CHAPTER 3

Visualization

MATLAB provides extensive capabilities for visualizing your data. You can produce 2D plots, 3D plots, and animations; view images; and create histograms, contour and surface plots, as well as other graphical representations of your data. You are probably familiar with making simple 2D plots with lines and markers, and pie and bar charts, but you may not be aware of the additional possibilities made available by the MATLAB low-level routines that underpin the frequently used functions like `plot`. There are also interactive capabilities for editing plots and figures and adding annotations before printing or exporting them.

MATLAB excels in scientific visualization and in engineering visualization of 3D objects. Three-dimensional visualization is used to visualize data that is a function of two parameters, for example, the height on the surface of the Earth, or to visualize objects. The former is used in all areas of science and engineering. The latter is particularly useful in the design and simulation of any kind of machine including robots, aircraft, automobiles, and spacecraft.

Three-dimensional visualization of objects can be further divided into engineering visualization and photo-realistic visualization. The latter helps you understand what an object looks like and how it is engineered. When the inside of an object is considered, we move into the realm of solid modeling which is used for creating models suitable for the manufacturing of the object. Photo-realistic rendering focuses on the interaction of light with the object and the eye. MATLAB does provide some capabilities for lighting and camera interaction but does not provide true photo-realistic rendering.

The main plotting routines are organized into several categories in the command-line help:

graphics – Low-level routines for figures, axes, lines, text, and other graphics objects.

graph2d – Two-dimensional graphs like linear plots, log scale plots, and polar plots.

graph3d – Three-dimensional graphs like lines, meshes, and surfaces; control of color, lighting, and the camera.

specgraph – Specialized graphs, the largest category. Special 2D graphs like bar and pie charts and histograms, contour plots, special 3D plots, volume and vector visualization, image display, movies, and animation.

M. Paluszek and S. Thomas, *MATLAB Recipes*,
https://doi.org/10.1007/978-1-4842-6124-8_3

The online help has an entire top-level section devoted to graphics, including plots, formatting and annotation, images, printing and saving, graphics objects and performance, and major changes to plotting internals that occurred in R2014b.

A good command of these functions allows you to create very sophisticated graphics as well as to adapt them to different publication media, whether you need to adjust the dimensions, color, or font attributes of your plot. In this chapter, we will present recipes that cover what you need to know to use MATLAB graphics effectively. We don't have space to discuss every available plotting routine, and that is well covered in the available help, but we will cover the basic functionality and provide recipes for common usage.

3.1 Plotting Data Interactively from the MATLAB Desktop

Problem

You would like to plot data in your workspace but aren't sure of the best method for visualizing it.

Solution

You can use the PLOTS tab in the MATLAB desktop to plot data directly by selecting variables in the Workspace display as shown in Figure 3.1. You select from a variety of plot options, and MATLAB automatically only shows you those which are applicable to the selected data set.

How It Works

Let's create some sample data to demonstrate this interactive capability, which is a fairly new feature in MATLAB. We'll start with some trigonometric functions to create sample data that oscillates.

```
theta = linspace(0,4*pi);
y = sin(theta).*cos(2*theta) + 0.05*theta;
```

We now have two vector variables available in the workspace. Select the PLOTS tab in the desktop as shown in Figure 3.1, then select the `y` variable in the Workspace display. The variable will appear on the far left of the PLOTS tab area, and various plot icons in the ribbon

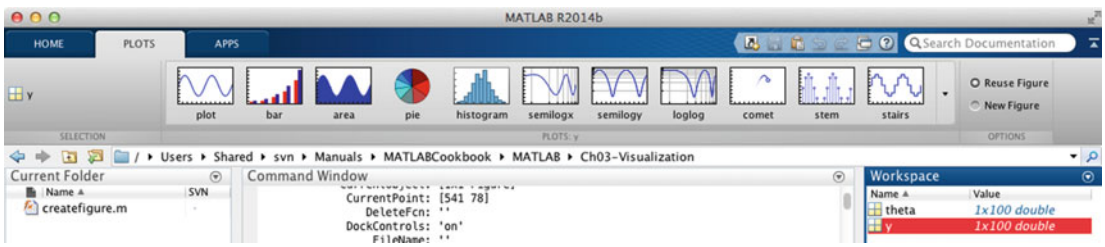

Figure 3.1: *PLOTS tab with plot icon ribbon.*

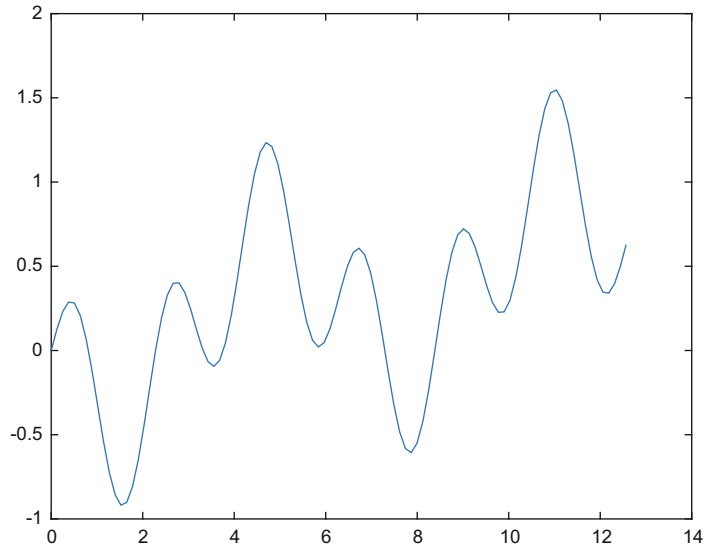

Figure 3.2: *Linear plot of trigonometric data.*

will become active: plot, bar, area, pie, and so on. Note the radio buttons on the far left for either reusing the current figure for the plot or creating a new figure.

Close all open figures with a `close all` and click the plot icon to create a new figure with a simple 2D plot of the data. Note that clicking the icon results in the plot command being printed to the command line:

```
>> plot(y)
```

The data is printed with linear indices along the x axis, as shown in Figure 3.2.

You simply click another plot icon to replot the data using a different function, and again the function call will be printed to the command line. The plot icons that are displayed are not all the plots available, but simply the default favorites from among all the many options; to see more icons, click the pop-up arrow at the right of the icon ribbon. The available plot types are organized by category, and there is a Catalog button that you can press to bring up a dedicated plot catalog window with the documentation for each function.

To plot our data `y` against our input `theta`, you need to select both variables in the workspace view. They will both be displayed in the plot ribbon with a button shown to reverse their order. Now click an area plot to get a plot with the angle on the x axis as shown in Figure 3.3.

```
>> area(theta,y)
```

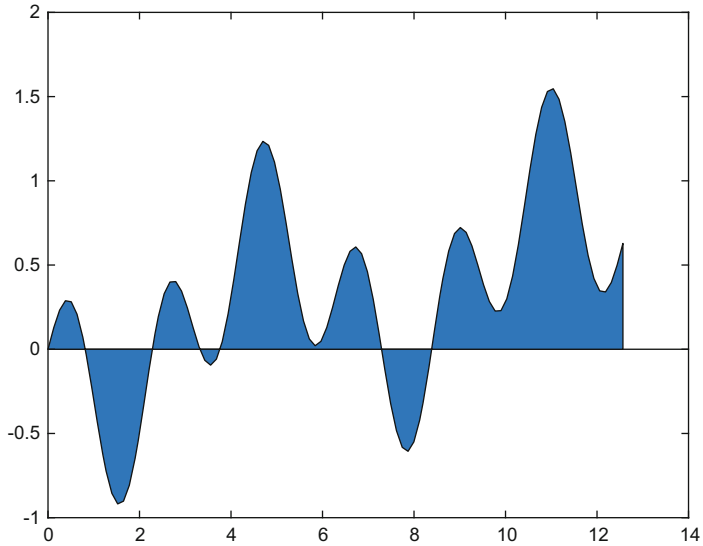

Figure 3.3: *Parametric area plot of trigonometric data.*

Note how this time the x-axis range is from 0 to 4π as expected.

You can annotate the plot interactively with arrows and text, add subplots, change line properties, and more using the Plot Edit toolbar and Figure Palette window shown in Figure 3.4. These are available from the View menu of the figure window and by clicking the "Show Plot Tools" button in the standard Figure Toolbar. For example, using the plot tools, we can select the axes, double-click it to open the property editor, type in an X Label, and turn on grid lines. We can add another subplot, plot the values of theta against linear indices, and then change the plot type to a stem plot, all from this window. See Figure 3.4.

The same changes can be made programmatically as will be shown in the following recipes. In fact, you can generate code from the Figure Palette, and MATLAB will create a function with all the commands necessary to replicate your figure from your data. The Generate Code command is under the File menu of the window. This allows you to interactively create a visualization that works with some example data and then programmatically adapt it to your toolbox. MATLAB calls the new autogenerated function `createfigure`. You can see the use of the following functions: `figure`, `axes`, `box`, `hold`, `ylabel`, `xlabel`, `title`, `area`, `stem`, and `annotation`.

