Quick answers to common problems

MATLAB Graphics and Data Visualization Cookbook

Tell data stories with compelling graphics using this collection of data visualization recipes

Nivedita Majumdar
Swapnonil Banerjee

[PACKT] PUBLISHING

MATLAB Graphics and Data Visualization Cookbook

Tell data stories with compelling graphics using this collection of data visualization recipes

Nivedita Majumdar

Swapnonil Banerjee

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

MATLAB Graphics and Data Visualization Cookbook

First published: November 2012

Production Reference: 1191112

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-316-5

www.packtpub.com

Cover Image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Authors

Nivedita Majumdar

Swapnonil Banerjee

Reviewers

Dr. John Bemis

Adee Ran

Ashish Uthama

David Woo

Acquisition Editor

Joanna Finchen

Lead Technical Editor

Kedar Bhat

Technical Editors

Dipesh Panchal

Copy Editor

Alfida Paiva

Project Coordinator

Yashodhan Dere

Proofreader

Stephen Swaney

Indexer

Rekha Nair

Graphics

Valentina D'silva

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Authors

Nivedita Majumdar is a software development engineer with extensive experience with MATLAB. She has a PhD in Computational Sciences and Informatics. She has been developing data analysis tools and algorithms for the communications and life sciences industries for the past decade. She is deeply interested in visualization as a tool for insightful data exploration. She is an enthusiastic proponent of MATLAB as the preferred environment for data visualization and algorithm prototyping.

Swapnonil Banerjee is a theoretical physicist with a PhD in Physics and a Bachelors degree in Electronics and Telecommunications Engineering. He has extensive MATLAB development experience in the areas of signal processing, numerical data modeling, curve fitting, differential calculus, and Monte Carlo simulations.

Acknowledgement

We gratefully acknowledge the support of several individuals and organizations in developing this book.

We would like to begin with a special thanks to David Woo for being constantly encouraging and providing valuable, actionable ideas for the book.

We are grateful to our family and friends for their love and the pride they take in our work. It has been very nice to have the enthusiasm of Shuman Majumdar on our behalf.

We would like to thank our reviewers John Bemis, Adee Ran, Ashish Uthama, and David Woo for patiently providing detailed critique of our work and great suggestions for improvement.

Importantly, we would like to thank Yashodhan Dere, Kedar Bhat, Dipesh Panchal, Joanna Finchen, and the rest of the team at Packt for being supportive throughout this project.

We would like to thank MathWorks™ for their book program that made software licenses available to us. We are grateful for their well maintained MATLAB Central File Exchange program that showcases the work of so many in the MATLAB community whose contributions we were able to build upon.

We would like to thank the University of California, Irvine and Stanford University for maintaining great public use data repositories that we were able to leverage.

Finally, we would like to thank Daniel B Carr, professor at George Mason University, who introduced us to the subject of data visualization.

About the Reviewers

Dr. John Bemis is a senior manager at Baker Hughes, Inc. John holds a BA degree in Chemistry from Grinnell College and a PhD in Inorganic Chemistry from the University of Wisconsin. John has 16 years of professional software development experience starting at TecMag Inc., designing and implementing user interfaces for magnetic resonance instrument data acquisition and control. He has spent the last 12 years at Baker Hughes, Inc., first developing MATLAB based data analysis software for magnetic resonance applications, and most recently as manager of the software technical project engineers for the Drilling and Evaluation Technology division.

Adee Ran received a BS degree in Electrical Engineering in 1991 and an MS degree in Electrical Engineering in 2000, both from the Israel Institute of Technology (Technion). He is a physical-layer communication systems architect at Intel's Israel Design Center in Haifa, Israel. He is also an active member of the IEEE 802.3 Ethernet Working Group and a devoted user and programmer of MATLAB ever since the days of Version 3.5.

Ashish Uthama is a developer in the Image Processing Toolbox team at MathWorks, makers of MATLAB. He has a Bachelor's degree in Electronics and Communication from PESIT, Bangalore, India and a Master's degree in Applied Science from UBC, Vancouver, Canada.

David Woo manages a team of algorithm developers in the Genetic Analysis R&D division at Life Technologies where data analysis and visualization are an important part of everyday work. He has a Master's degree in Electrical Engineering and 12 years of experience developing biotechnology instrumentation including DNA sequencers and real-time PCR thermal cyclers. He holds several patents in this area. In particular, he and his team focus on the data transformation from the time series images of biochemical reactions that produce fluorescence to biologically meaningful DNA base calls and gene quantification. Bridging the gap between engineering and biology is challenging, but ultimately rewarding, as the results improve health care and push the understanding of molecular biology. DNA sequencing has grown immensely since the completion of the first human genome, and genetic testing is rapidly becoming an indispensible tool to doctors, but as the volume of data increases, so does the need for data analysis and visualization.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Customizing Elements of MATLAB Graphics—the Basics 7

Introduction 7
Making your first MATLAB plot 10
Laying out long tick labels without overwriting 14
Using annotations pinned to the axes 18
Tufte style gridding for readability 22
Bringing order to chaos with legends 28
Visualizing details with data transformations 34
Designing multigraph layouts 37
A visualization to compare algorithm test results 44

Chapter 2: Diving into One-dimensional Data Displays 49

Introduction 50
Pie charts, stem plots, and stairs plots 50
Box plots 54
Sparklines 59
Stacked line graphs 63
Node link plots 66
Calendar heat map 71
Distributional data analysis 74
Time series analysis 79

Chapter 3: Graduating to Two-dimensional Data Displays 87

Introduction 88
Two-dimensional scatter plots 88
Scatter plot smoothing 92
Bidirectional error bars 95
2D node link plots 97

Dendrograms and clustergrams 100
Contour plots 103
Gridding scattered data 107
Choropleth maps 111
Thematic maps with symbols 114
Flow maps 117

Chapter 4: Customizing Elements of MATLAB Graphics—Advanced 123
Introduction 123
Transparency 123
Lighting 128
View control 135
Interaction between light, transparency, and view 138

Chapter 5: Playing in the Big Leagues with Three-dimensional Data Displays 143
Introduction 143
3D scatter plots 144
Slice (cross-sectional views) 148
Isosurface, isonormals, isocaps 156
Stream slice 160
Stream lines, ribbons, tubes 162
Scalar and vector data with a combination of techniques 166
Explore with camera motion 170

Chapter 6: Designing for Higher Data Dimensions 175
Introduction 176
Fusing hyperspectral data 176
Survey plots 180
Glyphs 186
Parallel coordinates 192
Tree maps 195
Andrews' curves 197
Downsampling for fast graphs 199
Principal Component Analysis 202
Radial Coordinate Visualization 206

Chapter 7: Creating Interactive Graphics and Animation 209
Introduction 209
Callback functions 210
Obtaining user input from the graph 216
Linked axes and data brushing 220

The magnifying glass demo 226
Animation with playback of frame captures 231
Stream particle animation 233
Animation by incremental changes to chart elements 236

Chapter 8: Finalizing Graphics for Publication and Presentations 243
Introduction 243
Export formats and resolution 244
Vector graphics for inclusion into documents 247
Preserving onscreen font size and aspect ratios 250
Publishing code and graphics to a webpage 253

Appendix: References 259

Index 261

Preface

MATLAB Graphics and Data Visualization is a cookbook with recipes providing a menu of graphs to rapidly identify the type of plot appropriate for your data. The step-by-step recipe style allows applying the techniques to your data within a short time. Several attractive customizations are provided as functions that can be easily integrated into your data analysis workflow. The hand created indexing into the recipes makes navigation through the book simple and powerful to quickly locate what you need. The book approaches the topic of visualization using data dimensionality and complexity as the central themes to organize the techniques.

What this book covers

Chapter 1, *Customizing Elements of MATLAB Graphics—the Basics,* introduces how to work with MATLAB handle graphics technology to customize graphs built in MATLAB. It covers recipes showing how to change basic graph elements such as layout, gridding, labels, and legends. It also forays into the use of color for depicting information.

Chapter 2, *Diving into One-dimensional Data Displays,* takes a tour of options available for one-dimensional data visualizations, beginning with common chart types such as line plots, bar plots, scatter plots, pie charts, stem plots, and stair plots. Further recipes cover box plots and specialized designs such as sparklines, stacked line graphs, and node link plots. A recipe is devoted to the use of heat maps for presenting daily data directly on a calendar. Final recipes point to analysis approaches such as distributional data analysis and time series data analysis, which may require specialized plots for visualizing the results.

Chapter 3, *Graduating to Two-dimensional Data Displays,* takes a tour of options available for two-dimensional data visualizations, beginning with common chart types, such as scatter plots, and options for scatter plot smoothing. Further recipes cover designs such as 2D node link plots, dendrograms, and clustergrams. Further recipes cover contour plots. A recipe is devoted to deal with data collected on non-uniform grids. Further recipes cover specialized graphics for presenting data on maps with choropleth maps, thematic maps with symbols, and flow maps.

Chapter 4, *Customizing Elements of MATLAB Graphics—Advanced,* introduces advanced features you can customize for graphics built with MATLAB, namely transparency, lighting, and view control.

Chapter 5, *Playing in the Big Leagues with Three-dimensional Data Displays,* takes a tour of options available for three-dimensional data visualizations with emphasis on volumetric data. It begins with 3D scatter plots. Further recipes cover designs using slices, isosurfaces, isonormals, and isocaps for scalar data visualization. Further recipes cover use of stream slices and various options for depicting direction using lines, ribbons, or tubes for vector data visualization. Several recipes pool the basic 3D techniques with lighting and view control mechanisms to create effective ways for 3D data exploration.

Chapter 6, *Designing for Higher Data Dimensions,* takes a tour of visualization options for higher data dimensions. Recipes cover the use of glyphs and parallel coordinates to demonstrate how to represent multiple dimensions in 2D. Further recipes show how to code the extra dimensions among available graphical features to achieve the same objective. Additional recipes show how to transform the data using techniques such as the principal component analysis or radial coordinate projections such that the key data dimensions that allow discrimination between them can be brought into focus.

Chapter 7, *Creating Interactive Graphics and Animation,* showcases MATLAB's capabilities of creating interactive graphics and animations. Recipes cover the essentials of programming callback functionality to add custom behavior to user interactions. Further recipes cover ways to obtain user input directly from the graph, including exploratory techniques such as data brushing and linking. Other recipes cover how to animate a sequence of frames, or use erase and redraw strategies to create animation effects.

Chapter 8, *Finalizing Graphics for Publication and Presentations,* covers options to adjust the image quality and formatting requirements for different presentation goals, including tips to keep in mind while designing graphics for presentation or publication in either hard copy or electronic formats.

Appendix, *References*, provides supplementary material.

What you need for this book

A basic MATLAB installation will be required. The code was developed using MATLAB R2012a.

One recipe needs the Image Processing Toolbox™. Several recipes require the Statistics Toolbox™. A couple of recipes make references to the Statistical Toolbox™, the Mapping Toolbox™, and the Bioinformatics Toolbox™ (however, a fallback implementation is provided in these cases so that you can use these recipes independent of these toolboxes).

Who this book is for

The book is targeted for practitioners in the academia and industry interested in either presenting the results of their specific analysis or doing exploratory data visualization. The data itself could come from any source, and the options to import the data into MATLAB are discussed in the book. A basic familiarity with MATLAB programming is assumed. However, advanced MATLAB experience is not needed. The recipes are detailed and broken into simple steps. It is intended as a handbook for creating compelling graphics that one can easily apply to new data. Several attractive options for customizations are made available as functions that can be easily integrated into any data analysis workflow.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The command `xlsread` allows you to read the numeric columns into the variable `numericData`, and the alphanumeric columns into the variable `headerLabels`."

A block of code is set as follows:

```
plot(x,y1);
line([mean1 mean1],get(gca,'ylim'));
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the drop-down arrow next to the **publish** icon on the toolbar to access the **Edit Publish Configurations for...** option."

Warnings or important notes appear in a box like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/3165OT_MATLAB_Graphics_and_Data_Visualization_Cookbook.pdf`

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Customizing Elements of MATLAB Graphics—the Basics

In this chapter, we will cover:

- Making your first MATLAB plot
- Laying out long tick labels without overwriting
- Using annotations pinned to the axes
- Tufte style gridding for readability
- Bringing order to chaos with legends
- Visualizing details with data transformations
- Designing multigraph layouts
- A visualization to compare algorithm test results

Introduction

MATLAB provides a rich and accessible environment for building data displays using **MATLAB graphics objects**. Each graphics object has a set of characteristics you can manipulate via their **property** settings. While each property has a default factory setting, you can set user-defined values for these properties by accessing them programmatically, via their unique identifier called a **handle**; or interactively, via the **property editor**. This is the fundamental way for customizing MATLAB graphics.

The different types of graphics objects may be hierarchically related. For example, a plot element such as a line needs an axes object to act as a frame of reference. The axes object needs the figure graphics object to hold it. Sometimes, it is possible to affect the property settings of a whole group of graphics objects using a single command, depending on the nature of their inter-relation. The recipes in this chapter show some of the commonly used customizations using handle graphics manipulation, applicable to all types of MATLAB plotting.

See MATLAB Product pages on **Handle Graphics Objects** for a complete exposition of the handle graphics technology.

Programmatic manipulation of graphics object properties

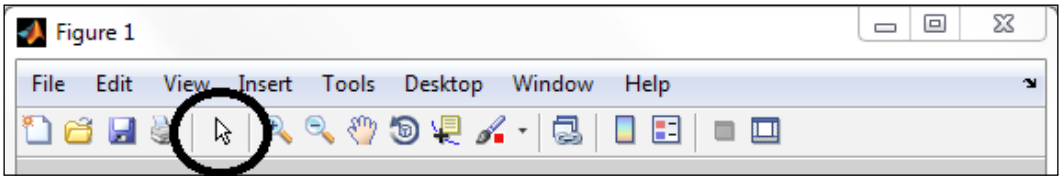
All plotting-related MATLAB commands implicitly create the figure and axes graphics objects and direct their output to the most recent figure and its most recent **child** axes object. Explicitly, you can use the command `figure` at the MATLAB console to launch a new MATLAB figure window; and the command `axes` to create a new axes object. You can create multiple axes objects on the same figure. Each axes object will be children of the **parent** figure object. Data is plotted onto the axes object with current focus. The current figure handle can be accessed by the command `get current figure` or `gcf`. The handle to the current axes can be accessed by the command `get current axes` or `gca`.

`get` (and `set`) commands apply to all MATLAB graphics objects and will allow to query (and define) their user-settable attributes as follows:

Select the **Plot Edit** button (the fifth button in the figure toolbar) to get into the plot edit mode. Then, select any object on the current figure (figure or axes or annotation objects). This becomes your graphic object with current focus. Run `get(gco)` at the console to see the complete list of user-definable attributes and their default settings for the graphic object in current focus. Use the `get` and `set` commands to alter their default values programmatically, as follows:

```
get(gco,'Property Name');
set(gco,'property Name',value);
```

The **Plot Edit** button is circled in the following screenshot:

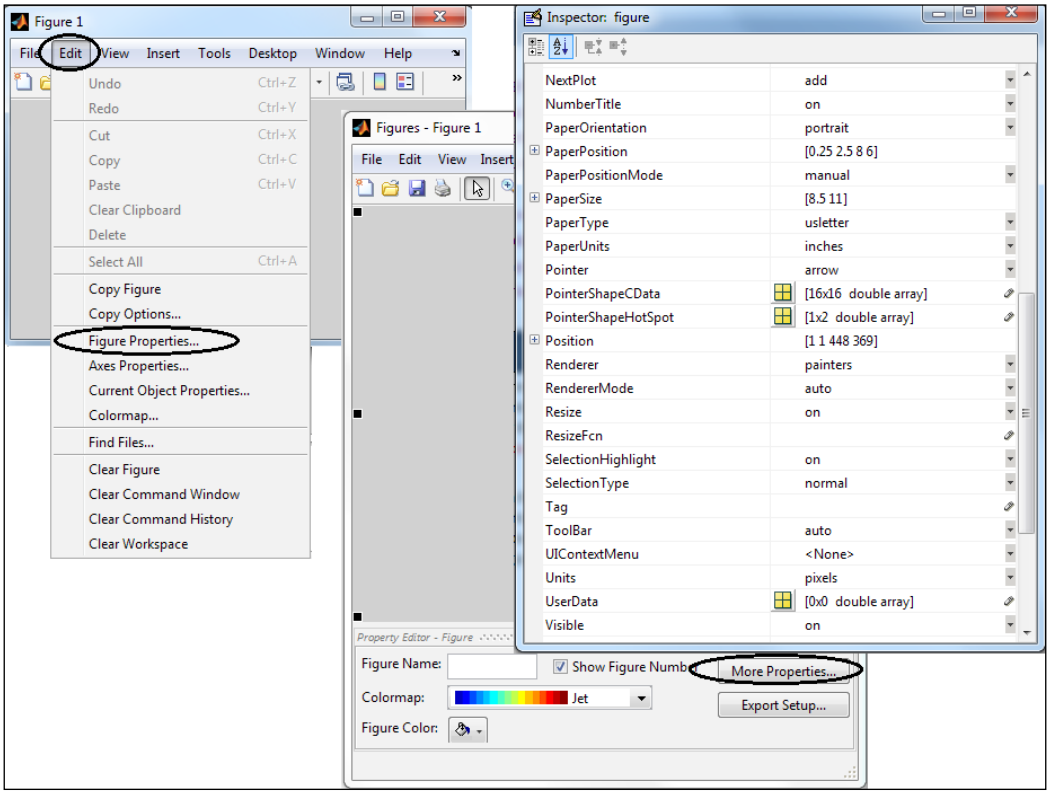


Altering graphics object properties via the Property Editor

An alternate way to change the figure and axes property values (and property values of other MATLAB graphic objects) is by means of the **MATLAB Property Editor**. Opening up the detailed property editor window will list every attribute that can be customized for the type of graphics object you are using.

The steps to use the figure property editor wizard are shown in the following screenshot: **Edit** | **Figure Properties** | **More Properties** bring up the **Property Inspector Table** where the entries can be directly altered. See **Axes Properties** and **Current Object Properties** in the drop-down options under the **Edit** menu item for the complete list of user-definable attributes.

The following screenshot shows steps to interact with the Property Editor for reviewing attributes available for customization for any MATLAB graphics object:



You can access the property editor for other graphics objects you may be using by selecting the object in plot edit mode, right-clicking on the object, and selecting **Show Property Editor**.

Once the appropriate parameters and their desired settings are identified using the Property Editor, the user can make a command line statement to set those properties to the new values and thus repeat the customizations every time the same graph is generated, programmatically.

Making your first MATLAB plot

This recipe takes you through the basic commands for creating a plot using MATLAB. It demonstrates how to import data from an Excel spreadsheet, how to create a basic plot with it, and how to add basic annotations. It will also teach how to add a linear least squares fit to the data. It will show how you locate the handle to this line object you created, and how to change some of its properties to impact your visualization.

Getting ready

The file `TemperatureXL.xls` is part of the code repository accompanying this book. This spreadsheet has two columns of numeric data with alphanumeric headers in the first row. The first step is to import the data into the MATLAB workspace with the `xlsread` command:

```
[numericData headerLabels]=xlsread('TemperatureXL.xls');
```

How to do it...

Perform the following steps:

1. Plot the data. (Sort the data before plotting if order is not important. Sorting helps to easily assess trends in the data or lack of it.)

   ```
   [sortedResults I] = sort(numericData(:,1));
   plot(numericData(I,1), numericData(I,2),'.');
   ```

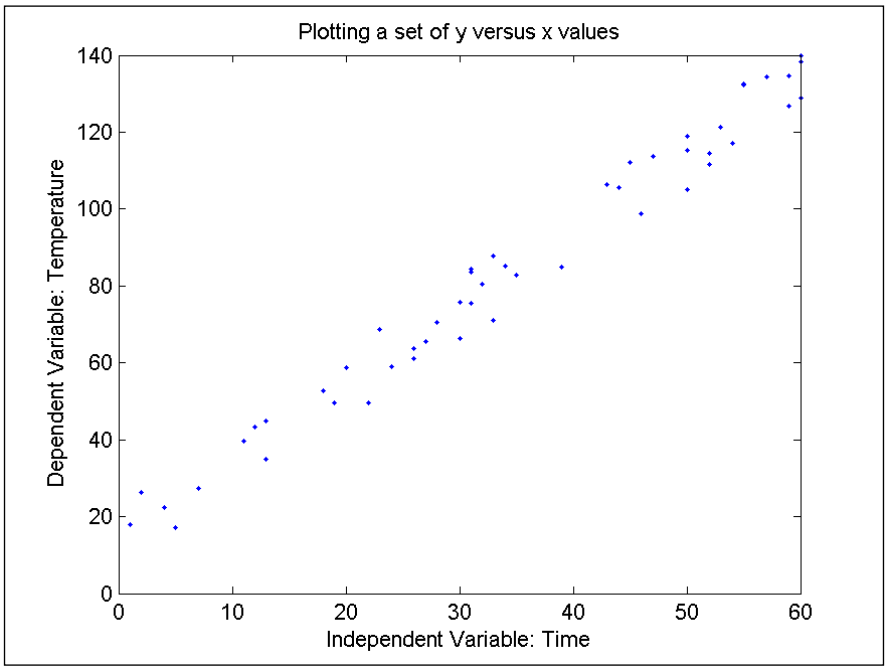
2. Label the `x` and `y` axis:

   ```
   xlabel(['Independent Variable: ' headerLabels{1}]);
   ylabel(['Dependent Variable: ' headerLabels{2}]);
   ```

3. Add a title:

   ```
   title('Scatter plot view of sorted data');
   ```

The output at this point should be as follows:



4. Estimate the trend (using a linear least squares fit):

```
p = polyfit(numericData(I,1),numericData(I,2),1);
y = polyval(p,numericData(I,1));
```

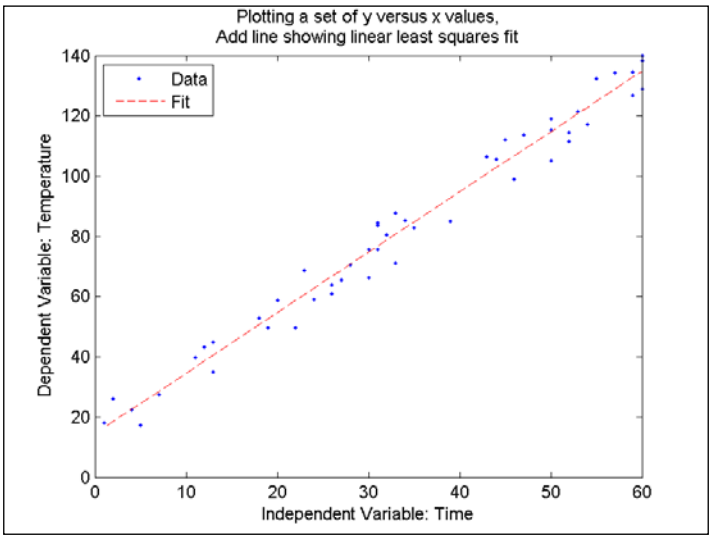
5. Overlay the trend line from step 4 on the current axes using a dashed line style. You can also specify the color of the line as part of the **linespec** definition.

```
hold on;
plot(numericData(I,1),y,'r--');
```

6. Add a legend:

```
legend({'Data','Fit'},'Location','NorthWest');
```

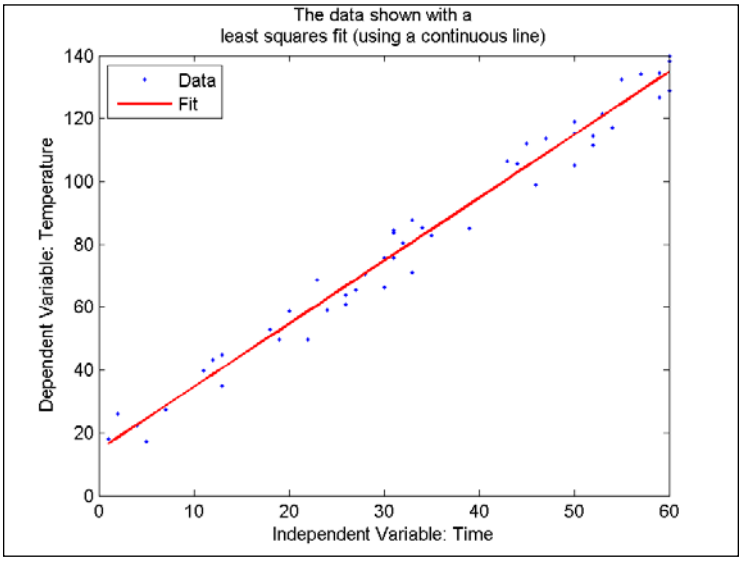
The output at this point should be as follows:



7. Locate the trend line based on the color you set for it. Change the line style to continuous instead of dashed. In this step, you should specify the color of the line with a three element vector of actual RGB values.

```
set(findobj(gca,'Color',[1 0 0]),...
  'Linestyle','-','Linewidth',1.5);
```

The effect of step 7 is as follows:



How it works...

The command `xlsread` allows you to read the numeric columns into the variable `numericData`, and the alphanumeric columns into the variable `headerLabels`. You can specify the location of the Excel file, the sheet name, and the exact columns to read by using the command `xlsread`.

In step 1, you sorted the data. The vector `I` held the sort index order of the data such that, `sortedResults = numericData(I,1)`.

In step 2, you plotted this ordered data. Note that the `plot` command does not use any specific marker for the data points, and connects the successive data points with a continuous line by default. Here, as you specified the marker style without the line style, no line style was assumed and the points denoted with the specified marker were not connected. Another alternative to create this type of graph is to use the command `scatter`.

In step 3, you labelled the `x` and `y` axis and added a title to the graph. The **string concatenation operator []** is used to construct the labels and titles using some hardcoded text and the column headers read in from the Excel spreadsheet. Note how you used cell arrays to **break the title into two lines** with each string representing the entry for a line.

In step 4, you calculated the **linear least squares fit** to this data using `polyfit` which fits a polynomial of a degree of your choice (here, degree = 1 for a linear fit). The parameters of the line derived by `polyfit` can be evaluated with a vector of your choice. In this case, you used the x values shown in this plot.

In step 5, you overlaid the trend line on the plot using a dashed style. The `hold` command ensures that the display area is not cleared before adding the new line.

In step 6, you added a legend to the graph.

In step 7, you located the handle to the line object you created based on the color you set for it. Using this handle you changed its style to a continuous line style, and its thickness to a user-defined value. This was a desirable change, since the dashed style is distracting the user without adding any valuable information.

The `findobj` command allows you to look for a graphics object with a property name and a property value pair. In this case, you looked for some graphic object that is a child of the current axis that is red in color. Once you found that handle (returned by `findobj`), you called the `set` function with that handle using a nested operation and reset the thickness and style of the trend line.

Note that positional coordinates, as used in this example, are a great way to present numerical data.

Takeaways from this recipe:

- Use positional coordinates to represent numerical data
- Sort data before plotting (if order is not important)
- Keep discontinuous lines that create visual noise to a minimum

See also

Look up **MATLAB help** on the `plot`, `polyfit`, `polyval`, `legend`, `sort`, `xlsread`, `set`, `get`, `findobj`, and `scatter` commands.

Laying out long tick labels without overwriting

You used **axis labels** in the *Making your first MATLAB plot* recipe. Another kind of label used in graphs is **tick labels**, which are the numeric or alphanumeric labels associated with the tick marks on the axes. When the plotting-related commands are invoked, MATLAB sets a default positioning and numerical tick labels. As this recipe shows, you can customize the content and positioning of these labels. When you have long tick labels (such as dates), this recipe shows how to rotate the labels by an arbitrary angle to avoid overwriting.

Getting ready

In this recipe, you will use gene expression data from 16,063 genes on 14 types of cancer for 198 samples. This data is part of the code repository accompanying this book. It was obtained from the machine-learning data repository maintained by the Department of Statistics at Stanford University.

```
load 14cancer.mat
```

How to do it...

Perform the following steps:

1. Display the expression levels for gene index 2,798 with a bar chart, error bars, and associated annotations:

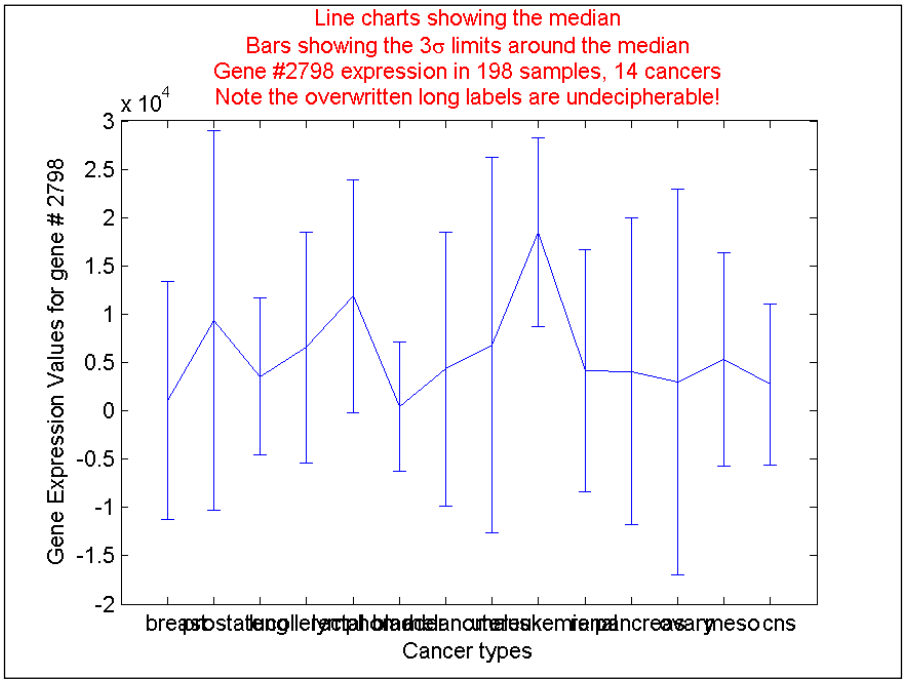
```
% Calculate the mean and standard deviations
% for each type of cancer
expressionLevel = [Xtrain(:,2798); Xtest(:,2798)];
cancerTypes = [ytrainLabels ytestLabels];
for j = 1:14
  indexes = expressionLevel(find(cancerTypes==j));
```

```
  meanExpressionLevel(j) = median(indexes);
  stdExpressionLevel(j) = 3*std(indexes);
end

% Plot the median data with bars around it showing the 3
% sigma extent of the data in that group
errorbar(1:14,meanExpressionLevel,...
  stdExpressionLevel,stdExpressionLevel);

% Add annotations
ylabel('Gene Expression Values for gene # 2798');
xlabel('Cancer types');
title({'Line charts showing the median',...
 'Bars showing the 3\sigma limits around the median',...
 'Gene #2798 expression in 198 samples, 14 cancers',...
 'Note the overwritten labels are undecipherable!'},...
 'Color',[1 0 0]);
```

2. Add the tick labels using a custom font size:

```
set(gca,'Fontsize',11,'XTick',1:14,'XTickLabel',...
  classLabels);
```

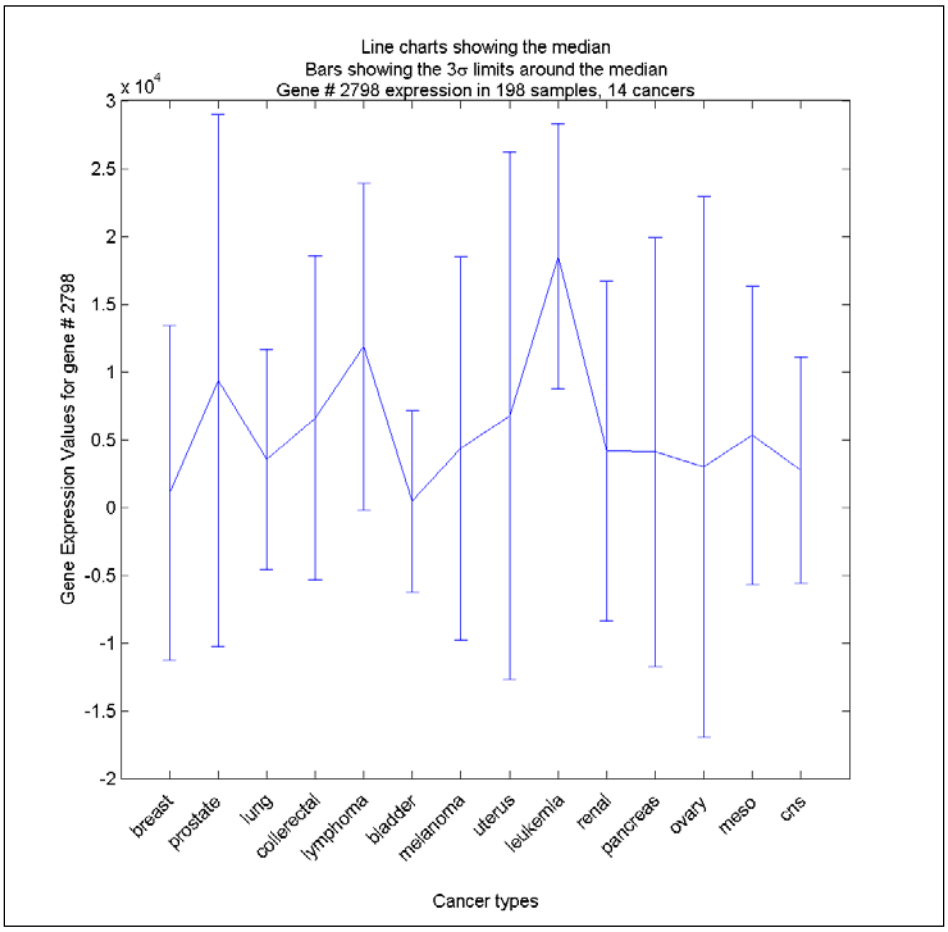
The output at this point should be as follows:



3. Rotate tick labels:

```
rotateXLabels(gca, 45);
```

`rotateXLabels` has the following effect on the x tick labels:



How it works...

The previous screenshot uses line chart with error bars to show the expression level for various cancers for a particular gene.

In step 1, you extracted the expression level data for gene index number 2,798 and collected the median and the three standard deviation values for each cancer type. Three standard deviations on either side encompass 99.7 percent of the data if a normal distribution is assumed. Next, you plotted this series with the `errorbar` command. This produced a plot with the median values for each group connected with a line; additionally for each group, it displayed bars above and below the median point. You plotted the three sigma values for each group on either side of the medians.

Note that, again, you have used positional coordinates to represent your data. This is a visualization best practice. You used error bars around your central representative data point to reflect a realistic picture of the difference in the data between groups. This is also a visualization best practice. From the previous screenshot, note that the expression level for leukemia has a higher mean. However, because the three sigma bars from other cancers overlap with the data from leukaemia, this gene expression level alone cannot be used as a definitive indicator for leukaemia.

In step 2, you added the cancer name labels to the x tick marks. You first set the tick positions with the vector 1 through 14 and then set the corresponding tick label entries for those positions with the array of strings containing the cancer class names. Since the label names were long, you observed significant overwriting that rendered the tick labelling unreadable. Resizing the figure and reducing the font size are some alternatives that could help in this case. However, a more compelling solution is to rotate the labels.

In step 3, you rotated the labels by 45 degrees so that the labels become readable. You used the function `rotateXLabels.m` that is part of the code repository accompanying this book. The function takes two arguments, the axis handle on which to work and the angle by which to rotate the labels in degrees. Internally, this function creates text annotations at the designated tick positions and rotates them by the angle specified in the second argument. This function is adapted from the submission by Ben Tordoff on MATLAB File Exchange. While the tick label rotation solves the problem of over-writing of the labels, remember that the steeper the angle, the more difficult it is to read.

A few additional steps (included in source code lines 38 – 43) for resizing the figure and the axes as well as updating the title will give you the screenshot shown earlier.

Takeaways from this recipe:

- Use positional coordinates to compare between data
- Use error bars (or some measure of variance) around your representative data point to realistically reflect the difference in the data between groups
- Use low angles of rotation for the x tick labels, when needed

See also

Look up **MATLAB help** on the `errorbar` command.

Using annotations pinned to the axes

MATLAB provides an interface to place custom elements on the graphics using the command `annotation`. Lines, arrows, double-ended arrows, text arrows, textboxes, ellipses, and rectangles are all valid elements you can overlay on your basic graphic to convey information.

Getting ready

In this recipe, you will plot the standard normal distribution and add a text arrow to point out the location of the mean value on the graphic.

