the object transparent using the `FaceAlpha` property of the patch object.

```
p.FaceAlpha = 0.3;   % make the object semi-transparent
```



If the `FaceColor` property is set to 'none', then the object appears as a wire frame diagram.

```
p.FaceColor = 'none';    % turn off the colors
```

Change Colormap

Change the colors for the object using the `colormap` function.

```
p.FaceAlpha = 1;           % remove the transparency
p.FaceColor = 'interp';    % set the face colors to be interpolated
p.LineStyle = 'none';      % remove the lines
colormap(copper)           % change the colormap
```

Light the Object

Add a light to make the object appear more realistic.

```
l = light('Position',[-0.4 0.2 0.9],'Style','infinite')
```

```
l = 
  Light with properties:

       Color: [1 1 1]
       Style: 'infinite'
    Position: [-0.4000 0.2000 0.9000]
     Visible: on

  Show all properties
```

```
lighting gouraud
```

These properties of the patch object affect the strength of the light and the reflective properties of the object:

- `AmbientStrength` - controls the strength of ambient light
- `DiffuseStrength` - controls the strength of diffuse light
- `SpecularStrength` - controls the strength of reflected light
- `SpecularExponent` - controls the harshness of reflected light
- `SpecularColorReflectance` - controls how reflected color is calculated.

You can set these properties individually. To set these properties to a predetermined set of values that approximate the appearance of metal, shiny, or dull material, use the `material` command.

```
material shiny
```

Adjust the position of the light using its `Position` property. The position is in *x*, *y*, *z* coordinates.

```
l.Position = [-0.1 0.6 0.8]
```

```
l =
  Light with properties:

       Color: [1 1 1]
       Style: 'infinite'
```

```
  Position: [-0.1000 0.6000 0.8000]
   Visible: on

Show all properties
```

Displaying Topographic Data

This example shows several ways to represent the Earth's topography. The data used in this example are available from the National Geophysical Data Center, NOAA US Department of Commerce under data announcement 88-MGG-02.

About the Topography Data

The data file, `topo.mat`, contains topographic data. `topo` is the altitude data and `topomap1` is a colormap for the altitude.

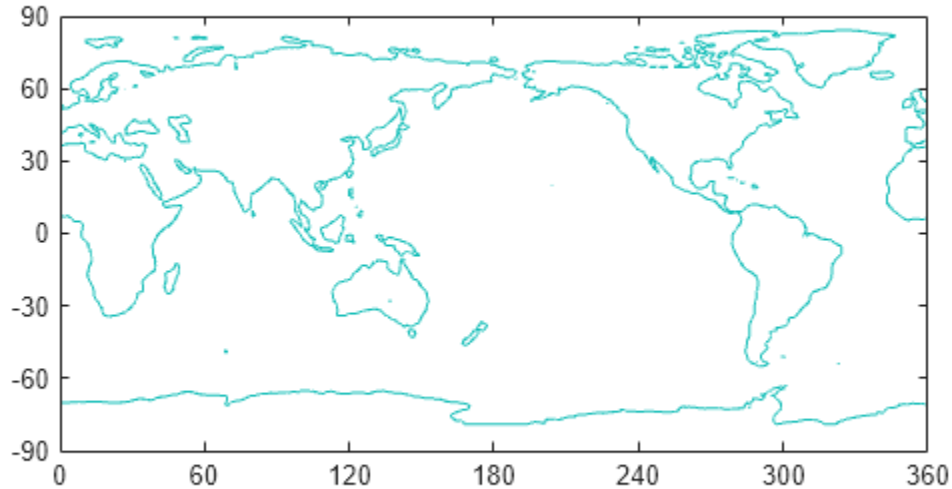
```
load topo topo topomap1    % load data
whos('topo','topomap1')
```

```
  Name            Size              Bytes  Class     Attributes

  topo          180x360            518400  double
  topomap1       64x3                1536  double
```

Create Contour Plot

One way to visualize topographic data is to create a contour plot. To show the outline of the Earth's continents, plot points that have zero altitude. The first three input arguments to `contour` specify the X, Y, and Z values on the contour plot. The fourth argument specifies the contour levels to plot.