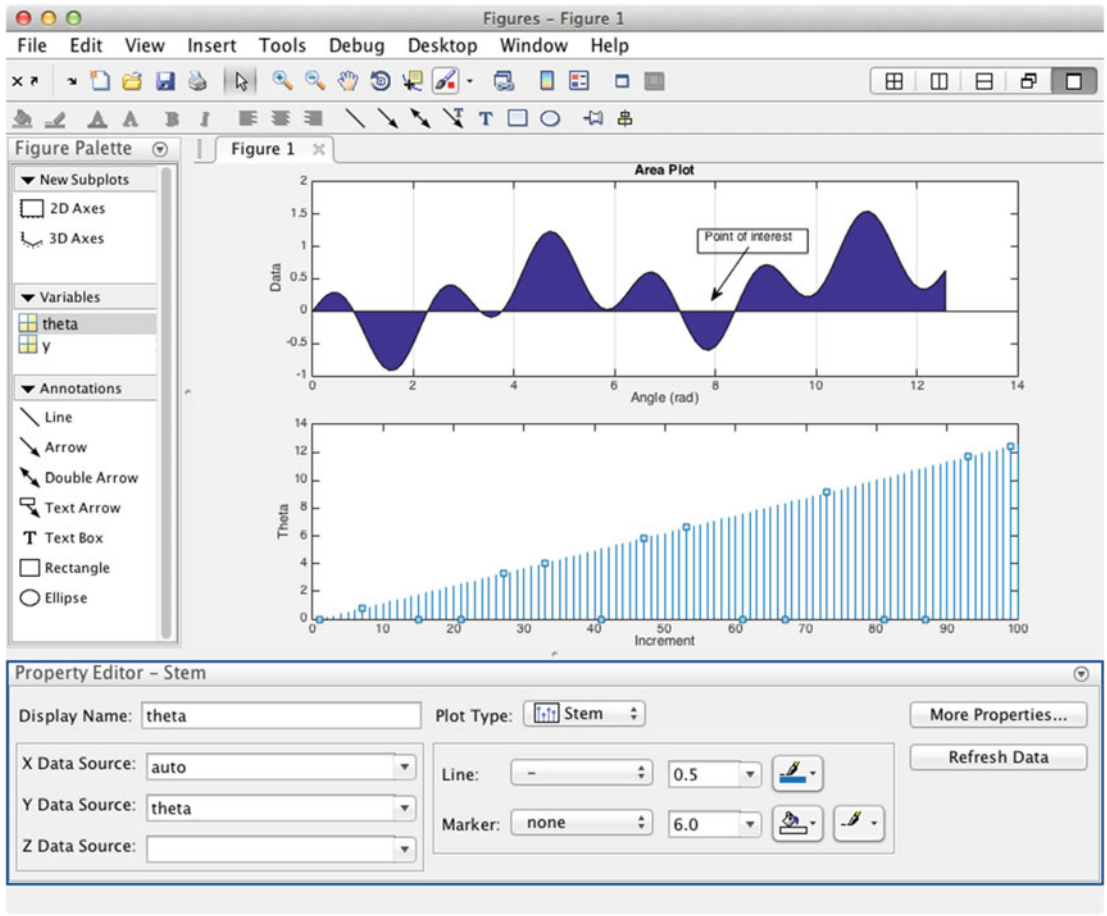

Figure 3.4: *Plot of trigonometric data in the Figure Palette.*

createfigure.m

```
function createfigure(X1, yvector1)
%CREATEFIGURE(X1, YVECTOR1) Autogenerated figure code.
%  X1:  area x
%  YVECTOR1:  area yvector

%  Auto-generated by MATLAB on 03-Jun-2015 14:32:43

% Create figure
figure1 = figure;

% Create axes
axes1 = axes('Parent',figure1,'XGrid','on','OuterPosition',[0 0.5 1
    0.5]);
box(axes1,'on');
```

```
14 hold(axes1,'on');
15
16 % Create ylabel
17 ylabel('Data');
18
19 % Create xlabel
20 xlabel('Angle (rad)');
21
22 % Create title
23 title('Area Plot');
24
25 % Create area
26 area(X1,yvector1,'DisplayName','Area','Parent',axes1);
27
28 % Create axes
29 axes2 = axes('Parent',figure1,'OuterPosition',[0 0 1 0.5]);
30 box(axes2,'on');
31 hold(axes2,'on');
32
33 % Create ylabel
34 ylabel('Theta');
35
36 % Create xlabel
37 xlabel('Increment');
38
39 % Create stem
40 stem(X1,'DisplayName','theta','Parent',axes2,'Marker','none',...
41   'Color',[0 0.447 0.741]);
42
43 % Create textarrow
44 annotation(figure1,'textarrow',[0.609822646657571
       0.568894952251023],...
45   [0.827828828828829 0.717117117117118]);
46
47 % Create textbox
48 annotation(figure1,'textbox',...
49   [0.553888130968622 0.814895792699917 0.120787482806052
        0.0489690721649485],...
50   'String',{'Point of interest'});
```

Note that this code did not in fact use the `subplot` function, but rather the option to specify the exact axes location in the figure with the `'OuterPosition'` property. Note also how the units of the axes position and of the annotations are between 0 and 1, that is, normalized. This is in fact an option for axes, as can be seen by the following call using `gca` to get the handle to the current axes:

```
>> set(gca,'units')
    'inches'
    'centimeters'
    'characters'
```

```
    'normalized'
    'points'
    'pixels'
```

Using other units may be helpful for certain applications, but normalized units are always the default.

There are additional interactive buttons in the Figure Toolbar we should mention:

- Zoom in
- Zoom out
- Hand tool – Move an object in the plane of the figure
- Rotate tool – Rotate the view
- Data cursor
- Brush/select data
- Colorbar
- Legend

The hand and rotate tools are very helpful with 3D data. The data cursor displays the values of a plot point right in the figure. The brush highlights a segment of data using a contrast color of your choosing using the colors pop-up. The colorbar and legend buttons serve as on/off switches.

3.2 Incrementally Annotate a Plot

Problem

You need to annotate a curve in a plot at a subset of points on the curve.

Solution

Use the `text` function to annotate the plot.

How It Works

We will call `text` within a for loop in `AnnotatePlot`. Use `sprintf` to create the text for the annotations, which gives you control over the formatting of any numbers. In this case, we will use %d for integer display. `linspace` creates an evenly spaced index array into the data to give us the selected points to annotate, in this case, five points. `linspace` is used to produce evenly spaced points.

AnnotatePlot.m

```
8  %% Parameters
9  nPoints = 5; % Number of plot points to have annotations
10
11 %% Create the line
12 v       = [1;2;3];
```

```
t        = linspace(0,1000);
r        = [v(1)*t;v(2)*t;v(3)*t];

%% Create the figure and plot
s = 'Annotated Plot';
h = figure('name',s);
plot3(r(1,:),r(2,:),r(3,:));
xlabel('X');
ylabel('Y');
zlabel('Z');
title(s)
grid

%% Add the annotations
n     = length(t);
j     = ceil(linspace(1,n,nPoints));

for k = j
  text(r(1,k), r(2,k), r(3,k), sprintf('- Time %d',floor(t(k))));
end
```

Note that we passed the index array `j` directly to the loop index `k`. Figure 3.5 shows the annotated plot. We create a three-dimensional straight line to annotate.

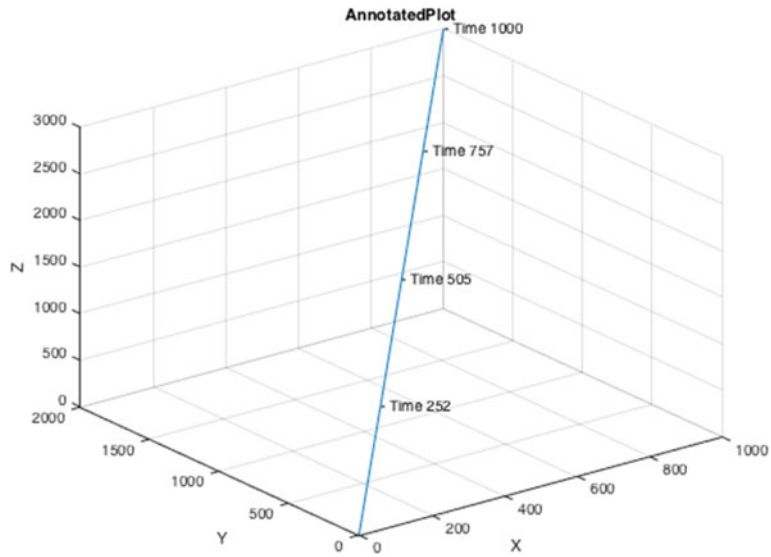

Figure 3.5: *Annotated three-dimensional plot.*

3.3 Create a Custom Plot Page with Subplot

Problem

You need multiple plots of your data for a particular application, and as you rerun your script, they are cluttering your screen and hogging memory. We often create many dozens of plots as we work on our commercial toolboxes.

Solution

Create a single plot with several subplots on it so you only need one figure to see the results of one run of your application.

How It Works

The `subplot` function allows you to create a symmetric array of plots in a figure in two dimensions. You generate an m-by-n array of small axes which are spaced in the figure automatically. A good example is a 3D trajectory with views from different angles. We can create a plot with a 2 x 2 array of axes, with the 3D plot in the lower left-hand corner and views from each direction around it. The function is `QuadPlot`. It has a built-in demo creating the figure in Figure 3.6.

Note that you must use the size of your axes array, in this case (2,2), in each call to `subplot`.

QuadPlot.m

```
%% QUADPLOT Create a quad plot page using subplot.
% This creates a 3D view and three 2D views of a trajectory in one
    figure.
%% Form
%  QuadPlot( x )
%% Input
%   x   (3,:)    Trajectory data
%
%% Output
% None. But you may want to return the graphics handles for further
    programmatic
% customization.
%

function QuadPlot(x)

if nargin == 0
  disp('Demo of QuadPlot');
  th = logspace(0,log10(4*pi),101);
  in = logspace(-1,0,101);
  x = [sin(th).*cos(in);cos(th).*cos(in);sin(in)];
  QuadPlot(x);
  return;
```

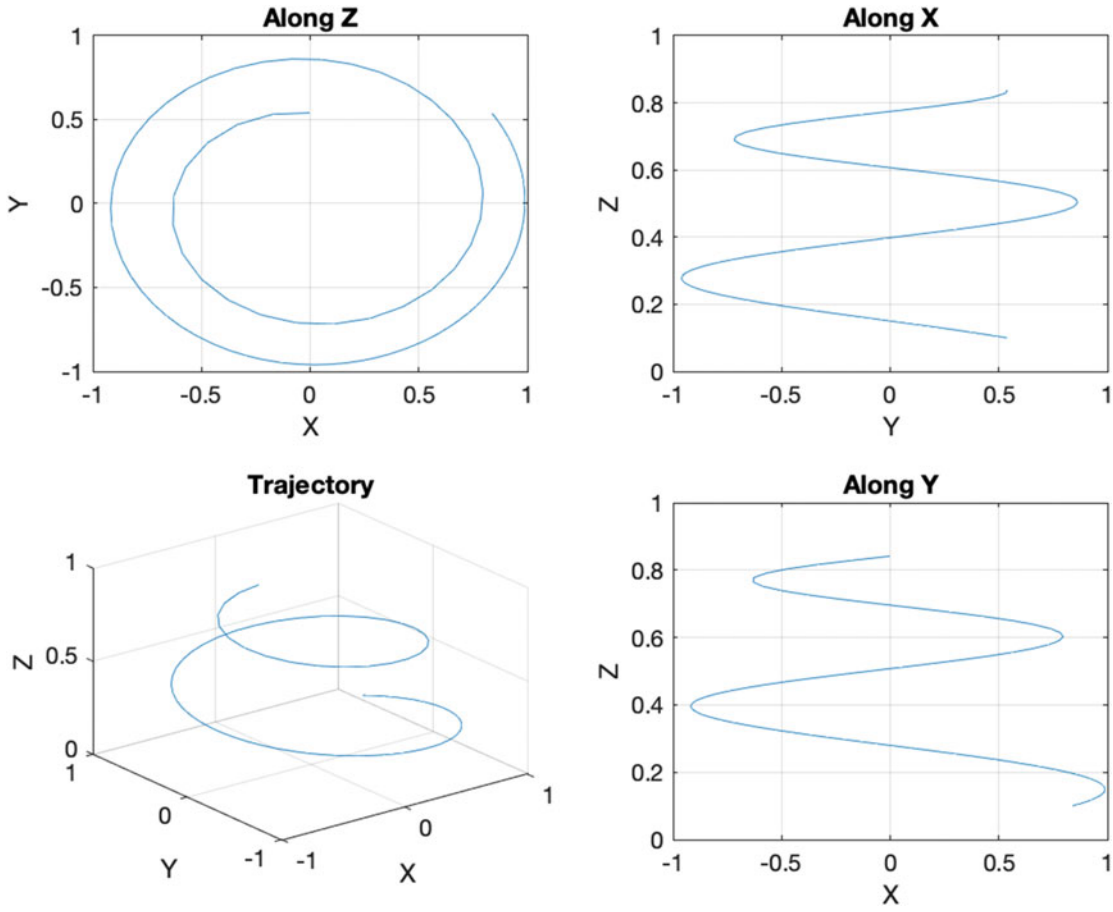

Figure 3.6: *QuadPlot using subplot for axes placement.*

```
22  end
23
24  h = figure('Name','QuadPage');
25  set(h,'InvertHardcopy','off')
26
27  % Use subplot to create plots
28  subplot(2,2,3)
29  plot3(x(1,:),x(2,:),x(3,:));
30  xlabel('X')
31  ylabel('Y')
32  zlabel('Z')
33  grid on
34  title('Trajectory')
35  rotate3d on
36
37  subplot(2,2,1)
```

```
38 plot(x(1,:),x(2,:));
39 xlabel('X')
40 ylabel('Y')
41 grid on
42 title('Along Z')
43
44 subplot(2,2,2)
45 plot(x(2,:),x(3,:));
46 xlabel('Y')
47 ylabel('Z')
48 grid on
49 title('Along X')
50
51 subplot(2,2,4)
52 plot(x(1,:),x(3,:));
53 xlabel('X')
54 ylabel('Z')
55 grid on
56 title('Along Y')
```