```
load stdNormalDistribution;
```

How to do it...

Perform the following steps:

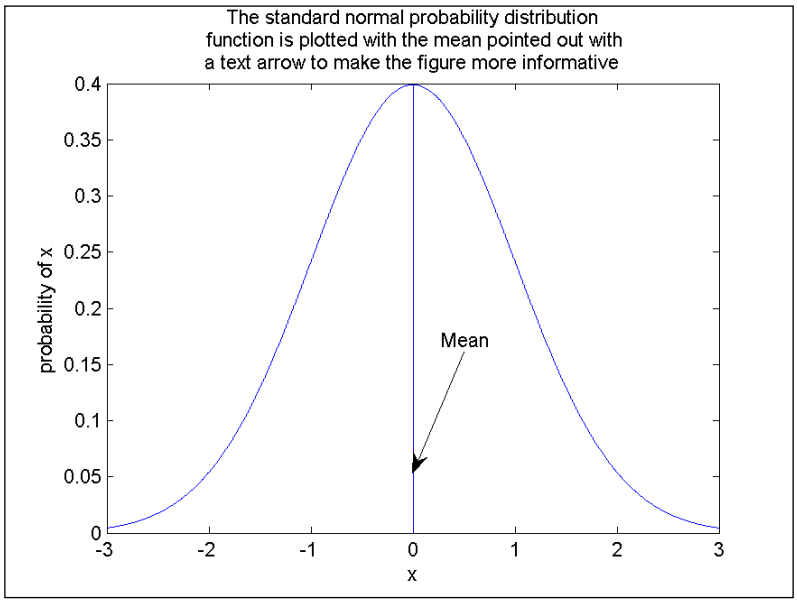
1. Plot the data. Add the line at the mean position:

```
plot(x,y1);
line([mean1 mean1],get(gca,'ylim'));
```

2. Add the text arrow annotation component by first converting the desired location for the arrow from data space coordinates to normalized figure units using `dsxy2figxy` and then invoking the `annotation` command:

```
[xmeannfu ymeannfu]= dsxy2figxy(gca,[.5,0],[.15,.05]);
annotation('textarrow',xmeannfu,ymeannfu,'String',...
  'Mean');
```

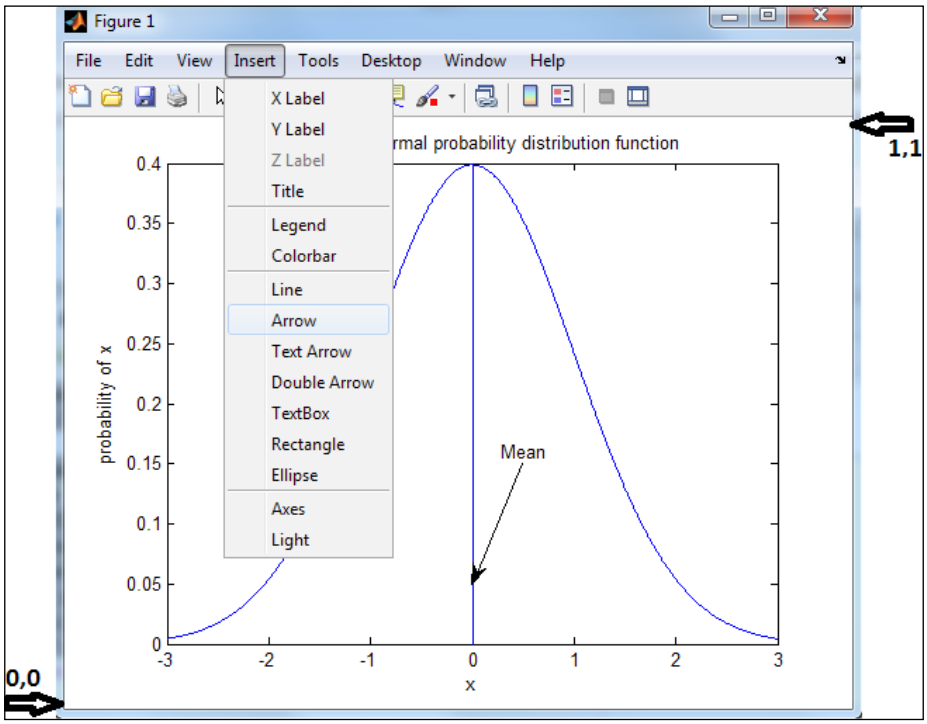
The output is as follows:



How it works...

The MATLAB command `annotation` works with **Normalized Figure Units**. As the name suggests, these units range from 0 to 1 in both, the horizontal and vertical directions and cover the entire area of the figure. The lower bottom left is 0, 0; the upper right corner is addressed with 1, 1. To place custom elements such as axes or buttons on the figure, you will need to use normalized figure units. The function `dsxy2figxy` accompanies the documentation on annotations from MATLAB. It allows you to convert coordinates from data space units to normalized figure units. It is provided as part of the code repository accompanying this book.

Note that when you need to place an annotation on the screen without referencing the data space axes, you can do that from the **Insert** file menu item directly, shown as follows:



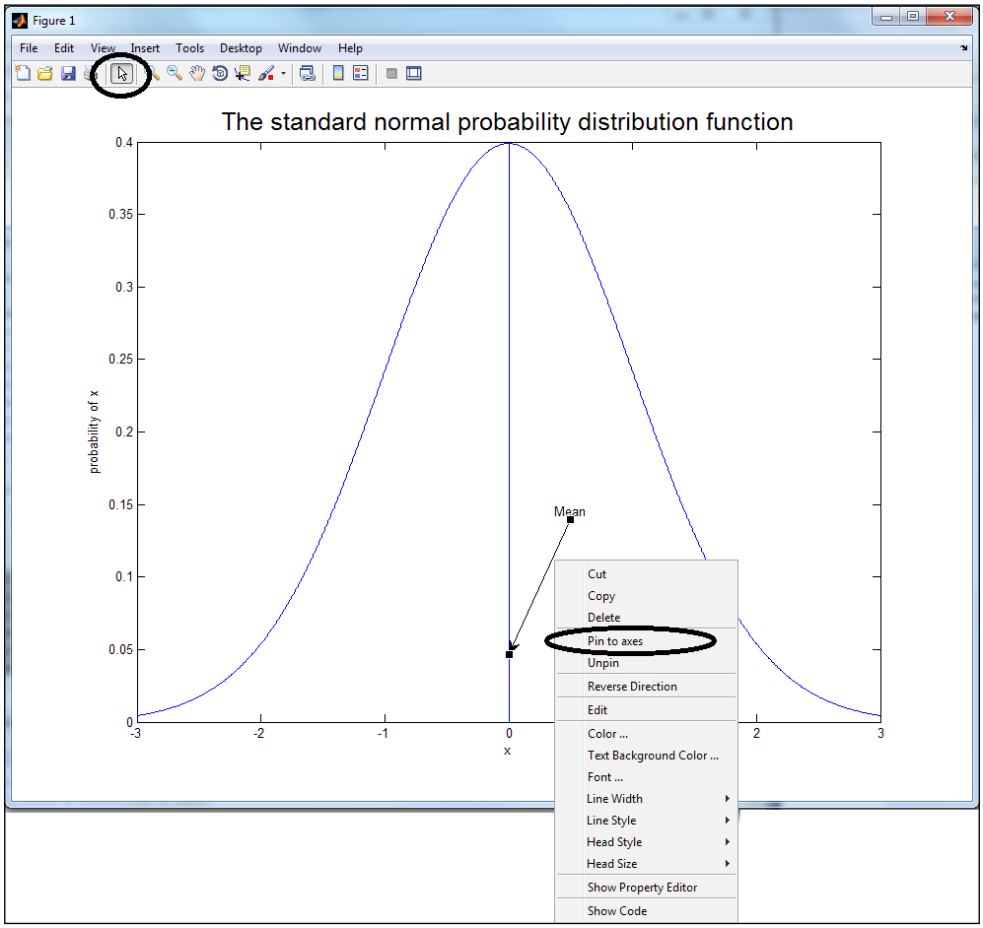
You can later look up the `Position` property value of your annotation object, using `get(gco,'position')` and then make the future placements of this object on your graphic using those positional coordinates, programmatically.

There's more...

If you resize the figure, the annotations can change shape but continue to point to the same locations on the graph because they and the figure use normalized coordinates. However, if you shift the axes up, down, left, or right within the figure, the annotations remain fixed in figure space and do not move with the axes. In order to ensure that annotations stay connected to the data with which you have associated them, you have to pin the annotation to the data space. The pinning can be achieved using the function `annotation_pinned` as shown in the following code snippet. This function is adapted from a submission by Fred Gruber on MATLAB File Exchange.

```
annotation_pinned('textarrow',[.5,0],[.15,.05],...
  'String','Mean');
```

The other alternative is to edit the plot in the **Plot Edit** mode and manually pin the annotation component to their location as shown in the following screenshot:



Takeaways from this recipe:

- Use components such as arrows and text labels to provide additional annotations to improve information content of the graphics

See also

Look up **MATLAB help** on the `linspace`, `annotation`, `dsxy2figxy`, and `line` commands.

Tufte style gridding for readability

Using grid lines is a great practice because they guide the eye and hence make numerical data easier to read and compare. The MATLAB command `grid` is used to turn the default grid lines on or off. This recipe shows the use of these default lines. This recipe also demonstrates how to add alternate grid lines in different line styles and at customized intervals. The MATLAB command `line` can be used to create the grid lines customized for your needs.

Getting ready

In this recipe, a marketing dataset with responses to 14 questions from 8,993 responders has been used. The data was obtained from the machine learning data repository maintained by the Department of Statistics at Stanford University, and is included as part of the code repository with this book.

```
load MarketingData.mat
```

How to do it...

Perform the following steps:

1. Extract the ethnicity and income group information:

```
% Initialize y
y = NaN(length(ANNUAL_INCOMEL),...
  length(ETHNIC_CLASSIFICATION));

% Each data point in y has the number of responses for a
% given income group and ethnic classification
for i = 1:length(ETHNIC_CLASSIFICATION)
  forThisGroup = find(data(:,13)==i);
  for j = 1:length(ANNUAL_INCOMEL)
    y(j,i) = length(find(data(forThisGroup,1)==j));
  end
end
```

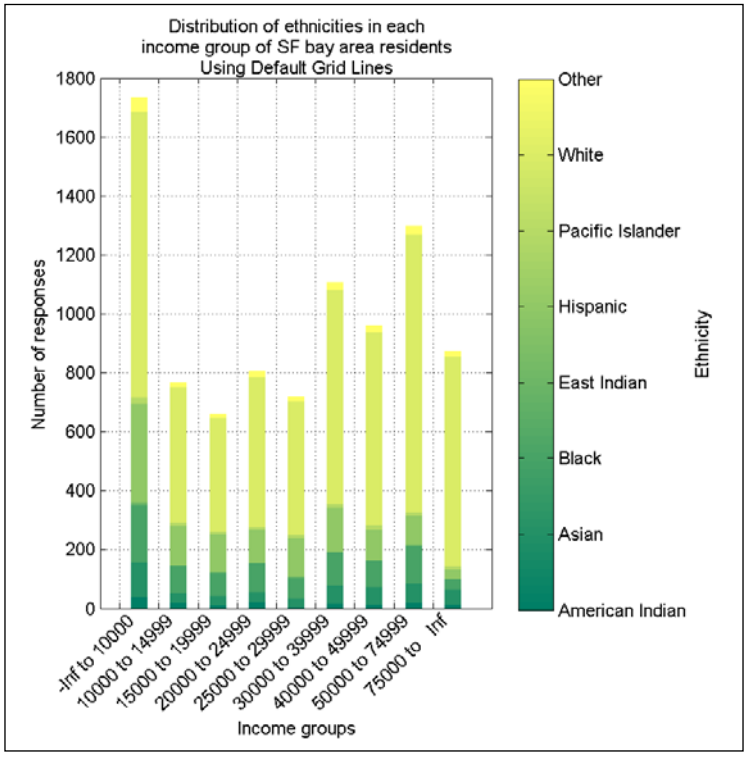
2. Generate a stacked bar plot to show the distribution of ethnicities within each with income group:

```
% Declare figure dimensions
figure('units','normalized',...
  'position',[ 0.3474    0.3481    0.2979    0.5565]);
axes('position',[ 0.1300    0.2240    0.6505    0.6816]);

% make the bar plot
bar(y,.4,'stacked','linestyle','none');

% Use an alternative predefined colormap
colormap('summer');

% Add annotations
set(gca,'Fontsize',11,...
  'Xtick',[1:9]-.5,...
  'XTickLabel', [num2str(ANNUAL_INCOMEL') ...
  repmat(' to ',9,1) ...
  num2str(ANNUAL_INCOMEU')]);
rotateXLabels(gca, 45);
ylabel('Number of responses','Fontsize',11);
xlabel('Income groups','Fontsize',11);
title({'Distribution of ethnicities in each',...
  'income group of SF bay area residents',...
  'Using Default Grid Lines'});
box on;

% Add annotations to the color bar
h=colorbar;
set(h,'Fontsize',11,'ytick',1:8,'yticklabel',...
  ETHNIC_CLASSIFICATION);
ylabel(h,'Ethnicity','Fontsize',11);
set(gcf,'Color',[1 1 1]);
```

3. Turn the automated grid on:

```
grid on;
```

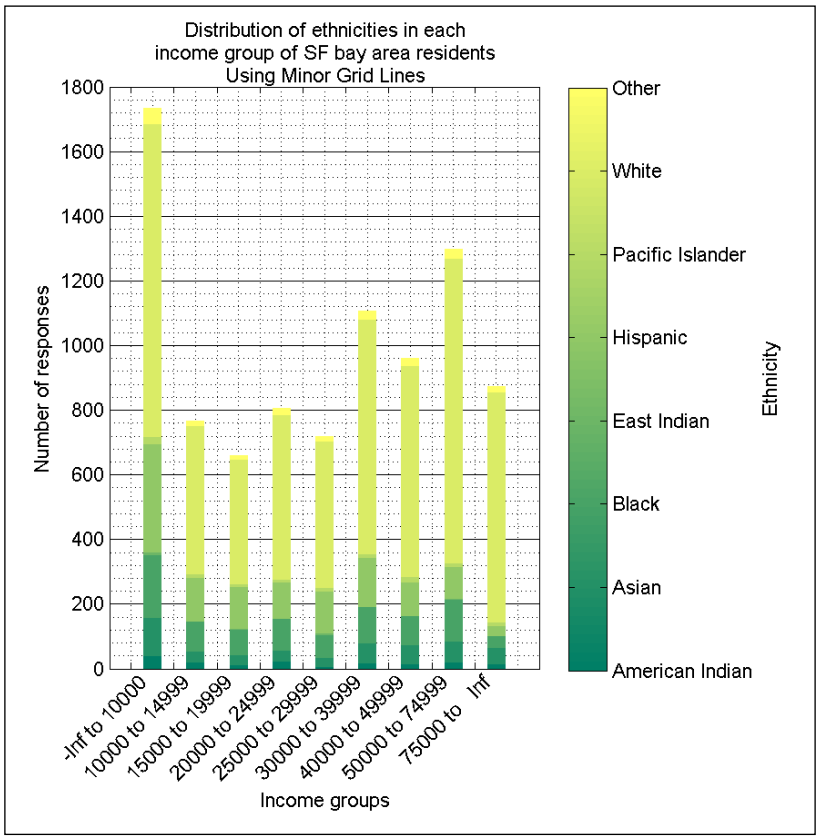
The output at this point should be as follows:



4. Turn the minor grid lines on (and update title):

```
grid minor;
title({'Distribution of ethnicities in each',...
  'income group of SF bay area residents',...
  'Using Minor Grid Lines'});
```

The output is as follows:



5. Turn the automated grid off:

```
grid off;
```

6. Add custom grid lines:

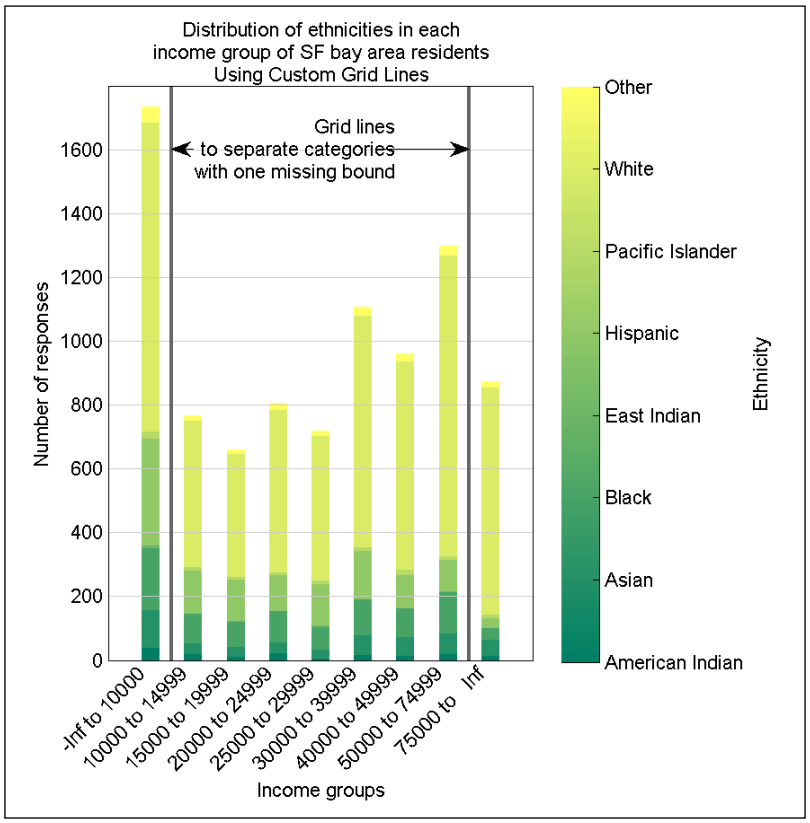
```
% Set axis limits
xlim([0 10]);ylim([0 1800]);

% Set y grid positions and draw lines (no x grid lines)
YgridPos = [0:200:1800];
set(gca,'ytick',YgridPos,'yticklabel',YgridPos);
xLimits = get(gca,'xlim');
line([xLimits(1)*ones(size(YgridPos),1); ...
  xLimits(2)*ones(size(YgridPos),1)],...
  [YgridPos; YgridPos],'Color',[.7 .7 .7],...
  'LineStyle','-');
```

```
XgridPos = [.5:9.5];
yLimits = get(gca,'ylim');

% Draw special y grid lines for separating data groups
line([XgridPos([2 9]); XgridPos([2 9])],...
  [yLimits(1)*ones(2,1) yLimits(2)*ones(2,1)]',...
  'Color',[.4 .4 .4],'LineStyle','-','Linewidth',2);

% Wipe out outer boundary for Tufte style bar plot
line([xLimits(1) xLimits(2)]',[YgridPos(1);...
  YgridPos(1)],'Color',[1 1 1],'LineStyle','-');
line([xLimits(1) xLimits(1)]',[YgridPos(1);...
  YgridPos(end)], 'Color',[1 1 1],'LineStyle','-');
line([xLimits(end) xLimits(end)]', [YgridPos(1);...
  YgridPos(end)],'Color',[1 1 1],'LineStyle','-');
line([xLimits(1) xLimits(2)]',[YgridPos(end); ...
  YgridPos(end)],'Color',[1 1 1],'LineStyle','-');
```

7. Add more annotations:

```
% Add an arrow and a text arrow annotation
[xmeannfu ymeannfu]= dsxy2figxy(gca,[2,1.5],[1600,1600]);
annotation('textarrow',xmeannfu,ymeannfu,...
  'String',{'Grid lines','to separate categories',...
  'with one missing bound'});
[xmeannfu ymeannfu]=...
  dsxy2figxy(gca,[6.5,8.5],[1600,1600]);
annotation('arrow',xmeannfu,ymeannfu);
title({['Distribution of ethnicities in each '...
  'income group'],'of SF bay area residents'});

% Remove the unnecessary ytick label at the end
for i = 1:length(YgridPos)-1;
cellticks{i} = num2str(YgridPos(i));
end
cellticks{i+1} = '';
set(gca,'ytick',YgridPos,'YTicklabel',cellticks);

% The colorbar doesn't need a box around it either
axes(h);box off;
```

The final output from this recipe is as follows:



How it works...

The previous screenshot shows the distribution of ethnicities in each of the nine income buckets among bay area residents. The largest number of responders belongs to the lowest income category. The ethnicity group White is the largest group within all income groups, followed by Hispanic, followed by Black.

This recipe explored grid lines with a stacked bar chart. The bar chart uses line length as the basic tool for comparison which is a recommended visualization practice. Automated grids and manually crafted grid lines were both demonstrated. For the custom grid lines, continuous and light color horizontal grid lines were chosen to aid in reading the data with reduced visual distraction. Unnecessary vertical lines were removed (as proposed by Edward Tufte). Special functional vertical grid lines were added to help visually distinguish categories that are missing one bound. Text arrow annotations were added to explain the intent of the bold grid lines. The principle of minimal and functional grid lines was thus demonstrated.

Note that you now have encountered three different ways of specifying colors for your graphic object. The first is by using the line spec (for example, 'r'), the second is by directly using the RGB values (for example, [0 0 1]), and in this recipe, you invoked an alternative built-in MATLAB `colormap summer` to get the colors for your data. A color map is a matrix of RGB values that represents a color scale, where the first row corresponds to the color with which to represent `cmin` and the last row corresponds to the color with which to represent `cmax`. `cmin` and `cmax` correspond to the minimum and maximum values in your data. When you make a plot, MATLAB automatically sets the `cmin` and `cmax` values from your data and ties it with the default colormap `jet`.

Note that another new option you exercised in this recipe was to manually set the **figure dimensions** in step 1. The default units for doing this are in pixel coordinates. However, you changed that to normalized coordinates by: `set(gcf,'units','normalized')`. This allowed you to specify positional dimensions independent of the resolution of the computer screen on which the code is executed.

Takeaways from this recipe:

- Use the length of a line to compare between numbers
- Use grid lines to make it easy to read the data
- Use the minimum number of grid lines needed
- Use grid lines to create data groups

See also

Look up **MATLAB help** on the `bar`, `line`, and `grid` commands.

Bringing order to chaos with legends

As graphics increase in complexity, it is common to use additional symbols, line styles, color, and such others to code for different layers of information. **Figure legends** help sort out this madness. Sometimes there are too many variables to code and the program needs to use clever combinations to code for the additional layers of information. This recipe demonstrates how the `legend` command from MATLAB helps to build legends of your choice. Furthermore, it shows how to make your own legends to accommodate a special need.

Getting ready

This recipe plots a set of ten normal distributions with different parameters.

```
load 10NormalDistributions
```

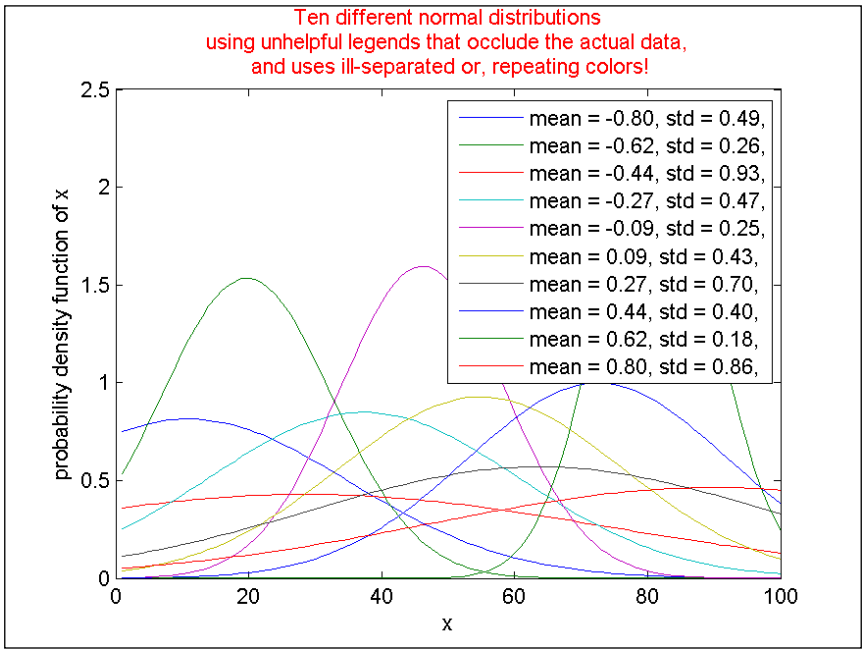
How to do it...

Perform the following steps:

1. Plot the data with basic labeling:

```
plot(dataVect');
title({'Ten different normal distributions',...
  ['using unhelpful legends that occlude the'...
  'actual data,'],...
  'and uses ill-separated or, repeating colors!'},...
  'Color',[1 0 0]);
xlabel('x');
ylabel('probability density function of x');
legend(legendMatrix);
```

   The output is as follows:



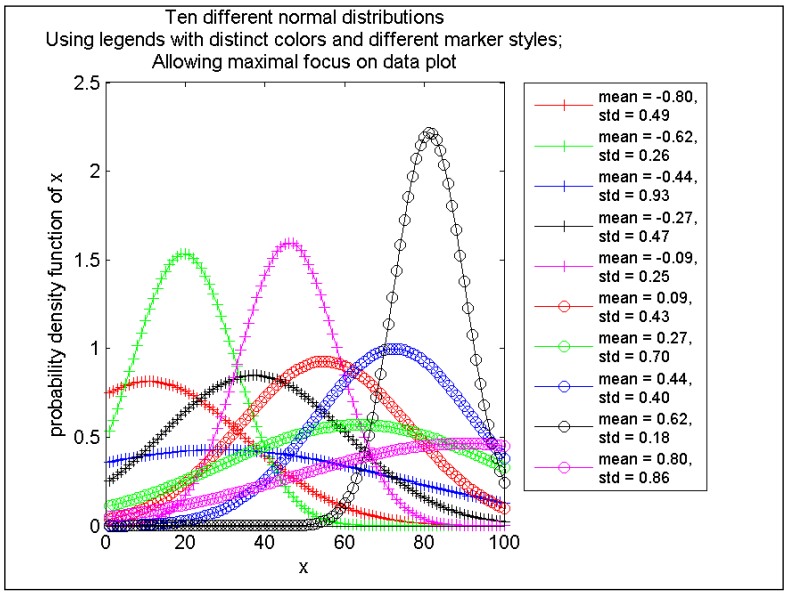
2. Define a set of line specifications:

```
LineStyles = {'-','--',':'};
MarkerSpecs = {'+','o'};
ColorSpecs = {'r','g','b','k'};
cnt = 1;
for i = 1:length(LineStyles)
  for j = 1:length(MarkerSpecs)
    for k = 1:length(ColorSpecs)
      LineSpecs{cnt} = [LineStyles{i} MarkerSpecs{j} ...
        ColorSpecs{k}];
      cnt = cnt+1;
    end
  end
end
```

3. Apply the new line specifications to visualize the distributions. Break the legend entries into two lines. Use smaller fonts to write them. And place the legend outside of graph area.

```
figure; hold on;
for i = 1:10
  dataVect(i,:) = (1/sqrt(2*pi*stdVect(i).^2))*...
    exp(-(x-meanVect(i)).^2/(2*stdVect(i).^2));
  plot(dataVect(i,:), LineSpecs{i});

% Multi line legend entries
  legendMatrix{i} = ...
    [sprintf('mean = %.2f, ',meanVect(i))...
    char(10) ...
    sprintf('std = %.2f',stdVect(i))];
end
title('Ten different normal distributions');
xlabel('x'); ylabel('probability density function of x');
legend(legendMatrix,'Location','NorthEastOutside',...
  'Fontsize',8);
box on;
```

The new output should be as follows:



How it works...

Line specs are composed of line style, line width, marker style, marker size, and the color of the line (which can be specified either with the character shorthand used here, or with the actual RGB values. All attributes of a line can be coded with information. The use of a few distinct colors combined with marker style and line style allows greater distinction between the set of ten lines in this example. Note that the colors chosen here were random. A sequential palette would have implied an order in the data. Too many colors in legends usually pose a perceptual challenge.

Adding the newline character `char(10)` between strings forces the legend entries to be broken into two lines.

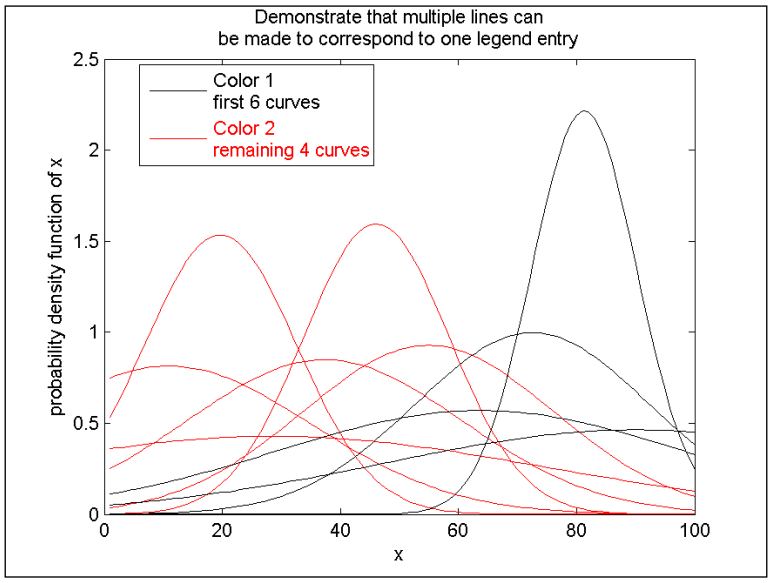
There's more...

The `legend` command internally increments a counter every time the plot command is called. Sometimes this automatic increment process is too restrictive. You may need several of your plot commands to correspond to just one legend entry shown as follows:

```
figure; hold;
plot(dataVect(1:6,:)','Color',[1 0 0]);
plot(dataVect(7:10,:)','Color',[0 0 0]);
h=legend(['Color 1' char(10) 'first 6 curves'],...
```

```
    ['Color 2' char(10) 'remaining 4 curves'],...
    'Location','Best');
c=get(h,'Children');
set(c(1:3),'Color',[1 0 0]);
set(c(4:6),'Color',[0 0 0]);
```

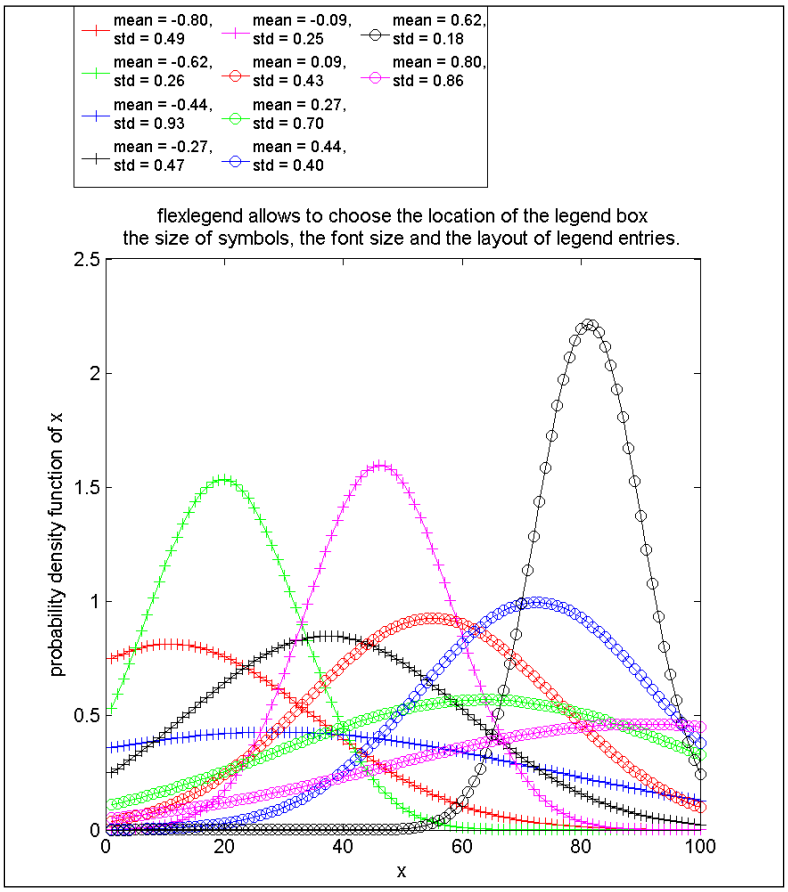
The output is as follows:



A MATLAB file exchange submission by Kelly Kearney extends the functionality of `legend` by making it possible to flexibly layout the components of the legend, instead of the serial columnar format. This code is included as part of the code repository with the book. Following is an example:

```
figure('units','normalized','position',...
  [ 0.4172    0.1769    0.2917    0.5861]);
hold on;
for i = 1:10
  h(i) = plot(dataVect(i,:), LineSpecs{i});
end
legendflex(h,...                %handle to plot lines
  legendMatrix,...              %corresponding legend entries
  'ref', gcf, ...               %which figure
  'anchor', {'nw','nw'}, ...    %location of legend box
  'buffer',[50 0], ...          %an offset wrt the location
  'nrow',4, ...                 %number of rows
  'fontsize',8,...              %font size
  'xscale',.5);                 %a scale factor for actual symbols
```

The output is as follows:



A few additional steps (included in source code lines 73 – 76) for resizing the figure and the axes as well as updating the title give you the preceding screenshot.

Takeaways from this recipe:

- Use legends that are carefully worded and judiciously placed, such that the data still has the maximum focus.
- Use line style and marker style over color to code information in legends.
- Do not use more than a handful of different colors in legends. (Color is best reserved for coding categorical variables. Choosing sequential colors may imply an order in the values. Use non-sequential colors when coding unordered categorical variables.)

See also

Look up **MATLAB help** on the `legend`, `grid`, and `flexlegend` commands.

Visualizing details with data transformations

The right transformation can reveal features of the data that are not observable in the original domain. In this recipe, you will see this principle at work.

Getting ready

In this recipe, you will use a data series with time variant rate of growth. It is common practice to use log transformations to effectively visualize the periods of growth in such cases. The original function generates a series of over 50 cycles following the given equation, where the growth efficiency E is a function of time (execute source code lines 11 – 15).

```
y1 = r * (1+E).^x;
```

How to do it...

Perform the following steps:

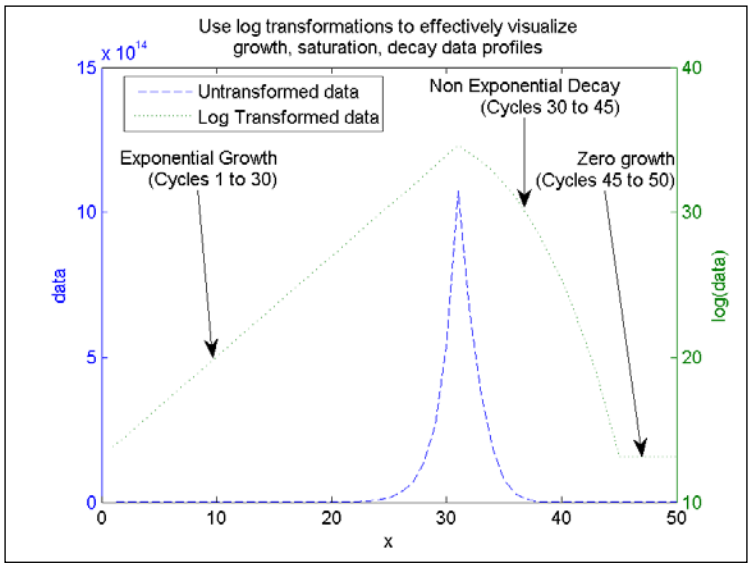
1. Plot the original and transformed data with the `plotyy` command:

```
y2 = log(y1);
axes('position',[0.1300    0.1100    0.7750    0.7805]);

[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
title({'Use log transformations to effectively ...
visualize','growth, saturation, decay data profiles'});
set(get(AX(1),'Ylabel'),'String','data');
set(get(AX(2),'Ylabel'),'String','log(data)');
xlabel('x'); set(H1,'LineStyle','--'); set(H2,'LineStyle',':');
```

2. Create annotations to reveal data characteristics:

```
annotation('textarrow',[.26 .28],[.67,.37],...
  'String',['Exponential Growth' char(10) ...
  '(Cycles 1 to 30)']);
annotation('textarrow',[.7 .7],[.8,.64],'String',...
  ['Non Exponential Decay' char(10) ...
  '(Cycles 30 to 45)']);
annotation('textarrow',[.809 .859],[.669,.192],...
  'String',...
  ['Zero growth' char(10) '(Cycles 45 to 50)']);
legend({'Untransformed data','Log Transformed data'},...
  'Location','Best');
```

The output should be as follows:



How it works...

The data in its original and transformed domains can both contain useful information. Transformations can reveal interesting characteristics about the data that is not apparent from the view of the original data.

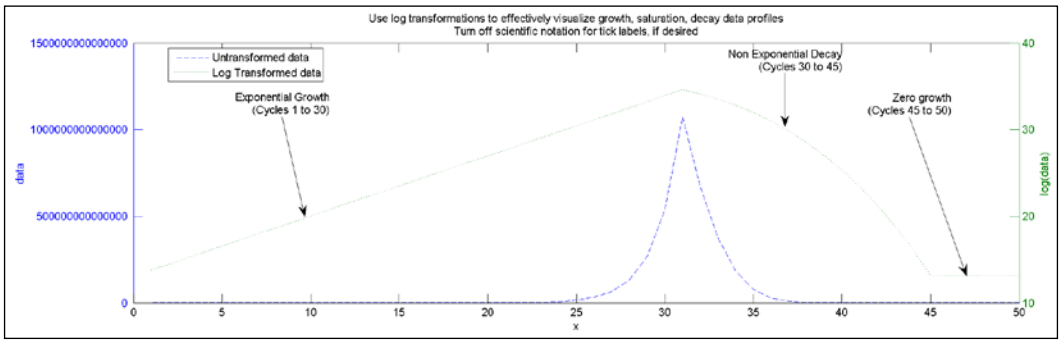
MATLAB offers the `plotyy` command to simultaneously use both, the left and the right `y` axis to plot two sets of data. This is especially useful when the independent variable `x` is the same (as in the present example).

The handles returned by `plotyy`, stored in the variable AX are used to put the desired labels (which in this case are the strings `data` and `log(data)` on the two sides of the `y` axis).

A side note: MATLAB uses scientific notation on the tick labels. Sometimes, this is an unintended effect. You can turn this off as follows:

```
% Resize the figure so you can see the huge numbers
set(gcf,'units','normalized','position',...
  [0.0411    0.5157    0.7510    0.3889]);

% AX(1) stores the handle to data on the untransformed axis
n=get(AX(1),'Ytick');
set(AX(1),'yticklabel',sprintf('%d |',n));
```

The output is as follows:

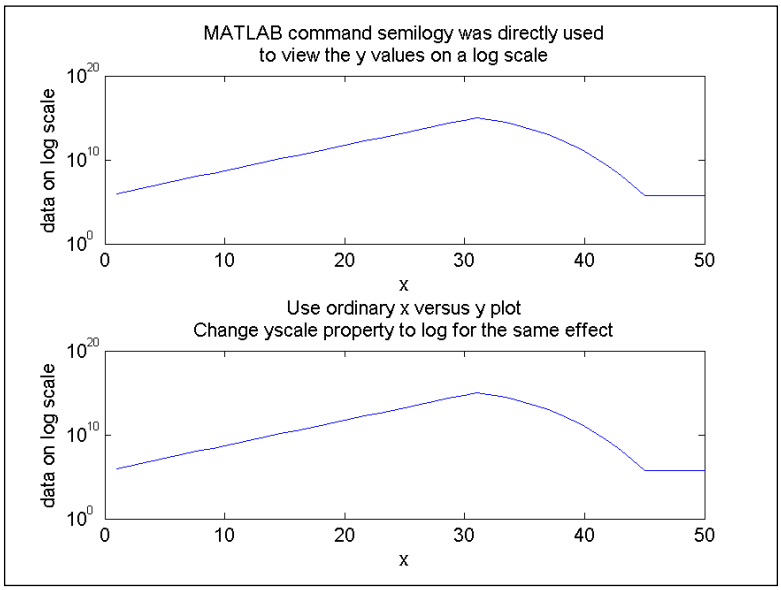


There's more...

Since logarithms are such a common transformation, MATLAB allows you to directly change the scale of `x` or `y` or both axis to a log scale by changing the `xscale` or `yscale` property. You can also use the `semilogx`, `semilogy`, and `loglog` plot types directly. For example:

```
subplot(2,1,1);
semilogy(x,y1);
xlabel('x');
ylabel('data on log scale');
title({'MATLAB command semilogy was directly used',...
  'to view the y values on a log scale'});
subplot(2,1,2);
plot(x,y1);
set(gca,'yscale','log');
xlabel('x');
ylabel('data on log scale');
title({'Use ordinary x versus y plot',...
  'Change yscale property to log for the same effect'});
```

The output is looks as follows:



Takeaways from this recipe:

- Use data transformations in your data exploration

See also...

Look up **MATLAB help** on the `plotyy`, `semilogx`, `semilogy`, and `loglog` commands that we encountered in this recipe.

Designing multigraph layouts

Related data is easier to interpret if they are placed in close proximity. MATLAB provides a default way to do this with the command `subplot`. Subplots are sufficient for creating graphs juxtaposed next to each other on a regular grid. Sometimes though, there is a need for an irregular grid, such as when there is one principal graph that needs more focus and hence more physical area dedicated to it. You can see detailed information on one graph, while another is a more abstracted view that provides context. This recipe concentrates on ways to lay out a set of graphics.

Getting ready

You will use stock price indices of the AAPL stock over year 2011:

```
% Load data and reverse the order to get earliest date first
[AAPL dateAAPL] = xlsread('AAPL_090784_012412.csv');
dateAAPL = datenum({dateAAPL{2:end,1}});
dateAAPL = dateAAPL(end:-1:1);
AAPL = AAPL(end:-1:1,:);

% Choose a time window for the main display
rangeMIN = datenum('1/1/2011');
rangeMAX = datenum('12/31/2011');
idx = find(dateAAPL >= rangeMIN & dateAAPL <= rangeMAX);
```

How to do it...

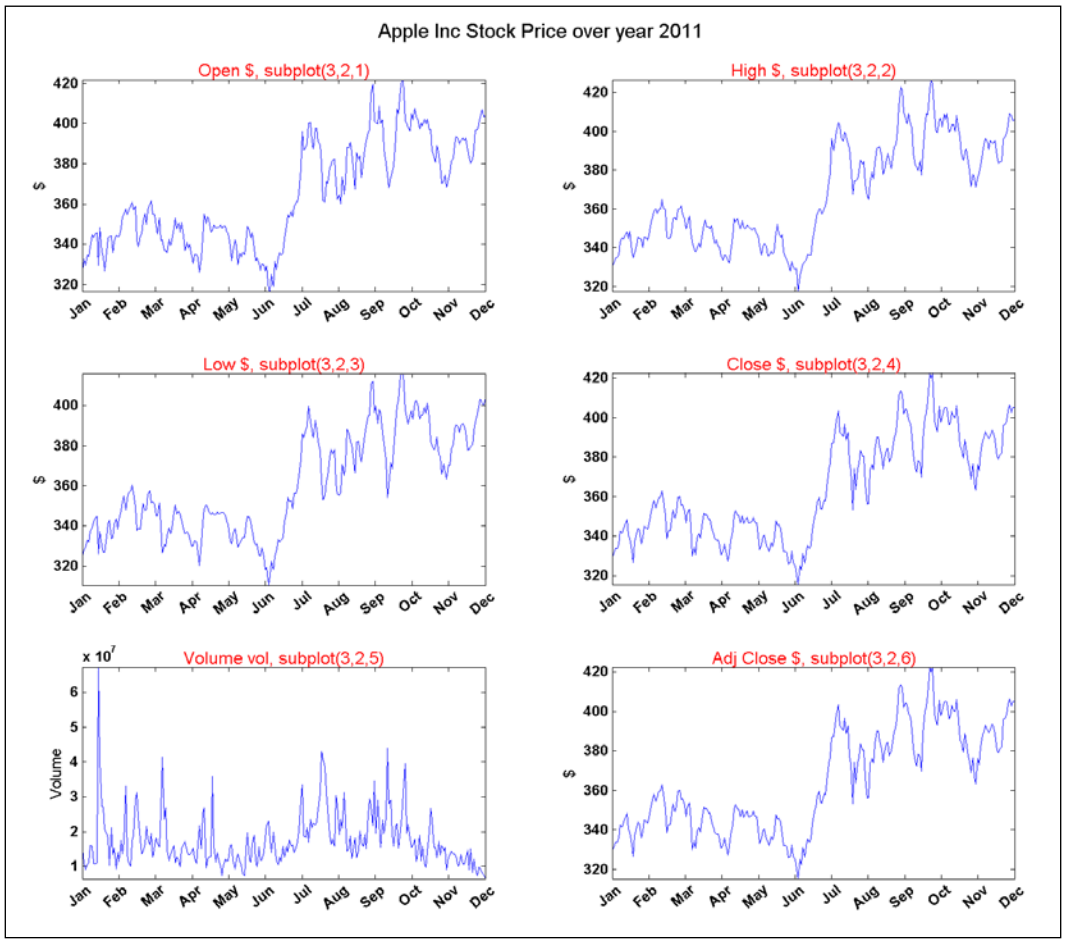
For the uniform grid layout, perform the following steps:

1. Use the `subplot` function for a regular grid layout. Notice the title on each subplot to understand how MATLAB accesses each consecutive position.