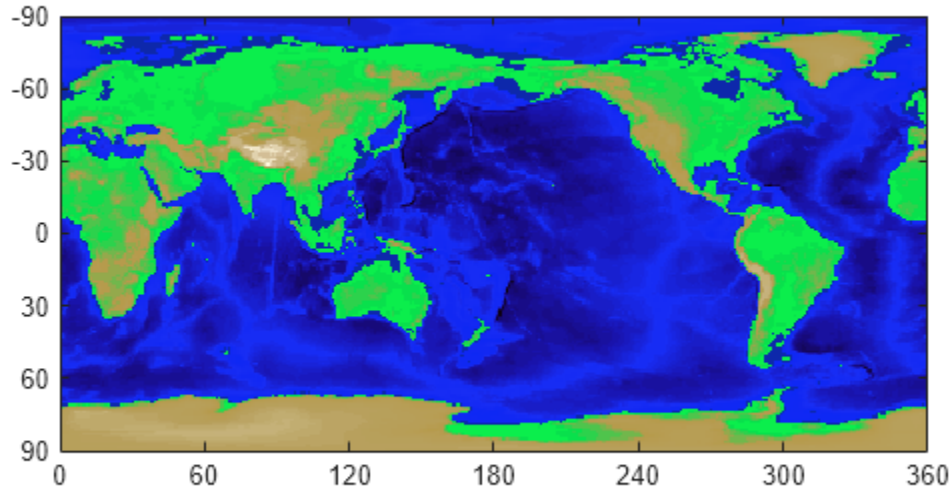
```
x = 0:359;                                % longitude
y = -89:90;                               % latitude

figure
contour(x,y,topo,[0 0])

axis equal                                % set axis units to be the same size
box on                                    % display bounding box

ax = gca;                                 % get current axis
ax.XLim = [0 360];                        % set x limits
ax.YLim = [-90 90];                       % set y limits
ax.XTick = [0 60 120 180 240 300 360];    % define x ticks
ax.YTick = [-90 -60 -30 0 30 60 90];      % define y ticks
```



View Data as Image

You can create an image of the topography using the elevation data and a custom colormap. The topography data is treated as an index into the custom colormap. Set the `CDataMapping` of the image to `'scaled'` to linearly scale the data values to the range of the colormap. In this colormap, the shades of green show the altitude data, and the shades of blue represent depth below sea level.

```
image([0 360],[-90 90], flip(topo), 'CDataMapping', 'scaled')
colormap(topomap1)

axis equal                                  % set axis units to be the same size

ax = gca;                                   % get current axis
ax.XLim = [0 360];                          % set x limits
ax.YLim = [-90 90];                         % set y limits
ax.XTick = [0 60 120 180 240 300 360];      % define x ticks
ax.YTick = [-90 -60 -30 0 30 60 90];        % define y ticks
```



Use Texture Mapping

Texture mapping maps a 2-D image onto a 3-D surface. To map the topography to a spherical surface, set the color of the surface, specified by the `CData` property, to the topographic data and set the `FaceColor` property to `'texturemap'`.

```
clf
[x,y,z] = sphere(50);           % create a sphere
s = surface(x,y,z);             % plot spherical surface

s.FaceColor = 'texturemap';     % use texture mapping
s.CData = topo;                 % set color data to topographic data
s.EdgeColor = 'none';           % remove edges
s.FaceLighting = 'gouraud';     % preferred lighting for curved surfaces
s.SpecularStrength = 0.4;       % change the strength of the reflected light

light('Position',[-1 0 1])      % add a light

axis square off                 % set axis to square and remove axis
view([-30,30])                  % set the viewing angle
```