In the latest versions of MATLAB, you can easily access figure and axes properties using field names. For instance, let's get the figure generated by the demo using `gcf`, then look at the children, which should include our four subplots.

```
>> h = gcf

h =

  Figure (5: PlotPage) with properties:

      Number: 5
        Name: 'PlotPage'
       Color: [0.94 0.94 0.94]
    Position: [440 378 560 420]
       Units: 'pixels'

  Show all properties

>> h.Children

ans =

  5x1 graphics array:

  ContextMenu
  Axes           (Along Y)
  Axes           (Along X)
  Axes           (Along Z)
  Axes           (Trajectory)
```

Note that the titles of our axes are helpfully displayed. If you wanted to add additional objects or change the properties of the axes, you could access the handles this way. Or, you might want to provide the handles as an output for your function. You can also make a subplot in a figure the current axes just by calling subplot again with the array size and ID.

```
subplot(2,2,1)
```

3.4 Create a Heat Map

Problem

You would like to create a heat map from data. A heat map shows the variation of magnitude using color in a two-dimensional image.

Solution

You can create a heat map using the `heatmap` function.

How It Works

We'll create a random set of data and two cell arrays for the x and y names.

HeatMapDemo.m

```
%% Heat map
% Heat map plot from random data

cD      = rand(4,3);
xV      = {'1' '2' '3'};
yV      = {'a' 'b' 'c' 'd'};

NewFigure('Heat Map')

heatmap(xV,yV,cD)
```

`heatmap` generates the map from the data shown in Figure 3.7.

```
>> HeatMapDemo

ans =

  Figure (3: Heat Map) with properties:

      Number: 3
        Name: 'Heat Map'
       Color: [0.9400 0.9400 0.9400]
    Position: [616 598 560 420]
       Units: 'pixels'

  Show all properties
```

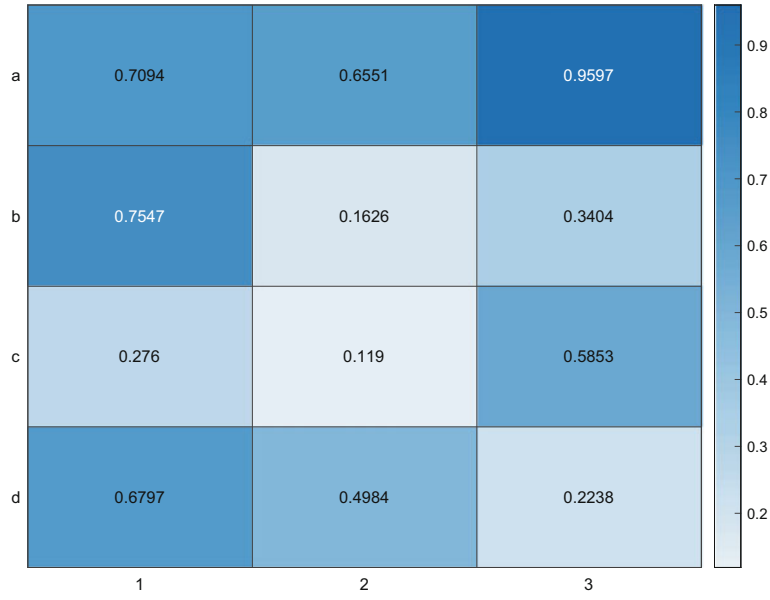

Figure 3.7: *A heat map from random data.*

```
ans =

  HeatmapChart with properties:

        XData: {3x1 cell}
        YData: {4x1 cell}
    ColorData: [4x3 double]

  Show all properties
```

3.5 Create a Plot Page with Custom-Sized Axes

Problem

You would like to group some plots together in one figure but not as evenly spaced subplots.

Solution

You can create custom-sized axes using the `'OuterPosition'` property of the axes, placing them anywhere in the figure you wish.

How It Works

We'll create a custom figure with two plots, one spanning the width of the figure and a second smaller axes. This will leave room for a block of descriptive text, which might describe the figure itself or display the results. In order to make the plots more interesting, we will add markers and text annotations using `num2str`.

The function is `PlotPage` shown in Figure 3.8. Using `'OuterPosition'` for the axes instead of `'Position'` means the limits will include the axes labels, so we can use the full range of the figure from 0 to 1 (normalized units). Figure 3.8 shows the resulting figure.

PlotPage.m

```
function PlotPage(t, x)

if nargin == 0
  disp('Demo of PlotPage');
  t  = linspace(0,100,101);
  th = logspace(0,log10(4*pi),101);
  in = logspace(-1,0,101);
  x = [sin(th).*cos(in);cos(th).*cos(in);sin(in)];
  PlotPage(t,x);
  return
end

h = figure('Name','PlotPage');
set(h,'InvertHardcopy','off')

% Specify the axes position as [left, bottom, width, height]
axes('outerposition',[0.5 0 0.5 0.5]);
plot(t,x);
xlabel('Time')
grid on

% Specify an additional axes and make a 3D plot
axes('outerposition',[0 0.5 1 0.5]);
plot3(x(1,:),x(2,:),x(3,:));
xlabel('X')
ylabel('Y')
zlabel('Z')
grid on

% add markers evenly spaced with time
hold on
for k=1:10:length(t)
  plot3(x(1,k),x(2,k),x(3,k),'x');
  % add a text label
  label = ['  ' num2str(t(k)) ' s'];
  text(x(1,k),x(2,k),x(3,k),label);
end
hold off

```

```
uh = uicontrol('Style','text','String','Description of the plots',...
           'units','normalized','position',[0.05 0.1 0.35 0.3]);
set(uh,'string',['You may wish to provide a detailed description '...
                 'of the visualization of your data or the results
                    right on the figure '...
                 'itself in a uicontrol text box such as this.']);
set(uh,'fontsize',14);
set(uh,'foregroundcolor',[1 0 0]);
```

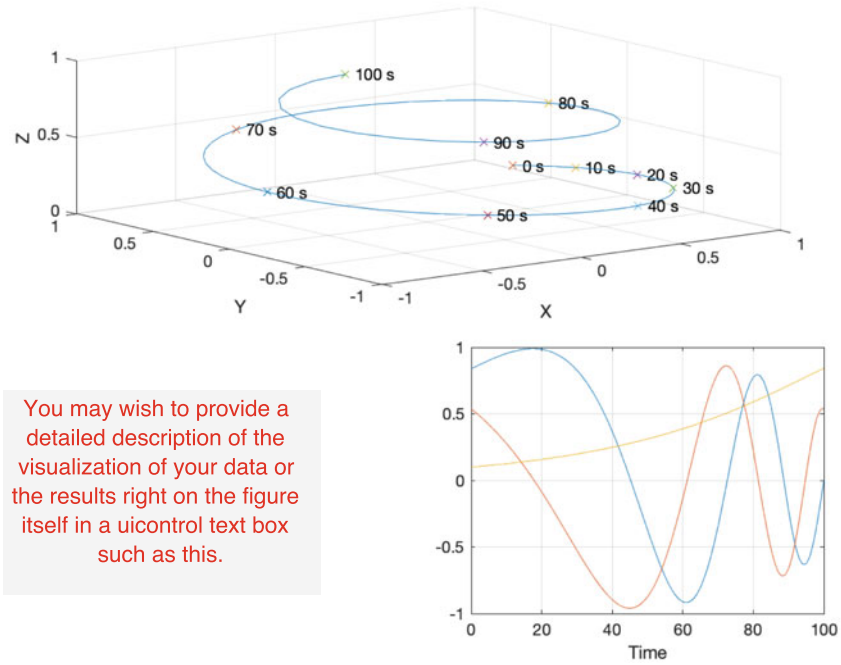

Figure 3.8: *`PlotPage` with custom-sized plots.*

3.6 Plotting with Dates

Problem

You want to plot data as a function of time using dates on the *x* axis.

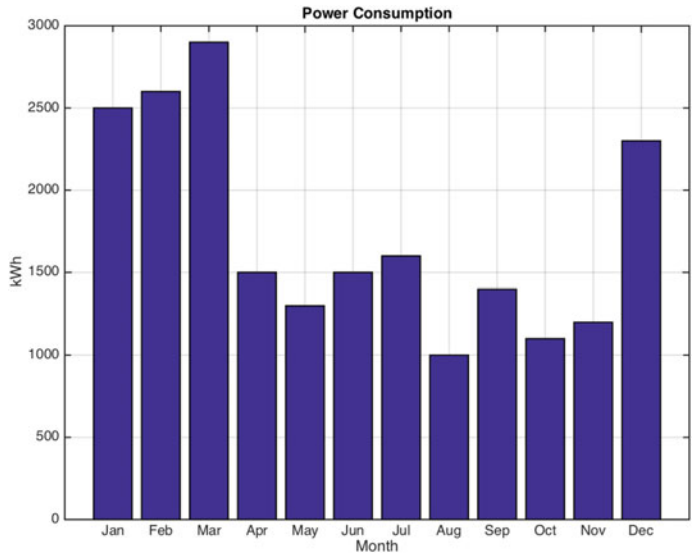

Figure 3.9: *Plotting with manual month labels.*

Solution

Access the tick labels directly using handles for the axis, or use `datetick` with serial date numbers.

How It Works

First, we will manually specify the tick labels. You plot the data as a function of index and then replace the x labels with strings of your choice, in this case specific months. For example, we will plot power consumption of a home in kilowatt hours (kWh). Note how we set the `xlim`, `xtick`, and `xticklabel` properties using `set` after generating the plot. The limits are set to [0 13] instead of [1 12] to accommodate the width of the bars. Figure 3.9 shows plotting with month labels.