```
% Declare the figure
figure('units','normalized','position',...
  [ 0.0609    0.0593    0.5844    0.8463]);

% Declare the data labels
matNames = {'Open','High','Low','Close','Volume',...
  'Adj Close'};

% Use subplots to lay it out
for i = 1:6
  subplot(3,2,i);
  plot(idx,AAPL(idx,i));
  if i~=5
    title([matNames{i} ' $, '...
      'subplot(3,2,' num2str(i) ')'],...
      'Fontsize',12,'Color',[1 0 0 ]);
    ylabel('$');
  else
    title([matNames{i} ' vol, '...
      'subplot(3,2,' num2str(i) ')'],...
    'Fontsize',12,'Color',[1 0 0 ]);
    ylabel('Volume');
  end
```

```
  set(gca,'xtick',linspace(idx(1),idx(end),12),...
    'xticklabel',...
    datestr(linspace(dateAAPL(idx(1)),...
      dateAAPL(idx(end)),12),...
    'mmm'),'Fontsize',10,'fontweight','bold');
  rotateXLabels(gca,40);
  box on; axis tight
end

% Add a title to tie the set of plots together
annotation('textbox',[ 0.37   0.96   0.48   0.03],...
  'String','Apple Inc Stock Price over year 2011',...
  'Fontsize',14,'Linestyle','none');
```

Presenting related information on a multigraph layout on a uniform grid:



For the **customized multigraph layout**, you will create a commonly used set of plots for viewing stock prices over a certain time window in the context of a bigger period of price variations. The data will be plotted in three panels. The bottom most panel has the entire data. The part in blue on the bottom panel is the part of the series that is blown up and presented in the top panel. The volume data from that same time window as the first panel is displayed with a bar chart in the central panel. Perform the following steps:

1. Plot the data in panel 1:

```
% Figure dimensions
figure('units','normalized','Position',...
  [ 0.0427    0.2102    0.6026    0.6944]);

% Layout the axis
Panel1 = axes('Position',...
  [ 0.0570    0.5520    0.8850    0.3730]);hold;

% use area graphs to create the plot with a filled out
% area under the curve
area(AAPL(idx,4),'FaceColor',...
    [188 210 238]/255,'edgecolor',...
    [54 100 139 ]/255);

% set axis view parameters
xlim([1 length(idx)]);
yminv = min(AAPL(idx,4))-.5*range(AAPL(idx,4));
ymaxv = max(AAPL(idx,4))+.1*range(AAPL(idx,4));
ylim([yminv ymaxv]);
box on;

% set up the grid lines
set(gca,'Ticklength',[0 0],'YAxisLocation','right');
line([linspace(1,length(idx),15);
  linspace(1,length(idx),15)],...
  [yminv*ones(1,15); ymaxv*ones(1,15)],...
  'Color',[.7 .7 .7]);
line([ones(1,10); length(idx)*ones(1,10)],...
  [linspace(yminv, ymaxv,10); ...
  linspace(yminv, ymaxv,10);],'Color',[.9 .9 .9]);

% set up the annotations
set(gca,'xtick',linspace(1,length(idx),15),...
  'xticklabel',datestr(linspace(dateAAPL(idx(1)),...
  dateAAPL(idx(end)),15),'ddmmmyy'));
title({'Apple Inc Stock Price,',...
  '(detailed view from selected time window)'},...
    'Fontsize',12);
```

2. Plot the data in panel 2 (Specially note how the date tick labels are generated):

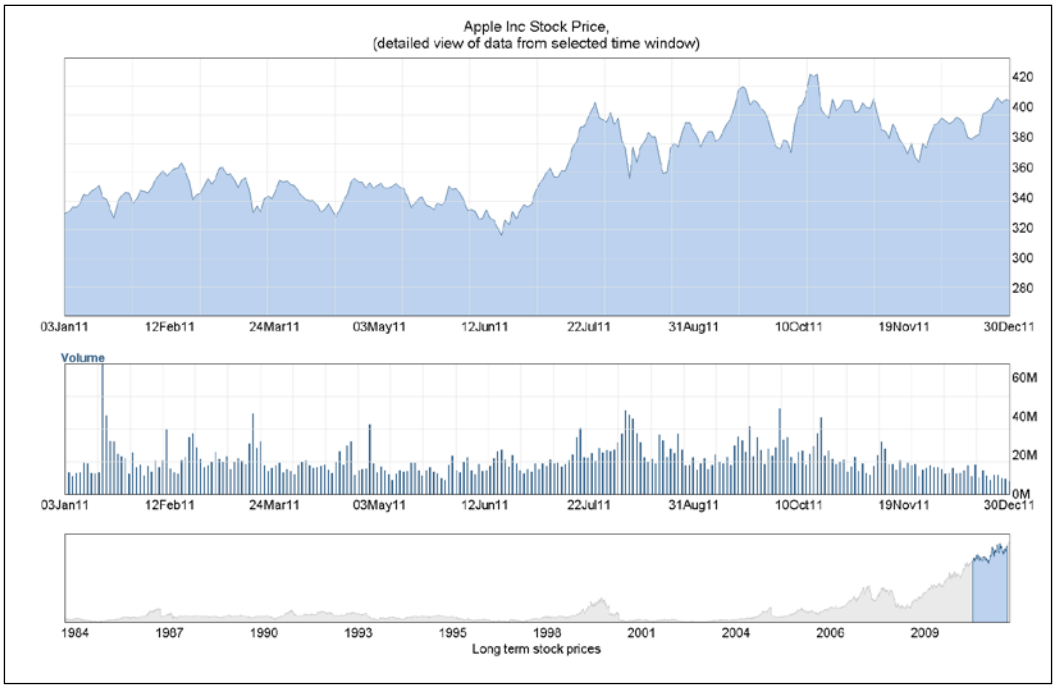
```
% Layout the axis
Panel2 = axes('Position',[.0570 .2947 .8850 .1880]);

% Plot the volume data with bar chart
bar(1:length(idx), AAPL(idx,5),.25,...
  'FaceColor',[54 100 139 ]/255);
hold; xlim([1 length(idx)]);hold on;

% Add grid lines
yminv = 0;
ymaxv = round(max(AAPL(idx,5)));
line([linspace(1,length(idx),30);...
  linspace(1,length(idx),30)],...
  [yminv*ones(1,30); ymaxv*ones(1,30)],...
  'Color',[.9 .9 .9]);
line([ones(1,5); length(idx)*ones(1,5)],...
  [linspace(yminv, ymaxv,5); ...
  linspace(yminv, ymaxv,5);],'Color',[.9 .9 .9]);
ylim([yminv ymaxv]);

% Set the special date tick labels
set(gca, 'Ticklength',[0 0],...
'xtick',linspace(1,length(idx),10),'xticklabel',...
  datestr(linspace(dateAAPL(idx(1)),...
  dateAAPL(idx(end)),10),'ddmmmyy'));
tickpos = get(Panel2,'ytick')/1000000;
for i = 1:numel(tickpos)
  C{i} = [num2str(tickpos(i)) 'M'];
end
set(Panel2,'yticklabel',C,'YAxisLocation','right');
text(0,1.1*ymaxv,'Volume','VerticalAlignment','top',...
  'Color',[54 100 139 ]/255,'Fontweight','bold');
```

3. Plot the data in panel 3 (Specially note that sub-selection is implied by plotting over the time window of interest, a segment with the same color as the detail view of panel 1, to establish the connection):

```
% Layout the axis
Panel3 = axes('Position',[.0570 .1100 .8850 .1273]);

% make the first plot muted underlying plot
area(dateAAPL, AAPL(:,4),'FaceColor',...
  [234 234 234 ]/255,'edgecolor',[.8 .8 .8]);
hold;
line([min(idx) min(idx)],get(gca,'ylim'),'Color','k');
line([max(idx) max(idx)],get(gca,'ylim'),'Color','k');
set(gca,'Ticklength',[0 0]);

% overplot for emphasis (use same color to establish
```

```
% connection)
area(dateAAPL(idx),AAPL(idx,4),'FaceColor',...
  [188 210 238]/255,'edgecolor',[54 100 139 ]/255);
ylim([min(AAPL(:,4)) 1.1*max(AAPL(:,4))]);
xlabel('Long term stock prices');

% Add additional grid lines
line([min(get(gca,'xlim')) min(get(gca,'xlim'))],...
  get(gca,'ylim'),'Color',[1 1 1]);
line([max(get(gca,'xlim')) max(get(gca,'xlim'))],...
  get(gca,'ylim'),'Color',[1 1 1]);
line(get(gca,'xlim'),[max(get(gca,'ylim')) ...
  max(get(gca,'ylim'))],'Color',[1 1 1]);
line(get(gca,'xlim'), [min(get(gca,'ylim')) ...
  min(get(gca,'ylim'))],'Color',[1 1 1]);
set(gca,'xticklabel',datestr(get(gca,'xtick'),...
  'yyyy'),'yticklabel',[]);
```

The resultant stock price charts with AAPL (Apple Incorporated Stock Price) is given in the following screenshot. The graphic illustrates how a combination of plots can be used to convey contextual information using the top to bottom drill down paradigm and how color can be used to associate different parts of the layout together.



How it works...

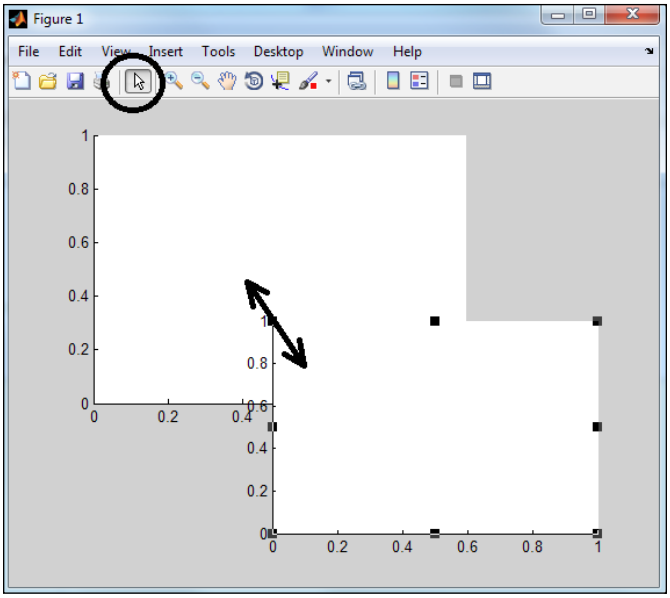
Designing displays that keep related information in close proximity helps the viewer integrate the different pieces of information.

For the uniform grid layout, you presented line charts of the six stock price indices for AAPL. The command `H = subplot(m,n,p)`, or `subplot(mnp)`, breaks the Figure window into an m-by-n matrix of small axes, selects the p-th axis for the current plot, and returns the axes handle. The axes are counted along the top row of the Figure window, then the second row. This is the most efficient way to make a set of plots on a regular grid in MATLAB.

For the **customized multigraph layout**, the way to generate the parameters for positioning the three different axes using the **normalized figure units** are given as follows:

1. Create a blank figure with the `figure` command and add axes to it (as many as you want) by using the `axes` command for each axis, without any parameters.
2. Enter the **Plot Edit** mode by selecting circled item of the toolbar.
3. Select each axis and drag and resize to position and size as desired.
4. Select each axis and execute `get(gca,'position')` at the command line to generate the parameters for each axis.
5. Add the axes command with these parameters to your code to generate the axes at the same position, programmatically, every time.

A set of steps for creating a multi plot graphic with a flexible layout is shown in the following screenshot:



Takeaways from this recipe:

- Place related graphics in close proximity when possible
- Use few and light colored grid lines
- Use color to create associations

See also

Look up **MATLAB help** on the `datestr`, `subplot`, and `axis` commands that we encountered in this recipe.

A visualization to compare algorithm test results

Data analysts often need to compare several methods for solving a problem. Input samples can usually be classed into several categories. The challenge is to choose the method that handles all the categories in the best way. A visual way to quickly compare the test results is to use a set of bar charts in a tabular format as shown in this recipe.

Getting ready

In the previous examples, you used predefined color schemes from MATLAB. In this recipe, the color palette was chosen by Colorbrewer, an online tool for color selection for maps and other graphics. Define a color matrix to correspond with the five different sample categories under comparison, using RGB values, as follows:

```
Colors = [  141 211 199;255 255 179;190 186 218;...
  251 128 114;128 177 211; ]/255;
```

The main data is contained in the matrix `MethodPerformanceNumbers` where each column represents one of the five algorithms under comparison and each row represents a different category of source samples. The matrix `CategoryTotals` is the total number of samples tested by the five methods in each category. Load the data into your workspace from the code repository for this book:

```
load algoResultsData
```

How to do it...

Perform the following steps:

1. Define the axes schematic: Parameters for axes positioning were chosen using methods described in the *Design multigraph layouts* recipe in this chapter. Use the `line` command to create bars representing the number of successes in each category using color scheme defined at the beginning of this recipe:

```
% Define the figure
figure('units','normalized',...
  'Position',[ 0.0880    0.1028    0.6000    0.6352]);

% X Tick labeling for the names of Algorithms under comparison
% Create an invisible axis; place the tick labels at an angle
hh = axes('Position',[.1,.135,.8,.1]);
set(gca,'Visible','Off',...
  'TickLength',[0.0 0.0],...
  'TickDir','out',...
  'YTickLabel','',...
  'xlim',[0 nosOfMethods],...
  'FontSize',11,...
  'FontWeight','bold');
set(gca,'XTick',.5:nosOfMethods-.5,...
  'XTickLabel',{'K Means','Fuzzy C Means',...
  'Hierarchical','Maximize Expectation','Dendogram'});
catgeoryLabels = {'Fresh Tissue','FFPE',...
  'Blood','DNA','Simulated'};
rotateXLabels(gca,20);
```

2. Place five different axes, one for each row corresponding to the five sample types. (Continue step 2 and 3 together as they are part of the same `for` loop.)

```
% Split the available vertical space into five
y = linspace(.142,.8,nosOfCategories);

% Place each of the axes: The height of the axes
% corresponds to the total number of samples in that
% category.

for i = 1 :nosOfCategories
  if CategoryTotals(i);
    ylimup = CategoryTotals(i);
  else
    ylimup = 1;
  end
```

```
  dat = [MethodPerformanceNumbers(i,:)];
  h(i) = axes('Position',[.1,y(i),.8,y(2)-y(1)]);
  set(gca,'XTickLabel','',...
    'TickLength',[0.0 0.0],...
    'TickDir','out',...
    'YTickLabel','',...
    'xlim',[.5 nosOfMethods+.5],...
    'ylim',[0 ylimup]);
```

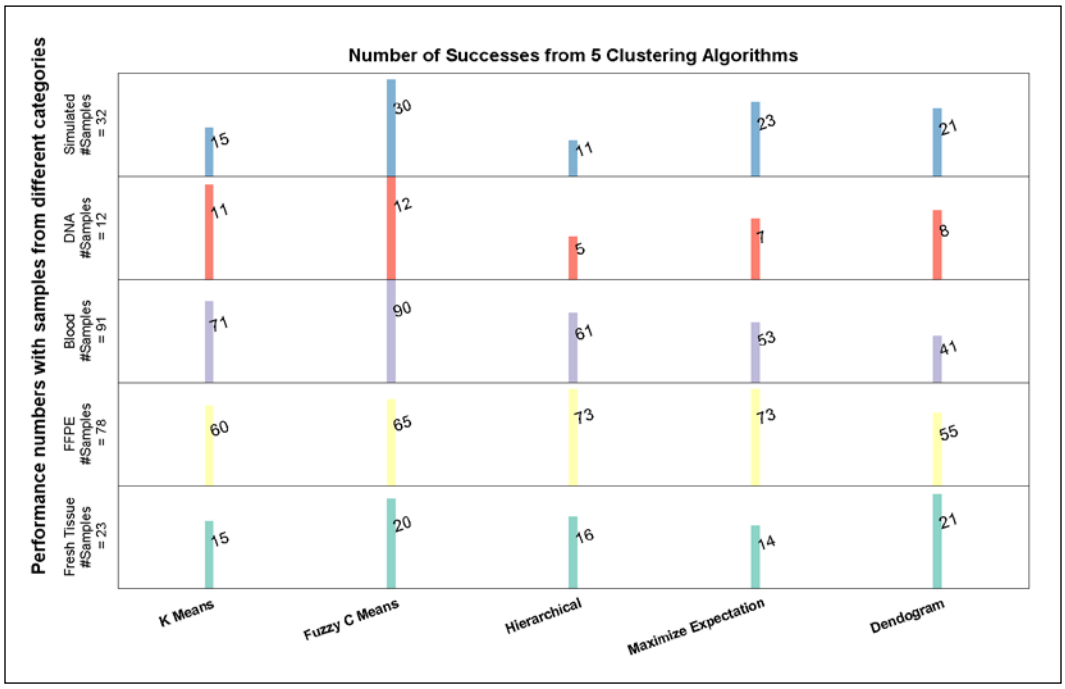
3. Plot the bars representing the number of successes and add labels by their side (this needs to be done for each category and hence is inside the `for` loop started in step 2):

```
% Use the line command to create bars representing the
% number of successes in each category using color
% defined at the beginning of this recipe.
line([1:nosOfMethods; 1:nosOfMethods],...
  [zeros(1,nosOfMethods); dat],...
  'Color',Colors(i,:),...
  'Linewidth',7);
  box on;

% Place the actual number as a text next to the bar
for j= 1:nosOfMethods
  if dat(j);
    text(j+.01,dat(j)-.3*dat(j),...
      num2str(dat(j)),'Rotation',20,'FontSize',13);
  end
end

% Add the category label
ylabel([catgeoryLabels{i} char(10) ...
  '#Samples' char(10) ' = ' num2str(ylimup)],...
  'Fontsize',11);
end
```

4. Add annotations:

```
title(['Number of Successes from 5 Clustering'...
  'Algorithms'],'Fontsize',14,'Fontweight','bold');
axes(h(3));
text(0.06,-170,...
  ['Performance #s with samples from different' ...
  'categories'],'Fontsize',14,'rotation',90,...
  'Fontweight','bold');
set(gcf,'Color',[1 1 1],'paperpositionmode','auto');
```

The output is as follows:



How it works...

The figure shows the number of successes obtained from testing five clustering algorithms. Five different types of input samples are processed with these algorithms. The results are presented with a bar chart in a tabular format. The **Fuzzy C Means** clearly outperforms the other options as it has the highest percent success across all sample categories.

This recipe brings together visualization techniques covered in previous recipes of Chapter 1.

Note that the length of the bar of the same color should be compared to each other. They represent the number of successes out of the total number of samples denoted on the left, in a given sample category. The length of the bars with different colors should not be compared to each other as the total number of samples is not the same in each input sample category. Note that here you used a simple color matrix which is a set of colors defined by their RGB values. You did not use the concept of a color map described in the *Tufte style gridding for readability* recipe.

Also note that for a small set of numbers such as this, a graphics design is usually unnecessary. A table would be sufficient to convey this information.

Takeaways from this recipe:

- Use the same color to create associations
- Use discontinuous colors to differentiate between categories

2

Diving into One-dimensional Data Displays

In this chapter, we will cover:

- Pie charts, stem plots, and stairs plots
- Box plots
- Sparklines
- Stacked line graphs
- Node link plots
- Calendar heat map
- Distributional data analysis
- Time series analysis

Introduction

This chapter focuses on one-dimensional data displays. The most common chart types used for this purpose are line charts, bar charts, and scatter plots. They use **positional coordinates** to represent numerical information. Positional coordinates are one of the best ways to represent numerical information according to visualization guru, Edward Tufte. It is the preferred choice over other dimensions such as angle, length, area, volume, or color for presenting numerical data. You used:

- One-dimensional **scatter plots** in *Chapter 1*, *Making your first MATLAB plot*
- **Line plots** with error bars in *Chapter 1*, *Laying out long tick labels without overwriting*
- **Bar plots** in *Chapter 1*, *Gridding for readability*

There are other plot types that could be appropriate too, depending on the nature of the data and the purpose of your graphic. The data analysis approach could also drive the visualization. This chapter brings together a set of recipes that cover plot types and data analysis approaches for one-dimensional data, using examples from different application domains.

Pie charts, stem plots, and stairs plots

In this recipe, you will see the popular chart types: pie charts, stem plots, and stairs plots.

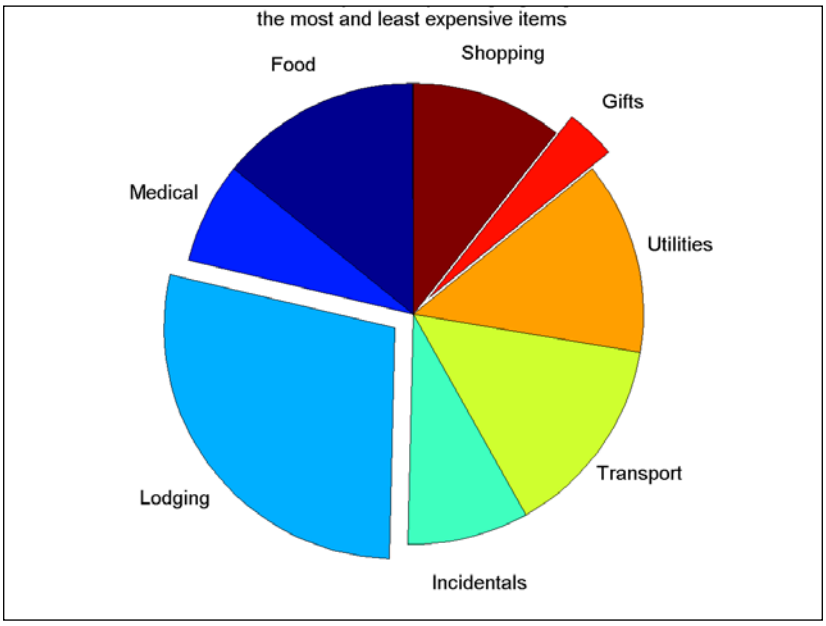
How to do it...

Perform the following steps:

1. **Pie charts** are created with the MATLAB command `pie`. Plot a pie chart and explode a slice out of the pie, for added emphasis:

```
Expenses = [20 10 40 12 20 19 5 15];
ExpenseCategories = {'Food','Medical','Lodging','Incidentals','Tra
nsport',...
  'Utilities','Gifts','Shopping'};
MostLeastExpensive = ...
  (Expenses==max(Expenses)|Expenses==min(Expenses));
h=pie(gca,Expenses,MostLeastExpensive,ExpenseCategories);

% Every alternate handle returned by pie is a text object. Use
% that to increment the font size of the labels
for i =2:2:16;set(h(i),'fontsize',14);end

% Add annotation
title('Annual Expense Report','fontsize',14);
```

The output is as follows:

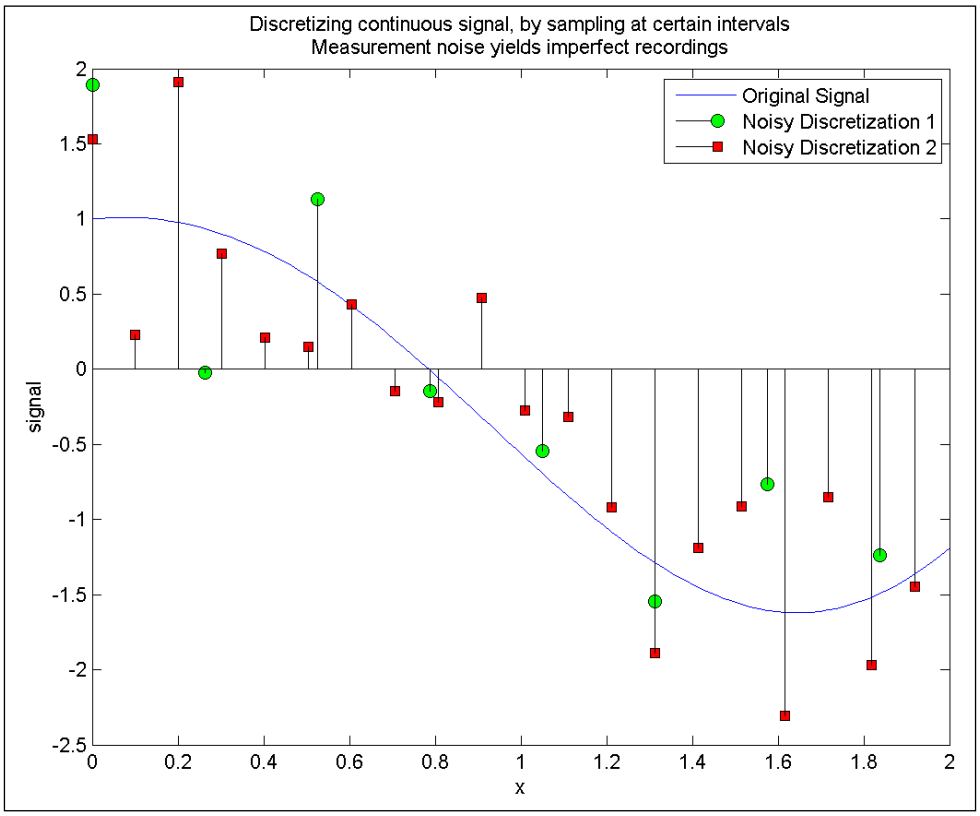


2. **Stem plots** are created with the MATLAB command `stem`. Use a stem plot to view the process of discretizing a continuous signal.

   Open the source code for this recipe and execute lines 25 – 32 to create the variables `x`, `y`, `x1`, `x2`, `y1`, `y2` (omitted here for the sake of brevity). Then, execute the following steps:

```
% x and y are the original signals, x1,y1 is obtained from a
% sampling of the original signal and x2,y2 is obtained from a
% different sampling of that original signal
plot(x,y); hold on;
h1 = stem(x1,y1);
h2 = stem(x2,y2);

% Choose marker size and style of your choice
set(h1,'MarkerFaceColor','green','Marker','o',...
  'Markersize',7,'Color',[0 0 0]);
set(h2,'MarkerFaceColor','red','Marker','square',...
  'Color',[0 0 0]);
xlabel('x');ylabel('signal');
legend({'Original Signal','Noisy Discretization 1',...
  'Noisy Discretization 2'});
```

The output is as follows:

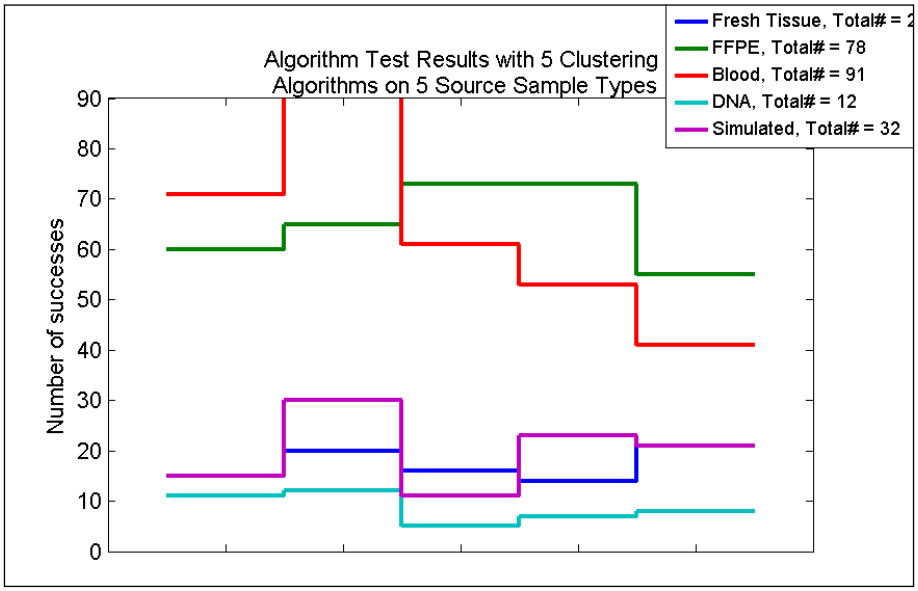


3. **Stairs plots** are created with the MATLAB command `stairs`. Use a stairs plot to plot the test results from five different algorithms on five different sample categories as follows:

```
% Load the data
load algoResultsData.mat

% adding a row of NaNs so that the last row gets represented
% as a horizontal line (rather than a rise)
h=stairs([MethodPerformanceNumbers nan(5,1)]');

legendMatrix = {'Fresh Tissue','FFPE',...
  'Blood','DNA','Simulated'};
for i = 1:5;
  set(h(i),'linewidth',2); % thicken the lines
  % add total # of samples to legend entry for this category
  legendMatrix{i} = [legendMatrix{i} ...
```

```
    ', Total# = ' num2str(CategoryTotals(i))];
end
set(gca,'xlim',[0.5 6.5],...
  'XTick',1.5:nosOfMethods+1,...
  'XTickLabel',{'K Means','Fuzzy C Means',...
  'Hierarchical','Maximize Expectation','Dendogram'});

%add annotations
title({'Algorithm Test Results with 5 Clustering',...
  ' Algorithms on 5 Source Sample Types'});
legendflex(h,...      %handle to plot lines
legendMatrix,...      %corresponding legend entries
'ref', gcf, ...       %which figure
'anchor', {'ne','ne'}, ...%location of legend box
'buffer',[0 0], ...       % an offset wrt the location
'fontsize',8,...          %font size
'xscale',.5);             %a scale factor for symbols
rotateXLabels(gca,20);
set(gca,'position',[0.1139    0.1989    0.7750    0.6638]);
```

The output is as follows:



How it works...

All three graphs demonstrated here were generated by single commands in MATLAB.

The pie chart showed the breakup of expenses in various categories. Pie charts code information in the area of its slices. Area is a hard dimension for humans to process. Pie charts should be used sparingly.

Next, you used stem plots to show the discretization of a continuous signal. The figure showed how measurement noise adds distortion in the process of sampling. Stem plots are useful for displaying discrete data points where joining the series may not make physical sense.

Next, you used stairs plots to compare the performance of five different clustering algorithms across five types of samples. Stairs plot add a twist to the traditional bar plots. A very dense series of bars could be replaced with a stairs plots design that boasts less visual chatter.

Other than the new plot types, this recipe reuses the functionalities and practices introduced in *Chapter 1*, *Customizing Elements of MATLAB Graphics—the Basics*, such as rotation of tick labels (*Laying out long tick labels without overwriting*) and flexible placement of legends (*Bringing order to chaos with legends*).

Takeaways from this recipe:

- Choose plot types that use positional coordinates to present numerical data, such as line charts, bar charts, scatter plots, stem plots, and stairs plots

See also

Look up **MATLAB help** on the `plot`, `bar`, `line`, `scatter`, `pie`, `stems`, and `stairs` commands.

Box plots

A **box plot** (also known as a **box-and-whisker** diagram) is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observation (sample minimum), lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation (sample maximum). The box is constructed between Q1 and Q3 and a special symbol is used to denote the median. The whiskers extend up to the smallest and largest observation on both sides of the distribution.

A box plot may also indicate which observations, if any, might be considered outliers (typically those outside 1.5 times the inter-quartile range) on both extremes of the distribution. If outliers are shown, then the whiskers extend only up to the smallest and largest observations inside 1.5 times the inter-quartile range on both extremes of the distribution. Outlier observations are marked with special symbols.

You will use the function `boxplotV`, which is part of the code repository with this book, to generate the parameters for drawing a box plot, and then use a series of `line` commands to construct the box plot, as shown in this recipe.

Getting ready

You will use the dataset with the set of gene expression levels for various cancer types, first introduced in *Chapter 1*, *Laying out long tick labels without overwriting*. Load the data:

```
load 14cancer.mat
```

How to do it...

There are two tasks demonstrated in this recipe. The first is how to build box plots. The second is how to work with multi-tiered tick labels.

Perform the following steps:

1. Generate the box plot parameters, one for each group of data. The grouping label for each data point is provided in the vector, which is the 2nd argument to the `boxplotV` function:

```
data = [Xtrain(:,2798); Xtest(:,2798)];
[lowerQuartile medianv upperQuartile ...
  upperOuter lowerInner outliers] = ...
  boxplotV(data, [ytrainLabels ytestLabels]');
```

2. The figure will have two axes. The first will hold the data and the second, the multi-tiered tick labels.

```
figure('units','normalized',...
  'Position',[0.3563    0.3019    0.6328    0.6028]);
hData = axes('position',[0.0831 0.2000 0.8930 0.7200]);box on;
```

3. Define the three tiers of tick labels in suitable order:

```
tier1 = classLabels;
tier2 = {'Upper body','Lower body','Upper body',...
'Lower body','Distributed','Lower body','Distributed',...
'Lower body','Distributed','Lower body','Lower body',...
'Lower body','Distributed','Upper body'};
tier3 = {'low','low','medium','medium','low',...
```

```
'low','low','low','medium','low','high','low',...
  'medium','medium'};
[tier1, tier2 tier3, sep2, sep3] = ...
  multiTierLabel(tier1, tier2, tier3);
```

4. Plot each of the `i` box plots, where `i` goes from 1 to the total number of groups you have. In this recipe, you have 14 cancer types.