PlottingWithDates.m

```
%% Plot using months as the x label
% First we will set the labels manually. Then we will use MATLAB's
    serial date
% numbers to set the labels automatically.

%% Specify specific months as labels
kWh   = [  2500 2600 2900 1500 1300 1500 1600 1000 1400 1100 1200
    2300];
month = {'Jan' 'Feb' 'Mar' 'Apr' 'May' 'Jun' 'Jul' 'Aug' 'Sep' 'Oct' '
    Nov' 'Dec'};

figure('Name','Plotting With Manual Date Labels');
bar(1:12,kWh)
xlabel('Month');
```

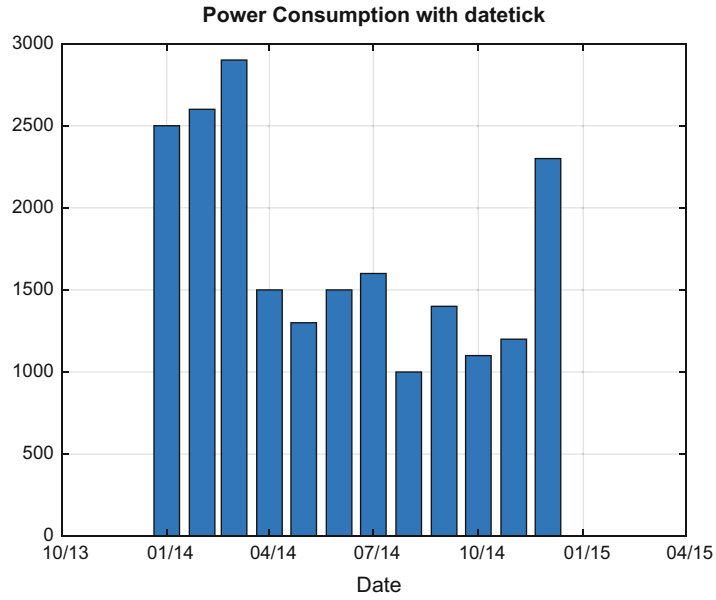

Figure 3.10: *Plotting using* `datetick` *with serial dates.*

```
ylabel('kWh')
title('Power Consumption');
grid on

set(gca,'xlim',[0 13],'xtick',1:12,'xticklabel',month);
```

If you are plotting data against complete dates, you can also use MATLAB's serial date numbers, which can be automatically displayed as tickmarks using `datetick`. You can convert between calendar dates and serial numbers using `datestr`, `datenum`, and `datevec`. A date vector is the six-component date as [year month day hour minute second]. So, for instance, let's assign our data in the preceding example to actual dates in the year 2014. The default date tickmarks will show months just like in our manual example, but for demonstration purposes, we specify a format including the year: `'mmmyy'`. Figure 3.10 shows plotting with serial dates.

```
%% Specify full dates and use serial dates to automatically produce
    labels
% Specifying only the month will use the current year by default. We
    will set
% the year to 2014 by using datevec.
N = datenum(month,'mmm');
V = datevec(N);
V(:,1) = 2014;
N = datenum(V);

figure('Name','Plotting With Serial Dates');
```

```
bar(N,kWh)
xlabel('Date');
title('Power Consumption with datetick');
datetick('x','mm/yy')
grid on
```

Note that the ticks themselves are no longer one per month; if you want to specify them manually, you now need to use date numbers. We have printed out the properties using `get` to show the `XTicks` used.

```
>> get(gca)
...
                      XLim: [735508 735965]
                 XLimMode: 'manual'
               XMinorGrid: 'off'
               XMinorTick: 'off'
                   XScale: 'linear'
                    XTick: [735508 735600 735690 735781 735873 735965]
               XTickLabel: [6x5 char]
```

MATLAB's serial date numbers do not correspond to other serial date formats like Julian date. MATLAB simply counts days from Jan-1-0000, so the year 2000 starts at a serial number of 2000*365 = 730,000. The following quick example demonstrates this as well as using `now` to get the current date:

```
>> v = datevec(now)
v =
         2015           7          31          11          37
            0.6198
>> n = datenum(v)
n =
   7.3618e+05
>> s = datestr(n,'local')
s =
31-Jul-2015 11:37:00
```

3.7 Generating a Color Distribution

Problem

You want to assign colors to markers or lines in your plot.

Solution

Specify the HSV components algorithmically from around the color wheel and convert to RGB.

How It Works

`ColorDistribution` chooses n colors from around the color wheel. The colors are selected using the hue component of HSV, with a full range from 0 to 1. Parameters allow the user to separately specify the saturation and value for all the colors generated. You could alternatively use these components to select a variety of colors of one hue.

Reducing the saturation (`sat`) lightens the colors while remaining on the same "spoke" of the color wheel. A saturation of 0 produces all grays. The value (`val`) keeps the ratio between RGB components remain the same, but lowering the magnitude makes colors darker, for example, [1 0.85 0] and [0.684 0.581 0]. See Figure 3.11.

ColorDistribution.m

```
%% Demonstrate a color distribution for an array of lines.
% Colors are calculated around the color wheel using hsv2rgb.

val     = 1;
sat     = 1;
n       = 100;
dTheta  = 360/n;
thetaV  = linspace(0,360-dTheta,n);

h       = linspace(0,1-1/n,n);
s       = sat*ones(1,n);
v       = val*ones(1,n);
colors  = hsv2rgb([h;s;v]');
y       = sin(thetaV*pi/180);
hF      = figure;
hold on;
set(hF,'name','Color Wheel')
l = gobjects(n);
for k = 1:n
        l(k) = plot(thetaV,k*y);
end
set(gca,'xlim',[0 360]);
grid on
pause

for k = 1:n
  set(l(k),'color',colors(k,:)*val);
end
```

Figure 3.11 plots a color distribution.

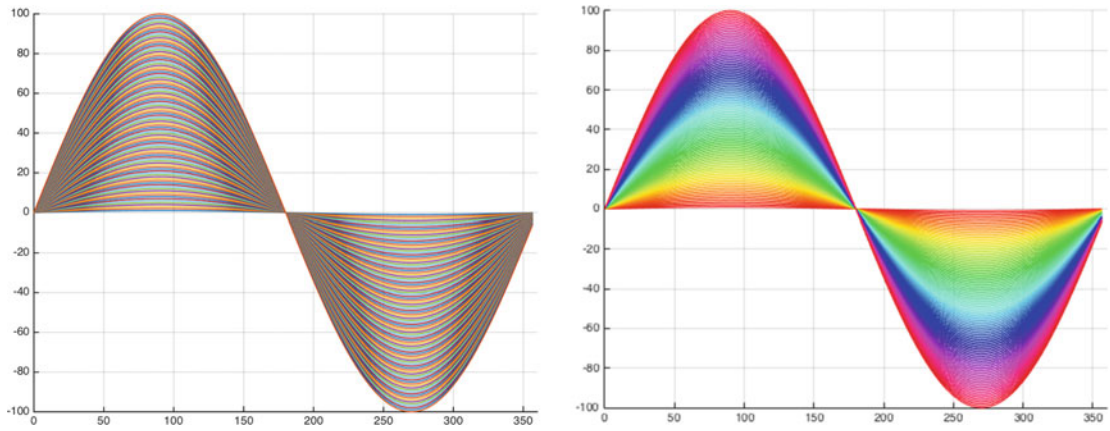

Figure 3.11: *Original lines and lines with a color distribution with values and saturation of 1.*

3.8 Visualizing Data over 2D or 3D Grids

Problem

You need to perform a calculation over a grid of data and view the results.

Solution

The function `meshgrid` produces grids over x and y that can be used for calculations and subsequently input to `surf`. This is also useful for `contour` and `quiver` plots.

How It Works

Our solution is in `GridVisualization.m`. First, you define the vectors in x and y that define your grid. You can perform your calculations in a for loop or in a vectorized function. The vectors do not have to be physical dimensions; indeed, in general, they are quite different quantities involved in a parametric study. The classic example is an exponential function of two variables, which is viewed as a surface in Figure 3.12.

GridVisualization.m

```
%% 2D example of meshgrid
figure('Name','2D Visualization');
xv = -1.5:0.1:1.5;
yv = -2:0.2:2;
[X,Y] = meshgrid(xv, yv);
Z = Y .* exp(-X.^2 - Y.^2);
surf(X,Y,Z,'edgecolor','none')
title('2D Grid Example')
zlabel('z = y exp( -x^2-y^2 )')
colormap hsv

```

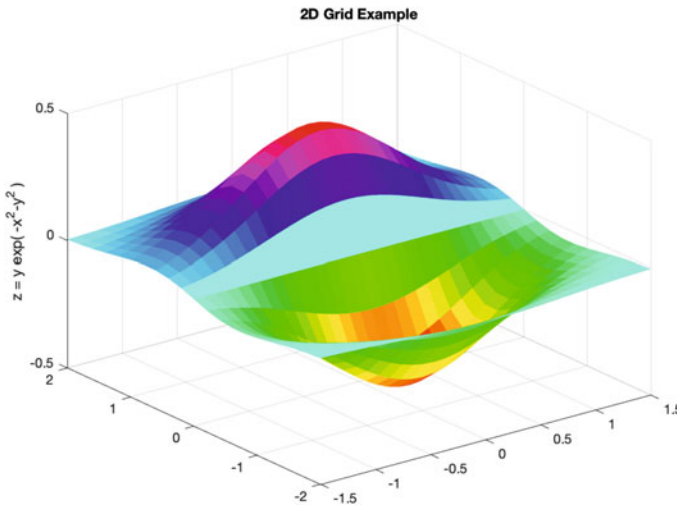

Figure 3.12: *3D surface generated over a 2D grid.*

```
size(X)
size(Y)
```

The generated matrices are square and consist of the input vector replicated in the correct dimension. You could achieve the same result by hand using `repmat`, but `meshgrid` eliminates the need to remember the details.

```
>> size(X)
ans =
    41    41
>> size(Y)
ans =
    41    41
>> X(1:5,1:5)
ans =
           -2         -1.9         -1.8         -1.7         -1.6
           -2         -1.9         -1.8         -1.7         -1.6
           -2         -1.9         -1.8         -1.7         -1.6
           -2         -1.9         -1.8         -1.7         -1.6
           -2         -1.9         -1.8         -1.7         -1.6
>> Y(1:5,1:5)
ans =
           -2           -2           -2           -2           -2
         -1.9         -1.9         -1.9         -1.9         -1.9
         -1.8         -1.8         -1.8         -1.8         -1.8
         -1.7         -1.7         -1.7         -1.7         -1.7
         -1.6         -1.6         -1.6         -1.6         -1.6
```

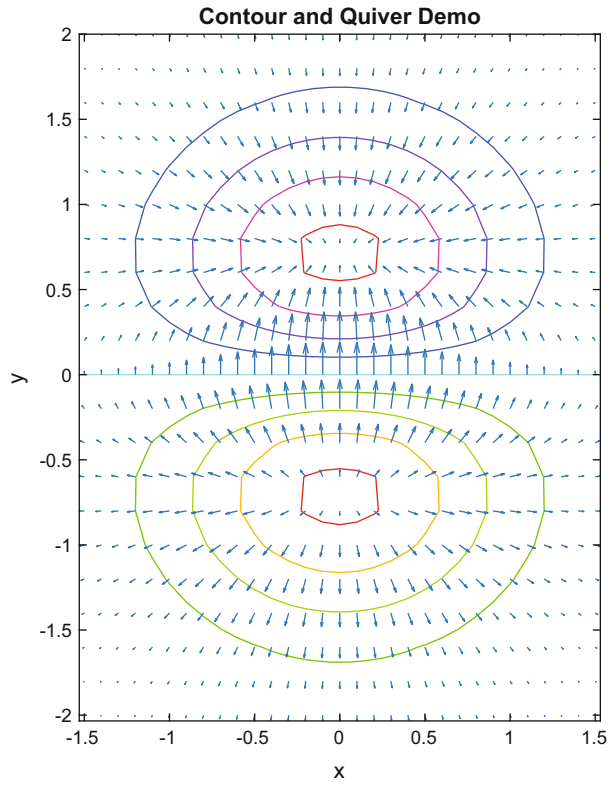

Figure 3.13: *3D surface visualized as contours.*

For fun, we can plot contours of the data as well. We can use the `gradient` function to calculate the slope and plot this using `quiver`. This uses `meshgrid` that returns a 2D mesh from x and y vectors. Figure 3.13 shows a contour plot.

```
figure('Name','Contour and Quiver')
[px,py] = gradient(Z,0.1,0.2);
contour(X,Y,Z), hold on
quiver(X,Y,px,py)
title('Contour and Quiver Demo')
xlabel('x')
ylabel('y')
colormap hsv
axis equal
```

You can also generate a 3D grid and compute data over the volume, for a fourth dimension. In order to view this extra data over the volume, you can use `slice`. This uses interpolation to draw slices at any location along the axes you specify. If you want to see the exact planes in your data, you can use `pcolor`, `surf`, or `contour` in individual figures. `quiver3` can be used to plot arrows in 3D space. We are going to generate five slices at three different x values and at two different z values. The result is shown in Figure 3.14.

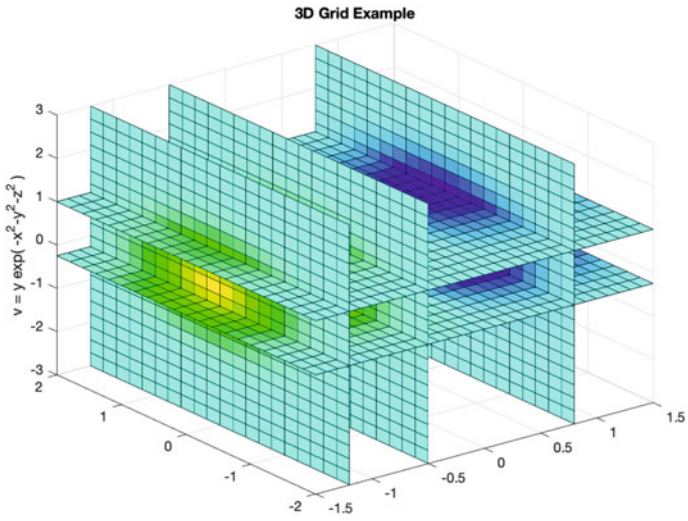

Figure 3.14: *3D volume with slices.*

```
%% 3D example of meshgrid
% meshgrid can be used to produce 3D matrices, and slice can display
    selected
% planes using interpolation.
figure('Name','3D Visualization');
zv = -3:0.3:3;
[x,y,z] = meshgrid(xv, yv, zv);
v = x .* exp(-x.^2 - y.^2 - z.^2);
slice(x,y,z,v,[-1.2 -0.5 0.8],[],[-0.25 1])
title('3D Grid Example')
zlabel('v = y exp( -x^2-y^2-z^2 )')
colormap hsv
```

3.9 Generate 3D Objects Using Patch

Problem

You would like to draw a 3D box.

Solution

You can create a 3D box using the `patch` function.

How It Works

The `patch` function in MATLAB uses vertices and faces to define an area in two or three dimensions. The vertex list is an `n`-by-3 array specifying the vertex locations. The faces array is an `n` by `m` array where `m` is the number of vertices per polygon. The faces array contains the row indices for the vertices. We usually set `m` to 3 since all graphics engines eventually reduce polygons to triangles. We draw a box in `BoxPatch` shown in the following. Generally, when drawing a physical object, we set `axis` to `equal` so that the aspect ratio is correct. `patch` has many properties. In this case, we just set the color of the faces to gray using RGB. The edge color, which can also be specified, is black by default. The `view(3)` call sets the camera to a position with equal x, y, and z values. `rotate3d on` lets us move the camera around. This is very handy for inspecting the model. Each line in face is the three vertex elements that form a triangle face. Figure 3.15 show a box generated with `patch`.

BoxPatch.m

```
%% Box design
x   = 3;
y   = 2;
z   = 1;

% Faces
f   = [2 3 6;3 7 6;3 4 8;3 8 7;4 5 8;4 1 5;2 6 5;2 5 1;1 3 2;1 4 3;5 6
    7;5 7 8];

% Vertices
v = [-x  x  x -x -x  x  x -x;...
      -y -y  y  y -y -y  y  y;...
      -z -z -z -z  z  z  z  z]'/2;

```

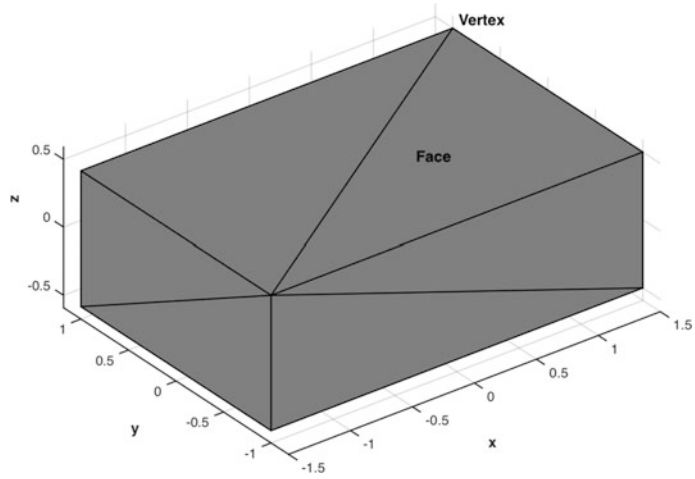

Figure 3.15: *Box generated using* `patch`.

```
%% Draw the object
h = figure('name','Box');
patch('vertices',v,'faces',f,'facecolor',[0.5 0.5 0.5]);
axis equal
grid on
axis([-3 3 -3 3 -3 3])
xlabel('x')
ylabel('y')
zlabel('z')
view(3)
rotate3d on
```

3.10 Working with Light Objects

Problem

You would like to illuminate the 3D box drawn in the previous recipe.

Solution

You can create ambient or directed light objects using the `light` function. Light objects affect both patch and surface objects, which are created by `surf`, `mesh`, `pcolor`, `fill`, `fill3`, and `patch`.

How It Works

The main properties for working with light objects are color, style, position, and visible. The style may be infinite, with the light shining in parallel rays from a specified direction, or local, with a point source shining in all directions. The position property has a different meaning for each of these styles. `PatchWithLighting` adds a local light to the box script. We modify the box surface properties using `material` to get different effects.

PatchWithLighting.m

```
%% Add lighting to the cube patch
% We use findobj to locate the patch drawn in Patch, then change its
    properties
% to be suitable for lighting. We add a local light.

%% Create the box patch object
BoxPatch;

%% Find and update the patch object
p = findobj(gcf,'type','patch');
c = [0.7 0.7 0.1];
```

```
15 set(p,'facecolor',c,'edgecolor',c,...
16       'edgelighting','gouraud','facelighting','gouraud');
17 material('metal');
18
19 %% Lighting
20 l = light('style','local','position',[10 10 10]);
```

Figure 3.16 shows dull and metal material with the same lighting. The lighting produced by MATLAB is limited by being an OpenGL lighting. Modern 3D graphics use textures and shaders for photo-realistic scene lighting. You also cannot generate shadows in MATLAB. The one on the right has a somewhat sharper color gradient at the corner.

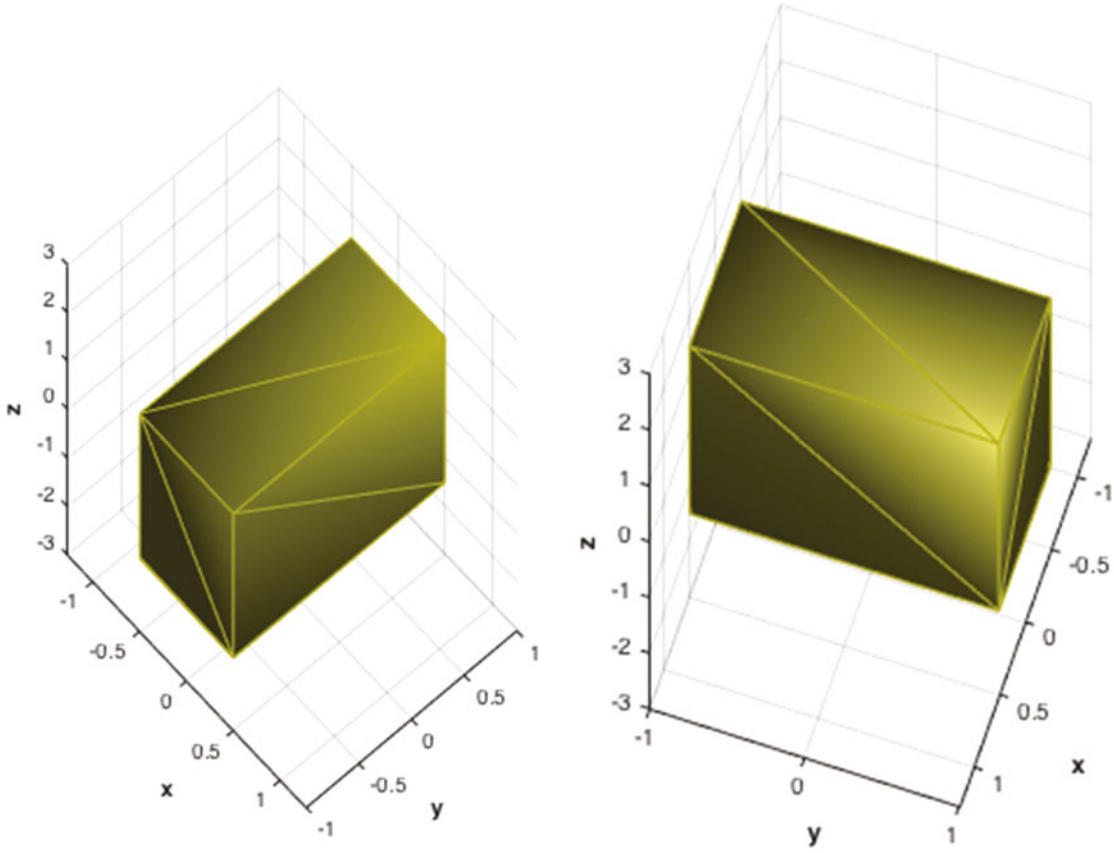

Figure 3.16: *Box illuminated with a local light object. The left box has "dull" material. The one on the right has "metal."*

The dull, shiny, and metal settings for `material` set the patch properties to produce these effects. We can easily print the effects to the command line using `get`.

```
>> material dull
>> get(p)
              DiffuseStrength: 0.8
...
    SpecularColorReflectance: 1
             SpecularExponent: 10
             SpecularStrength: 0

>> material metal
>> get(p)
              DiffuseStrength: 0.3
              ...
    SpecularColorReflectance: 0.5
             SpecularExponent: 25
             SpecularStrength: 1

>> material shiny
>> get(p)
              DiffuseStrength: 0.6
              ...
    SpecularColorReflectance: 1
             SpecularExponent: 20
             SpecularStrength: 0.9
```

Note that the `AmbientStrength` is 0.3 for all the material settings listed earlier. If you want to see the effect of only your light objects without ambient light, you have to manually set this to zero. In Figure 3.17, we have set the ambient strength to zero and applied the shiny material.