```
% Define the boxwidth
boxwidth = .5/2;

axes(hData);hold on;
for i = 1:length(tier1)

  % draw the inter quartile box in blue and mark the
  % location of the median in red
  line([i-boxwidth i-boxwidth i-boxwidth i+boxwidth; ...
    i+boxwidth i+boxwidth i-boxwidth i+boxwidth;],...
    [lowerQuartile(i) upperQuartile(i) ...
    lowerQuartile(i) upperQuartile(i); ...
    lowerQuartile(i) upperQuartile(i) ...
    upperQuartile(i) lowerQuartile(i);],...
    'Color',[0 0 1]);
  line([i-boxwidth; i+boxwidth;],...
    [medianv(i); medianv(i);],'Color',[1 0 0]);

  % draw the whiskers in black
    line([i i i-.5*boxwidth i-.5*boxwidth; ...
    i i i+.5*boxwidth i+.5*boxwidth;],...
    [upperQuartile(i) lowerQuartile(i) ...
    lowerInner(i) upperOuter(i); ...
    upperOuter(i) lowerInner(i) ...
    lowerInner(i)  upperOuter(i);],...
    'Color',[0 0 0]);

  % draw the outliers with the red + marker
  plot(repmat(i,size(outliers(i).mat)),outliers(i).mat,
    'r+');
end

% Add annotations
set(gca,'xticklabel',[],'ticklength',[0 0],...
  'ylim',[1.1*min(data) 1.1*max(data)]);
ylabel('Gene Expression levels', 'fontsize',10);
title({'Gene expression boxplot for 198 samples',...
  'for a specific gene across 14 cancers'},...
  'fontsize',10);
```

5. Estimate the location of grid lines and add the primary grid lines:

```
% Estimate location of grid lines
sepLine = unique([sep2 sep3]) +.5;
if sepLine(end)==length(tier1)
  sepLine = sepLine(1:end-1);
end

% add primary grid lines
line([sepLine; sepLine], ...
  [min(get(gca,'ylim'))*ones(size(sepLine));
  max(get(gca,'ylim'))*ones(size(sepLine))],...
  'Color',[.8 .8 .8]);
```

6. Declare and prepare the second axis reserved for the labels and add the meta labels:

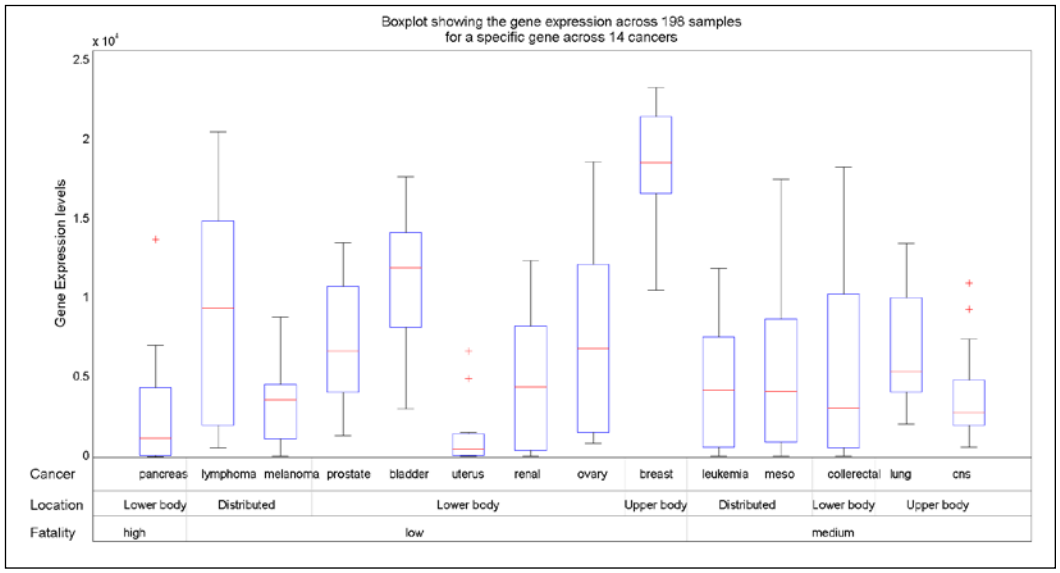
```
hLabels = axes('position',[0.0831,.05,0.8930,.15]);box on;
set(hLabels,'ticklength',[0 0],'xticklabel',[],...
  'yticklabel',[],'xlim',get(hData,'xlim'),'ylim',[0 1]);
line(get(gca,'xlim'),[1 1],'Color',[0 0 0]);
line(get(gca,'xlim'),[0 0],'Color',[0 0 0]);

% add meta labels
text(-1,.1,'Fatality','fontsize',12);
text(-1,.43,'Location','fontsize',12);
text(-1,.8,'Cancer','fontsize',12);
```

7. For each tier of tick labels, place the tick labels, place the vertical and horizontal divider lines:

```
% add tier 1 labels
text([1:14]-boxwidth,.8*ones(size(tier1)),...
  strtrim(tier1),'Fontsize',10);
% add the grouping grid lines
line([sepLine; sepLine], [.5*ones(size(sepLine)); ...
max(get(gca,'ylim'))*ones(size(sepLine))],
  'Color',[.8 .8 .8]);
% add separator line
line(get(gca,'xlim'),[.6 .6],'Color',[0 0 0]);

% add tier 2 labels
x=[0 sep2 length(tier1)];
x = (x(1:end-1)+ x(2:end))/2;
text(x,.43*ones(length(sep2)+1,1),...
  tier2([sep2 length(tier1)]),'Fontsize',10);
% add the grouping grid lines
line([sepLine; sepLine], [.3*ones(size(sepLine)); ...
```

```
max(get(gca,'ylim'))*ones(size(sepLine))],
  'Color',[.8 .8 .8]);
% add separator line
line(get(gca,'xlim'),[.3 .3],'Color',[0 0 0]);

% add tier 3 labels
x=[0 sep3 length(tier1)];
x = (x(1:end-1)+ x(2:end))/2;
text(x,.1*ones(length(sep3)+1,1),...
  tier3([sep3 length(tier1)]),'Fontsize',10);
% add the grouping grid lines
line([sep3; sep3]+.5, ...
[min(get(gca,'ylim'))*ones(size(sep3));
  3*ones(size(sep3))],...
  'Color',[.8 .8 .8]);
```

The output should be:



How it works...

This recipe uses box plots to visualize the expression levels of a certain gene across 14 different cancers. Leukemia clearly has a higher expression level, but since the whiskers from other types of cancer overlap with the data from leukemia, this expression level alone cannot be used as a diagnostic for leukemia.

In step 2, the `boxplotV` function calculated the parameters for plotting a box plot, that is the upper and lower quartiles, the median, the upper and lower outer values, and outliers as per the definition given in the introduction to this recipe, for each group of data. In step 4, you used the `line` command as shown, to actually make the plots. These were directed into the data axis you declared in step 3.

For making the **multi-tiered tick label**, you called the `multiTierLabel` function, which essentially sorts the ordering of the tick labels as per what is provided as the outermost set of labels, and then within each group, sorts the ordering of the tick labels as per what is provided as the 2nd set of labels; and then further sorts the ordering as per what is provided as the 1st set of labels.

In step 7, you used the three sets of tick labels and put them on three different rows. This was directed into the label axis you declared in step 3.

The `multiTierLabel` function will order any number of triplets of tick label entries. The `boxplotV` function will generate box plot parameters for any amount of data and any number of groups. You will have to adapt the actual plotting and tick label placements as shown in this recipe to customize to your own dataset.

Takeaways from this recipe:

- Use box plots to present distributional summaries for your datasets
- Use tiered labeling to convey additional information without using long tick labels

See also

Look up **MATLAB help** on the `boxplot` command that comes as part of the MATLAB Statistics Toolbox™.

Sparklines

A **sparkline** is a type of information graphic characterized by its small size and high data density. They are used to present trends and variations in the data in a simple and condensed way. The term **sparkline** was proposed by Edward Tufte for "small, high resolution graphics embedded in a context of words, numbers, and images". While the typical chart is designed to show as much data as possible, and is set off from the flow of text, Sparklines are intended to be succinct, memorable, and located where they are discussed.

Getting ready

The book data repository comes with several comma separated value (`.csv`) files with the daily stock price histories over a year. In this recipe, this stock index data is used to construct **Sparklines** highlighting the maximum swing over a year. You already learned how to import data from Excel in *Chapter 1*, *Making your first MATLAB plot*. Refer to that or the source code to this recipe (lines 12 – 21) to import the data into the variable `dt`, a cell array of vector data, with one set of stock price indices per vector.

Next, preprocess the data by normalizing between 0 and 1:

```
for i = 1:length(dt)
  % convert date to a numeric format
  dateD{i} = datenum({dateD{i}{2:end,1}});

  % find dates in range
  idx = find(dateD{i} >= rangeMIN & dateD{i} <= rangeMAX);
  dt{i} = dt{i}(idx);

  % extract data in range
  dateD{i} = dateD{i}(idx);

  % normalize
  dtn{i} = dt{i}./max(dt{i});
  clear idx
  labels2{i} = num2str(dt{i}(end));
end
```

You use a new function `datenum` to convert date notations to absolute numbers that then become amenable to numerical operations such as choosing between our desired time windows.

How to do it...

Perform the following steps:

1. Call the function `sparkline`:

   ```
   sparkline(dateD,dtn,stocks,labels2);
   ```

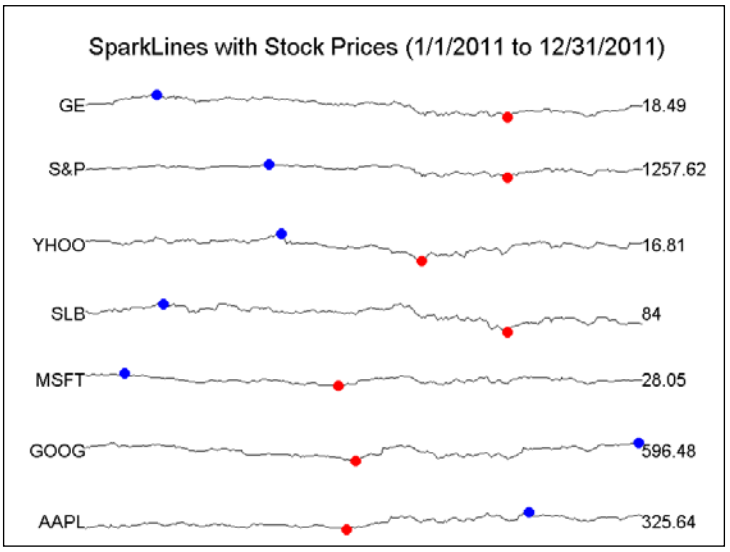
2. Following are the steps you execute inside the function `sparkline`:

   ```
   % Each sparkline is stacked up next to one another, separated
   % by an arbitrary unit of separation. Here unitOfSep=1;
   unitOfSep=1;
   ```

```
figure;

% No borders necessary - span the axes out to total available %
space
axes('position',[0 0 1 .9]);hold on;
endPt = -1; startPt = 1e100;

for i = 1:length(xdata)

  % Plot the SparkLines
  plot(xdata{i}, ydata{i}+ (i-1)*+unitOfSep,'k');

  %Locate the minimum and maximum points and mark with red
  %and blue
  maxp{i} = find(ydata{i}==max(ydata{i}));
  minp{i} = find(ydata{i}==min(ydata{i}));
  plot(xdata{i}(maxp{i}),...
    ydata{i}(maxp{i})+ (i-1)*+unitOfSep,...
    'bo','MarkerFaceColor','b');
    plot(xdata{i}(minp{i}),...
    ydata{i}(minp{i})+ (i-1)*+unitOfSep, ...
    'ro','MarkerFaceColor','r');

  %Place the two labels at start and end of the sparkline
  text(xdata{i}(end), mean(ydata{i})+
    (i-1)*+unitOfSep,...
    labels1{i},'HorizontalAlignment','right');
    text(xdata{i}(1), mean(ydata{i})+
    (i-1)*+unitOfSep,...
    labels2{i},'HorizontalAlignment','left');

    % Keep track of the start and end in order to correctly
    % set the x limits later
  endPt = max([xdata{i}(1) endPt]);
  startPt= min([xdata{i}(end) startPt]);
end

% Set the title
text(startPt+2, i*unitOfSep+.7,...
'SparkLines with Stock Prices (1/1/2011 to 12/31/2011)',...
  'fontsize',14);
set(gca,'visible','off',...
  'ylim',[0+unitOfSep/2 i*unitOfSep+unitOfSep/2],...
  'yticklabel',[],...
```

```
    'xlim',...
      [startPt-.15*(endPt-startPt) endPt+.15*(
        endPt-startPt)],...
      'xticklabel',[],...
      'TickLength',[0 0]);
set(gcf,'Color',[1 1 1],'Paperpositionmode','auto');
```

The output is as follows:



How it works...

The preceding screenshot shows the daily price fluctuations in seven different stocks over the year 2011. Red and blue dots mark the occurrence of the highest and lowest values in the series. The numeric label at the end of the series gives an approximate idea of the range of values represented. **MSFT**, **GOOG**, and **AAPL** shows a slight upswing past the middle of the year.

Sparklines are intended to be data summaries. The normalization makes the data comparable despite the lack of any real axes. Also, the one data point showing the actual value provides the context to interpret the data in a relative sense. Sparklines allow a quick assessment of the trend across a large number of time series.

The function `sparkline` is called with two numerical cell arrays, containing vectors of `x` and corresponding `y` values. Also, it takes two cell arrays of strings called `LABELS1`, which have labels for each spark line to be located at the start of the line and `LABELS2`, which have the labels for each spark line to be located at the end of the line.

Takeaways from this recipe:

- Use spark lines for a quick trends assessment of a large number of time series data.

See also

Look up **MATLAB help** on the `datenum` command.

Stacked line graphs

This recipe illustrates how to make **stacked line graphs** using the MATLAB command `area`. The idea of this graphic is inspired from the website `namevoyager.com`, which tracks the popularity of thousands of baby names. The graphic shows the popularity of 15 baby names over several decades. The names are visible on the right. In any given year, the widest blue line is the most popular male baby name and the widest pink line is the most popular female baby name.

Getting ready

Load the data:

```
[ranksoverdecades names] = ...
  xlsread('MockDataNameVoyager.xlsx');
sex = names(2:end,2);
names = names(2:end,1);
years = ranksoverdecades(1,:);
ranksoverdecades = ranksoverdecades(2:end,:)';
```

How to do it...

Perform the following steps:

1. Split data by male and female names:

```
% split into male and female names
males = find(strcmp(sex,'M'));
fmales = find(strcmp(sex,'F'));
ymax=max(max(cumsum(ranksoverdecades,2)));
```

2. Generate a location for placing of name labels:

```
%generate the y coordinates where the name will be placed
nameLoc = cumsum(ranksoverdecades(end,:));
nameLoc = [0 nameLoc];
nameLoc = (nameLoc(1:end-1) + nameLoc(2:end))/2;
```

3. Set up the two `y` axis (one for the data and one for placing the labels):

```
% set up the figure
figure('units','normalized','Position',...
  [ 0.3432    0.1472    0.6542    0.7574]);

%create main axes for the data
axes('position',[.05,.1,.87,.85],'ylim',[0 ymax],...
  'xlim',[min(years) max(years)],...
  'YAxisLocation','right', 'ytick',nameLoc,...
  'yticklabel',names, 'ticklength',[0.01 0.05],...
  'tickdir','out','fontsize',14);

% create a secondary axes to place the name labels
axes('Position',get(gca,'Position'));
```

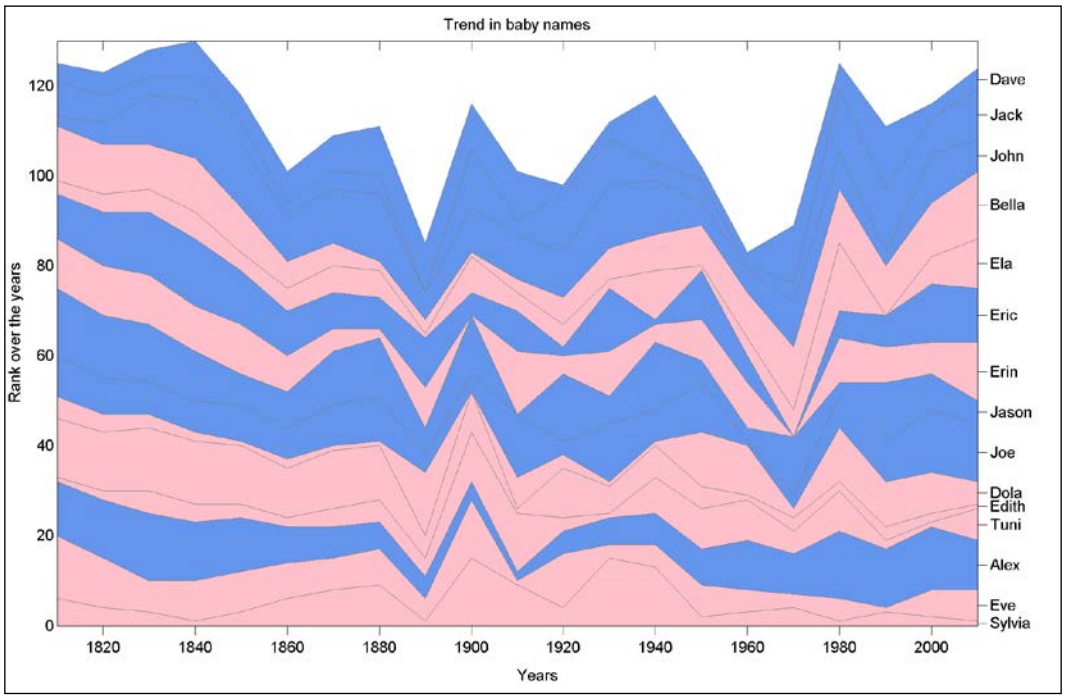
4. Draw the stacked line graphs; color line graphs by sex of the baby:

```
% draw the stacked line graphs
h = area(years,ranksoverdecades);

% set the area graph colors per sex of the baby name
set(h(males),'FaceColor',[100   149 237]/255)
set(h(fmales),'FaceColor',[255  192 203]/255);
```

5. Define the edge color, the `x` and `y` limits, and annotations:

```
% fix edgecolor and x and y limits
set(h,'edgecolor',[.5 .5 .5])
set(gca,'ylim',[0 ymax],'xlim',[min(years) max(years)],...
  'xticklabel',[] ,'fontsize',14);
box on;

% annotate the graph
title('Trend in baby names','Fontsize',14);
ylabel('Rank over the years','Fontsize',14);
text(mean(get(gca,'xlim')),-11,'Years','Fontsize',14);
```

The output is as follows:



How it works...

The figure uses stacked line graphs/**area graphs** to show the popularity of 15 baby names over several decades. You can see that names like Bella and Alex were popular in the 1800's and have become popular again recently.

Essentially, MATLAB supports a single command `area` to create these types of graphs. Other customization such as the double y axis, color, and addition of tick labels follow the standard rules you have seen in previous recipes.

Area graphs allow to present large number of graphs without overplotting.

Takeaways from this recipe:

- Use area graphs to present large number of line graphs without over plotting

See also

Look up **MATLAB help** on the `cumsum` and `area` commands.

Node link plots

This recipe talks about graphics that could be used for showing a relationship between pairs of things aka **node link plots**. There are a couple of variations available to represent this type of data. This recipe shows two alternatives.

Getting ready

In this dataset, you have intercity distances between 128 US cities and the coordinate locations for those cities. The data was obtained from a website maintained at the Department of Scientific Computing at the Florida State University. Load the data:

```
[XYCoord] = xlsread('inter_city_distances.xlsx','Sheet3');
[intercitydist citynames] =  ...
  xlsread('inter_city_distances.xlsx','Distances');

% A city should not be detected as the nearest to itself
howManyCities = 128;
for i =1:howManyCities; intercitydist(i,i)=Inf; end
```

How to do it...

The recipe will show how to generate a graphic where each city location is connected to its nearest neighbor with a line.

Perform the following steps:

1. Define the adjacency matrix:

```
adjacency = zeros(howManyCities,howManyCities);

% find min distance and define adjacency = 1 for those entries
for i = 1:howManyCities
  alls = find(intercitydist(i,:)==
    min(intercitydist(i,:)));
  for j = 1:length(alls)
    adjacency(i,alls(j)) = 1;
    adjacency(alls(j),i) = 1;
  end
  clear alls
end
```

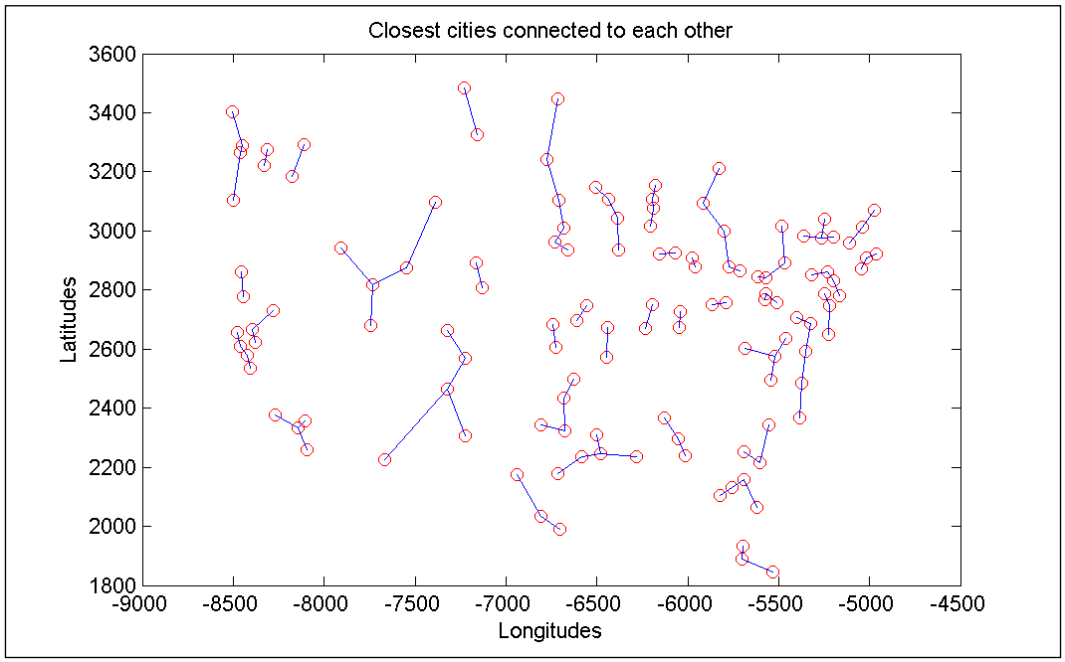
2. Plot the city coordinate with a marker. Overlay the connections defined in the adjacency matrix using the MATLAB command `gplot`:

```
plot(XYCoord(1:howManyCities,1),XYCoord(1:howManyCities,2),...
  'ro');hold on;
gplot(adjacency, XYCoord);
```

3. Add annotations:

```
title('Closest cities connected to each other');
xlabel('Longitudes');ylabel('Latitudes');
```

The output should be as follows:



How it works...

The figure shows 128 US cities with connections between their nearest neighbors. The recipe showed how to define an **adjacency matrix**, which is a matrix for n vertices of size n x n, where entry `i, j = 1`, if the vertices are to be connected, otherwise `i, j = 0`. The function `gplot` uses the definition from the adjacency matrix along with a second argument providing the coordinates positions for the nodes, to create a graph with all the nodes that are connected by a user definable line, as per the relationship defined in the adjacency matrix.

There's more...

A variation on this type of node link plot is to create a plot where the `x` axis holds all the nodes in a line and arcs connecting them show the relationship between pairs of nodes. In this section, you will use this said design to show cities that are within 100 miles of each other by road.

Perform these steps:

1. Sort the first row entry of the `intercitydist` matrix and then sort the city name list in the same order. Although this does not guarantee that cities are exactly sorted according to their intercity distance, it will pull cities near each other closer and this ordering will ultimately yield a cleaner looking visual when you put the arcs connecting the nodes. You will have to recalculate the intercity distances, because of this sorting:

```
% reset city distances to itself to zeros
for i = 1:howManyCities;  intercitydist(i,i) = 0; end

% rearrange city name list by intercity distance, use
% distances to any one city for sorting and that will put
% approximately close cities together
[balh I] = sort(intercitydist(1,:));
citynames = citynames(I);
XYCoord = XYCoord(I,:);

% recalculate intercity distance matrix
for i = 1:howManyCities;
  for j = 1:howManyCities
    if i==j
      intercitydist(i,i) = Inf;
    else
    intercitydist(i,j) = ...
      sqrt((XYCoord(i,1)-XYCoord(j,1))^2 + ...
        (XYCoord(i,2)-XYCoord(j,2))^2);
    end
  end
end
```

2. Define the adjacency matrix for this problem:

```
adjacency = zeros(howManyCities,howManyCities);
for i = 1:howManyCities
  alls = find(intercitydist(i,:)<100);
  for j = 1:length(alls)
    adjacency(i,alls(j)) = 1;
```

```
    adjacency(alls(j),i) = 1;
  end
  clear alls
end
```

3. Define figure and axes positioning:

```
%figure and axes positioning
figure('units','normalized','position',...
  [0.0844    0.2259    0.8839    0.4324]);
axes('Position',[0.0371    0.2893    0.9501    0.6296]);
xlim([1 howManyCities]);
ylim([0 100]);
hold on;
```

4. Put city names on the `x` axis. Rotate the `x` tick labels to 90 degrees. The city name list is too long to be accommodated by any lower angle of rotation.

```
set(gca,'xtick',1:howManyCities,'xticklabel',citynames,...
  'ticklength',[0.001 0]);
box on;
rotateXLabels(gca,90);
```

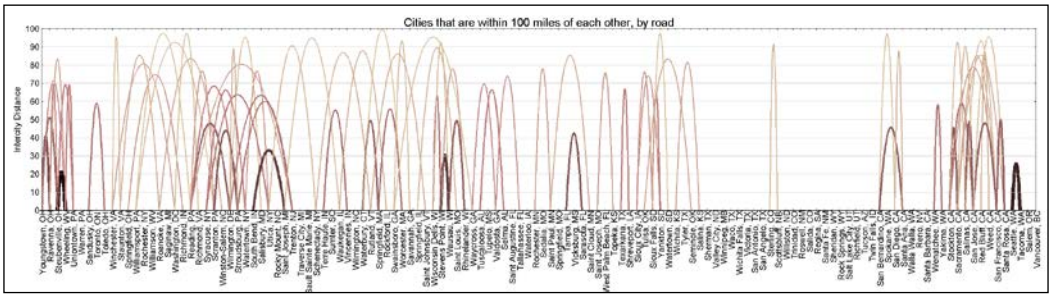
5. Draw the connecting arcs as per the adjacency matrix, such that the thickness and color of the arc approximately change with intercity distance. This will help to draw focus on cities that are more close than others.

```
m = colormap(pink(howManyCities+1));
cmin = min(min(intercitydist));
cmax = 150;

% plot arcs
for i = 1:howManyCities
  for j = 1:howManyCities
    if adjacency(i,j)==1

% draw parabolas for the arcs
      x=[i (i+j)/2 j];
      y=[0 intercitydist(i,j) 0];
      pol_camp=polyval(polyfit(x,y,2),linspace(i,j,25));
      plot(linspace(i,j,25),pol_camp,...
      'Color',m(fix((intercitydist(i,j)-cmin)/...
      (cmax-cmin)*howManyCities)+1,:),...
      'linewidth',100/intercitydist(i,j));
    end
  end
end.
```

6. Add annotations:

```
title('Cities within 100 miles of each other, by road',...
  'fontsize',14);
ylabel('Intercity Distance');

%add horizontal grid lines for readability
line(repmat(get(gca,'xlim'),9,1)',...
  [linspace(10,90,9); linspace(10,90,9)],'Color',
    [.8 .8 .8]);

%reposition the axes in case it moved
set(gca,'Position',[0.0371    0.2893    0.9501
  0.6296]);
ylim([0 max(get(gca,'ylim'))]);
```

The output is as follows:



How did you extract the color of the arc such that it corresponded to the intercity distance value? Here is how you translate **data value to a color** in your color map in MATLAB:

As introduced in *Chapter 1*, *Customizing Elements of MATLAB Graphics—the Basics*, the color map is essentially a color scale with RGB values. On invoking plot functions that use color, MATLAB automatically maps the color scale defined by the default color map, linearly, between the two extreme values in your dataset. You can manually define the color map to use. In this case, in step 5, you extracted a color map definition with at least as many distinct levels as the number of cities with `colormap(pink(howManyCities+1))`, where `pink` is one of the built-in color scales that come with MATLAB.

Now, given that your color map had `L` number of distinct rows, the way to extract the color that corresponds to a given value of intercity distance `V`, is to use the formula `fix((V-cmin)/(cmax-cmin)*L)+1`, where `cmax` and `cmin` are the two extreme data values you want to map to the two extreme colors in your color map. You set `cmax` to 150 (slightly larger than the 100 miles to make sure that the higher intercity distance values are not drawn in too light a shade); and `cmin` to the minimum intercity distance in your dataset.

Takeaways from this recipe:

- Use node link plots to convey relational information between data points

See also

Note that instead of a horizontal x axis, there exist variations of this design, where the nodes are laid out in a circular format (with the labels radiating outward) and the connections are represented as chords of this circle.

A related type of node link plot is the **tree plot**, designed to represent parent child relationship structure between nodes. MATLAB supports a single command to make a tree plot.

Look up **MATLAB help** on the `gplot` and `treeplot` commands.

Calendar heat map

This visualization is for showing any type of time series that is a daily reading directly superimposed on the calendar. In this example, the daily closing stock price for Google stocks for 2010 and 2011 is presented as a heat map on a 6 by 4 monthly calendar.

Getting ready

The daily price for Google stocks between January, 2010 to December, 2011 is loaded and sorted in ascending chronological order. A `NaN` value is recorded for those dates for which we have no records. See the code for this recipe (lines 12 – 36) for the exact commands to execute.

How to do it...

Perform the following:

1. The axes' positions for the 4 by 6 layout can be calculated as in *Chapter 1*, *Designing multigraph layouts*. The parameters used in this recipe are suited for a 6 by 4 format for data up to 2 years:

```
figure('units','normalized',...
  'Position',[ 0.3380    0.0889    0.6406    0.8157]);
colormap('cool');
xs = [0.03 .03+.005*1+1*.1525 0.03+.005*2+2*.1525 ...
  0.03+.005*3+3*.1525 0.03+.005*4+4*.1525...
```

```
  0.03+.005*5+5*.1525];
ys = [0.14 .14+0.04*1+1*.165   .14+0.04*2+2*.165  ...
  .14+0.04*3+3*.165];
```

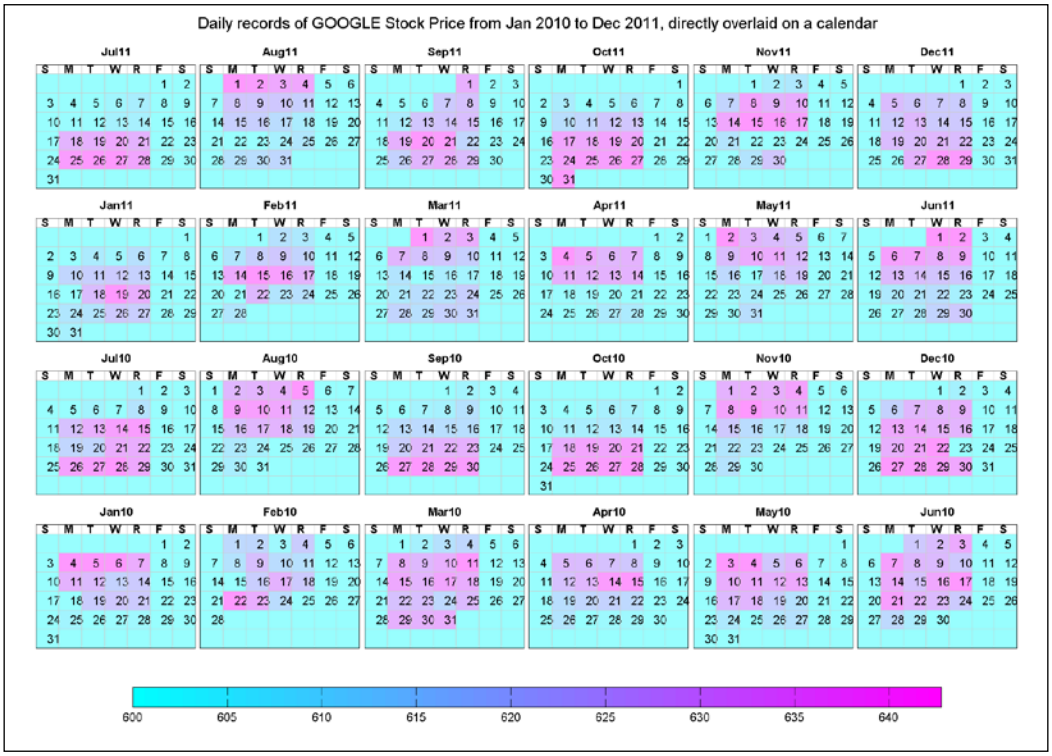
In order to generate the month headings, you have to increment a pointer for every month. Estimate how many days to count for each month by executing lines 46 - 55. It is a simple list of days to expect in a given month.

2. For each month, you have to extract the data, layout the calendar with the grid lines and date labels, and plot the heat map with the actual data. Here `(i, j)` are two indices for the year and month. For example, `Dcnt=0, i = 1, j = 1` refers to January 2010. Proceed as follows:

```
% position calendar for the month on screen
axes('Position',[xs(j) ys(i) .1525    0.165]);

% identify which newDatedata days belong to this segment
idx = find(newDateData >= ...
  datenum([datestr(newDateData(1)+Dcnt,'mm') '/01/'...
  datestr(newDateData(1)+Dcnt,'yyyy')]) & ...
  newDateData <= ...
    datenum([datestr(newDateData(1)+Dcnt,'mm') '/31/'...
    datestr(newDateData(1)+Dcnt,'yyyy')]));

% identify the calendar entries for this segment
A = calendar(newDateData(1)+Dcnt);

% pull out data for corresponding days using calendar format
data = NaN(size(A));
for k = 1:max(max(A))
  [xx yy] = find(A==k);
  data(xx,yy) = newData(idx(k));
end

% make a heatmap with some transparency to bleach out some
% color; add day labels (with blanks for days that dont
% belong in your month)
imagesc(data); alpha(.4);hold on;
set(gca,'fontweight','bold');
xlim([.5 7.5]); ylim([0 6.5]);
for m = 1:6
  for n= 1:7
    if A(m,n)~=0
      text(n,m,num2str(A(m,n)));
    end
  end
end

% add annotations
```

```
text(.75,.25,'S','fontweight','bold'); text(1.75,.25,'M','fontweig
ht','bold');
text(2.75,.25,'T','fontweight','bold');
text(3.75,.25,'W','fontweight','bold');
text(4.75,.25,'R','fontweight','bold');
text(5.75,.25,'F','fontweight','bold');
text(6.75,.25,'S','fontweight','bold');

title([datestr(newDateData(1)+Dcnt,'mmm') ...
  datestr(newDateData(1)+Dcnt,'yy')]);
set(gca,'xticklabel',[],'yticklabel',[],
  'ticklength',[0 0]);
line([-.5:7.5; -.5:7.5], [zeros(1,9); 6.5*ones(1,9)],...
  'Color',[.8 .8 .8]);
line([zeros(1,9); 7.5*ones(1,9)],[-.5:7.5; -.5:7.5],...
  'Color',[.8 .8 .8]);
box on;

% Increment day counter with # days as expected for this month
Dcnt=Dcnt+D(i,j);
```

The output is as follows:



How it works...

Laying the data directly on the calendar creates a strong relationship between the day the data is recorded and the data itself. Records that contain daily readings can benefit from this type of visualization as the association of the reading to the time line is very strong by design. It is easy to correlate trends in the data to a specific event or some annual, seasonal, monthly phenomenon.

There's more...

The final step is the addition of a color legend and overall title to the graphic:

```
colorbar('Location','SouthOutside','Position',...
  [ 0.1227    0.0613    0.7750    0.0263]);alpha(.4);

annotation('textbox',[0.1800 0.9354 0.8366 0.0571],...
  'String','Daily records of GOOGLE Stock Price from ...
  Jan 2010 to Dec 2011, directly overlaid on a calendar',...
  'LineStyle','none','Fontsize',14);
```

Takeaways from this recipe:

- Use calendar heat maps to present daily numerical records

See also

Look up **MATLAB help** on the `colorbar`, `alpha`, `imagesc`, and `calendar` commands.

Distributional data analysis

This recipe demonstrates some common first order visualizations used to investigate the empirical distribution of a one-dimensional dataset.

Getting ready

Load `distriAnalysisData.mat`:

```
load distriAnalysisData;
```

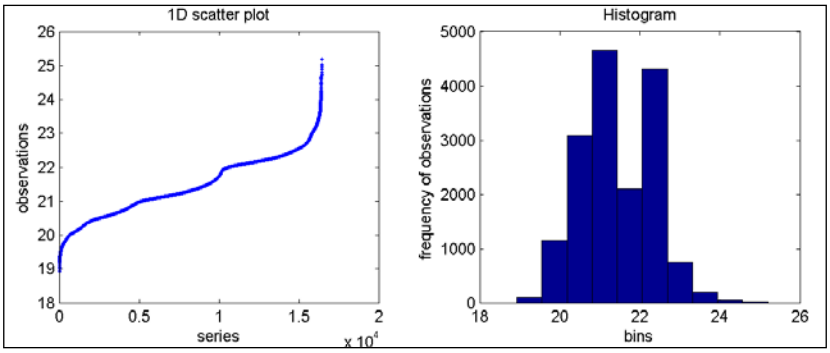
How to do it...

Perform the following:

1. Look at a sorted scatter plot and the histogram:

```
subplot(1,2,1);
plot(sort(B),'.');
xlabel('series');ylabel('observations');
title('1D scatter plot');
subplot(1,2,2);
hist(B);
xlabel('bins');ylabel('frequency of observations');
title('Histogram');
```

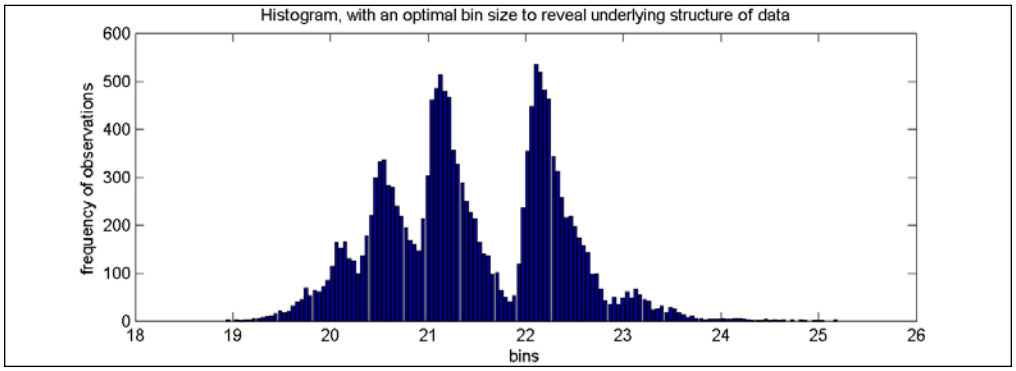
   The output is as follows:



2. Try an alternative **bin size** for the **histogram**:

```
hist(B,200);
title('Alternate binning, bin size = 200');
```

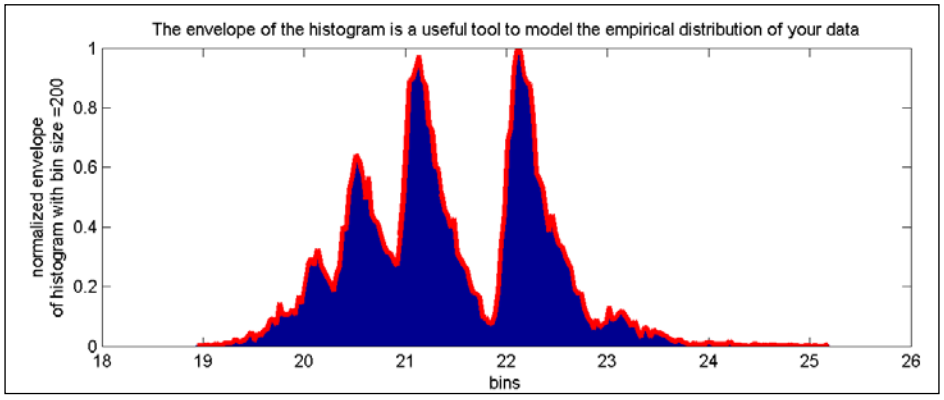
   The output is as follows:



3. Plot the **envelope** of the histogram as an estimate for the probability density function for this dataset:

```
% Re-compute the histogram with nbins = 200
[N c] = hist(B,200);

% Compute the envelope with a spline fit
env = interp1(c,N,c,'spline');

%plot normalized envelope with actual data on xaxis
bar(c,N./max(N));hold;
plot(c,env./max(env),'r','Linewidth',2);

% annotate
xlabel('bins');
ylabel({'normalized envelope',...
  'of histogram with bin size =200'});
```

The output is as follows:



How it works...

In this recipe, the bin size is chosen as the square root of the number of points. There are other alternatives people recommend. The bin size will impact how the histogram looks. A good choice of bin size is essential to understand the underlying structure in the data. Often, the lack of empty internal bins is used to choose the lower limit on the size of bins. The smooth appearance of the histogram profile (versus a square finish) is used to choose the upper limit of the size of bins.

The envelope of the histogram is a useful tool to model the empirical distribution of the data.

There's more...

It is often desired to identify if two datasets come from the same distribution or, if data can be assumed to come from a known distribution such as the normal distribution. In the latter case, the advantage is that, a lot of mathematics becomes directly applicable if the dataset can be assumed to be normal (or other known distributions).

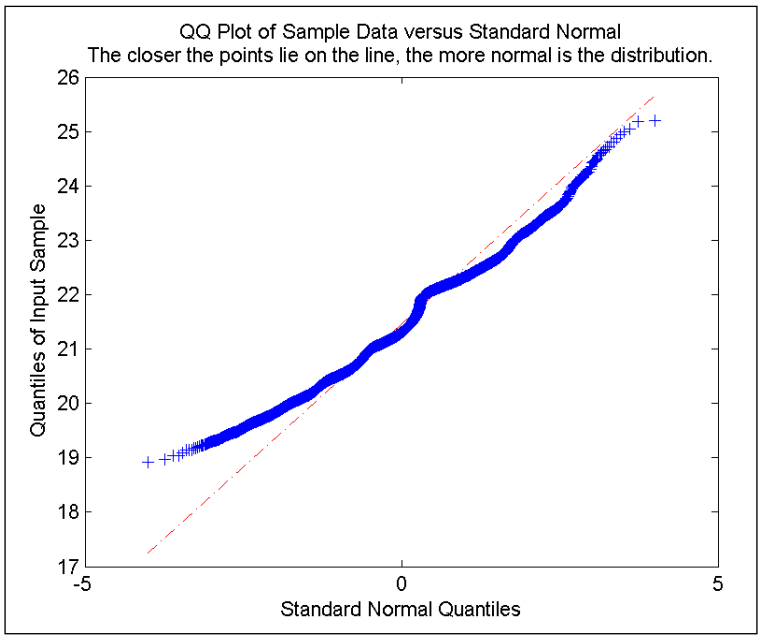
One of the visual methods to test for if two datasets come from the same distribution is the use of quantile-quantile plots, where the quantiles from the two distributions are plotted against each other. Here, you compared the dataset to the quantiles of the standard normal distribution.

If they come from the same distribution, the plot is expected to be close to linear.

The MATLAB function `qqplot`, part of the MATLAB statistical toolbox, will display such a quantile-quantile plot; in this section, you will use it to compare the sample quantiles of the current dataset to the theoretical quantiles from a normal distribution:

```
qqplot(B);
```

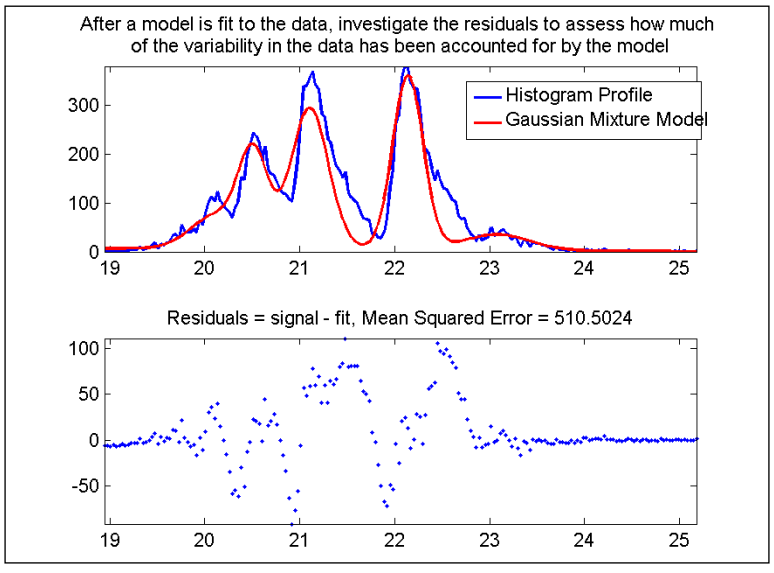
The output is as follows:



As a final step, you will fit a mix of normal distributions to the histogram profile. You will graphically evaluate the **residuals** to assess how well the model fits the profile. The matrix `sigma_ampl` and `mu` hold the mean, standard deviations, and contribution coefficients for six normal distributions. These are added to create the data from the model as follows:

```
sigma_ampl = [79.26  8.12 5 6.25 5.06 11.11 577.45 ...
  531.38 962.45 1800 1800 357.92];
mu=[29 38 51 70 103 133];

% Gaussian mixture model
f_sum=0;x=1:200;
for i=1:6
  f_sum=f_sum+sigma_ampl(i+6)./...
  (sigma_ampl(i)).*exp(-(x-mu(i)).^2./(2*sigma_ampl(i).^2));
end
subplot(2,1,1);
h(1)=plot(c,env,'Linewidth',1.5);hold on;
h(2)=plot(c,f_sum,'r','Linewidth',1.5); axis tight
legendflex(h,{'Histogram Profile',...
  'Gaussian Mixture Model'},'ref',gcf,...
  'anchor',{'ne','ne'},'xscale',.5,'buffer',[-50 -50]);
title('Overlay histogram profile with the Gaussian mixture model');
subplot(2,1,2);
plot(c,env-f_sum,'.');axis tight;
title(['Residuals = signal - fit, Mean Squared Error = ' ...
  num2str(sqrt(sum(abs(env-f_sum).^2)))]);
```

The output is as follows:



Takeaways from this recipe:

- Use histograms of appropriate bin size to investigate your data
- Use quantile-quantile plots (QQ plots) to investigate the normality of your data
- Investigate your residuals after fitting a model to the data to assess goodness of fit

See also

Look up **MATLAB help** on the `hist`, `qqplot`, and `interpl` commands.

Time series analysis

A time series is a set of data points recorded over time, usually at uniform intervals. The important thing that distinguishes time series from other one-dimensional data is the property of natural temporal ordering present in the data. In this recipe, you will analyze human heart rate data and visualize your findings.

Getting ready

In this recipe, you will use two time series that contain evenly-spaced measurements of instantaneous heart rate (in units of beats per minute) at 0.5 second intervals from two subjects. The data was downloaded from an ECG website maintained at MIT (GoldBerger and Mordy).

Load the data as follows:

```
load timeseriesAnalysis;
```

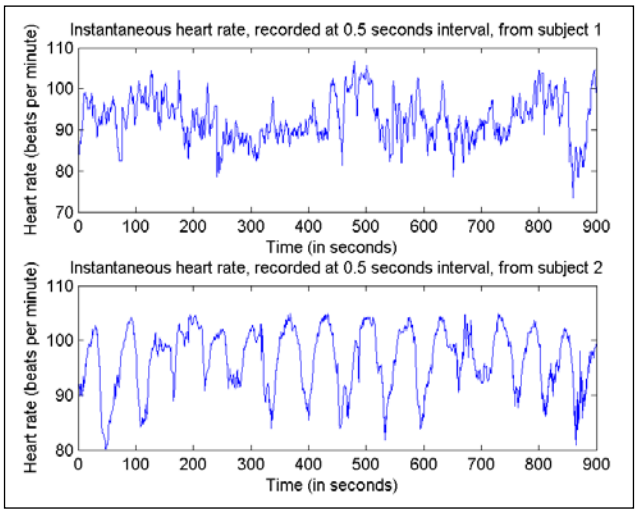
How to do it...

Perform the following steps (Code is shown here for one series. Apply similarly to the other):

1. Plot the data:

```
subplot(2,1,1);
plot(x,ydata1);
title('Instantaneous heart rate, recorded at 0.5 seconds interval,
from subject 1');
xlabel('Time (in seconds)');
ylabel('Heart rate (beats per minute)');
```

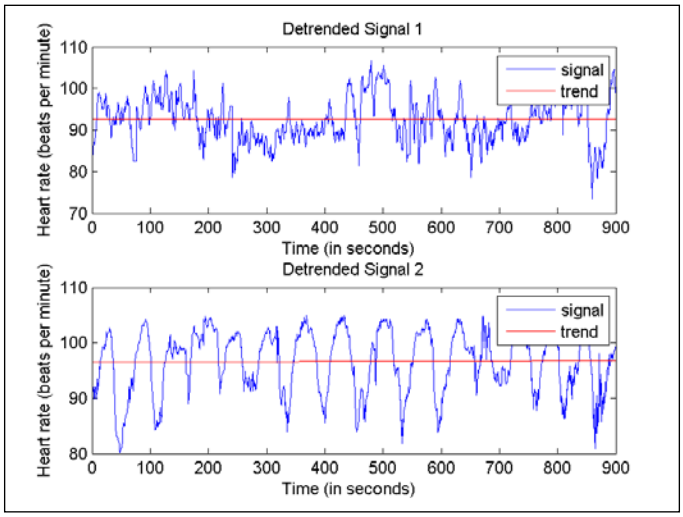
The output is as follows:



2. **De-trend** (remove the best straight line fit from the data):

```
y_detrended1 = detrend(ydata1);
plot(x, ydata1,'-',x, ydata1-y_detrended1,'r');
title('Detrended Signal 1');
legend({'signal','trend'});
xlabel('Time (in seconds)');
ylabel('Heart rate (beats per minute)');
```

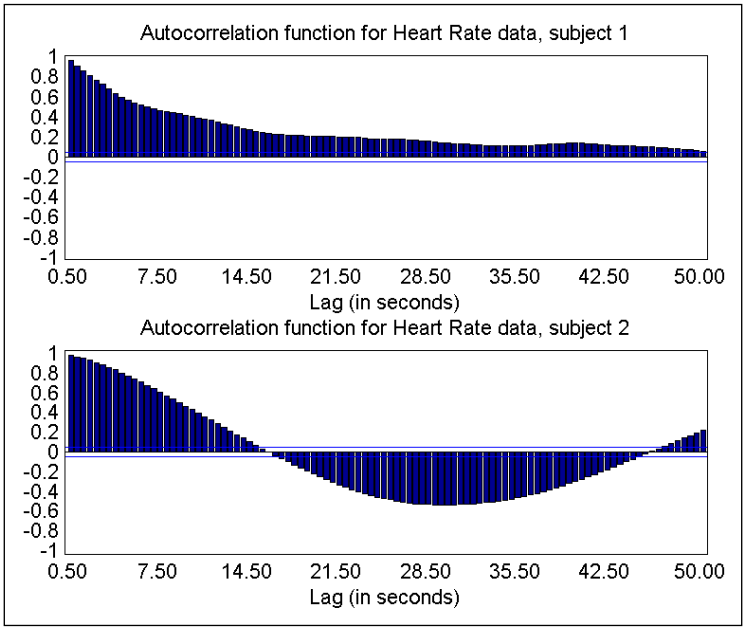
The output is as follows:



3. Compute the **auto correlation function**. It is also known as a **correlogram**.

```
y_autoCorr1 = acf(subplot(2,1,1),ydata1,100);

% annotations
set(get(gca,'title'),'String',...
  'Autocorrelation function for Heart Rate data,
    subject 1');
set(get(gca,'xlabel'),'String','Lag (in seconds)');
tt = get(gca,'xtick');
for i = 1:length(tt); ttc{i} =
  sprintf('%.2f ',0.5*tt(i));
end
set(gca,'xticklabel',ttc);
```

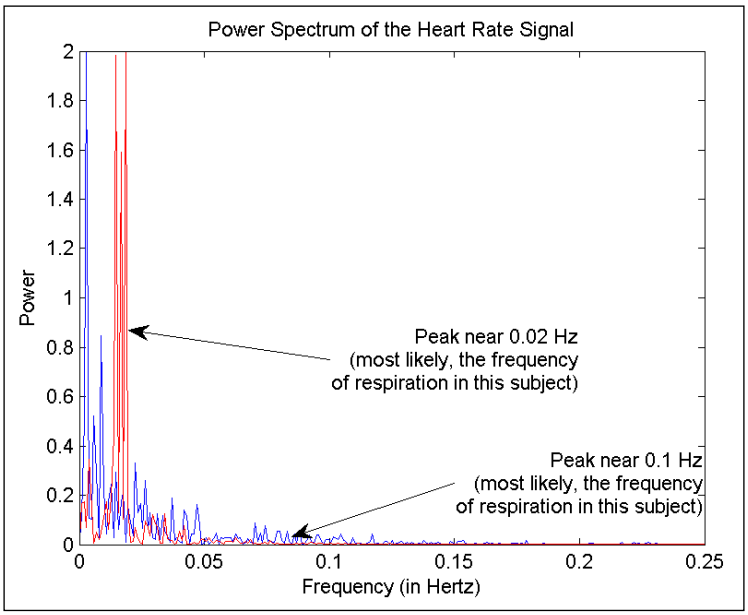
The output is as follows:



4. Take a **Fourier transform** to investigate the signal in frequency domain. Plot the **power spectrum** of the signal (plot of **Power** versus **Frequency**):

```
% Use next highest power of 2 greater than or equal to
% length(x) to calculate fft
nfft = 2^(nextpow2(length(x)));

% Take fft, padding with zeros with zero padding
ySpectrum1 = fft(y_detrended1,nfft);
NumUniquePts = ceil((nfft+1)/2);

% FFT is symmetric, throw away second half and use the
% magnitude of the coefficients only
powerSpectrum1 = abs(ySpectrum1(1:NumUniquePts));

% Scale the fft
powerSpectrum1 = powerSpectrum1./max(powerSpectrum1);

% Calculate power spectrum, preserving total energy
powerSpectrum1 = powerSpectrum1.^2;

% odd nfft excludes Nyquist point
if rem(nfft, 2)
  powerSpectrum1(2:end) = powerSpectrum1(2:end)*2;
else
  powerSpectrum1(2:end -1) = powerSpectrum1(2:end -1)*2;
end

% Sampling frequency
Fs = 1/(x(2)-x(1));
f = (0:NumUniquePts-1)*Fs/nfft;

%plotting
plot(f,powerSpectrum1,'-');
```

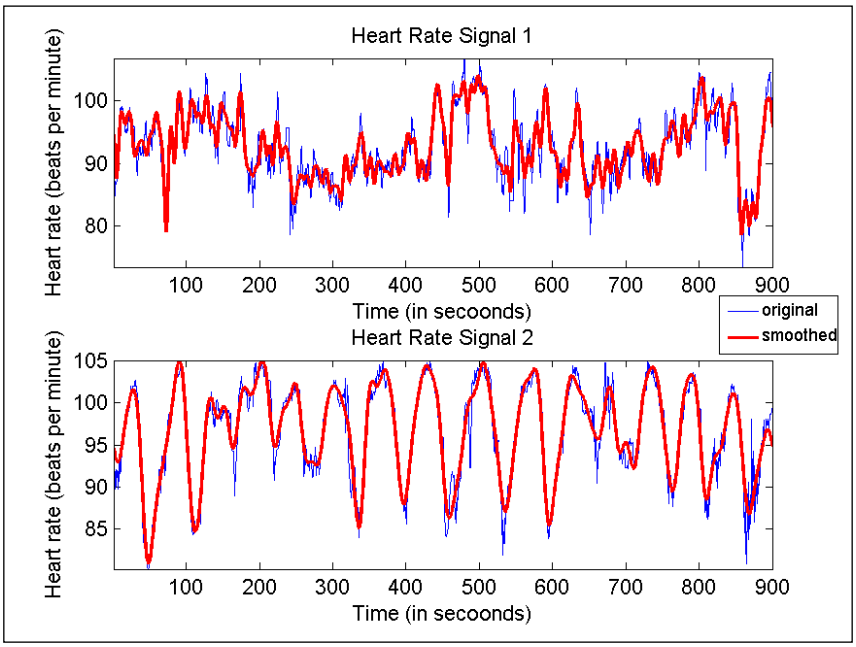
The output is as follows:



5. Compute a **global smoothing** by zeroing the less significant Fourier coefficients (usually they will correspond to the higher frequencies that give the noisy appearance to the signal):

```
% Dont use zero padding
ySpectrum1 = fft(ydata1);

% Zero out less significant coeffs
freqInd1=find(abs(ySpectrum1)<400);
ySpectrum1(freqInd1)=0;

% Reconstruct signal
y_cyclic1=ifft(ySpectrum1);

%plotting
h(1)= plot(x,ydata1,'b');hold on;
h(2)=plot(x,y_cyclic1,'r','linewidth',1.5);
title('Smoothed Heart Rate Signal 1');axis tight;
xlabel('Time (in secoonds)');
ylabel('Heart rate (beats per minute)');
```

The output is as follows:



How it works...

Time series data analysis is a complex and mature discipline. A small snapshot was presented in this recipe to illustrate the special ways this type of data is handled.

Line charts are the most common ways of representing time series data.

Time series can be investigated in both time and frequency domains. In the time domain, techniques such as the auto-correlation function reveal the similarity between observations as a function of the time separation between them. In the frequency domain, techniques such as the Fourier transform are used to investigate the spectral properties.

In the analysis performed in this recipe, the de-trending step did not show much effect (possibly because the data was already flat and did not have a significant trend that could be removed).

The auto-correlation function showed a distinct difference in the signatures from the two signals and the non-zero correlation coefficients clearly established that the data is far from random. The file exchange submission by Calvin Price was used to compute the auto-correlation function in this recipe.

The rapid oscillation visible in series 1 was reflected in its power spectrum by a peak near 0.1 Hz. This component of Heart Rate Variation is probably respiratory sinus arrhythmia, a modulation of heart rate that is greatest in young subjects, and gradually decreases in amplitude with increasing age. By contrast, almost all of the power in series 2 was concentrated at a much lower frequency (about 0.02 Hz). These dynamics are commonly observed in the context of congestive heart failure, where circulatory delays interfere with the regulation of carbon dioxide and oxygen in the blood, leading to slow oscillations of heart rate.

Finally, a global smooth for the signals was shown, computed by zeroing the less significant Fourier coefficients. In this case, the higher frequency components were the less significant ones and hence the process produced the effect of reducing the high frequency noise in the signal.

Takeaways from this recipe:

- Use line charts to present time series data
- Use correlograms to check for randomness of your dataset
- Use Fourier transforms to investigate spectral properties of your dataset

See also

Look up **MATLAB help** on the `acf`, `fft`, and `ifft` commands.

3

Graduating to Two-dimensional Data Displays

In this chapter, we will cover:

- 2D scatter plots
- Scatter plot smoothing
- Bidirectional error bars
- 2D node link plots
- Dendrograms and clustergrams
- Contour plots
- Gridding scattered data
- Choropleth maps
- Thematic maps with symbols
- Flow maps

Introduction

This chapter focuses on two-dimensional data displays. The most common chart types for two-dimensional data visualization are **scatter plots** and **heat maps**. Scatter plots use positional coordinates to show numerical data, which is a visualization best practice, as discussed in *Chapter 2*, *Diving into One-dimensional Data Displays*. Heat maps use color to code numerical data. Heat maps typically use a sequential color scale which works well to convey the sense of ordering in the values and also reveal spatial patterns. MATLAB supports many additional chart options for 2D data as you will see in this chapter, such as contour maps, dendrograms, and flow maps. Several recipes also present methods to plot geo-specific data on a map.

The core technologies for visualizing multidimensional data in MATLAB are briefly discussed in the following section.

Surface, patch, and shading

MATLAB uses **surface** and **patch** elements as the basic planar building blocks. Surface objects are quadrilaterals and more suitable for presenting planar topographies. Patch objects are polygons, geared toward 3D modeling. These objects work with data on a uniform grid. The color for the face of the grid element is determined from the values it represents. First, MATLAB transforms the data value in your matrix to a color value by linearly mapping the data value to the color map (the two extreme colors in the color map correspond to the highest and lowest value in your data matrix). As per the **shading** algorithm, the color for the face of a grid element is interpreted at the vertex of the grid element (the color within the grid element being a bilinear function of the local coordinates) as in shading type **interp**; alternately, that color is a constant associated with that grid element as in shading type **flat** or **faceted**. The `pcolor` command is a surface object with faceted shading and no grid lines. Another option is to use a wireframe or **mesh**. (But this is still a surface object plot with the face color set to the background color to simulate wireframes).

Two-dimensional scatter plots

Two-dimensional **scatter plots** are paired values plotted one versus the other. In this recipe, you will explore a variety of techniques used in making scatter plot visualizations.

Getting ready

You will use a famous dataset from pattern recognition literature first reported in Fisher's classic 1936 paper. The iris dataset contains four attributes of the iris plant (sepal length, sepal width, petal length, and petal width). The data is widely available, and also part of the code repository with this book.

```
[attribclassName] = xlsread('iris.xlsx');
```

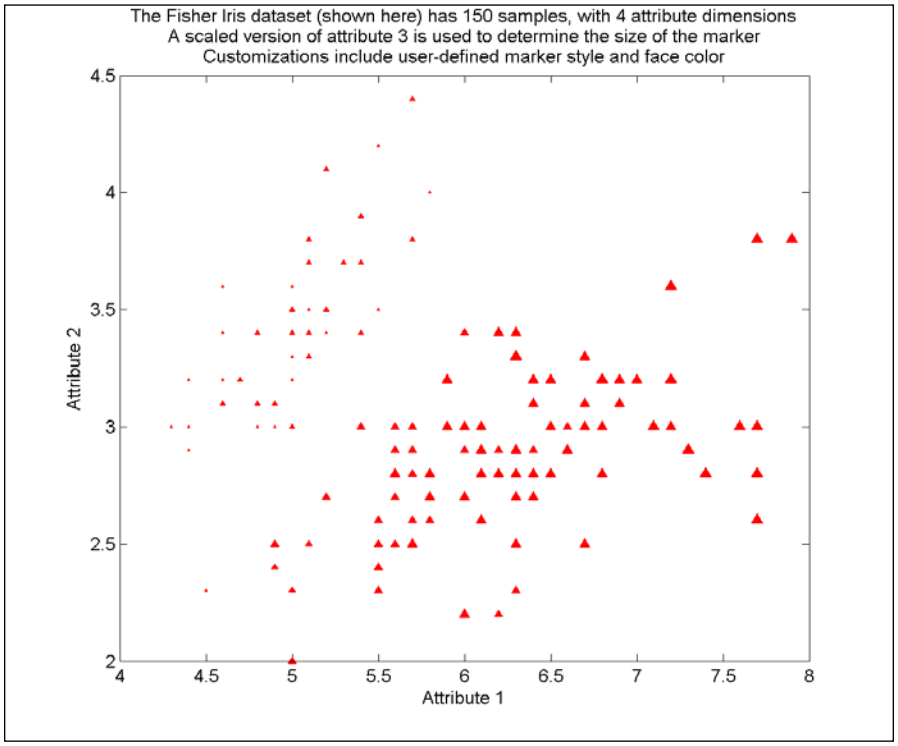
How to do it...

Perform the following steps:

1. Make the basic scatter plot using the command `scatter`. Execute lines 16 – 21 from the source code of this recipe to generate the annotations:

```
scatter(attrib(:,1),attrib(:,2),10*attrib(:,3),...
  [1 0 0],'filled','Marker','^');
```

   The output is as follows:



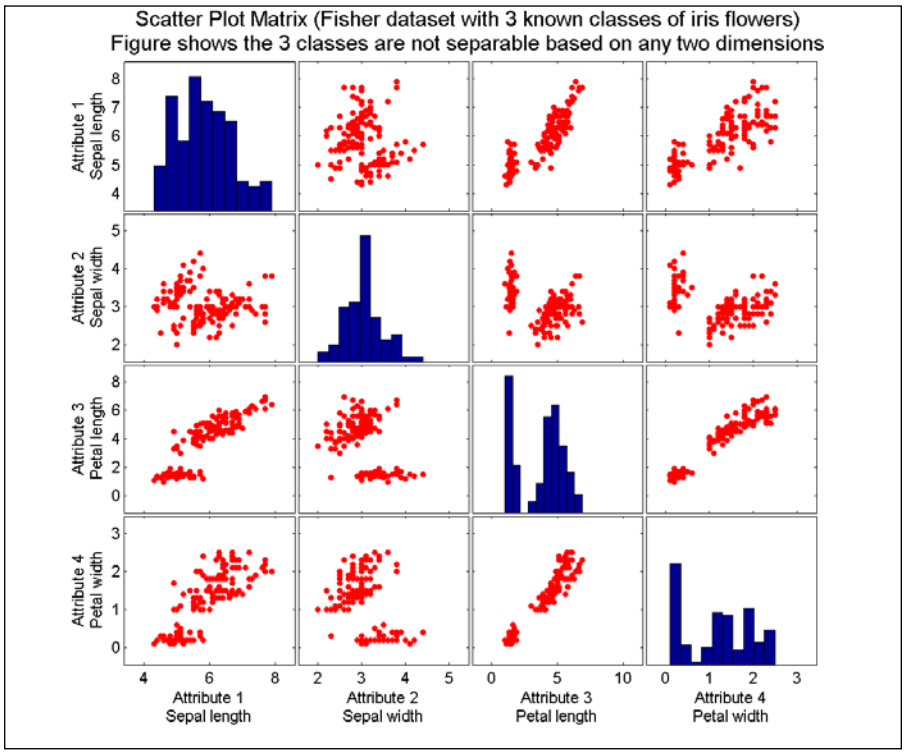
2. Make a **scatter plot matrix** using the command `plotmatrix`. This brings up scatter plots for all pairs of attributes, in a gridded format, and the histogram of each attribute.

```
[H,AX,BigAx,P] = plotmatrix(attrib,'r.');
```

3. You can use the output vector from the `plotmatrix` call to add annotations to the figure, as shown in lines 33 – 40 of source code for this recipe. For example:

```
for i = 1:4
  set(get(AX(i,1),'ylabel'),'string',...
  ['Attribute ' num2str(i)]);
  set(get(AX(4,i),'xlabel'),'string',...
  ['Attribute ' num2str(i)]);
end
```

The output is as follows:



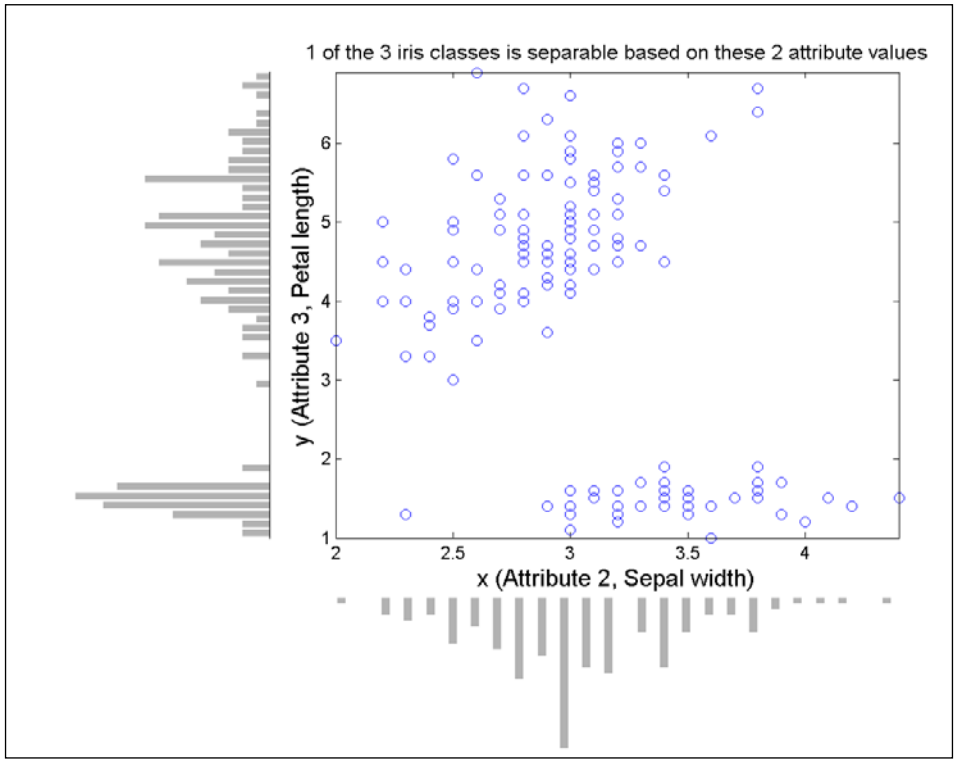
4. Review the distribution of the `x` and `y` values in one dimension at the same time as you review the 2D scatter plot, using the function `scatterHistV`(supplied with this book):

```
[mainDataAxesxhistAxesyhistAxes] = ...
  scatterHistV(attrib(:,2),attrib(:,3),50, 50);
```

5. Add annotations using handles returned by the function as shown in lines 47 – 49 of the source code for this recipe. For example:

```
set(get(mainDataAxes,'title'),'String',...
  ['Scatter plot view alongside distribution'...
  ' of x and y'],'Fontsize',14);
```

The output is as follows:



How it works...

In this recipe, you learned to make 2D scatter plots, a scatter plot matrix, and then a view to simultaneously look at the scatter plot alongside the univariate distributions of the two data dimensions. The scatter plot matrix showed that the three classes of the iris flower are not clearly separable using any two data dimensions. In the previous figure, one of the classes is clearly separated using two of the four attribute dimensions.

The 2D scatter plot and the scatter plot matrix were built using direct MATLAB commands. The `scatterhistV` is a custom function that internally splits the figure area into three axes. It uses the `hist` function to calculate the histogram and then uses the `bar` function to plot the univariate histograms into two of those axes positions. It sets the `xdir` and `ydir` properties to `reverse` to turn the histograms around. The main data plot is done using the `scatter` command. The code for this function is part of the source code with the book.

Takeaways from this recipe:

- Use scatter plots to review your two dimensional data

See also

Look up **MATLAB help** on the `scatter`, `scatterhist`, and `plotmatrix` commands.

Scatter plot smoothing

When the data volumes are huge, there may be significant over-plotting in a simple scatter plot view. In that case, one may need to look at a higher abstraction of the data than all the data points at once. To this end, there exist techniques such as **density plots** and **scatter plot smoothing**, which are shown in this recipe.

Getting ready

In this recipe, you will look at a dataset with 3000 points that shows significant over-plotting in the scatter plot view. Generate this as follows:

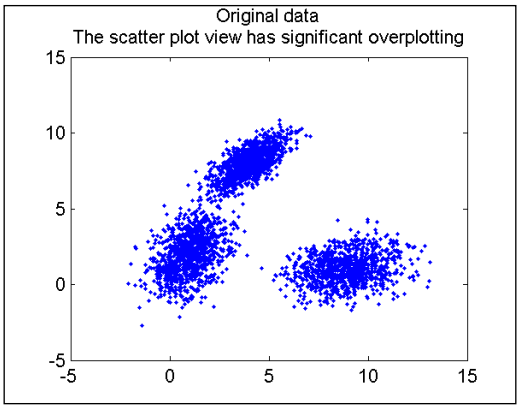
```
z = [repmat([1 2],1000,1) + randn(1000,2)*[1 .5; 0 1.32];...
  repmat([9 1],1000,1) + randn(1000,2)*[1.4 .2; 0 0.98];...
  repmat([4 8],1000,1) + randn(1000,2)*[1 .7; 0  0.71];];
```

How to do it...

Perform the following steps:

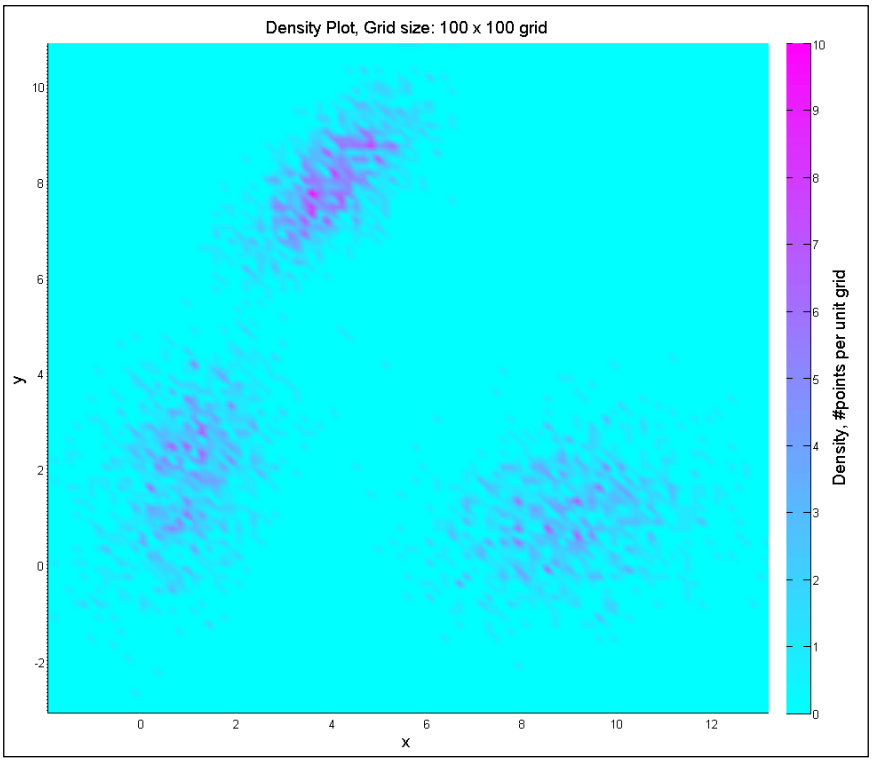
1. Plot the original raw data and observe the heavily over-plotted zones where the data is hidden and the user has no idea about the actual number of points in that blob. See the results in the following screenshot:

   ```
   plot(z(:,1),z(:,2),'.');
   ```



2. Create a density plot on a 100 x 100 grid to obtain the following plot:

```
densityPlot2D(z(:,1),z(:,2),100);
```



How it works...

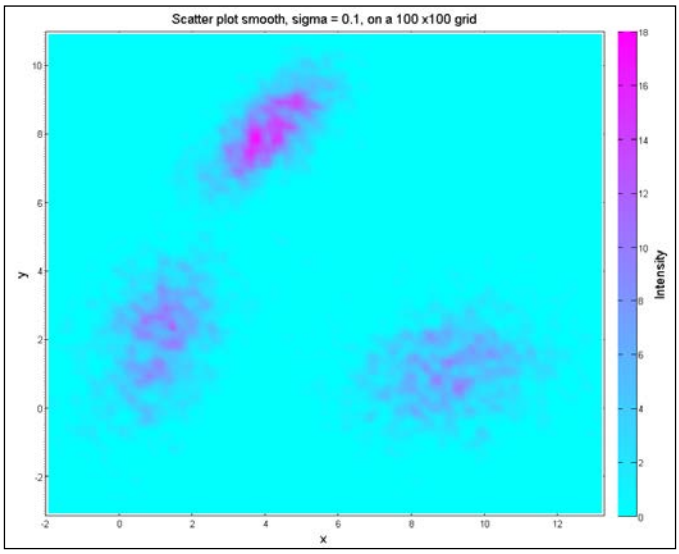
For creating the **density plots**, data is gridded onto a coarser grid and the frequency of data points in each grid is computed. This 2D histogram is then presented using a heat map. Note that you implemented heat maps in *Chapter 2*, *Calendar Heat Map*, with the `imagesc` command (which uses MATLAB's image viewing technology). In this recipe, you used the `surf` command, which fits a surface to the data. The `shading` algorithm used is the default (`faceted`), so that each grid element has one constant color that corresponds to the frequency of points in that bin. The `densityPlot2D.m`, a custom function that is part of the code repository with this book, implements this strategy. It takes the `x` and `y` data vectors and also the grid size on which to compute the density plot as inputs. With this strategy, over-plotting is no longer a problem and the plot now reveals the true picture of the data distribution to the user.

There's more...

An alternate way to view high density scatter plots is to use **scatter plot smooths**. For this, at each point, a Gaussian distribution with a certain spread is assumed, and the sum of all the Gaussians is computed on a fine data grid. This represents the smoothed version of the scatter plot (the spread assumed for the Gaussian function acts as a tunable parameter in this visualization; the higher the spread, the more diffused or smoothed is the result). The `scatterPlotSmooth2D.m`, a custom function that is part of the code repository with this book, implements this strategy. It takes the two dimensions of the data, the spread, and the grid size as inputs.

```
scatterPlotSmooth2D(z(:,1),z(:,2),.1,300);
```

The output is as follows:



An interactive version of this plot could allow the user to zoom in on these heat map views. When the total number of points in focus falls under a certain threshold, the graphic could switch mode and reveal the actual points. One of the drawbacks of these methods is that they are costly to evaluate (particularly true for the scatter plot **smooth**). This cost magnifies when the plot is interactive and the zoom functionality requires regridding the data and refitting the surface in real time.

Takeaways from this recipe:

- Use scatter plot smoothing or density plot views to review your high volume two-dimensional data at a manageable scale

See also

Look up **MATLAB help** on the `surf`, `interp2`, `intersect`, `cat`, and `meshgrid` commands.

Bidirectional error bars

In *Chapter 1*, *Laying out long tick labels without overwriting*, you plotted error bars that were three standard deviations out, on both sides of the median gene expression value, for each cancer type. In that case, the biological and measurement variation was only in the variable plotted along the y axis. There was no such ambiguity in the variable plotted on the x axis. When plotting two-dimensional data, there may be variability (both real and measurement related) along both data dimensions. This recipe shows how to present **bidirectional error bars**.

Getting ready

In this recipe you will use recordings of position and velocity for turbulent wind flow. Load the data:

```
load flatPlateBoundaryLayerData
```

How to do it...

Perform the following steps:

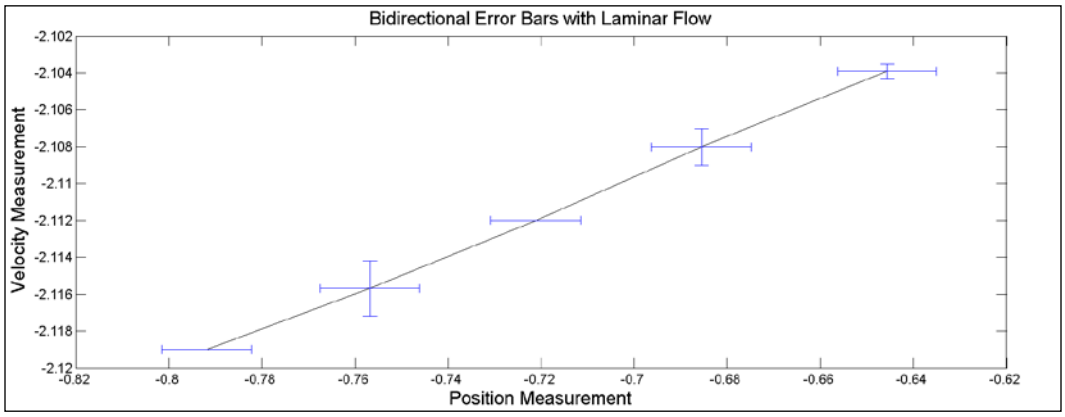
1. Bin the data by executing the lines 14 – 31.
2. Call the custom function `binDirErrBar.m` (part of the code bundle with this book) with the four parameters: `x` and `y`, the average data for position and velocity, and `e_x` and `e_y`, the error bars around the average position and velocity for each bin:

```
h = biDirErrBar(x,y,e_x,e_y);
```

3. Add annotations using the handle returned by the previous function:

```
set(get(h,'title'),'string',...
  'Bidirectional Error Bars','Fontsize',15);
set(get(h,'xlabel'),'string',...
  'Position Measurement','Fontsize',15);
set(get(h,'ylabel'),'string','Velocity Measurement',...
  'Fontsize',15);
```

The output is as follows:



How it works...

The previous figure shows the variation in both x and y dimensions. Note that in a real situation the measurement variation in x and y will be available to plug into this view. In this case, you artificially generated it by binning the data in x and y and calculating a spread to represent uncertainty about the mean value. The mean value in x and y in each bin is joined with a line. The three standard deviation values in x and y directions in each bin are plotted as the two error bars respectively.

The `binDirErrBar.m` function internally calls the MATLAB built-in `errorbar` function to plot the vertical error bars. Then, it uses the MATLAB command `line` to construct the horizontal error bars. The vertical bars at the end of the horizontal error bars are put at 2 percent of the maximum y observation. For a different range of data, experiment with these parameters to modify the bidirectional error bar function.

Takeaways from this recipe:

- Use bidirectional error bars to present the physical and/or measurement related uncertainty in both data dimensions

See also

Look up **MATLAB help** on the `errorbar` command.

2D node link plots

In this recipe, you will see the two-dimensional analog of **node link plots** you built in *Chapter 2*, *Node Link Plots*.

Getting ready

The data was obtained from UCI's website and provided as part of the book. It presents the network of co-appearances of characters in the novel *Les Miserables*, by *Victor Hugo*. Nodes represent characters as indicated by the labels. Edges hold values connecting two nodes (or zero for no connection). Non-zero edge values in the matrix represent the number of co-appearances of the pair of characters. Load the data:

```
load characterCoOccurences
```

How to do it...

Perform the following steps:

1. Set up the figure and axes:

```
figure('units','normalized','Position',...
  [0.2 0.13 0.49 0.77]);
mainAx = axes('position',[0.14 0.01 0.80 0.80]);
```

2. Set up a color map. Reverse the color map matrix so that low values correspond to lighter colors:

```
m = colormap(copper);m = m(end:-1:1,:);colormap(m);
```

3. Create the heat map of the data with a surface plot:

```
r=surf(lesMiserables);view(2);

% set the edge to semi-transparent and the color limits
set(r,'edgealpha',0.2);
set(gca,'clim',[min(lesMiserables(:)) max(lesMiserables(:))]);
```

4. Add annotations:

```
% position the colorbar
h=colorbar('northoutside');
% add tick labels
set(mainAx,'xAxisLocation','top','xtick',0:78,...
  'xticklabel',{' ' LABELS{:} ' '},...
  'ytick',0:78,'yticklabel',{' ' LABELS{:} ' '},...
  'ticklength',[0 0],'fontsize',8);
axis tight;
rotateXLabels(gca,90);
```

5. Resize axes for proper visibility of all elements:

```
% reposition to correct impact from tick label rotation
set(h,'position',...
  [0.1006    0.9209    0.8047    0.0128]);
set(get(h,'title'),'String',['Number of '...
  'co-appearances of characters in Les Miserables'...
  'by Victor Hugo']);
set(mainAx,'position',...
  [ 0.1361    0.0143    0.8042    0.8038]);
```

The output is as follows:

How it works...

The screenshot shows the co-appearance of characters in *Les Miserables*, by *Victor Hugo*. The most common co-characters are Javerte and Marguerite.

In step 2, you reversed the color map matrix and made the lightest to darkest colors correspond to the lowest to the highest values. This is a useful measure for visualizing sparse matrix in a heat map (otherwise the dark color makes the plot appear very busy).

In step 3, you used the `surf` command to make the actual data heat map.

In step 4, among others, you set the x axis location property to `top`, which made the x labels appear above. Also notice how you add two blank strings to the cell array containing the character names, at the first and last positions so that the names lines up with the columns instead of the tick positions.

Takeaways from this recipe:

- Use heat maps to present two relationships between two data dimensions
- Use light color to represent low values for sparse matrix to reduce color saturation

See also

Look up **MATLAB help** on the `surf` command.

Dendrograms and clustergrams

A **dendrogram** is a tree diagram used to illustrate the arrangement of clusters in the data, produced by agglomerative or hierarchical clustering. The outer most set of nodes represents individual observations, and the remaining nodes represent the clusters to which the data belong, with the arrows representing the distance (dissimilarity). The distance between merged clusters is monotone increasing with the level of the merger; the height of each node in the plot is proportional to the value of the intergroup dissimilarity between its two daughters.

A **clustergram** is essentially a heat map with dendrograms attached to the top of the y axis and the left of the x axis (so that the heat map is sorted along each variable). It is a nice way to observe patterns within the data (in terms of the effect of either of the variables on the data).

Getting ready

This recipe will use the `linkage`, `pdist`, and `dendrogram` functions from the MATLAB Statistics Toolbox™ package.

You will use the cancer gene expression dataset (supplied with this book) to look at expression levels from 30 genes across 30 samples for leukemia. Extract the data as follows:

```
load 14cancer.mat
data = [Xtrain(find(ytrainLabels==9),genesSet); ...
  Xtest(find(ytestLabels==9),genesSet)];
```

How to do it...

Perform the following steps:

1. Layout the three principal components to this plot. Two of them will hold the dendrograms. One of them will hold the heat map.