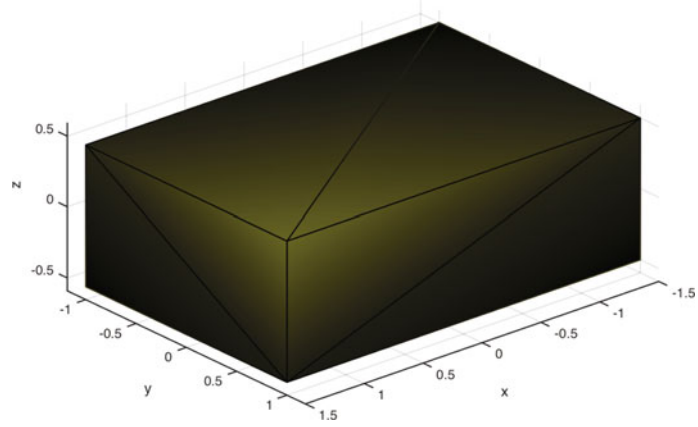

Figure 3.17: *Shiny box with ambient lighting removed (`AmbientStrength` set to 0) and a different camera viewpoint.*

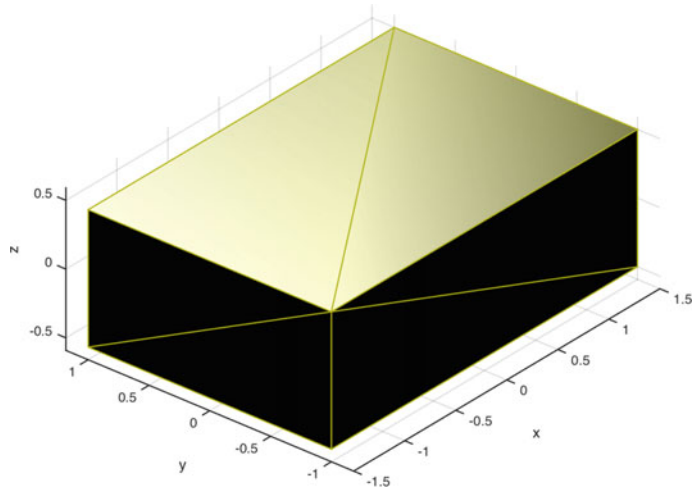

Figure 3.18: *Shiny box with flat lighting.*

MATLAB has a `lighting` function to control the lighting model with four settings: none, Gouraud, Phong, and flat. Gouraud interpolates the lighting across the faces and gives the most realistic effect. Note that setting the lighting to Gouraud for our box sets the `FaceLighting` property to `gouraud` but the `EdgeLighting` to `none`, which will give a different effect than in our script earlier where the edge lighting was also set to Gouraud via its property. Flat lighting gives each entire face a uniform lighting, as in Figure 3.18, where we set the view to (-50,30) and the lighting to flat.

The MATLAB recommendations are to use flat lighting for faceted objects and Gouraud lighting for curved objects. The easiest way to compare these is to create a sphere, which is simple using the `sphere` function and generating a surface. This is done in the following `SphereLighting`. The infinite light object shines from the x axis. See Figure 3.19 for the resulting plots.

SphereLighting.m

```
%% Create and light a sphere

%% Make the sphere surface in a new figure
[X,Y,Z] = sphere(16);
figure('Name','Sphere Demo')
s = surf(X,Y,Z);
xlabel('x')
ylabel('y')
zlabel('z')
axis equal
view(70,15)

%% Add a lighting object and display the properties
light('position',[1 0 0])
disp(s)
```

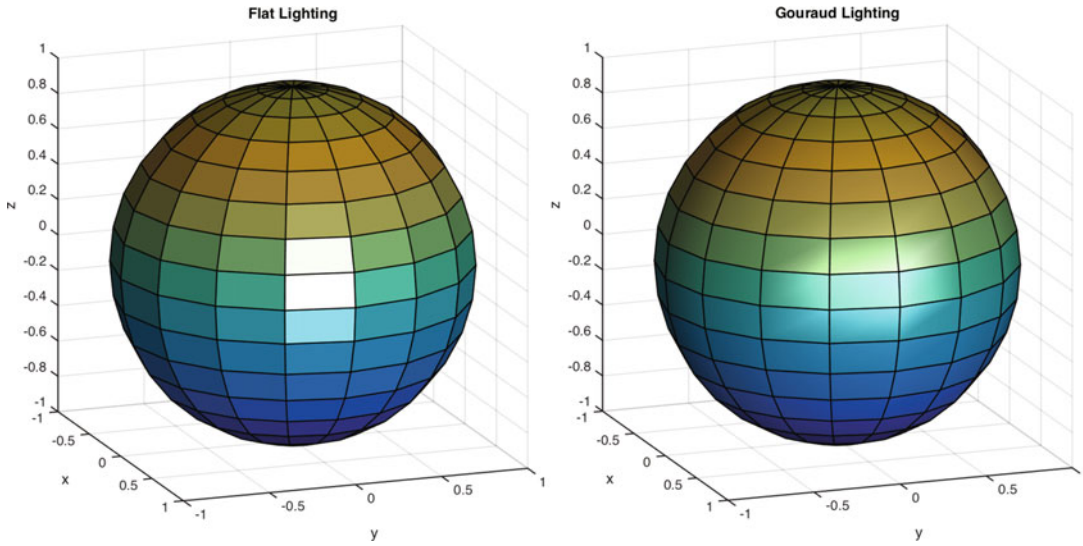

Figure 3.19: *Sphere illuminated with an infinite light object. The left sphere has flat lighting. The one on the right has Gouraud.*

```
16 title('Flat Lighting')
17 pause
18
19 %% Change to Gouraud lighting and display again
20 lighting gouraud
21 title('Gouraud Lighting')
22 disp(s)
```

In addition to a `sphere` function, MATLAB also provides `cylinder` and `ellipsoid`.

3.11 Programmatically Setting the Camera Properties

Problem

You would like to have a camera in your scene that can be pointed.

Solution

Use the MATLAB `cam` functions. These provide the same functionality as the buttons in the camera toolbar, but with repeatability and the ability to pass in variables for the parameters. We demonstrate this in the script `PatchWithCamera.m`.

How It Works

We make two boxes in the scene. One is scaled and displayed from the other by 5 in x. We use the MATLAB functions `camdolly`, `camorbit`, `campan`, `camzoom`, and `camroll` to control the camera. We put all of these functions in the `PatchWithCamera.m` script and provide examples of two sets of parameters. Note that without lighting, the edges disappear.

PatchWithCamera.m

```
%% Generate two cubes using patch and point a camera at the scene
% The camera parameters will be set programmatically using the cam
    functions.

%% Camera parameters
% Orbit
thetaOrbit      = 0;
phiOrbit    = 0;

% Dolly
xDolly      = 0;
yDolly      = 0;
zDolly      = 0;

% Zoom
zoom        = 1;

% Roll
roll        = 50;

% Pan
thetaPan    = 1;
phiPan      = 0;

%% Box design
x   = 1;
y   = 2;
z   = 3;

% Faces
f   = [2 3 6;3 7 6;3 4 8;3 8 7;4 5 8;4 1 5;2 6 5;2 5 1;1 3 2;1 4 3;5 6
    7;5 7 8];

% Vertices
v = [-x  x  x -x -x  x  x -x;...
     -y -y  y  y -y -y  y  y;...
     -z -z -z -z  z  z  z  z]'/2;

%% Draw the object
```

```
h = figure('name','Box');

c = [0.7 0.7 0.1];
patch('vertices',v,'faces',f,'facecolor',c,'edgecolor',c,...
      'edgelighting','gouraud','facelighting','gouraud');

c = [0.2 0 0.9];
v      = 0.5*v;
v(:,1) = v(:,1) + 5;
patch('vertices',v,'faces',f,'facecolor',c,'edgecolor',c,...
      'edgelighting','gouraud','facelighting','gouraud');

material('metal');
lighting gouraud
axis equal
grid on
xlabel('x')
ylabel('y')
zlabel('z')
view(3)
rotate3d on

%% Camera commands
campan(thetaPan,phiPan)
camzoom(zoom)
camdolly(xDolly,yDolly,zDolly);
camorbit(thetaOrbit,phiOrbit);
camroll(roll);

s = sprintf('Pan %3.1f %3.1f\nZoom %3.1f\nDolly %3.1f %3.1f %3.1f\
    nOrbit %3.1f %3.1f\nRoll %3.1f',...
thetaPan,phiPan,zoom,xDolly,yDolly,zDolly,thetaOrbit,phiOrbit,roll);

text(2,0,0,s);
```

Additional functions for interacting with the scene camera include `campos` and `camtarget`, which can be used to set the camera position and target. This can be used to image one object from the vantage point of another. `camva` sets the camera view angle, so you can model a real camera's field of view. `camup` specifies the camera "up" vector or the direction of the top of the frame.

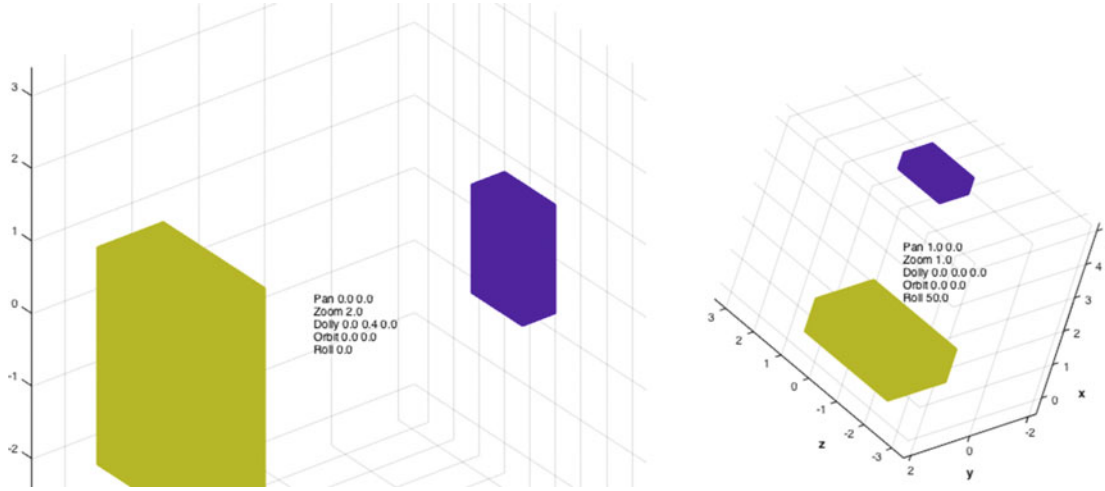

Figure 3.20: *Boxes with different camera parameters.*

3.12 Display an Image

Problem

You would like to draw an image.

Solution

You can read in an image directly from an image file and draw it in a figure window. MATLAB supports a variety of formats including GIF, JPG, TIFF, PNG, and BMP. Our solution is in the script `ReadImage.m`.

How It Works

We read in a black and while image using `imread` and display it using `imagesc`. `imagesc` scales the color data into the colormap. It is necessary to apply the grayscale colormap; otherwise, you'll get the colors in the default colormap. In parula, this is blue and yellow.

ReadImage.m

```
%% Draw a JPEG image in a figure multiple ways
% We will load and display an image of a mug.
%% See also
% imread, pcolor, imagesc, imshow, colormap

%% Read in the JPEG image
i = imread('Mug.jpg');

%% Draw the picture with imagesc
```

```
% This preserves an axes. Each pixel center of the image lies at integer
% coordinates ranging between 1 and M or N. Compare the result of imagesc to
% that of pcolor. axis image sets the aspect ratio so that tick marks on both
% axes are equal, and makes the plot box fit tightly around the data.
h = figure('name','Mug');
subplot(1,2,1)
pcolor(i)
shading('interp')
colorbar
axis image
title('pcolor with colorbar')
a = subplot(1,2,2);
% scale the image into the colormap
imagesc( i );
colormap(a,'gray')
axis image
grid on
title('imagesc with gray colormap')
```

Figure 3.21 shows the mug first using `pcolor`, which creates a pseudocolor plot of a matrix, which is really a `surf` with the view looking down from above. To highlight this fact, we added a colorbar. Then on the right, the image is drawn using `imagesc` with a gray

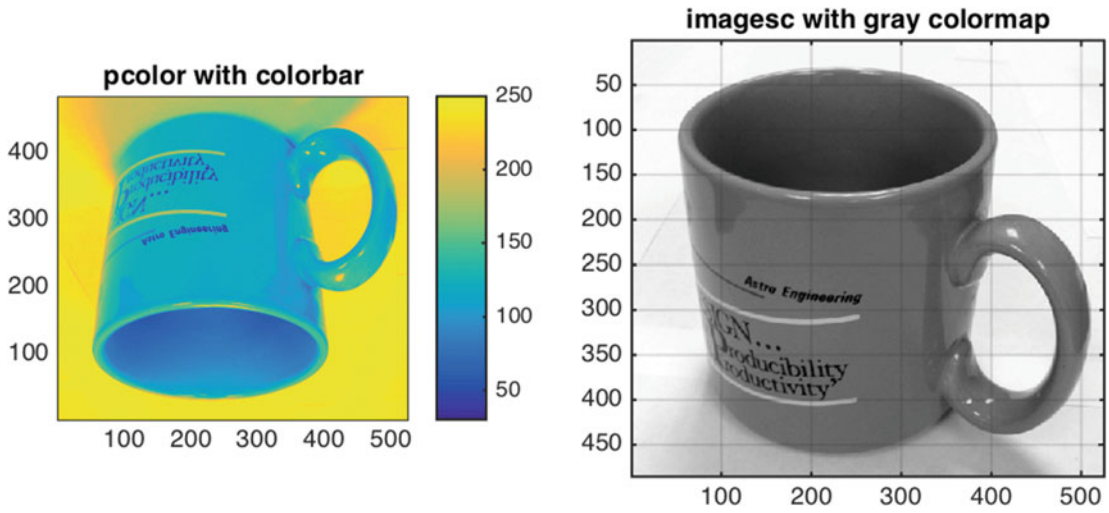

Figure 3.21: *Mug displayed using* `pcolor` *and* `imagesc`.

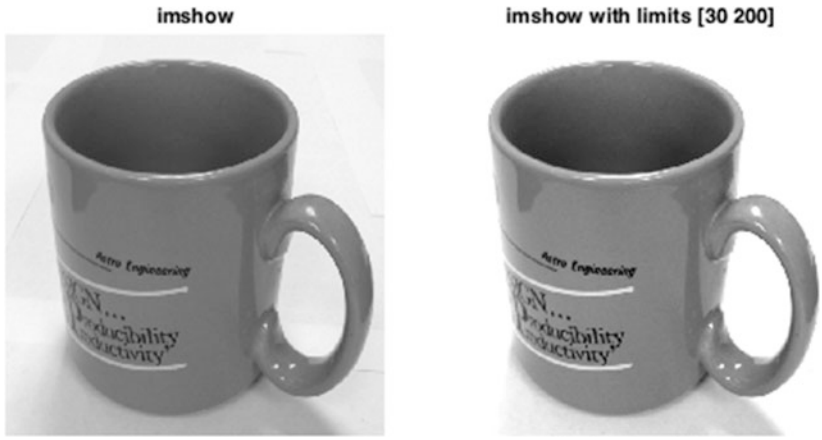

Figure 3.22: *Mug displayed using* `imshow`*, with color limits applied on the right.*

colormap. Observe that `imagesc` has changed the direction of the axes so that the image appears right-side up. Both plots have axes with tickmarks.

MATLAB has another image display function called `imshow`, which is considered the fundamental image display function. This optimizes the figure, axes, and image object properties for displaying an image. If you have the Image Processing toolbox, `imtool` extends `imshow` with additional features. Notice how the image is displayed without the axes box. This function scales and selects the gray colormap automatically. Figure 3.22 shows the use of `imshow`

```
33 %% Draw with imshow
34 % The axes will be turned off. The image will be scaled to fit the
       figure if it
35 % is too large.
36 f = figure('Name','Mug Image');
37 subplot(1,2,1)
38 imshow(i)
39 title('imshow')
40 subplot(1,2,2)
41 imshow(i,[30 200])
42 title('imshow with limits [30 200]')
```

Not all images use the full depth available; for instance, this mug image has a minimum value of 30 and a maximum of 250. `imshow` allows you to set the color limits of the image directly, and the pixels will be scaled accordingly. We can darken the image by increasing the lower color limit and brighten the image by lowering the upper color limit.

3.13 Graph and Digraph

Problem

We have a stochastic process for which we want a graphical representation.

Solution

Use graph and digraph in the script `RandomWalk.m`.

How It Works

Generate a transition matrix showing the probability of transition from one state to a second state.

The code in `RandomWalk.m` creates a digraph, graph, and a random walk. The first part creates a transition matrix.

RandomWalk.m

```
%% Demonstrate a digraph and graph

% Generate a transition matrix
% x ranges from -5 to 5
p = zeros(11,11);
for k = 2:10
  p(k,k-1) = 0.5;
  p(k,k+1) = 0.5;
end

p(1,2)    = 1;
p(11,10)  = 1;

fprintf('%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f%4.1f\n',p);
```

When we run `RandomWalk` at the command line we get the below output:

```
>> RandomWalk
 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
 1.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0
 0.0 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0 0.0
 0.0 0.0 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0
 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0
 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.5 0.0 0.0
 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.5 0.0
 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 1.0
 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0
```

The next part of RandomWalk creates a digraph shown in Figure 3.23 and a graph shown in Figure 3.24.