```
figure('units','normalized','Position',...
  [0.5641    0.2407    0.3807    0.6426]);
mainPanel = axes('Position',[.25 .08 .69 .69]);
leftPanel = axes('Position',[.08 .08 .17 .69]);
topPanel =  axes('Position',[.25 .77 .69 .21]);
```

2. Construct the dendrograms:

```
Z_genes =   linkage(pdist(data'));
Z_samples = linkage(pdist(data));
```

3. Manipulate the color map such that lower values appear in light color:

```
m = colormap(pink); m = m(end:-1:1,:);
colormap(m);
```

4. Plot the dendrograms at their target axis (with thick lines):

```
axes(leftPanel);
h = dendrogram(Z_samples,'orient','left');
set(h,'color',[0.1179 0 0],'linewidth',2);
axes(topPanel); h = dendrogram(Z_genes,0);
set(h,'color',[0.1179 0 0],'linewidth',2);
```

5. Extract the ordering proposed by the dendrograms, rearrange the data in that order, and create a heat map view of the ordered data:

```
%% get the ordering proposed by the dendrograms
Z_samples_order = str2num(get(leftPanel,'yticklabel'));
Z_genes_order = str2num(get(topPanel,'xticklabel'));
axes(mainPanel);
% plot a heat map of the ordered data
surf(data(Z_samples_order,Z_genes_order),...
  'edgecolor',[.8 .8 .8]);view(2);
set(mainPanel,'Xticklabel',[],'yticklabel',[]);
```

6. Align the x and y axis between the dendrograms and the heat map:

```
set(leftPanel,'ylim',[1 size(data,1)],'Visible','Off');
set(topPanel,'xlim',[1 size(data,2)],'Visible','Off');
axes(mainPanel);axis([1 size(data,2) 1 size(data,1)]);
```

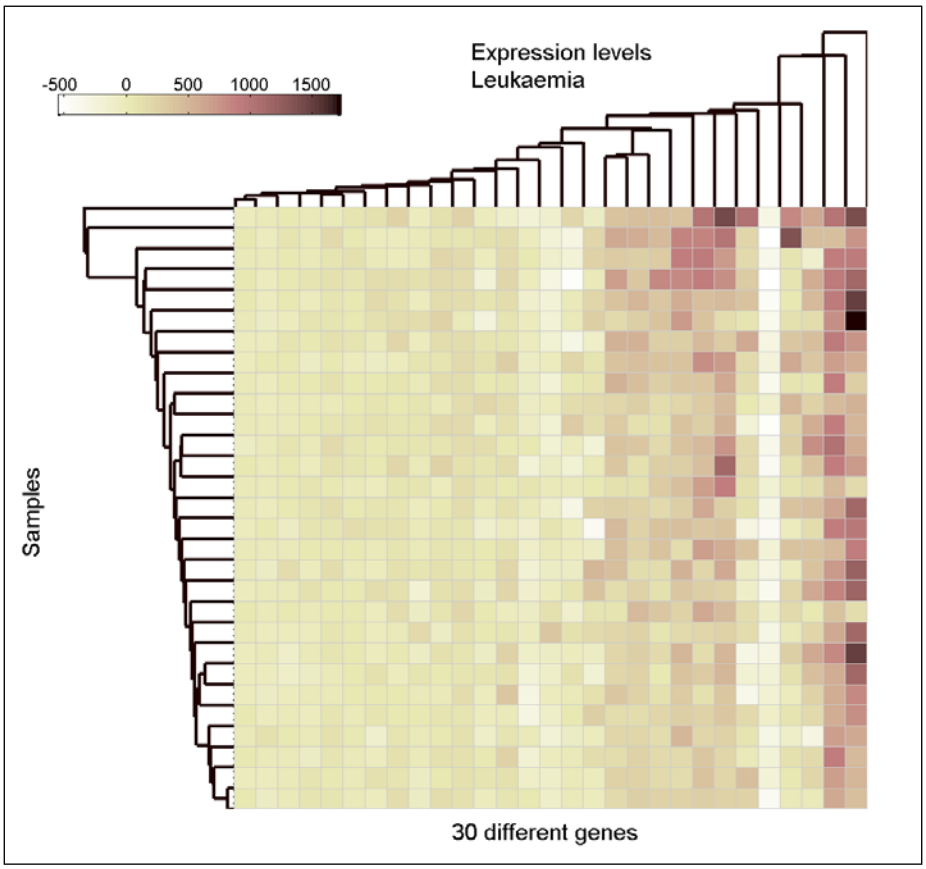
7. Add annotations:

```
axes(mainPanel); xlabel('30 different genes','Fontsize',14);
colorbar('Location','northoutside','Position',...
  [ 0.0584    0.8761    0.3082    0.0238]);
annotation('textbox',[.5 .87 .4 .1],'String',...
  {'Expression levels',...
  'Leukaemia'},'Linestyle','none',...
  'fontsize',14);
```

8. Apply this to show axis labels, even when axis is invisible:

```
set(leftPanel,'yaxislocation','left');
set(get(leftPanel,'ylabel'),'string','Samples',...
  'Fontsize',14);
set(findall(leftPanel, 'type', 'text'),...
  'visible', 'on');
```

The output is as follows:



How it works...

For the hierarchical clustering, the `linkage` function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The `linkage` function then links these newly formed clusters to each other and to other objects to create bigger clusters until all the objects in the original dataset are linked together in a hierarchical tree. By default, `pdist` uses the Euclidean metric. You can choose others or define your own.

Takeaways from this recipe:

- Use dendrograms to investigate the natural ordering andclusters in your data
- Use clustergrams to order your heat map so that trends are directly visible

See also

If you have MATLAB's Bioinformatics Toolbox™, the previous graphic can be created with command `clustergram`.

Look up **MATLAB help** on the `dendrogram`, `linkage`, `cluster`, and `pdist` commands.

Contour plots

A **contour** line is a curve along which the function has a constant value. MATLAB provides the function `contour` to construct these lines.

Getting ready

In this recipe, you will use an elevation dataset from MATLAB (part of MATLAB installation) to explore contour maps. Load the data:

```
load('topo.mat');
```

How to do it...

Perform the following steps:

1. Set up the figure and color map for its use:

```
figure('units','normalized','Position',...
  [.1063 .4083 .513 .4963]);
axis equal; box on
colormap(topomap1);
```

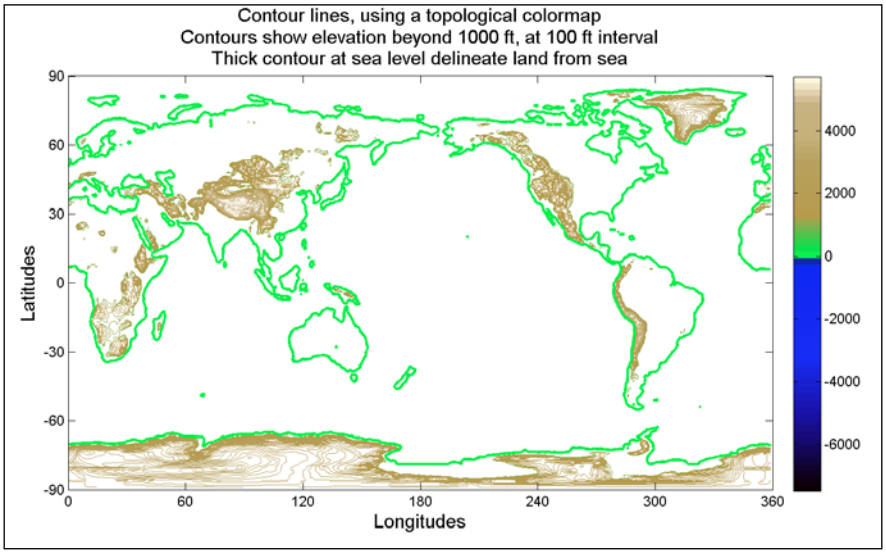
2. Construct the contour lines at specific values of elevation:

```
% contour lines above 1000 ft, at every 100 ft interval
contour(0:359,-89:90,topo,[1000:100:5800]); hold on;
set(gca,'clim',[min(topo(:)) max(topo(:))]);

% add a thick contour to 0 sea level to denote the land
contour(0:359,-89:90,topo,[0 0],'Linewidth',2);
```

3. Add a color bar and annotations:

```
colorbar;
set(gca,'XLim',[0 360],'YLim',[-90 90], ...
  'XTick',[0 60 120 180 240 300 360], ...
  'Ytick',[-90 -60 -30 0 30 60 90]);
xlabel('Longitudes','Fontsize',14);
ylabel('Latitudes','Fontsize',14);
title({'Contour lines, using a topological colormap',...
  ['Contours show elevation > 1000 ft, at 100 ft' ...
  'interval'],['Thick contour at sea level ',...
  'delineate land from sea']},'Fontsize',14);
```

The output is as follows:



How it works...

The `contour` command creates contour lines joining points on the grid with the same value. In this figure, you drew contour lines on the world map at elevation values greater than 1000 ft, at every 100 ft interval. The Rockies, Andes, Himalayas, Transatlantic Mountains, Sumatran /Javan ranges, Tien Shan, and Altai ranges among others, are visible.

You can specify styling for the contour lines, as you did for the contour at zero elevation, to effectively highlight the outline of land mass separated from sea.

Note that in this case you used a preconstructed nonuniform color scale that you loaded with the data. This ensured that all values under zero mapped to a shade of blue, and all values approximately greater than 1000 mapped to a shade of brown, in keeping with the standards of topographical data presentation.

There's more...

Notice how MATLAB used a nonuniform color map. This section discusses how to coerce MATLAB to use a nonuniform color map with continuous data values. Suppose you have a Gaussian function that denotes the spread of a cluster. The distance of an observation from the center of the cluster is inversely proportional to the confidence level of its cluster membership. The goal is to color the regions of the graph where the confidence levels are high with more discernable granularity than the regions with lower confidence levels.

Perform the following:

1. Define a color map with the handful of RGB values you will be using:

```
ColorMat = ...
  [255 255 255; 255 98 89; 255 151 99; 255 234 129; 227 249 149;...
  155  217 106; 76 202 130; 0 255 0;]/255;
```

2. Here is the definition for the Gaussian function (data generation step):

```
[X,Y] = meshgrid(linspace(0,1,50), linspace(0,1,50));
Z = exp(-(((X-.25).^2)./(2*.5^2) + ...
  ((Y-.5).^2)./(2*.5^2)));
```

3. Define the values at which you would like to change the contour color in the vector `x`:

```
x = [0, 0.4, 0.7, 0.9, 0.95, 0.98, .99, 1];
```

4. Now, construct the vector `y` as a fine grid from the lowest to highest values in your dataset (`0` and `1` in this example). For all values in `y` between two consecutive values in `x`, pad with the greater of the two `x` values:

```
clim = [0 1];
y = 0:0.005:1.0;
for k=1: length(x)-1,
  y(y>x(k) & y<= x(k+1)) = x(k+1);
end
```

5. Generate the color map corresponding to `y` by interpolating the color map corresponding to `x`:

```
cmap2 = [interp1(x(:),ColorMat(:,1),y(:)) ...
  interp1(x(:),ColorMat(:,2),y(:)) ...
  interp1(x(:),ColorMat(:,3),y(:))];
```

6. Create filled contour lineswith the `surf` command:

```
surf(X,Y,Z); box on; view(2);shading flat;
```

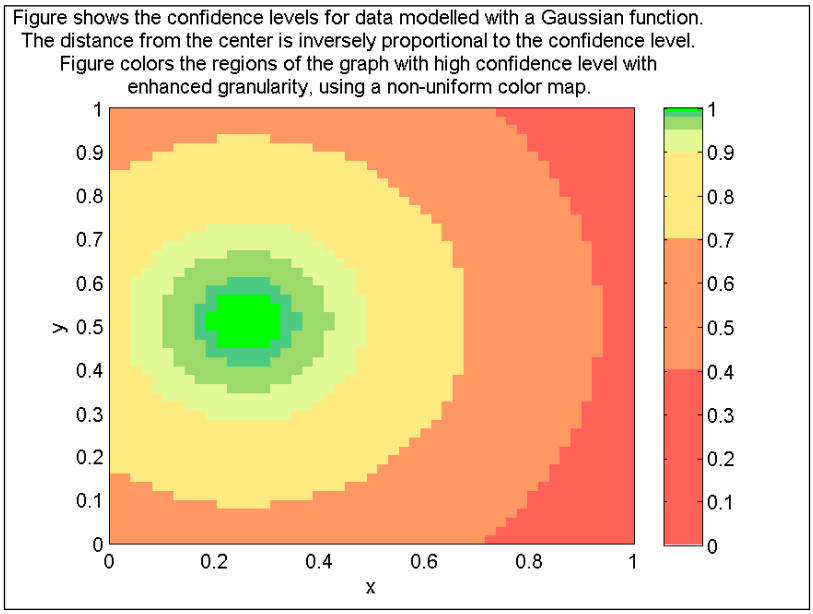
7. Set the color map to the newly defined matrix:

```
colormap(cmap2);
```

8. Manually set the color axis from `0` to `1`. Add a color bar:

```
caxis(clim);
colorbar;
```

The output is as follows:



Takeaways from this recipe:

- Use contour lines to investigate the patterns in your two-dimensional data
- Use nonuniform color scales to focus on data ranges of interest

See also

Look up **MATLAB help** on the `contour`, `clabel`, `peaks`, `contourc`, and `contourf` commands that we encountered in this recipe.

Gridding scattered data

If the two-dimensional data does not exist on a uniform grid, methods such as `surf` will not be applicable. MATLAB provides functionality to fit scatter points with a uniform grid and thereby facilitate the use of the standard techniques to 2D visualization on such data, as shown in this recipe.

Getting ready

Generate a scattered dataset in 2D space:

```
load griddataExample;
R = sqrt(x.^2 + y.^2) + eps; z = sin(R)./R;
```

How to do it...

Perform the following steps:

1. Define a uniform grid:

```
xx = linspace(min(x),max(x),30);
yy = linspace(min(y),max(y),30);
[X,Y] = meshgrid(xx,yy);
```

2. Fit a surface to the scattered data using MATLAB command `griddata`:

```
Z_griddata = griddata(x,y,z,X,Y);
```

3. An alternate method is using MATLAB command `triscatteredinterp`.

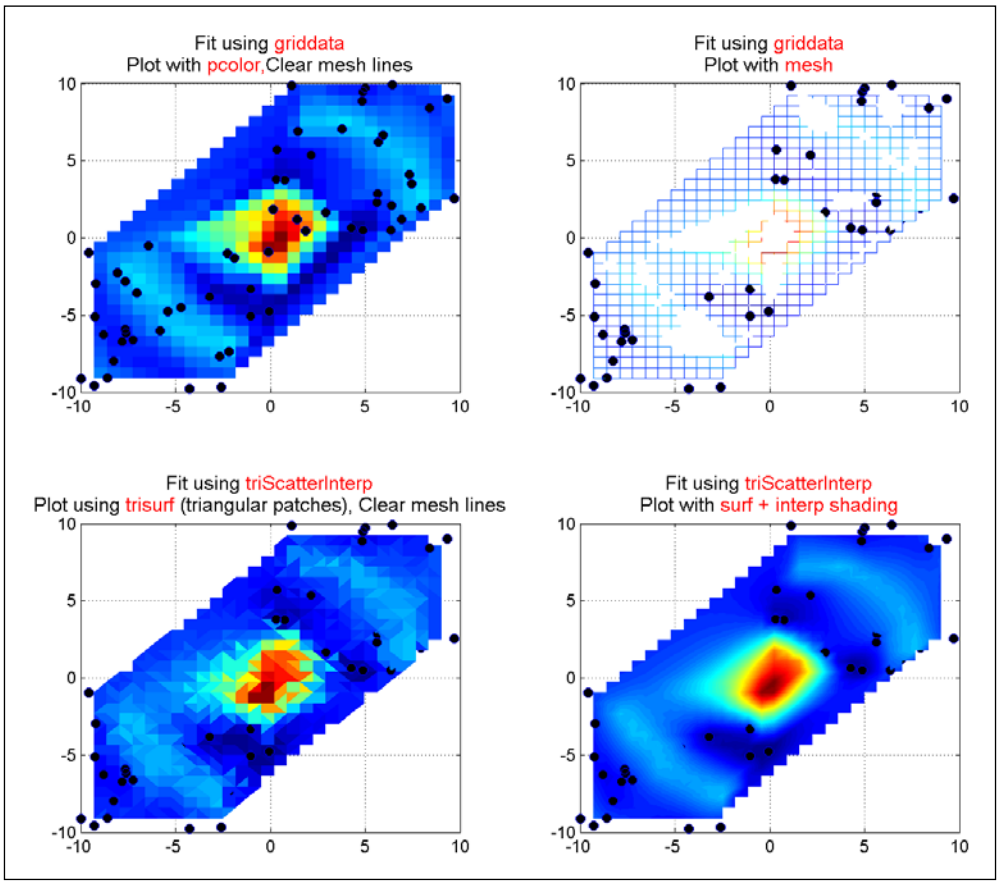
```
triScatterInterp_F = TriScatteredInterp(x,y,z,'natural');
Z_triScatterInterp = triScatterInterp_F(X,Y);
```

4. Visualize using several techniques as follows:

```
%% set up the figure
figure('units','normalized','Position',...
  [.312 .1463 .488 .712]);

%% plot 1 = griddata, with pcolor, no grid lines
subplot(2,2,1);
```

```
h=pcolor(X,Y,Z_griddata);set(h,'edgecolor','none');  hold on;
title({'Fit using \color{red}griddata',...
  '\color{black}Plot with \color{red}pcolor',...
  'color{black}Clear grid lines'},'Fontsize',12);
plot(x,y,'o','markerfacecolor',[0 0 0]);
box on; grid on;axis([-10 10 -10 10]);

%% plot 2 - griddata, mesh
subplot(2,2,2);
mesh(X,Y,Z_triScatterInterp); hold on;
h=mesh(X,Y,Z_griddata);view(2);
set(h,'edgecolor','none');  hold on;
title({'Fit using griddata',...
  'Plot with mesh'},'Fontsize',12);
plot(x,y,'o','markerfacecolor',[0 0 0]);
box on; grid on;axis([-10 10 -10 10]);

%% plot 3 - 'TriScatterInterp + triangular patches
subplot(2,2,3);
tri = delaunay(X,Y);
h= trisurf(tri,X,Y,Z_triScatterInterp); set(h,'edgecolor','none');
view(2); hold on;
title({'Fit using triScatterInterp',...
  'Plot using trisurf triangular patches)'},...
  'Fontsize',12);
plot(x,y,'o','markerfacecolor',[0 0 0]);
box on; grid on;axis([-10 10 -10 10]);

%% plot 4 - triScatterInterp, surf + interp shading
subplot(2,2,4); hold on;
surf(X,Y,Z_triScatterInterp); view(2); shading interp;
title({'Fit using triScatterInterp',...
  'Plot with surf + interp shading'},'Fontsize',12);
plot(x,y,'o','markerfacecolor',[0 0 0]);
box on; grid on;axis([-10 10 -10 10]);
```

The output is as follows:



How it works...

In this recipe, you investigated options for representing data from a nonuniform grid. Of the two available options to fit scatter data, the `TriScatteredInterp` is faster than the `griddata`. Both use the same triangulation to construct the results. However, the latter caches the interpolant and eliminates the need to recompute the triangulation each time you evaluate, change the interpolation method, or change the values at the sample locations. Another advantage to `TriScatterDataInterp` is also the **Natural Neighbor** interpolation method that you can use with it. This has an area-of-influence weighting associated with each sample point, which is C1 continuous (except at the sample locations) and performs well in both clustered and sparse data locations. `TriScatteredInterp` also uses the CGAL Delaunay triangulations (as does `DelaunayTri`), which are robust against numerical problems, faster, and more memory efficient.

The recipe also explored several MATLAB options for viewing the gridded data including `pcolor`, `surf`, `mesh`, and `trisurf` with different shading algorithms.

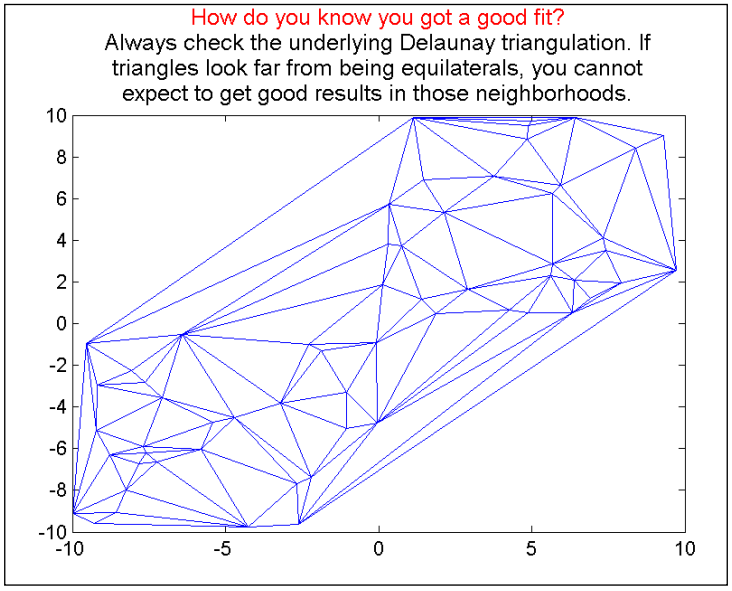
Note how you used `\color{red}griddata` to format part of the title string in a different color (`griddata` in this case was colored red).

There's more...

How do you know if the interpolated surface returned by MATLAB is a good one? Check the Delaunay triangulation. If triangulations look far from being equilateral, you cannot expect to get good results in those neighborhoods.

```
dt = DelaunayTri(x,y)
triplot(dt);
```

The output is as follows:



Takeaways from this recipe:

- Use `TriScatteredInterp` to interpolate large scattered datasets and infer patterns in your data.
- If the 2D scattered data interpolation produces unexpected results, check the Delaunay triangulation. If some of the triangles are very elongated, you cannot expect to get good results in that neighborhood.

See also

Look up **MATLAB help** on the `griddata`, `meshgrid`, `mesh`, `surf`, `pcolor`, `delaunay`, `DelaunayTri`, `triplot`, and `trisurf` commands that we encountered in this recipe.

Choropleth maps

Choropleth maps are a type of thematic maps in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map. The choropleth map provides an easy way to visualize how a measurement varies across a geographic area.

Getting ready

In this recipe, you will visualize a dataset with the rate of cancer incidence inthe US. The data was obtained from the CDC website and provided as part of this book. You will need the boundary information for the US, which was downloaded from the US government census website and also provided as part of this book.

Load a dataset:

```
[deathRatesstateNames] = xlsread('cancerByRegion.xlsx');
stateNames = {stateNames{2:end,1}};
load('USStateboundaries.mat');
```

How to do it...

Perform the following steps:

1. Layout the figure and set the color scale between the max and min data to present:

```
figure('units','normalized','Position',...
  [.11 .34 .48 .56]);

% set the color scale between the max and min data
climMat = [min(deathRates(:,1)) max(deathRates(:,1))];
set(gca,'clim',climMat); hold on;
m = colormap;
```

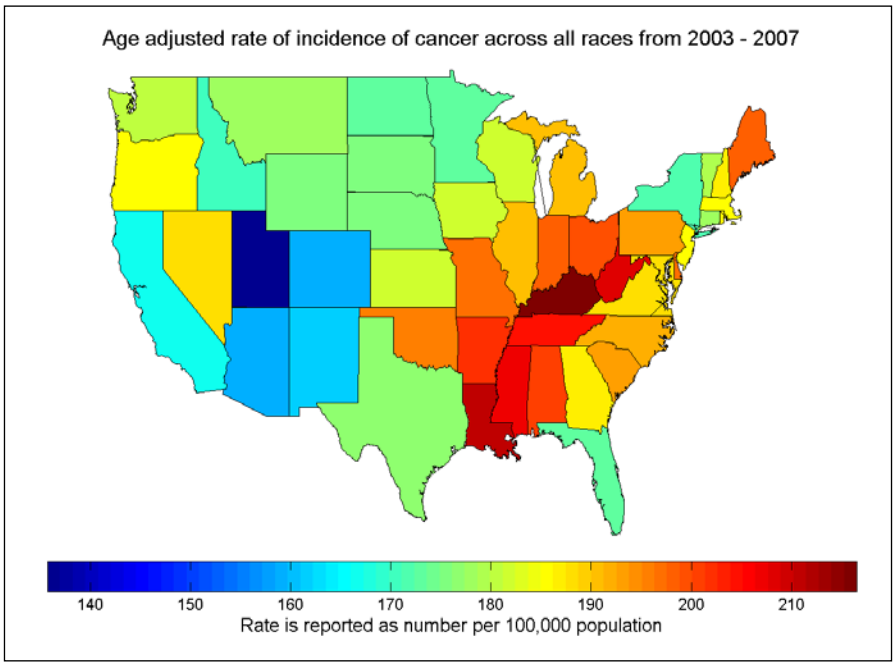
2. For each state, you will have to draw a filled polygon. The coordinates of the polygon should correspond to the latitude longitude boundary for the state. The color of the polygon should correspond to the data value to show.

```
% for each state,
for i = 1:49

% extract the data value for this state
dataPoint = deathRates(find(strcmp(states(i).Name, ...
  stateNames)),1);

% extract the color index into the color map that
% corresponds to this data value
index = fix((dataPoint-climMat(1))/...
  (climMat(2)-climMat(1))*63)+1;

% draw a filled polygon (fill color corresponding to
% colormap value at aforementioned index position
% Need to delete NaN values for fully connected polygon
fill(states(i).Lon(~isnan(states(i).Lon)),...
  states(i).Lat(~isnan(states(i).Lat)),m(index,:));
end
```

3. Add annotations and a colorbar:

```
title({'Age adjusted rate of incidence of cancer '...
  'across all races from 2003 - 2007'},'Fontsize',14);

% dont want to see axis
set(gca,'visible','off');

% but want to see title on invisble axis
set(findall(gca, 'type', 'text'), 'visible', 'on');

% add a color bar
h=colorbar('Location','Southoutside');
xlabel(h,['Rate is reported as number per 100,000 '...
  'population'],'Fontsize',13);
```

The output is as follows:



How it works...

The screenshotshows that the incidence of cancer is higher in the mid-eastern block of the US.

The recipe can be broken into two essential steps (which you accomplish for each state in step 2). The first is the use of polygons to plot the boundary of the state (you used MATLAB function `fill` to achieve this); the second is to map a data value to a color from the color map, so that polygon could be colored appropriately. The formula to map a data value is given by:

```
index = fix((dataPoint-climMat(1))/...
  (climMat(2)-climMat(1))*63)+1;
```

Here `63` corresponds to the number of color levels defined in your color map minus `1`. The `63` comes from the fact that the default color map type `jet` has 64 levels. `climMat` provides the two extremes of the data values that map to your current color map (you linked the data values to the color map by setting the `clim` property in step 1).

Takeaways from this recipe:

- Use choropleth maps to convey statistical data from a geographic region, to highlight patterns, variations, and trends across the region

See also

Look up **MATLAB help** on the `fill`, `clim`, `surf`, and `shading` commands that we encountered in this recipe.

Thematic maps with symbols

The **thematic maps with symbols** technique uses symbols of different sizes to represent data associated with different areas or locations within the map. For example, in this recipe, you will use arrows to depict the gradient of the elevation for the Himalayan region, overlaid on a topographic map of the region.

Getting ready

In this recipe, you will use the same elevation data you used in *Chapter 2*, *Diving into One Dimensional Data Displays*. You will focus on the Himalayan region. Load the data:

```
load topo
```

How to do it...

Perform the following steps:

1. Set up the figure as follows:

```
% layout the figure
figure('units','normalized','position',...
  [.36 .42 .38 .48]);

%% define grids
xx = [0:180 -179:-1];
yy = -89:90;
[XX, YY] = meshgrid(xx,yy);

%% calculate the gradient
[FX,FY] = gradient(topo);
```

2. Place the background image:

```
surf(xx,yy,topo);shading interp; view(2);
% Use a special color map that came with the dataset
colormap(topomap1);

% Make it semi-transparent to be able to see the additional
% information you will be overlaying
alpha(.5);
hold on;
```

3. Place the gradient data with `quiver`, with a scaling factor greater than one:

```
quiver(XX,YY,FX,FY,1.7,'Color',[0 0 0],'linewidth',1.5);
```

4. Focus on the region of interest:

```
xlim([65 110]); ylim([18 44]);
```

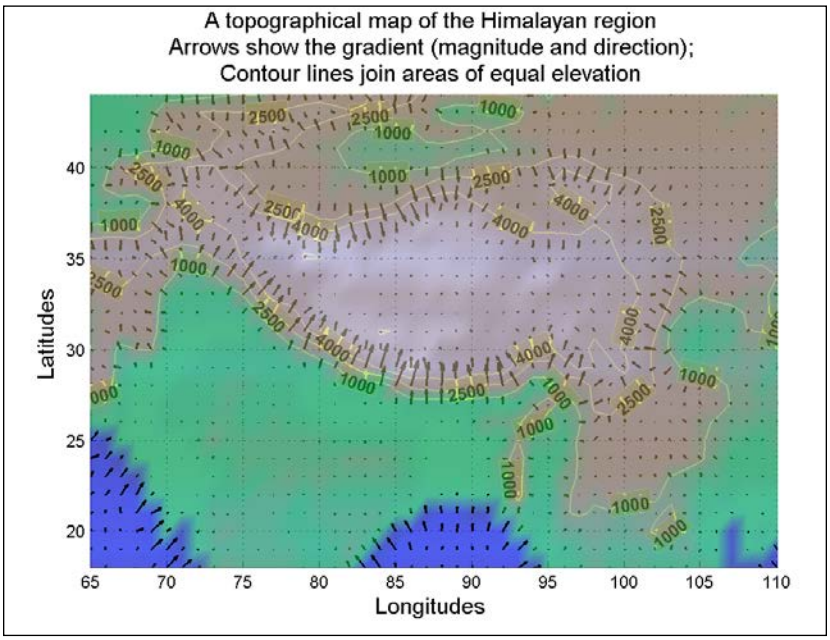
5. Add contour lines joining area of equal interval (beyond 1,000ft, every 1,500ft):

```
% Customize contour line color and width
[C, h] = contour(xx,yy,topo,[1000:1500:max(topo(:))],...
  'Color',[.8 .8 .8],'linewidth',1);

% Add text labels to contour lines with custom formatting
text_handle = clabel(C,h);
set(text_handle,'BackgroundColor',[1 1 .6],...
  'Edgecolor',[.7 .7 .7],'fontweight','bold')
```

6. Add annotations for overall graph:

```
title({'A topographical map of the Himalayan region',...
  ['Arrows show the gradient (magnitude and'...
  'direction);'],...
  'Contour lines join areas of equal elevation'},...
  'Fontsize',14);
xlabel('Longitudes','Fontsize',14);
ylabel('Latitudes','Fontsize',14);
```

The output is as follows:



How it works...

The graphic uses arrows to show the gradient information on the topographic map. You can see the steep rise of the mountains in the Himalayan region.

You altered the transparency of the image overlay in step 2 with the command `alpha`, because the **quiver lines** were being over-plotted. Look at *Chapter 4*, *Customizing Elements of MATLAB Graphics—Advanced*, for more on transparency.

In step 3, you used the `quiver` command to present the gradient information. The length of the arrow represents the magnitude and the direction represents the direction of the gradient value. You used a scaling factor of `1.7` on the gradient values to stretch the arrows such that they are more pronounced and visible.

In step 5, you added contour lines with custom formatting, along with labels with custom formatting on them.

Takeaways from this recipe:

- Use thematic maps along with symbols to convey data from a geographic region highlighting patterns, variations, and trends across the region

See also

Look up **MATLAB help** on the `surf`, `quiver`, `contour`, and `alpha` commands that we encountered in this recipe.

Flow maps

Flow maps are a mix of maps and flow charts depicting the movement of objects from one location, indicating information such as what is flowing, the direction of the flow, source and destination, how much is flowing, and so on.

Getting ready

In this recipe, you will show the itinerary of a travel plan to tour Italy. Load the data:

```
load romanHoliday
```

Note that the Google developer API was used to obtain the map at runtime. So you will need an Internet connection for this recipe to successfully execute.

How to do it...

Perform the following steps:

1. Get the map using a Google Map static image API:

```
%% Parameter Definition
height = 640; width = 640;

% construct the query string with all the lat/lonvalues
pos = [];
for i = 1:length(Lats)-1
  pos= [posnum2str(Lats(i)) ',' ...
  num2str(Lons(i)) '&markers='];
end
pos = ['http://maps.google.com/staticmap?markers=' ...
  posnum2str(Lats(end)) ',' num2str(Lons(end)) ...
  '&size=' num2str(width) 'x' num2str(height) ...
  '&scale=2'];

% retrieve as image from google maps
[I map]=imread(pos,'gif');
RGB=ind2rgb(I,map);
```

2. Layout the figure and plot the map in a main axes panel:

```
figure('units','normalized','position',...
  [.09 .09 .34 .81]);
mapPanel = axes('position',[.11 .36 .78 .57]);
image(RGB);hold on;
```

3. Set the `x` and `y` tick labeling and limits for the map panel:

```
axes(mapPanel);
set(gca,'XTickLabel',[],'YTickLabel',[]);
xlim([170 466 ]); ylim([110 528]);
```

4. Declare the additional flow info panel and set its tick labels, axis limits:

```
metaPanel = axes('position',[.11 .11 .78 .25]);
ylim([0 12]);xlim([170 466 ]);
set(metaPanel,'ytick',1:11,'yticklabel',...
  {'Day 1','Day 2','Day 3','Day4','Day 5','Day 6',...
  'Day 7','Day8','Day9','Day10','Day 11'});
[bl I] = sort(X);
set(metaPanel,'xtick',X(I),'xticklabel',stops(I));
rotateXLabels(metaPanel,90);
grid on;hold on;
```

5. Plot the path showing length of stay at each stop and method of transportation between stops:

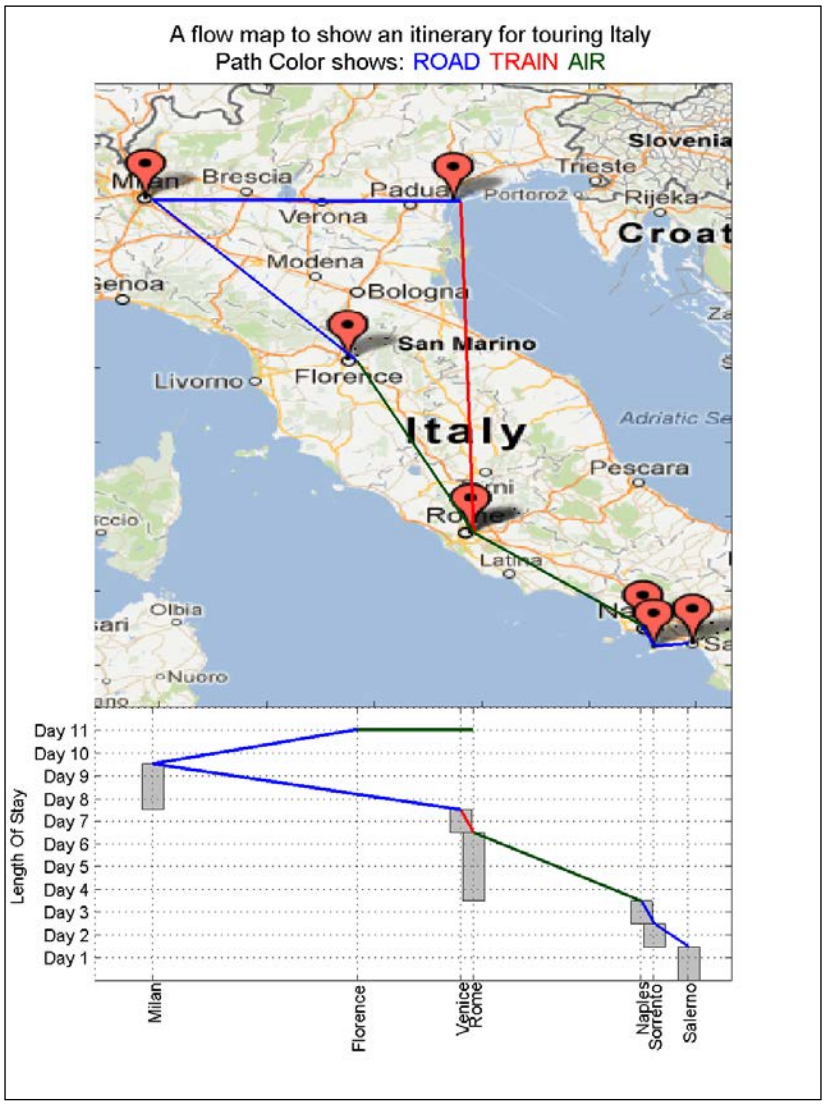
```
for i = 1:7
  j = i+1;
% leave out of Rome
  if i==7, j = 4;
end

% Draw lines joining stops using color to depict what
% transportation was used to travel between locations
axes(mapPanel);
if strcmp( meansOfTransportation{i},'Road')
  c = [0 0 1];
  line([X(i) X(j)],[Y(i) Y(j)],...
    'Color',[0 0 1],'linewidth',2);
elseif strcmp( meansOfTransportation{i},'Air')
  c = [0 .3 0];
  line([X(i) X(j)],[Y(i) Y(j)],...
    'Color',[0 .3 0],'linewidth',2);
```

```
elseif strcmp( meansOfTransportation{i},'Train')
  c = [1 0 0];
  line([X(i) X(j)],[Y(i) Y(j)],...
    'Color',[1 0 0],'linewidth',2);
end