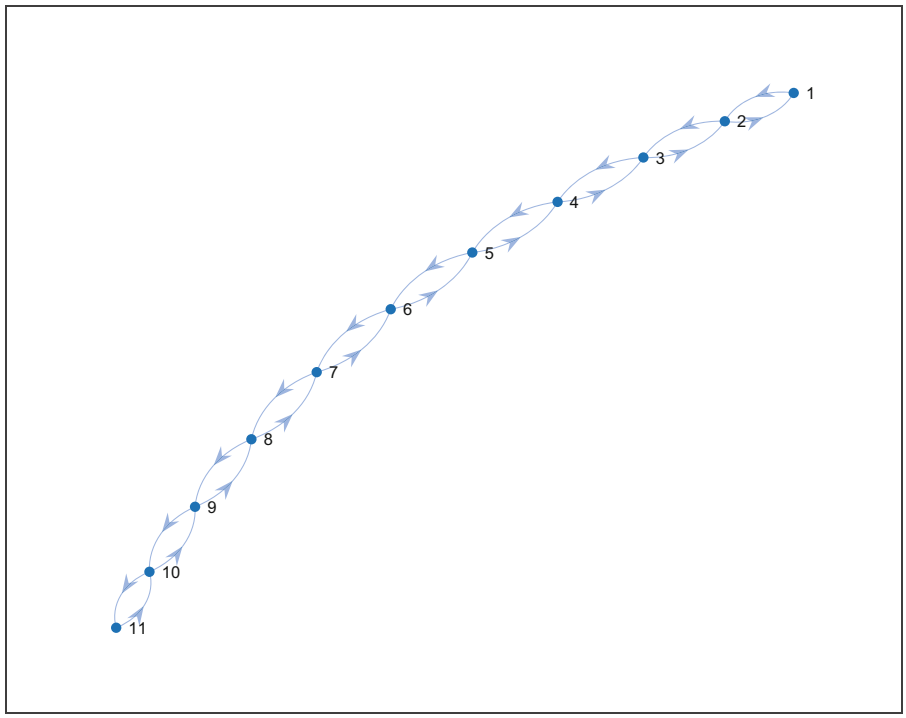

Figure 3.23: *Digraph for the random walk.*

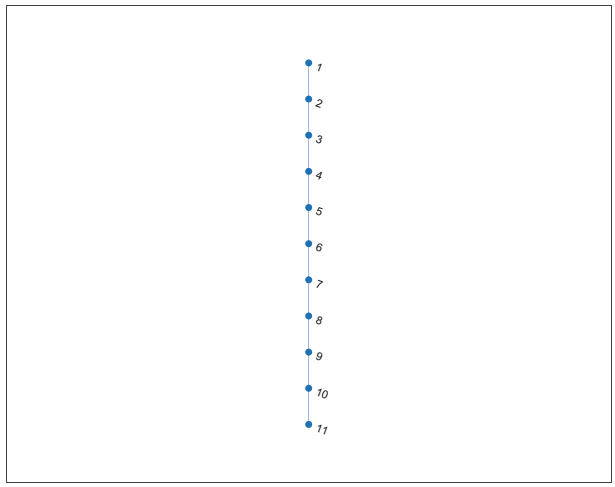

Figure 3.24: *Graph for the random walk.*

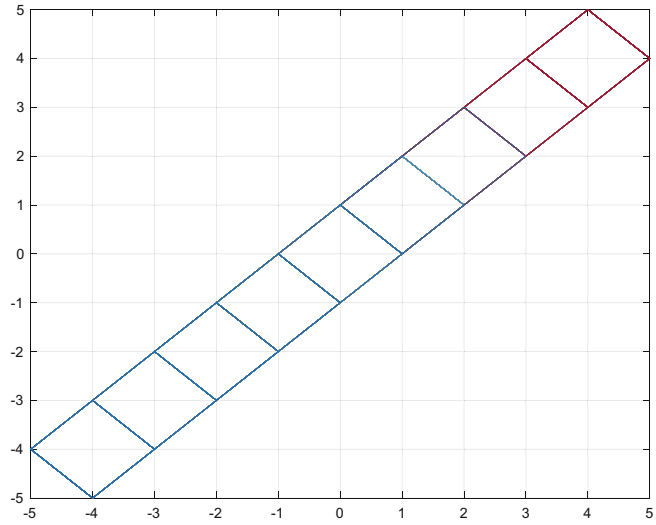

Figure 3.25: *The random walk. The lines show the connections between the nodes in the random walk. All possible paths are shown.*

```
g = digraph(p);

NewFigure('Digraph');
plot(g)
grid on
```

```
g = graph(p,'upper');

NewFigure('Graph');
plot(g)
grid on
```

The random walk based on the transition matrix is shown in Figure 3.25.

```
n = 100;
m = 50;

NewFigure('Random Walk');
for k = 1:m
  x = zeros(1,n);
  for j = 2:n
    if(x(j-1) == -5)
      x(j) = -4;
    elseif( x(j-1) == 5 )
      x(j) = 4;
    else
      x(j) = x(j-1) + sign(randn);
```

```
    end
  end
  plot(x(1:n-1),x(2:n))
  hold on
end
grid on
```

3.14 Adding a Watermark

Problem

You have a lot of great graphics in your toolbox, and you would like them to be marked as having been created by your company. Alternatively, or additionally, you may want to mark images with a version number or date of the software that generated them.

Solution

You can use low-level graphics functions to add a textual or image watermark to figures that you generate in your toolbox. The tricky part is adding the items to the figure at the correct time so they are not overridden.

How It Works

The best way to add watermarks is to make a special axis for each text or image item you want to add. You turn the axis box off so all that you see is the text or image. In the first example, we add an icon and text to the lower left-hand corner of the plot. We add a color for the edge around the text so that it is nicely delineated. This is shown in Figure 3.26 using the `Watermark.m` function. In the example, we set the hard copy inversion to off, so that when we print the figure, we will get a gray background – this makes it easier to see in the book.

```
>> h = figure;
>> set(h,'InvertHardCopy','off')
>> axes
>> Watermark(h)
```

Watermark.m

```
%% WATERMARK Add a watermark to a figure.
% This function creates two axes, one for the image and one for the
    text.
% Calling it BEFORE plotting can cause unexpected results. It will
    reset
% the current axes after adding the watermark. The default position is
% the lower left corner, (2,2).
%% Form
%   Watermark( fig, pos  )
%% Inputs
%   fig        (1,1) Figure hangle
```

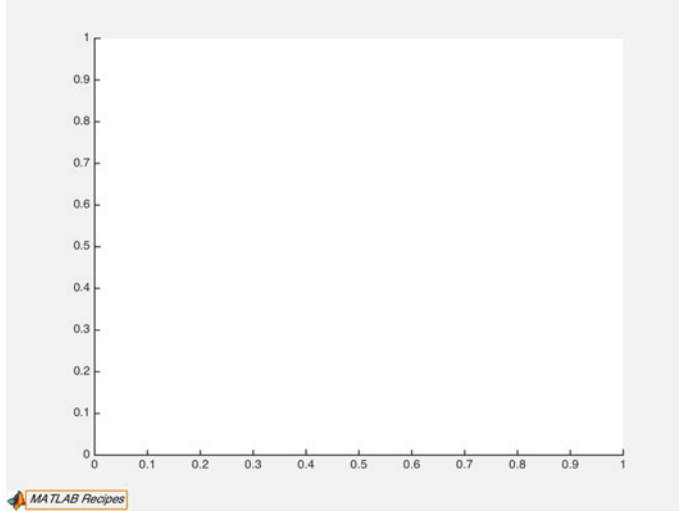

Figure 3.26: *Company watermark.*

```
%   pos         (1,2) Coordinates, (left, bottom)
%% Outputs
% None.

function Watermark( fig, pos  )

if (nargin<1 || isempty(fig))
  fig = figure('Name','Watermark Demo');
  set(fig,'color',[0.85 0.9 0.85]);
end

if (nargin<2 || isempty(pos))
  pos = [2 2];
end

string = 'MATLAB Recipes';

% Save the current axes so we can restore it
aX = [];
if ~isempty(get(fig,'CurrentAxes'))
  aX = gca;
end

% Draw the icon
%--------------
[d,map] = imread('matlabicon','gif');
posIcon = [pos(1:2) 16 16];
a = axes( 'Parent', fig, 'box', 'off', 'units', 'pixels', 'position',
    posIcon );
image( d );
colormap(a,map)
```

```
axis off

% Draw the text
%--------------
posText = [pos(1)+18 pos(2)+1 100 15];
axes( 'Parent', fig, 'box', 'off', 'units', 'pixels', 'position',
    posText );
t = text(0,0.5,string,'fontangle','italic');
set(t,'edgecolor',[0.87 0.5 0])
axis off

% Restore current axes in figure
if ~isempty(aX)
  set(fig,'CurrentAxes',aX);
end

set(fig,'tag','Watermarked')
```

As an additional example, we added text along the left- and right-hand sides of a figure using text rotation in the function `DraftMark.m`. We gave the text a light color. This marks the figure as a draft. We create a blank figure and axis before adding the draft mark, as shown in Figure 3.27.

```
>> h = figure('Name','Draftmark Demo');
>> set(h,'color',[0.85 0.9 0.85]);
>> set(h,'InvertHardCopy','off')
>> axes;
>> Draftmark(h);
```

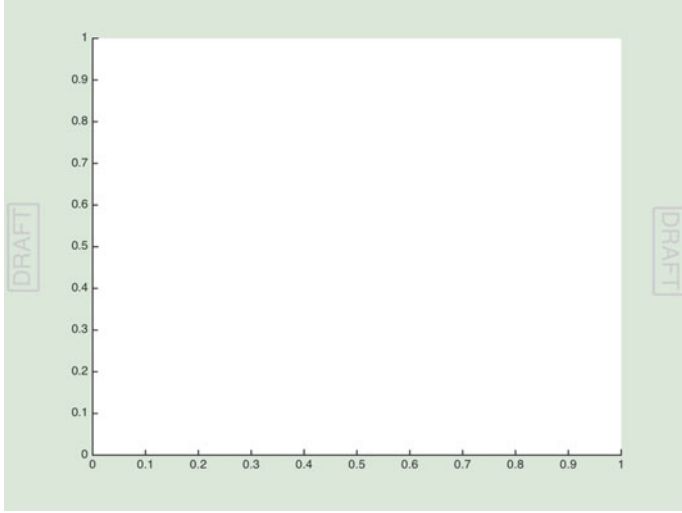

Figure 3.27: *Draft watermark.*

Draftmark.m

```
%% DRAFTMARK Add a draft marking to a figure.
% This function creates two axes, one each block of text.
% Calling it BEFORE plotting can cause unexpected results. It will reset
% the current axes after adding the watermark. The default position is
% the lower left corner, (2,2).
%% Form
%   Draftmark( fig, pos  )
%% Inputs
%   fig         (1,1) Figure hangle
%   pos         (1,2) Coordinates, (left, bottom)
%% Outputs
% None.

function Draftmark( fig, pos  )

if (nargin<1 || isempty(fig))
  fig = figure('Name','Draft Demo');
  set(fig,'color',[0.85 0.9 0.85]);
end

if (nargin<2 || isempty(pos))
  pos = [2 2];
end

string = 'DRAFT';

% Save the current axes so we can restore it
aX = [];
if ~isempty(get(fig,'CurrentAxes'))
  aX = gca;
end

% Draw the text
%--------------
pf = get(fig,'position');
posText = [pos(1)+5 pos(2)+0.5*pf(4)-40 20 80];
axes( 'Parent', fig, 'box', 'on', 'units', 'pixels', 'outerposition', posText );
t1 = text(0,0,string,'fontsize',20,'color',[0.8 0.8 0.8]);
set(t1,'rotation',90,'edgecolor',[0.8 0.8 0.8],'linewidth',2)
axis off

posText = [pos(1)+pf(3)-25 pos(2)+0.5*pf(4)-40 20 80];
axes( 'Parent', fig, 'box', 'on', 'units', 'pixels', 'outerposition', posText );
t2 = text(0,1,string,'fontsize',20,'color',[0.8 0.8 0.8]);
set(t2,'rotation',270,'edgecolor',[0.8 0.8 0.8],'linewidth',2)
axis off

```

```
% Restore current axes in figure
if ~isempty(aX)
  set(fig,'CurrentAxes',aX);
end
```

If you want to get very fancy, you could draw objects across the front of the figure and give them transparency, but it has to be fill or patch objects; text cannot be given transparency.

Summary

In this chapter, we reviewed key features of MATLAB visualization, from basic plotting to 3D visualization including objects and lighting. We demonstrated accessing figure and axes handles and setting properties programmatically, as well as using the interactive tools for figures. Creating helpful visualization routines is a key part of any toolbox. MATLAB provides excellent data management routines, including for large grids of data, and many options for colorization. Table 3.1 lists the code developed in the chapter.

Table 3.1: *Chapter Code Listing*

File	Description
`AnnotatePlot`	Add text annotations evenly spaced along a curve
`BoxPatch`	Generate a cube using patch
`ColorDistribution`	Demonstrate a color distribution for an array of lines
`DraftMark`	Add a draft marking to a figure
`GridVisualization`	Visualize data over 2D and 3D grids
`PatchWithCamera`	Generate two cubes using patch and point a camera at the scene
`PatchWithLighting`	Add lighting to the cube patch
`PlotPage`	Create a plot page with several custom plots in one figure
`PlottingWithDates`	Plot using months as the x label
`QuadPlot`	Create a quad plot page using subplot
`ReadImage`	Draw a JPEG image in a figure multiple ways
`SphereLighting`	Create and light a sphere
`Watermark`	Add a watermark to a figure