% add lines showing length of stay to the meta data map
axes(metaPanel);
if i==1
  fill([X(i)-5 X(i)+5 X(i)+5 X(i)-5 X(i)-5],...
    [0 0 sum(daysSpent(1:i)) sum(daysSpent(1:i)) 0],...
    [.5 .5 .5]);alpha(.5);
  line([X(i) X(j)],[sum(daysSpent(1)) ...
    sum(daysSpent(1:2))],'Color',c,'linewidth',2);
else
  if j < i
    line([X(i) X(j)],[sum(daysSpent(1:i)) ...
      sum(daysSpent(1:i))],'Color',c,'linewidth',2);
  else
    line([X(i) X(j)],[sum(daysSpent(1:i)) ...
      sum(daysSpent(1:j))],'Color',c,'linewidth',2);
    fill([X(i)-5 X(i)+5 X(i)+5 X(i)-5 X(i)-5],...
      [sum(daysSpent(1:i-1)) sum(daysSpent(1:i-1)) ...
      sum(daysSpent(1:i)) sum(daysSpent(1:i)) ...
      sum(daysSpent(1:i-1))],[.5 .5 .5]);alpha(.5);
    end
  end
end
```

6. Add annotations:

```
axes(metaPanel);
ylabel('Length Of Stay'); box on;
axes(mapPanel);
title({'Vacation Plan for Italy',...
  ['Path Color shows: \color{blue}ROAD '...
  '\color{red}TRAIN \color[rgb]{0 .3 0}AIR']},...
  'Fontsize',14);
```

The output is as follows:



How it works...

This recipe shows how data flow information is demonstrated on a map.

The recipe lays out the map and plots the path of the travel itinerary. Color is used to depict the means of transportation. A shorthand legend is used including color coded words in the title of the figure.

The days spent panel below ties the location information to time. Notice how you used non-equal grid lines by setting xticks at the X locations of the image (step 4).

The Google developer API was used with a set of marker latitude and longitude values, and the size of the image as inputs. Google does not allow querying the exact boundaries using this static image retrieval API, so the image coordinates for the latitude and longitude values for the various locations was manually evaluated. References for the API are available in the source code for this recipe.

Takeaways from this recipe:

- Use flow maps to convey a mix of data and flow behavior

See also

Look up **MATLAB help** on the `line` and `alpha` commands that we encountered in this recipe.

4

Customizing Elements of MATLAB Graphics—Advanced

In this chapter, we will cover:

- Transparency
- Lighting
- View Control
- Interaction between light, transparency, and view

Introduction

In *Chapter 1*, *Customizing Elements of MATLAB Graphics—the Basics*, you learned about a set of figure and axes properties that enable powerful customization of MATLAB graphs, applicable across all data dimensions. In this chapter, you will learn about programmability to affect **transparency**, **lighting,** and **view** of graphs and how they can be effectively used for visualization.

Transparency

You have used color in the recipes of the earlier chapters to encode information. An analogous property of graphics objects is their face transparency. Both **color** and **transparency** can be used to encode information. You have used transparency in the recipes of the earlier chapters to reveal elements hidden from the exposed view. In this recipe, you will learn to use transparency to also encode information.

Getting ready

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

You will use a section of galactic survey data recorded in the radio frequency 327 MHz by the Westerbork Synthesis Radio Telescope (WSRT) in the Netherlands over several years. This region covers several pulsar locations which will be overlaid on the primary dataset. Load the data:

```
load W6
```

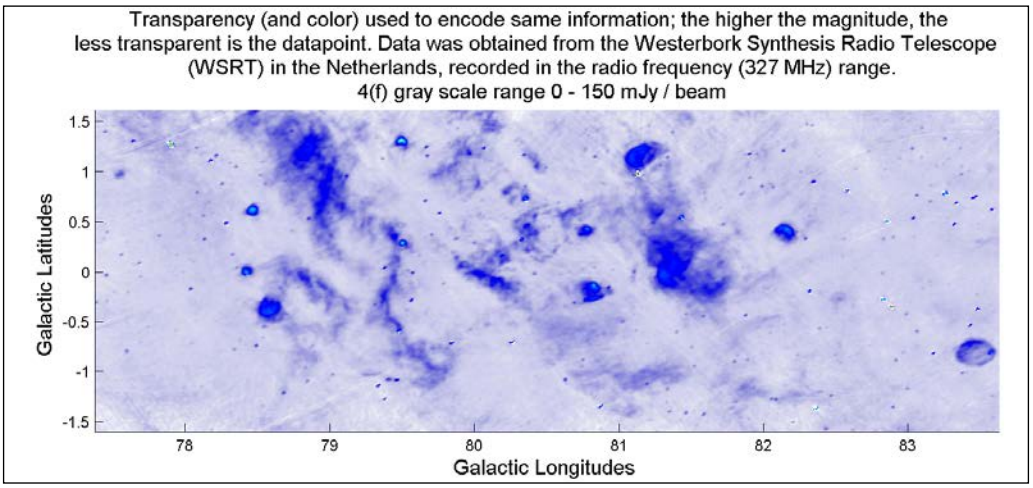
How to do it...

Perform the following steps:

1. Plot the data using the `surf` command. Use the same data as the source of information for both color and transparency for each grid face:

```
figure('units','normalized','position',[.2 .5 .55 .4]);

% define grids
xx=[84-3/8:-(((84-3/8)-(77+3/8))...
  /(size(W6,2)-1)):77+3/8];
yy=[-1.5-.1:(abs((-1.5-.1)-(1.5+.1))/...
  (size(W6,1)-1)):1.5+.1];

% surf the data. Note both data and alphadata set to W6
surf(xx,yy,W6,'alphadata',W6,'facealpha','interp'); view(2);
shading interp; axis tight;

% set the alpha limits
alim([-0.05 .2]);

% add annotations for overall graph
title({['Transparency (&color) used to encode same'...
  'information. Data recorded in radio frequency'],...
  [' (327 MHz) by the Westerbork Synthesis Radio' ...
  'Telescope (WSRT) in the Netherlands'],...
  ['4(f) gray scale range 0 - 150 mJy /'...
  'beam']},'Fontsize',14);
xlabel('Galactic Longitudes','Fontsize',14);
ylabel('Galactic Latitudes','Fontsize',14);
```

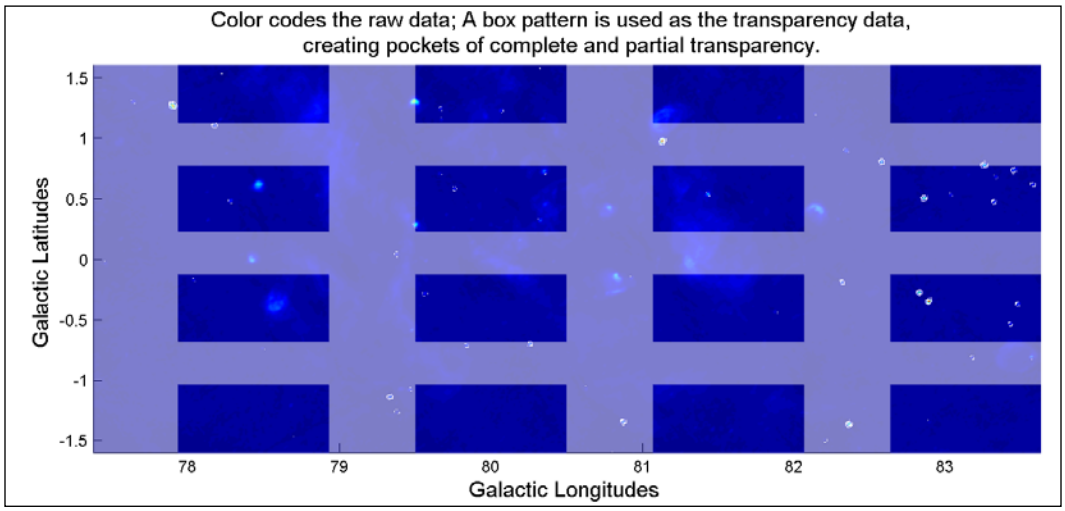
The output is as follows:



2. Next, plot the data again. This time, use different data as the source of information for the color and the transparency respectively:

```
% Plot the original data
h=surf(xx,yy,W6); view(2); shading interp; axis tight;

% create a box mask to act as the transparency data
tt = .5*ones(size(W6));
tt1 = repmat([.5*ones(1,140) 0*ones(1,80) ...
  0.5*ones(1,140) 0*ones(1,80) 0.5*ones(1,140)...
  0*ones(1,80) 0.5*ones(1,140) 0*ones(1,79) ],365,1);
tt2 = repmat([0.5*ones(1,65) 0*ones(1,40) ...
  0.5*ones(1,63) 0*ones(1,40) 0.5*ones(1,62) ...
  0*ones(1,40) 0.5*ones(1,55)]',1,879);
tt = .5*(double(tt1&tt2)) + .5;

% set the alpha data to this mask
set(h,'alphadata',tt,'facealpha','interp');

% set the alpha limits
alim([0 1]);

% add annotations for overall graph
title({['Color for raw data; Pattern for '...
  'transparency']},'Fontsize',14);
xlabel('Galactic Longitudes','Fontsize',14);
ylabel('Galactic Latitudes','Fontsize',14);
```

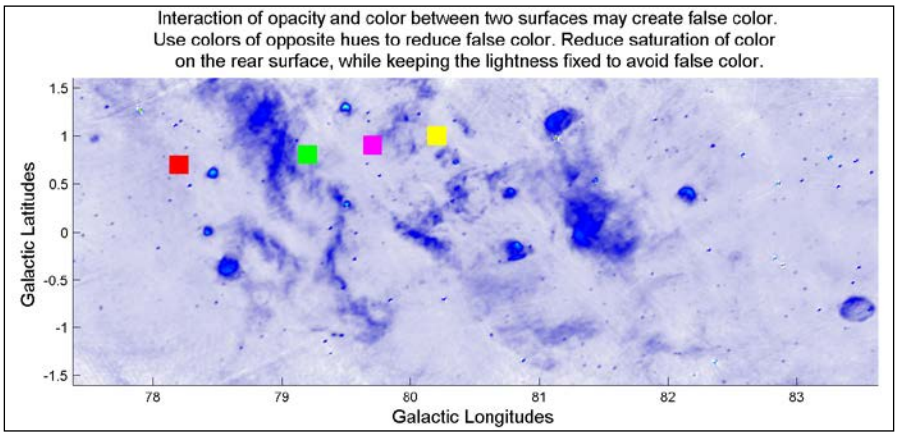
The output is as follows:



3. Add markers to the original figure to indicate positions of known pulsars:

```
surf(xx,yy,W6,'alphadata',W6,'facealpha','interp'); view(2);
hold on; shading interp; axis tight;
alim([-0.05 .2]);

% add markers to indicate location of pulsars
plot(pulsars(2).lon,pulsars(2).lat,'s',...
  'MarkerEdgeColor','none','Markersize',18,
  'Markerfacecolor',[1 1 0]);...
plot(pulsars(2).lon-.5,pulsars(2).lat-.1,'s',...
  'MarkerEdgeColor','none','Markersize',18,
  'Markerfacecolor',[1 0 1]);...
plot(pulsars(2).lon-1,pulsars(2).lat-.2,'s',...
  'MarkerEdgeColor','none','Markersize',18,
  'Markerfacecolor',[0 1 0]);...
plot(pulsars(2).lon-2,pulsars(2).lat-.3,'s',...
'MarkerEdgeColor','none','Markersize',18,
'Markerfacecolor',[1 0 0]);...

% add annotations for overall graph
title({['Interaction of opacity and color between '...
  'two surfaces may create false color.']},...
  'Fontsize',14);
xlabel('Galactic Longitudes','Fontsize',14);
ylabel('Galactic Latitudes','Fontsize',14);
```

The output is as follows:



How it works...

MATLAB allows the manipulation of transparency over a range of values from 0 to 1, and refers to them as **alpha** values. An alpha value of 0 means completely transparent (that is, invisible); an alpha value of 1 means completely opaque (that is, no transparency).

Alpha data defines the transparency information for the object. By default, objects have single-valued alpha data. However, you can define surface and image alpha data with a data array via the `AlphaData` property. An alpha data array has to be of the same size as the data points. It works very similar to how MATLAB determines the color to use from the data. When you create a **surface** or **patch** object, the MATLAB rendering software maps each element in the data array to a color in `colormap`. Similarly, each element in the alpha data array maps to a transparency value in `alphamap`. Now the `alphadatamapping` property determines whether the data is to be interpreted as indexes into `alphamap` (`direct`) or linearly mapped between limits prescribed by `alim` (`scaled`). You can select the face and edge rendering you want to use via the `facealpha` and `edgealpha` properties. `Flat` uses one transparency value per face, while `interpolated` performs bilinear interpolation of the values at each vertex. For patch objects, you will need to use the `FaceVertexAlphaData` property. By default, objects have scalar alpha data (`AlphaData` and `FaceVertexAlphaData`) set to the value 1.

In previous recipes, you altered the transparency values of the surface layer to either reduce the color saturation in the graph or to reveal the over-plotted data underneath the surface (*Chapter 1*, *Calendar Heat Map* and *Chapter 2*, *Thematic map with symbols*). In this recipe, you used transparency to code information.

In step 1, you defined `alphadata` and `Cdata` with the same matrix of data. The result was that the high values of data retained its color. The low values of data appeared faint or washed out (highly transparent). In step 2, you defined `CData` with the data matrix. For `alphadata`, you used a mask with pockets of high and low transparency values. Note the use of `alim` in both cases. This command helps to define the extreme data values to which the two extreme values of `alphamap` should correspond. In step 1, you arbitrarily set the higher `alim` value to a value much lower than the maximum value of the data matrix. This value was chosen because it corresponds to the physical value of the areas with the interesting features in the data. You were making sure that these areas did not get their color bleached out by the transparency.

Note that when overlaying transparent surfaces together, the color and degree of opacity of both surfaces will interact and may create what is referred to as **false color**. Use colors with opposite hues for semi-transparent layering to avoid false color. Reduce saturation of color on the rear surface, while keeping the lightness fixed to avoid false color.

Takeaways from this recipe:

- Use transparency to reveal hidden or occluded features of your data
- Use transparency as an additional dimension of your graphic for encoding information
- Use colors with opposite hues for semi-transparent layering to avoid false color. Reduce saturation of color on the rear surface, while keeping the lightness fixed to avoid false color

See also

Look up **MATLAB help** on the `alpha`, `alphamap`, and `alim` commands that we encountered in this recipe. Look up MATLAB help on the `AlphaData`, `AlphaDataMapping`, `FaceAlpha`, `EdgeAlpha`, `FaceVertexAlphaData`, `ALim`, `ALimMode`, and `Alphamap` transparency related figure properties.

Lighting

Lighting is a technique used for adding realism to a graphical scene. This is done by simulating the highlights and dark areas that occur on objects under natural lighting. The default light source is the **ambient** light, which is a directionless light that shines uniformly on all objects in the scene. You cannot see the light sources themselves. Objects created by functions such as `surf`, `mesh`, `pcolor`, `fill`, and `fill3` as well as the `surface` and `patch` functions can show the effect of light and have several properties which govern how they will be impacted by the light sources.

Getting ready

In this recipe, you will use `teapotdemo` supplied by MATLAB to investigate the effects of lighting on visualization. The demo has a console window that shows what commands were executed to create the various effects, if you made the changes using their GUI.

How to do it...

Perform the following steps:

1. Launch the demo:

   ```
   teapotdemo
   ```

   The default parameters of the demo produce the following view of the Newell teapot:

2. Change the lighting style to `flat`and material type to `dull`:

   ```
   lighting flat;
   material dull;
   ```

The effect on the Newell teapot in the demo window is as follows:

3. Demonstrate the effect of altering default values for `SpecularExponent`, `SpecularStrength`, `DiffuseStrength`, and `SpecularcolorRefectance`:

```
%Locate the handle to a patch object
h=findobj(gca,'type','patch');

% set the various lighting related properties
set(h,'FaceLighting','phong',...
  'FaceColor',[1 1 0],...
  'EdgeColor','none',...
  'SpecularExponent',12,...
  'DiffuseStrength',1,...
  'SpecularStrength',5, ...
  'SpecularColorReflectance',.5);
```

The effect on the Newell teapot in the demo window is as follows:

4. Demonstrate the effect of altering default values for `AmbientStrength` and`AmbientLightColor`:

```
set(h,'AmbientStrength', 0.5,...
  'DiffuseStrength', 0.6,...
  'SpecularStrength',0.9,...
  'SpecularExponent',10,...
  'SpecularColorReflectance',1);

% AmbientLight color is an axes object property
set(findobj(gca,'type','axes'),'AmbientLightColor', ...
  [1 0 1]);
```

The effect on the Newell teapot in the demo window is as follows:

How it works...

MATLAB allows you to create light sources with the command `light`. The MATLAB rendering software determines the effect of light sources on the target objects in your graphic. The demo creates two light sources. Execute `findobj(gca)` and then check the type of the objects to identify the handles to these two light objects. You can then query the object properties using the handles to the light sources to understand the settings for the light color, style, and position. `Color` determines the color of the light cast by the light object; `Style` defines the source at either infinity or local; and `Position` defines the direction for infinite light sources or the location for local light sources.

In step 2, you set the lighting algorithm and the property of the target material being viewed. The three lighting algorithms supported by MATLAB are `Flat`, `Gouraud`, and `Phong`. Faceted objects should be visualized with `Flat` lighting whereas curved surfaces should be visualized with `Gouraud` or `Phong` lighting. `Phong` takes the longest to render because it calculates the reflectance at the pixel level but produces the finest results. `Gouraud` calculates the colors at the vertices and then interpolates colors across the faces. The `material` command sets the surface reflectance properties of the surface and patch objects.

In step 3, you explicitly defined the reflectance properties of the target object. A higher specular strength enhances the reflectivity of the material. A higher diffuse strength makes the object bright (giving off more light), but the effect is not shiny as the surface is assumed to be rough. The higher the specular exponent, the more prominent is the glare from specular reflection. The color reflectance controls what additional colors get reflected (other than the true color of the object).

In step 4, you modified the color of the ambient light, which is a directionless light that shines uniformly on all objects in the scene. There are two properties that control ambient light —`AmbientLightColor` (an axes property that sets the color), and `AmbientStrength` (a property of target objects that determines the intensity of the ambient light on the particular object). Note that the true color is visible when there is a white light object present. Otherwise, the surface object color interacts with the color of the light shining on it, to produce the final color displayed.

There's more...

In this section you will look at two things: Effect of **vertex normals** and use of the **backface lighting**.

Effect of vertex normals

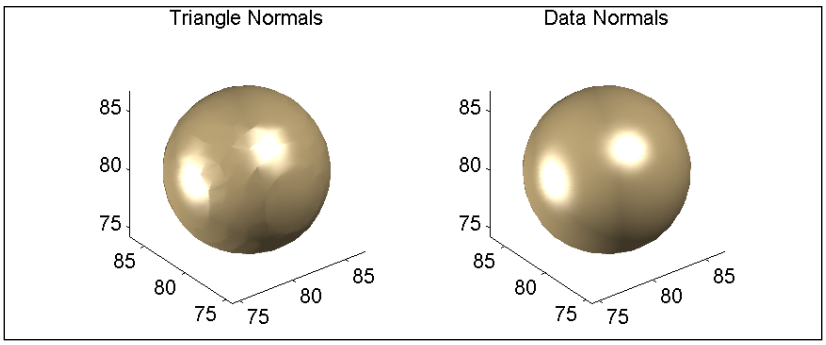
`VertexNormals` is a patch and surface property that contains normal vectors for each vertex of the object. MATLAB uses vertex normal vectors to perform lighting calculations. While MATLAB automatically generates this data, you can also specify your own vertex normals. The `NormalMode` property determines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

This example compares the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the `isonormals` function uses the data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

```
% generate some data
x_lim=1-.01; y_lim=1-.01; z_lim=1-.01; step=.1;
x=[-x_lim:step:x_lim]; y=[-y_lim:step:y_lim];
z=[-z_lim:step:z_lim];
[x y z]=meshgrid(x,y,z);r=sqrt(x.^2+y.^2+z.^2);
r_s=.01;
w=2*sqrt(r_s*(r-r_s));
w = interp3(w,3,'cubic');
% make the plots
subplot(1,2,1); p1 = patch(isosurface(w,.05),...
```

```
    'FaceColor', [.8 .7 .5],'EdgeColor','none');

% set the view
view(3); daspect([1,1,1]); axis tight

% adjust lighting
camlight; camlight(-80,-10); lighting phong;
title('Triangle Normals')

%make the plot with isonormal calculations
subplot(1,2,2); p2 = patch(isosurface(w,.05),...
 'FaceColor', [.8 .7 .5], 'EdgeColor','none');
isonormals(w,p2);

% set the view
view(3); daspect([1 1 1]); axis tight

% adjust lighting
camlight;  camlight(-80,-10); lighting phong;
title('Data Normals')
```

The output is as follows:

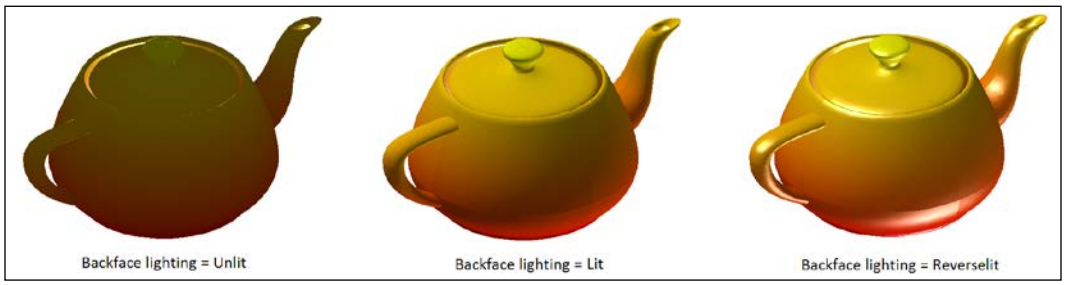


Use of the back face lighting option

Back face lighting is useful for showing the difference between internal and external faces. The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point away from the camera. See the effects of different values of `Backfacelighting` on the Newell teapot as follows:

```
teapotdemo;
h=findobj(gca,'type','patch');
set(h,'BackFaceLighting','unlit');
```

The output is as follows:



Takeaways from this recipe:

- Use lighting to add realism to your 3D data
- Use `Flat` lighting algorithm for faceted objects and `Gouraud` or `Phong` for curved surfaces
- Use the `isonormals` command to calculate the vertex normals based on the gradient of the data points for a finer finish to your 3D rendering

See also

Look up **MATLAB help** on the `material`, `light`, `isonormals`, `patch`, and `camlight` commands that we encountered in this recipe.

View control

The **view** is the vantage point from which you observe the data in your graphic. It has a high impact on how the graphic is perceived. Several factors guide how the view is defined such as the aspect ratio, whether or not you manipulate the object in terms of rotation or panning, and whether you choose to define a region of interest in terms of the axis limits or a zoom in. In this recipe, you will explore some of the options and practices for controlling the view.

Getting ready

You will use data generated by the function $w=2\sqrt{\frac{r_s}{r-r_s}}$. Load the data:

```
load viewCntrolDataSet
```

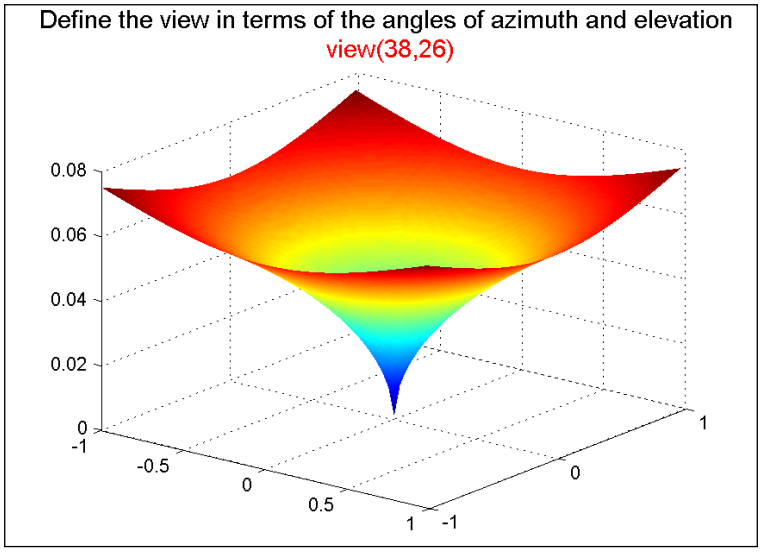
How to do it...

Perform the following steps:

1. Define the view in terms of the angles of azimuth and elevation:

   ```
   surf(x,y,w); shading interp; view(38,26);
   ```

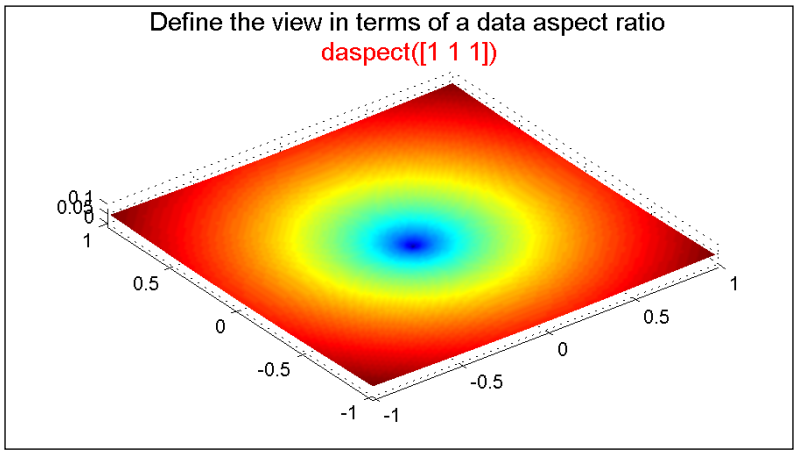
   The output is as follows:



2. Define the view by specifying the data aspect ratio:

   ```
   surf(x,y,w); shading interp; daspect([1 1 1]);
   ```

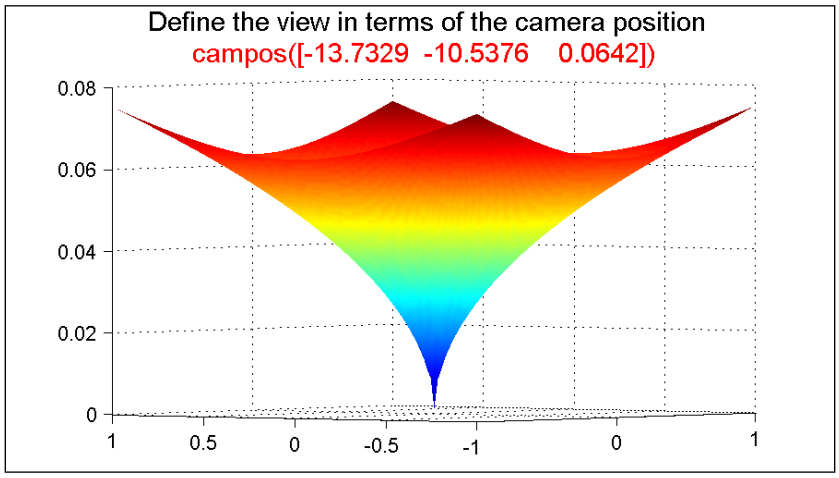
The output is as follows:



3. Define the view by specifying the camera position:

```
surf(x,y,w); shading interp;
campos([-13.7329  -10.5376    0.0642]);
```

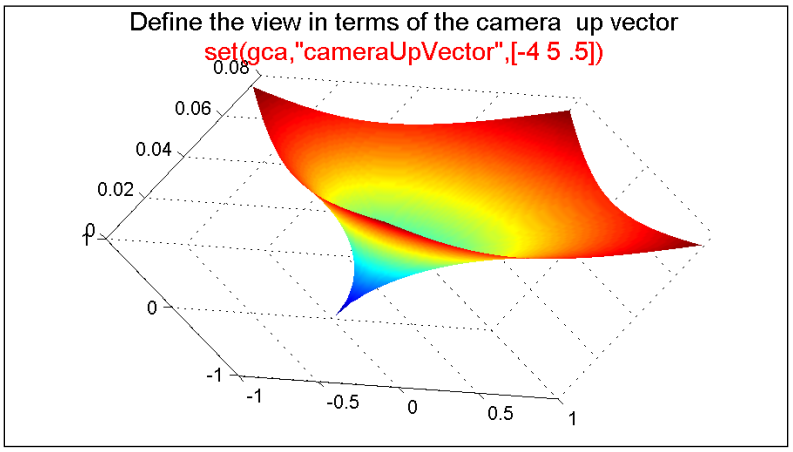
The output is as follows:



4. Define the view by specifying the camera up vector (redefine what is considered vertical):

```
surf(x,y,w); shading interp;
set(gca,'cameraUpVector',[-4 5 .5]);
```

The output is as follows:



How it works...

The **aspect ratio** of the graph can exaggerate or undermine a trend for those who do not read labels carefully. The same effect can also be caused by leaving out the zero, or by a deliberate choice for x, y, and z axis limits. MATLAB provides a set of powerful camera tools that you can also harness to create interesting perspectives for your data. The view is thus an important ingredient to the graphic design and one of the ways to influence the viewer's **perception** of your data.

In step 1, you specified the viewpoint in terms of the **azimuth** and **elevation**. You can also use a point in three-dimensional space with the `view` command to specify the viewpoint. The azimuth is the horizontal rotation about the z axis as measured in degrees from the negative y axis. Positive values indicate counterclockwise rotation of the viewpoint. The elevation is the vertical elevation of the viewpoint in degrees measured from the x – y plane. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object. In step 2, you specified the view in terms of the data aspect ratio, which is the relative scaling of the data units along the x, y, and z axes.

MATLAB also has the `axis` command which impacts the data aspect ratio. There are several useful settings such as `equal`, which sets the aspect ratio so that tick mark increments on the x, y, and z axes are equal in size (same as `daspect([1 1 1])`). You can also specify the exact x, y, and z coordinate limits to focus the graphic. Refer to the MATLAB help for more options with `axis`.

In step 3 and 4, you used camera tools to direct the view of the graph.

Takeaways from this recipe:

- Use a careful choice of view point or data aspect ratio for your graphic to influence viewer perception

See also

Look up **MATLAB help** on the `view`, `axis`, `daspect`, `campos`, and `cameraupvector` commands.

Interaction between light, transparency, and view

This recipe brings together the three aspects of visualization discussed in this chapter: light, transparency, and view.

Getting ready

In this recipe, you will explore the Klein bottle, which is a nonorientable surface in four-dimensional space. It is formed by attaching two Mobius strips along their common boundary. Klein bottles cannot be constructed without intersection in three-space. The code for constructing this is obtained from MATLAB's help documentation (see lines 11 – 29 in the source code for this recipe).

How to do it...

Perform the following steps:

1. Layout the axis:

```
axes('position',[-.02 .12 1 .8]);
```

2. Construct the two components of the Klein bottle (body and handle) using surface objects:

```
handleHndl=surf(x1,y1,z1,X); shading interp;
hold on;
bulbHndl=surf(x2,y2,z2,Y);shading interp;
```

3. Declare a colormap and add a color bar:

```
colormap(hsv);
colorbar('position',[0.9071 0.1238 0.0149 0.8000]);
```

4. Set some properties of the Klein bottle:

```
% set wireframe line color to gray
set(handleHndl,'EdgeColor',[.5 .5 .5]);
set(bulbHndl,'EdgeColor',[.5 .5 .5])

% make transparency proportional to y coordinate
set(handleHndl,'alphadata',y1,'facealpha','interp');
set(bulbHndl,'alphadata',y2,'facealpha','interp');
```

5. Fix the view:

```
% freeze aspect ratio properties to enable rotation of
% 3-D objects and overrides stretch-to-fill.
axis vis3d

% set the view
view(13,-18);

% do not show the axis
axis off

% zoom by a certain factor
zoom(1.4);
```

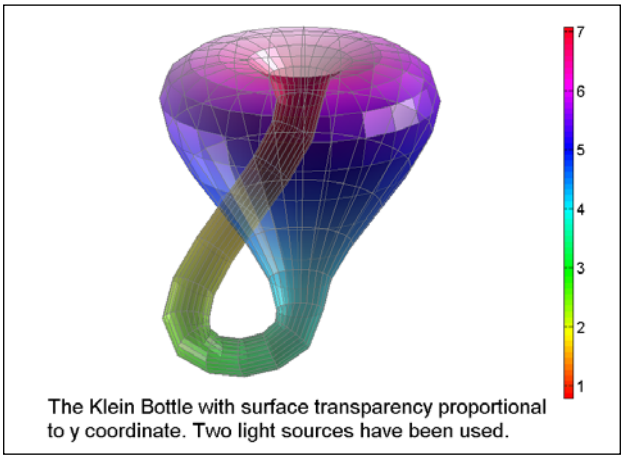
6. Add two light sources with their position defined in terms of data coordinates:

```
light('Position',[.5 .5 .5]);
light('Position',[-.67 -.1 -.7]);
```

7. Add annotations:

```
h=annotation('textbox','position',[.06 .04 .95 .1],...
  'String',{['The Klein Bottle with surface'...
  ' transparency proportional'],...
  ['to y coordinate. Two light sources have been'...
  'used.']},'linestyle','none', 'fontsize',14);
```

The output is as follows:

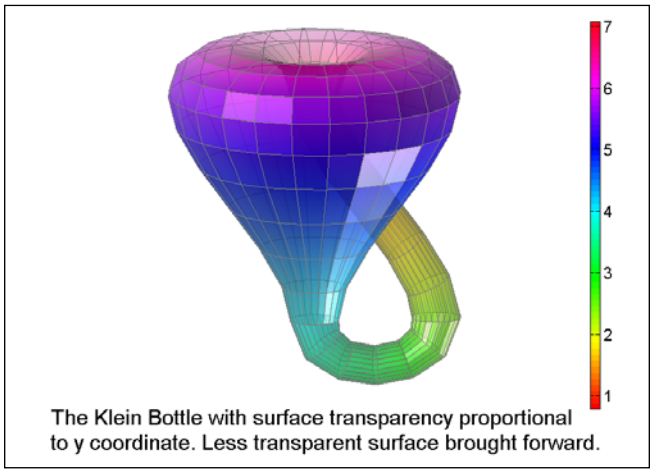


The Klein Bottle with surface transparency proportional to y coordinate. Two light sources have been used.

8. Rotate camera to bring less transparent surface closer to the viewer:

```
% delete previous annotation
delete(h);

% rotate camera to change the view point
campos([1.5017   14.2881    3.7572]);

% add new annotation
annotation('textbox','position',[.06 .04 .95 .1],...
  'String',{['Less transparent surface brought '...
  'forward.']},'linestyle','none', 'fontsize',14);
```

The output is as follows:



The Klein Bottle with surface transparency proportional to y coordinate. Less transparent surface brought forward.

How it works...

This recipe brings the interplay of lighting, transparency, and view point into focus. The Klein bottle is essentially a four-dimensional data with the three independent parameters x, y, z, and the value at the function surface. You used color to code the function contour. You used transparency to reveal the internal hidden structure in the data. You added lights to two locations to create a realistic rendering. You exercised the view control to create two different views of the dataset. Note that you used the command `zoom` with a factor parameter of 1.4 to zoom in.

Takeaways from this recipe:

- Use transparency to reveal hidden structures
- Use color to code continuous data
- Use lights to add realism
- Use view control to expose your graphic to maximal advantage

See also

Look up **MATLAB help** on the `view`, `axis`, `daspect`, `campos`, `cameraupvector`, and `zoom` commands.

5

Playing in the Big Leagues with Three-dimensional Data Displays

In this chapter, we will cover:

- 3D scatter plots
- Slice (cross sectional views)
- Isosurface, isonormals, and isocaps
- Stream slice
- Stream lines, ribbons, tubes
- Scalar and vector data with a combination of techniques
- Exploring with camera motion

Introduction

Three-dimensional data poses a set of unique challenges for visualization. Since visualization is necessarily happening on a two-dimensional plane, you have to use special techniques to explore 3D data space. Taking slices and visualizing one slice at a time, or using transparency for seeing a structure behind the outermost layer are some of the common techniques used for 3D data visualization.

A special case of 3D data is volumetric data. Volumetric data may be scalar or vector. **Scalar** data has only magnitude and is defined per point in the 3D grid. **Vector** data has both magnitude and direction. Direction is represented by components in the three axis directions. MATLAB supports the use of **isosurface**, **slice plane**, and **contour slice** for viewing scalar data. For vector data, MATLAB provides **stream lines** (**particles**, **ribbons**, and **tubes**) and **cone plots**. The recipes in this chapter employ a combination of techniques to visualize volumetric data.

3D scatter plots

This recipe explores the concept of **3D scatter plots**.

Getting ready

In this recipe, the data is contained in a 3D matrix representing the electron hopping probability computed at each point of a 3D lattice structure. Load the data:

```
load latticeExample
```

How to do it...

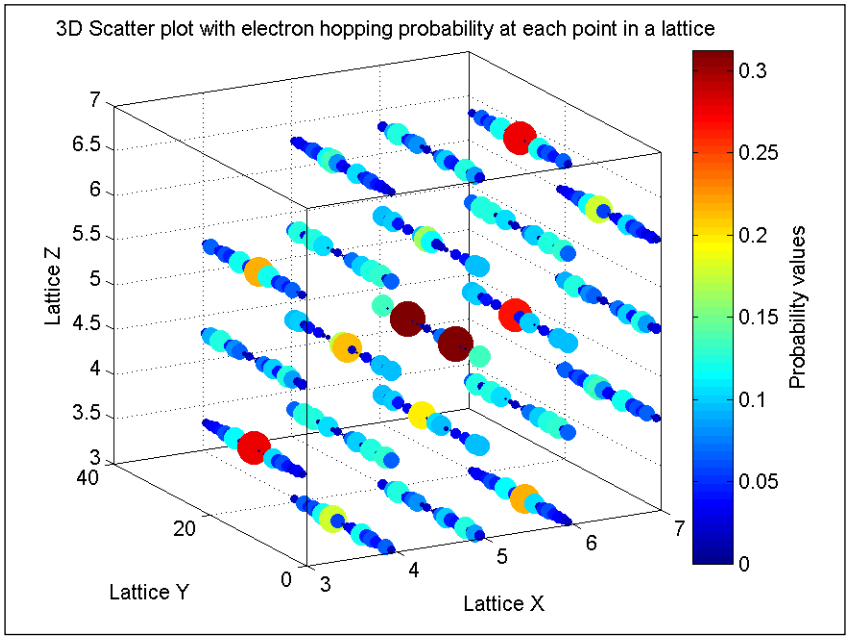
Perform the following steps:

1. Use 3D scatter plots to construct a basic scatter plot view of the data:

```
% unwrap the x,y and z matrices
xx = x(:);yy = y(:);zz = z(:);

% locate the non-zero points
a = find(T~=0);

% plot the non-zero points using a scatter plot; use the
% values of T to represent both color and size of symbols
scatter3(xx(a),yy(a),zz(a),1000*T(a),T(a),'filled');

% set the view
view(3);
campos([ -7.8874 -217.1200   13.7208]);

% add colorbar to read probability values
h=colorbar;set(get(h,'ylabel'),'String','Probability');

% add annotations
ylabel('Lattice Y');
xlabel('Lattice X');
```

```
zlabel('Lattice Z');
title('3D Scatter plot with probability values');
grid on; box on;
```

The output is as follows:



2. Next, generate a cloud at each lattice point to represent the interaction between these molecules. This is done with a Gaussian kernel as follows:

```
data = smooth3(TB_Eigen,'gaussian');

% unwrap the smoothed data
d = data(:);
```

3. Color will be used to represent the interaction magnitudes. Generate color values for each point by querying the color map:

```
idx = find(d~=0);
cmapH(:,1:3) = colormap;
cmin=min(5*d(idx));cmax=max(5*d(idx));
caxis([cmin cmax]);
Cidxs = fix((5*d(idx)-cmin)/(cmax-cmin)*length(cmapH))+1;
Cidxs(find(Cidxs>64))=64;
```

4. Transparency will also be used to represent the interaction magnitudes. Generate alpha values for each point by querying the alpha map:

```
% generate the alpha map (with 100 levels)
amapH = linspace(0,1,100);

% make max value most transparent by reversing the alphamap
amapH = amapH(end:-1:1);

amin=min(d(idx));amax=max(d(idx));
Aidxs = fix((d(idx)-amin)/(amax-amin)*length(amapH))+1;
Aidxs(find(Aidxs>100))=100;
```

5. Call `bubbleplot3(x,y,z,r,c,alpha)`, a three-dimensional bubbleplotter, for creating bubbles of radii `r`, at corresponding positions in `x`, `y`, `z`. The variable `c` is an RGB-triplet for color. Alpha data provides transparency values for each bubble.

```
figure('units','normalized','position',...
  [.24 .28 .41 .63]);
bubbleplot3(xx(idx),yy(idx),zz(idx),5*d(idx),...
  cmapH(Cidxs,:),amapH(Aidxs));
```

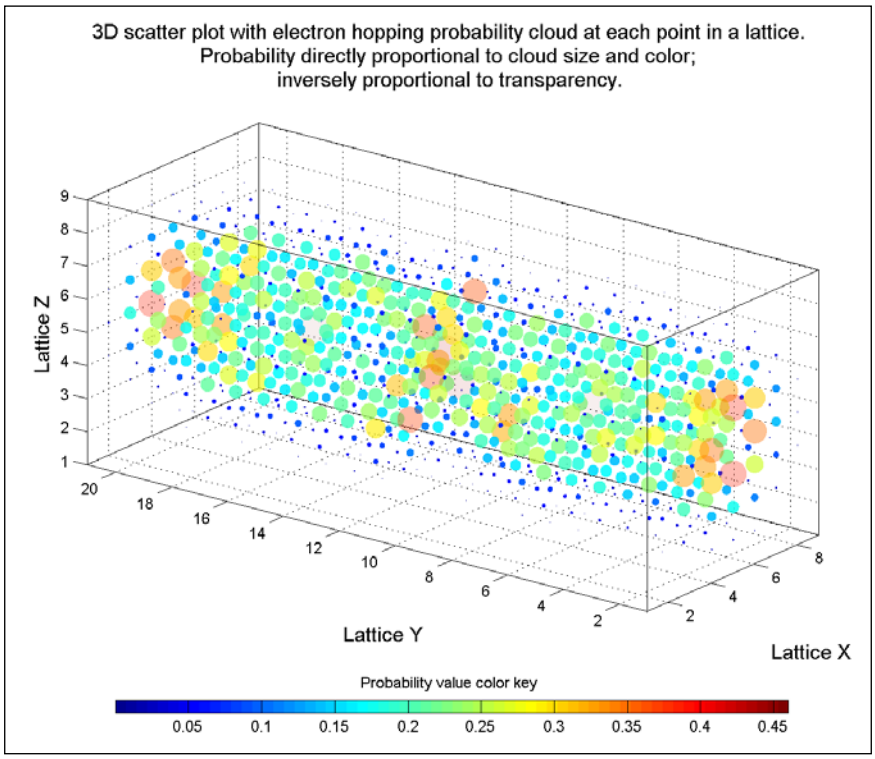
6. Set the view:

```
view(3);
axis([min(xx) max(xx) min(yy) max(yy) min(zz) max(zz)]);
box on;
campos([-80.6524  -54.7234   44.2951]);
zoom(1.2);
```

7. Add annotations:

```
ylabel('Lattice Y','Fontsize',14);
xlabel('Lattice X','Fontsize',14);
zlabel('Lattice Z','Fontsize',14);
title({['3D scatter plot with electron hopping '...
  'probability cloud at each point in a lattice.'],...
  ['Probability directly proportional to cloud size'...
  ' and color;'],['inversely proportional to'...
  ' transparency.'],''},'Fontsize',14);
h=colorbar('location','SouthOutside','position',...
  [ 0.1286    0.0552    0.7750    0.0172]);
set(get(h,'title'),'string',...
  'Probability value color key');
```

The output is as follows:



How it works...

`scatter3` is the 3D analog for ordinary scatter plot and is commonly used for displaying discrete data points in the three dimensions.

You used a custom function, `bubbleplot3`, adapted from a submission by Peter Bodin on MATLAB file exchange. This function allowed you to provide `x`, `y`, and `z` coordinates as well as the radius of the spheres to be drawn to represent each point, and the color and transparency values to use with each point. You used the probability data for both the color and size of the spheres. You used the inverse of the probability data to set the transparency values. Color and transparency values were both manually extracted from the color map and a custom alpha map in steps 3 and 4.

Takeaways from this recipe:

- Use 3D scatter plots to present scatter points in 3D.
- Additionally, you can use color, shape, size, and transparency of scatter points to code other dimensions of your data; however, recall from the first two chapters that these attributes are not high on the list of easily perceived attributes for visualization.

See also

Look up **MATLAB help** on the `bubbleplot3`, `smooth3`, `campos`, and `colorbar` commands.

Slice (cross-sectional views)

In this recipe, you will visualize cross sections or **slices** of a 3D dataset to understand its structure.

Getting ready

You will use a human brain magnetic resonance imaging (MRI) dataset that is included with MATLAB. Load the data:

```
load mri
```

How to do it...

Perform the following steps:

1. The MRI data, `D`, is stored as a 128-by-128-by-1-by-27 array. The third array dimension has no information and can be removed using the `squeeze` command:

   ```
   D = squeeze(D);
   ```

2. Create horizontal slices:

   ```
   % layout figure; use colormap that came with dataset
   figure('units','normalized','position',...
     [0.3016  0.3556  0.3438  0.5500],'Colormap',map);

   % choose axes position
   axes('position',[0.1300  0.2295  0.7750  0.8150]);

   % view horizontal slices
   ```

```
whichSlices = 3:5:27;
h=slice(1:128,1:128,1:27,double(D),[],[],whichSlices);
shading interp;

% set alphadata = data; set upper alim to a low value
% so that color is only bleached from the background image
for i =1:length(h)
  set(h(i),'alphadata',double(D(:,:,whichSlices(i))),...
  'facealpha','interp');alim([0 2]);
end

% set the view
zoom(1.2);campos([-706 -778  111]);axis off;
zlim([1 25]);

% annotate
annotation('textbox',[.05 .07 .9 .1],'String',...
  {['Horizontal \color{red} slice \color{black} views '...
  'from MRI dataset.']},'fontsize',14,'linestyle','none');
```

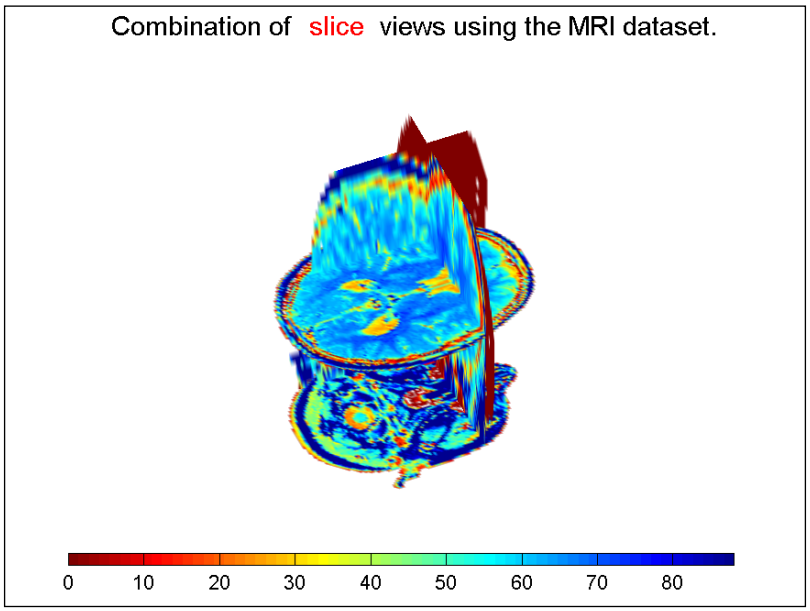
The output is as follows:

Horizontal slice views from the MRI dataset. Visual uses transparency to hide extreme low values that would otherwise showup in black

3. Explore with a combination of slices:

```
% pick colormap, invert it so low values get light colored
m=colormap('jet');m = m(end:-1:1,:);colormap(m);

% do your slices
h=slice(1:128,1:128,1:27,double(D),90,50,[1 15]);
shading interp; axis tight;

% set the transparency so low values are invisible
set(h(1),'alphadata',squeeze(double(D(90,:,:))),...
  'facealpha','interp'); alim([0 2])
set(h(2),'alphadata',squeeze(double(D(:,50,:))),...
  'facealpha','interp'); alim([0 2])
set(h(3),'alphadata',squeeze(double(D(:,:,1))),...
  'facealpha','interp'); alim([0 2])
set(h(4),'alphadata',squeeze(double(D(:,:,15))),...
  'facealpha','interp'); alim([0 2])

% set the view
daspect([128 128 27]);
zoom(1.3);
campos([-637 366 177]);
axis off;

% annotate
colorbar('location','southoutside',...
  'position',[.08 .07 .83 .02]);
annotation('textbox',[.24 .88 .9 .1],'String',...
  ['Combination of \color{red} slice \color{black}'...
  'views using MRI dataset. '],'fontsize',14,...
  'linestyle','none');
```

The output is looks as follows:

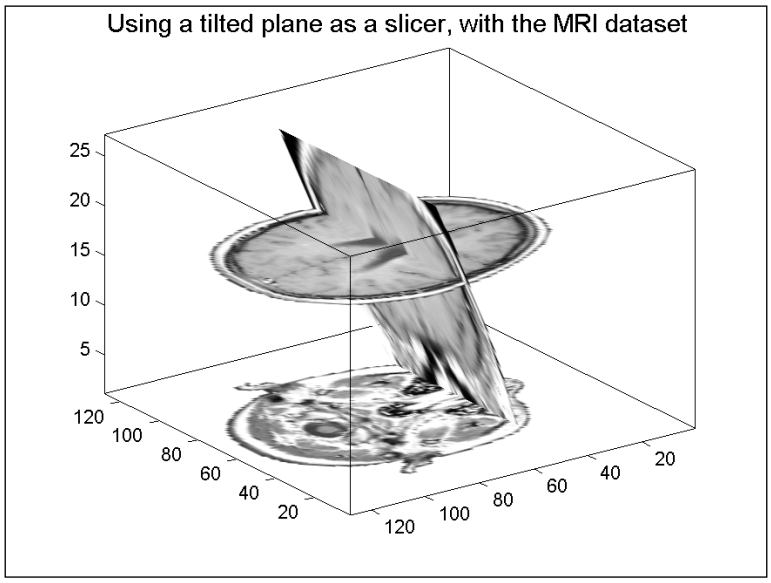


4. Create slices at an angle:

```
figure('Colormap',map);hold on;view(3);

% define the slice and rotate
hslice = slice(1:128,1:128,1:27,double(D),[],[],15);
shading interp; axis tight;
rotate(hslice,[-1,0,0],-35);

% extract the x, y and z data from rotated slice, remove it
xd1 = get(hslice,'XData');
yd1 = get(hslice,'YData');
zd1 = get(hslice,'ZData');
delete(hslice);

% call slice function with extracted data
h=slice(1:128,1:128,1:27,double(D),xd1,yd1,zd1);
shading interp; axis tight;

% set transparency to correspond with the data values
set(h,'alphadata',squeeze(double(D(:,:,15))),...
  'facealpha','interp'); alim([0 2])
```

```
% declare two other horizontal slices
h=slice(1:128,1:128,1:27,double(D),[],[],[1 18]);
shading interp; axis tight;

% set the transparencies for the additional slices
set(h(1),'alphadata',squeeze(double(D(:,:,1))),...
  'facealpha','interp'); alim([0 2])
set(h(2),'alphadata',squeeze(double(D(:,:,18))),...
  'facealpha','interp'); alim([0 2])

% set the view
zlim([1 27]);box on;
campos([-710.945 617.6196 126.5833]);

% annotate
title('Using a tilted planewith MRI dataset');
```

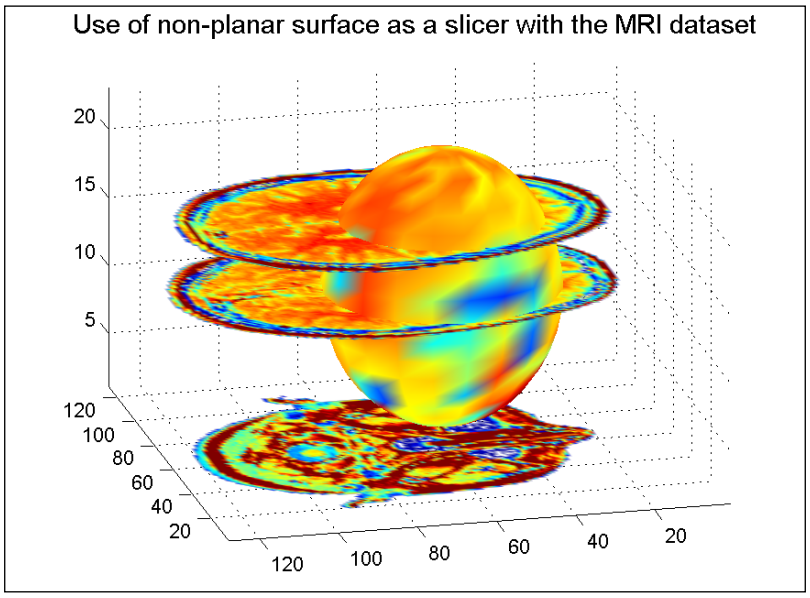
The output is as follows:



5. Use a non-planar surface as a slicer:

```
% add a boundary slice
h=slice(1:128,1:128,1:27,double(D),[],[],[1 13 18]);
shading interp; axis tight; hold on;

% set transparency
set(h(1),'alphadata',squeeze(double(D(:,:,1))),...
  'facealpha','interp'); alim([0 2])
set(h(2),'alphadata',squeeze(double(D(:,:,13))), ...
  'facealpha','interp'); alim([0 2])
set(h(3),'alphadata',squeeze(double(D(:,:,18))), ...
  'facealpha','interp'); alim([0 2])

% create a surface on which you want the projection; scale
% and translate it so that it covers the data zone
[xsp,ysp,zsp] = sphere;
hsp = surface(30*xsp+60,30*ysp+60,10*zsp+13);

% get the data out
xd = get(hsp,'XData');
yd = get(hsp,'YData');
zd = get(hsp,'ZData');

% delete the temporary surface
delete(hsp);

% plot the surface as part of the slice command
hslicer = slice(1:128,1:128,1:27,squeeze(double(D)),...
  xd,yd,zd);shading interp;

% set the view
axis tight
view(-103.5,28);

% annotate
title('Use of non-planar surface as a slicer',...
  'fontsize',14);
```

The output is as follows:



How it works...

This recipe shows how to plot systematic cross sections to reveal the structure of the data. You used a combination of planar and non-planar surfaces as slices into your volumetric data space. Note that for the non-planar or orthogonal slicing, you needed to extract the `x`, `y`, `z` data from the surface objects before you could plot them using the `slice` command. Slicing is a powerful tool to explore 3D datasets.

There's more...

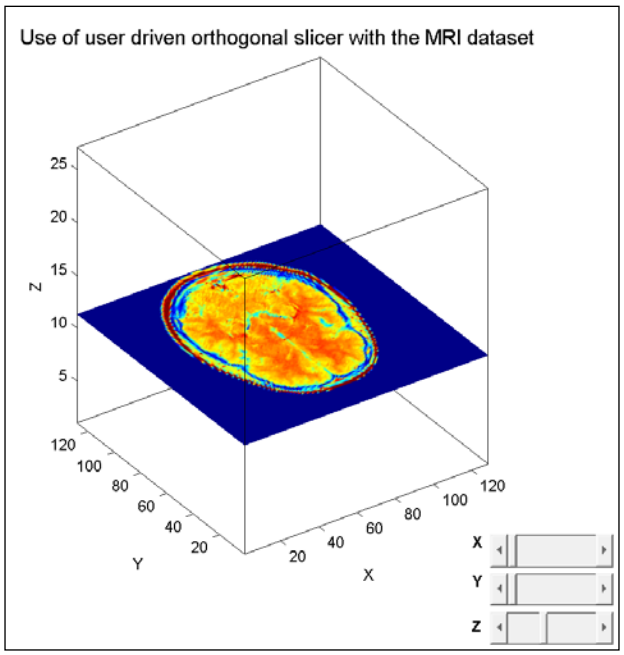
This section discusses a function that allows you to visualize your 3D data with slice positions controlled with three sliders from a GUI. It is adapted from a File Exchange submission by Loren Shure on volume visualization. Perform the following steps to initialize the visualization with the MRI data and add UI Control elements. The following code excerpt shows how to add one slider and the label associated with it. Execute lines 139 – 163 of the source code with this recipe to add the other two sliders and labels.

```
% Initialize the visualization with the data
s = volumeVisualization(1:128,1:128,1:27,double(D));

% Add uicontrol for x (and a label)
annotation('textbox',[.75,.1388,.06,.05],'String','X',...
  'fontweight','bold','linestyle','none');
```

```
hSliderx = uicontrol(...
  'Units','normalized', ...
  'Position',[.79 .13 .2 .05], ...
  'Style','slider', ...
  'Min',s.xMin, ...
  'Max',s.xMax, ...
  'Value',s.xMin, ...
  'tag','x',...
  'userdata',s,...
  'Callback',@volVisSlicesUpdateSliderPosition);
```

The function `volumeVisualization` takes the input parameters `X`, `Y`, `Z`, and `V` where `X`, `Y`, `Z` are the coordinates and `V` is the value of the 3D dataset at each of those coordinate positions. It returns a structure with the three member functions namely, `addSlicePlanex`, `addSlicePlaney`, and `addSlicePlanez` that contain function handles to add slices at the location defined by the current value of the respective sliders from an external GUI. Also, it provides the handle to a fourth member function `deleteLastSlicePLane` to delete the last slice plane added. The **callback** function associated with the sliders, `volVisSlicesUpdateSliderPosition`, has code to make the calls to the above set of functions provided by `volumeVisualization` as needed, based upon user input. The concepts around making interactive graphics are covered in detail in *Chapter 7*, *Creating Interactive Graphics and Animation*. Refer there to learn more about callback functions.

The output is as follows:



Takeaways from this recipe:

- Use a series of slices in various orientations to investigate the structure of your 3D data
- Use interactive designs to allow the user to discover regions of interest with ease
- Use surface transparency for slices and view control to maximize information visualization

See also

Look up **MATLAB help** on the `slice`, `contourslice`, `campos`, and `squeeze` commands.

Isosurface, isonormals, isocaps

Isosurfaces are surfaces that join points of equal magnitude. They are the 2D analog of contour lines. In this recipe, you will explore isosurfaces and their use in 3D visualization.

Getting ready

You will once again use the MRI dataset that comes with MATLAB installation. Load the data:

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

How to do it...

Perform the following steps:

1. Construct the human face using an `isosurface` and the MRI plates. The three-dimensional smoothing creates a smoothed surface for presenting the human face. The color of the patch is chosen to match common skin tones.

```
% create figure with predefined colormap
figure('Colormap',map)

% smooth the data
Ds = smooth3(D);

% create a isosurface; use patch to construct the image
% choose face color for common skin tone
hiso = patch(isosurface(Ds,5),...
  'FaceColor',[1,.75,.65],...
  'EdgeColor','none');
```

2. The `isonormals` are constructed based on the gradient values of the actual values of the surface elements.

```
isonormals(Ds,hiso);
```

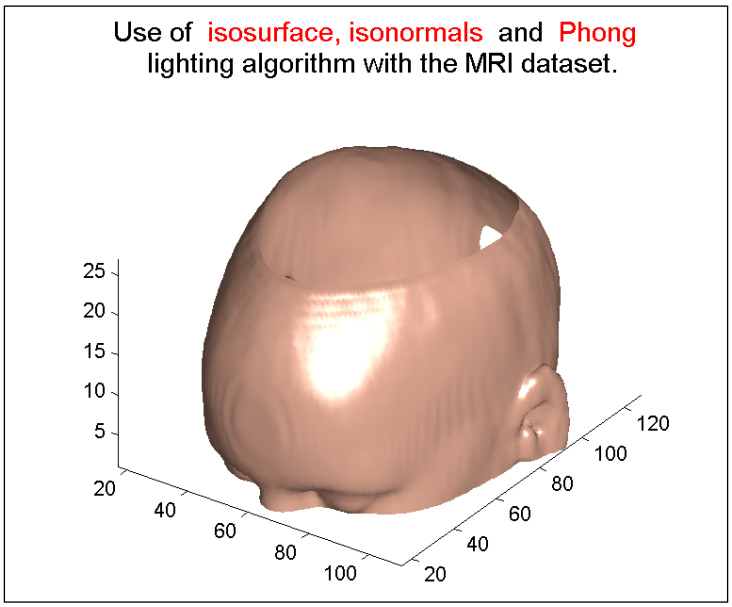
3. Adjust the view:

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

4. Adjust the lighting to create a realistic rendering:

```
lightangle(45,30);
lighting phong
```

The output is as follows:



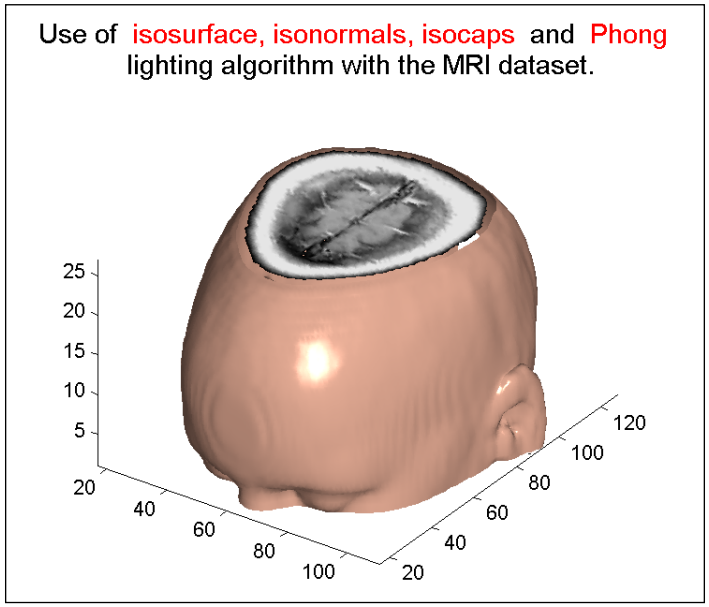
5. Add an isocap to create the effect of a cut away slice:

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

6. Adjust the lighting parameters one more time to create a realistic rendering, in this case, by reducing the specular shine:

```
set(hcap,'AmbientStrength',.6)
set(hiso,'SpecularColorReflectance',0,...
  'SpecularExponent',50);
```

The output is as follows:



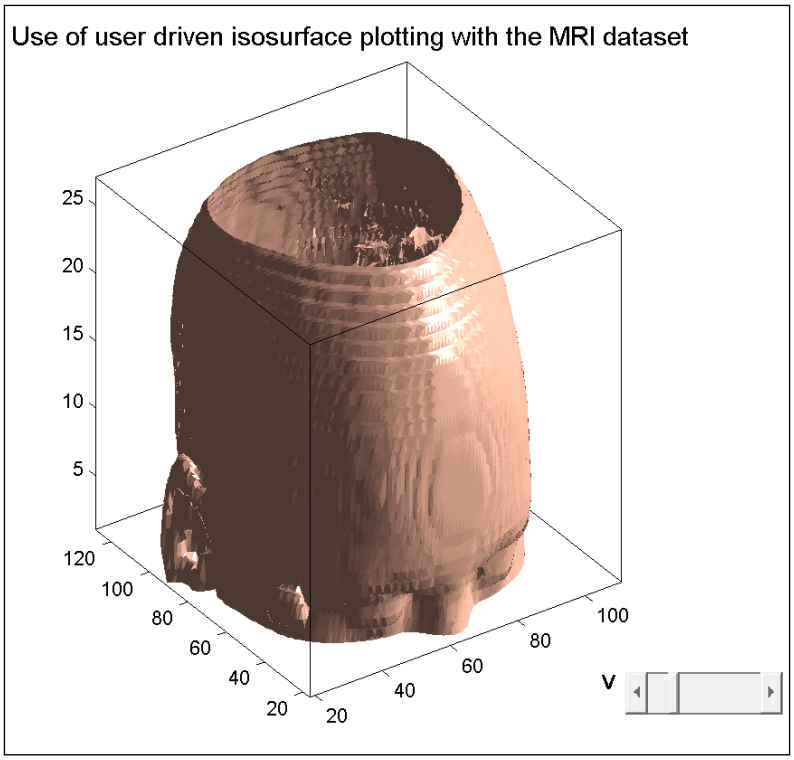
How it works...

MRI images contain scalar data. Several techniques such as the use of isosurface, isocaps, camera positioning, and lighting effects are utilized to bring the data to life.

There's more...

The user interactivity paradigm for data exploration demonstrated in previous recipe for the `slice` function is extended here to **isosurfaces**. The function `volumeVisualization_isosurface` takes the input parameters `X`, `Y`, `Z`, and `V` where `X`, `Y`, `Z` are the coordinates and `V` is the value of the 3D dataset at each of those coordinate positions. It returns a structure with two member functions, one being `plotIsoSurface` that contains a function handle to add an isosurface for the value defined by the current value of the slider from an external GUI. Also, it provides the handle to the function `deleteIsoSurface` to delete the last isosurface added. The **callback** function associated with the slider, `volVisIsoSurfaceUpdateSliderPosition`, has code to make the calls to the above functions provided by `volumeVisualization_isosurface` as needed based upon user input. The concepts around making interactive graphics are covered in detail in *Chapter 7*, *Creating Interactive Graphics and Animation*.

The output is as follows:



Takeaways from this recipe:

- Use isosurfaces, isonormals, and isocaps to create realistic 3D visualizations
- Use interactive designs to allow user to discover regions of interest with ease

See also

Look up **MATLAB help** on the `isonormals`, `isosurface`, `isocap`, and `patch` commands.

Stream slice

Vector data needs to show direction as well as magnitude. This requires an additional data dimension to be displayed at each point and thus presents a challenge for the techniques demonstrated so far. MATLAB provides the ability to draw **stream lines** on the slices to show the direction at each location.

Getting ready

In this recipe, you will use data from the electric dipole.You will calculate the electric field generated by the dipole antenna in 3D space and visualize it using the `streamslice` MATLAB function. Execute lines 12 – 24 of the source code for this recipe to generate the data.

How to do it...

Perform the following steps:

1. Plot the data using the MATLAB command `streamslice`:

```
streamslice(x,y,z,E_ang_x,E_ang_y,E_ang_z,...
  [-.3 .1],[],[]);
```

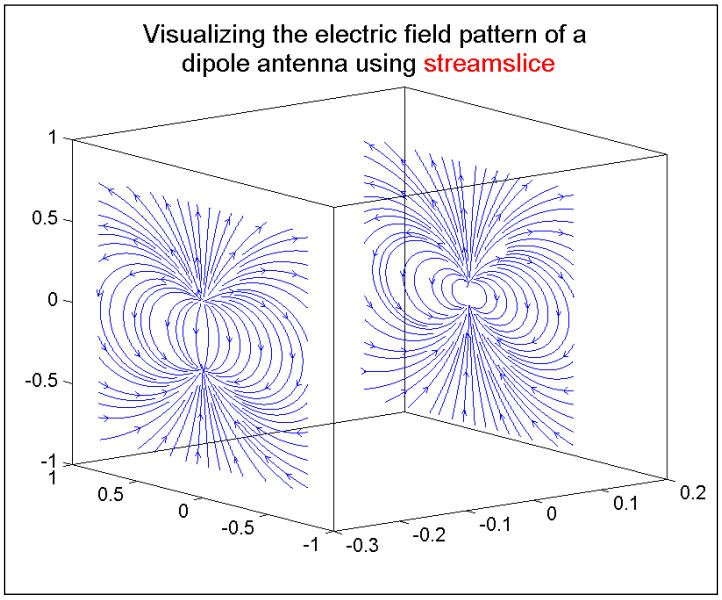
2. Set the view:

```
campos([-3,-15,5]);
box on
```

3. Set a grey background and black axes markings:

```
set(gca,'Color',[.8,.8,.8] ,'XColor','black', ...
  'YColor','black','ZColor','black');
```

The output is as follows:



How it works...

The Electric Field is a vector data set. This means this data has both magnitude and direction. The direction is represented by three different numbers denoting the `x`, `y`, and `z` components of the direction vector. You used the MATLAB command `streamslice` to present the vector data. In this case, slices are placed at `x= -0.3` and `x =0.1`. The empty matrices imply that all points along that axis should be used. `Streamslice` works similarly to `slice` commands. It extracts a certain plane in which to display the information. The contours draw the vector data using `streamlines`.

Takeaways from this recipe:

- Use stream slices to investigate the structure of your 3D vector dataset

See also

Look up **MATLAB help** on the `streamslice`, and `streamlines` commands.

Stream lines, ribbons, tubes

MATLAB offers a variety of stream plots (**stream lines**, **stream ribbons**, **stream tubes**, and **cones**) to illustrate the flow of a 3D vector field.

Getting started

You will use the wind dataset that comes with the MATLAB installation to explore these functions. Load the data:

```
load wind
```

How to do it...

Perform the following steps:

1. Define the starting points for `streamlines`:

```
[sx sy sz] = meshgrid(80, 20:10:50, 0:4:16);
plot3(sx(:),sy(:),sz(:),'bo','MarkerFaceColor','b');
hold on;
```

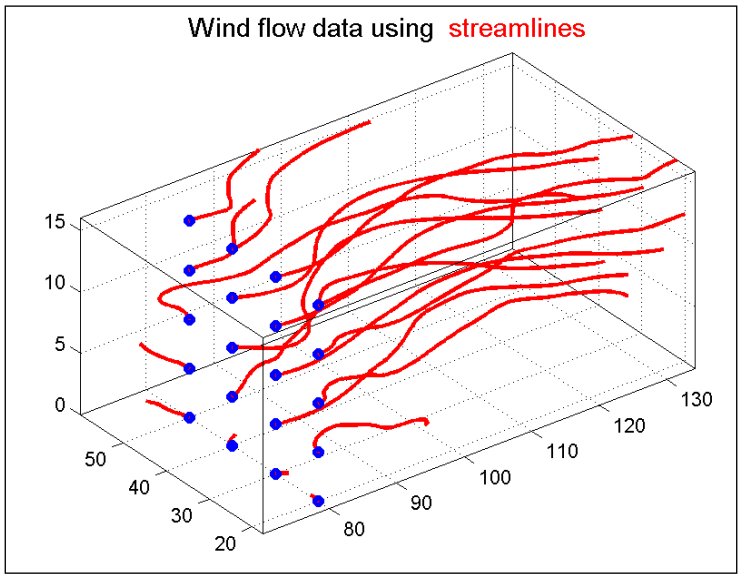
2. Plot the stream lines with custom thickness and color:

```
h=streamline(x,y,z,u,v,w,sx(:),sy(:),sz(:));
set(h,'linewidth',2','color',[1 0 0]);
```

3. Set the view:

```
axis(volumebounds(x,y,z,u,v,w))
grid; box; daspect([2.5 3 1.5]);
```

The output is as follows:



4. Next, compute the curl, which is a vector data and the wind speed, which is a scalar data:

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

5. Generate vertices from which the stream ribbons should start. Use the `stream3` command to generate `streamlines`:

```
[sx sy sz] = meshgrid(80,20:10:50,0:4:16);
verts = stream3(x,y,z,u,v,w,sx,sy,sz,1);
```

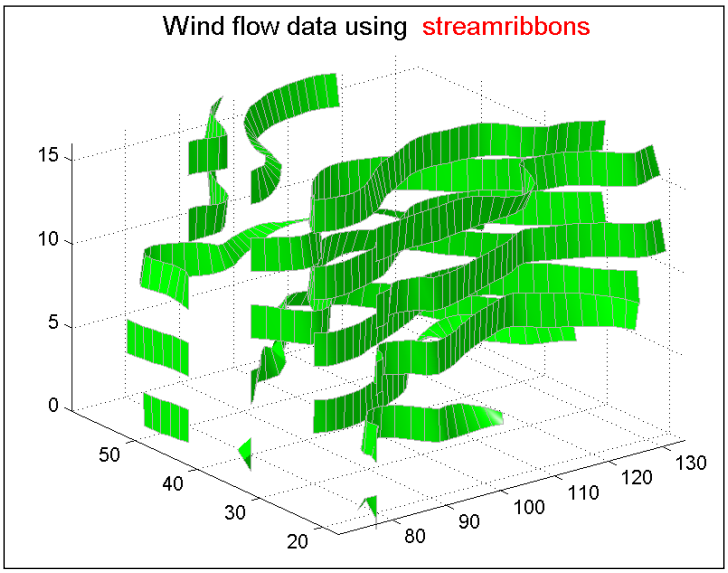
6. Plot customized `stream ribbons`. The width factor is set to 2 to enhance visibility. The face color is set to green, edge color to grey and ambient light strength to 0.6:

```
h = streamribbon(verts,x,y,z,cav,wind_speed,2);
set(h,'FaceColor','g',...
  'EdgeColor',[.7 .7 .7], 'AmbientStrength',.6);
```

7. Set the view for the plot (including lighting parameters):

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

The output is as follows:



8. Next, present the divergence of this vector data using `streamtubes`. The same vertices (or start points) for the streamlines are used.

```
div = divergence(x,y,z,u,v,w);
```

9. Use `stream3` to calculate the streams and plot with tubes:

```
[sx sy sz] = meshgrid(80,20:10:50,0:4:16);
verts = stream3(x,y,z,u,v,w,sx,sy,sz,1);
```

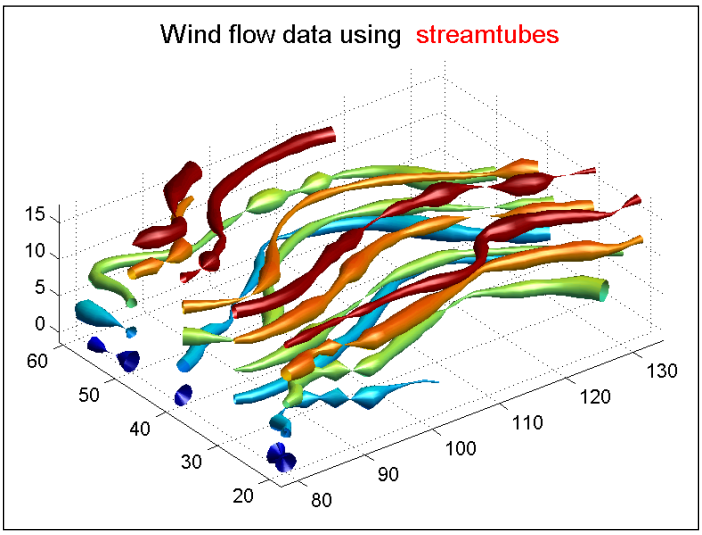
10. Plot the `streamtubes`:

```
h=streamtube(verts,x,y,z,div);
shading interp;
```

11. Set the view for the plot (including lighting parameters):

```
daspect([1 1 1]);axis tight
grid on;view(3);
camlight; lighting gouraud
```

The output is as follows:



How it works...

Vector data has magnitude and direction. The stream commands offer a way to visualize the path. The `u`, `v`, `w` vectors are tangents to any given point on the `streamline`. Note that some prior knowledge of your data's characteristics such as the primary direction of flow and the range of the data coordinates helps in the selection of the starting points for your streamlines.

The color, thickness, and other attributes can be used to encode additional information. You did this in step 6 by encoding the rotation about the flow axis information with a proportional twist of the ribbon-shaped flow line.

Takeaways from this recipe:

- Use stream lines, ribbons, and tubes to investigate the structure of your 3D vector dataset

See also

Look up **MATLAB help** on the `streamline`, `stream3`, `streamtube`, and `volumebounds` commands.

Scalar and vector data with a combination of techniques

This recipe was created to demonstrate how to combine various techniques such as the use of **isosurface**, **isonormal** computations, **lighting** parameters considerations, **view** angle considerations, **transparency**, **contourslice**, **streamlines**, **streamtubes**, and **cone** plots to create an enhanced visualization.

Getting ready

You will use the `wind` dataset that comes with the MATLAB installation to explore these functions. Load the data. Generate some additional variables from the data:

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
xmin = min(x(:));xmax = max(x(:));
ymax = max(y(:));ymin = min(y(:));
```

How to do it...

Perform the following steps:

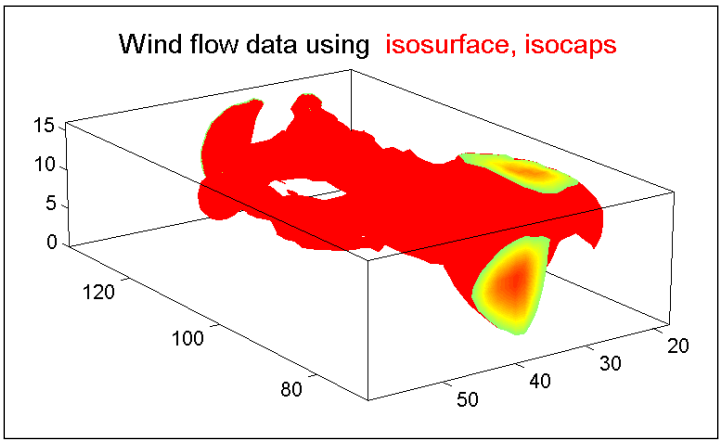
1. Create an isosurface with isocaps at wind speed `40`. The isocaps create an effect of cutting off the surface and reveal the distribution of values on that plane.

```
% add isosurface
p = patch(isosurface(x,y,z,wind_speed, 40));

% use isonormals for a smoother surface
isonormals(x,y,z,wind_speed, p);

% set the facecolor to red and no edge marks
set(p, 'FaceColor', 'red', 'EdgeColor', 'none');

% add isocaps
p2 = patch(isocaps(x,y,z,wind_speed, 40));
set(p2, 'FaceColor', 'interp', 'EdgeColor', 'none')
```

2. Set the view:

```
box on;camproj perspective;
axis(volumebounds(x,y,z,wind_speed))
campos([ -203.7953  253.0409  129.3906]);
daspect([1 1 1]);
lighting phong;
```

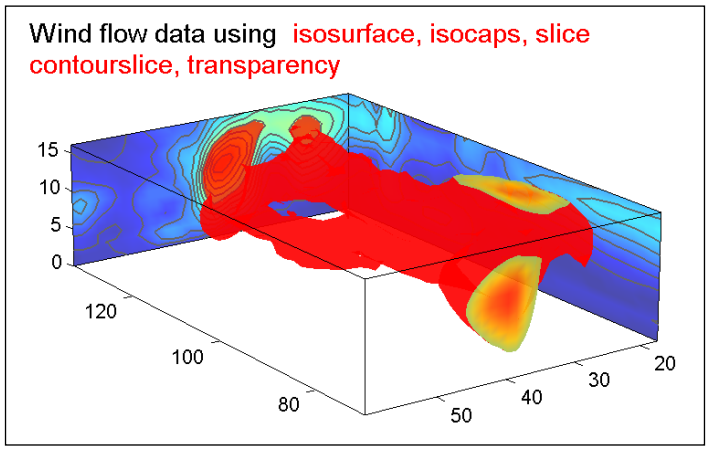
The output is as follows:



3. Draw **contour slices** on the two back walls. Change the transparency of the isosurface to make contour lines approximately visible:

```
% Create one slice at max y and add color from the data
hold on;hslice = slice(x,y,z,wind_speed,xmax,ymin,[]);
set(hslice,'FaceColor','interp','EdgeColor','none')

% Change the transparency of the isosurface to make contour
% lines approximately visible.
alpha(0.7);
color_lim = caxis;

% add contour intervals on it
cont_intervals = linspace(color_lim(1),color_lim(2),17);
hcont = contourslice(x,y,z,wind_speed,xmax,ymin,...
  [],cont_intervals,'linear');
set(hcont,'EdgeColor',[.4 .4 .4],'LineWidth',1);
```

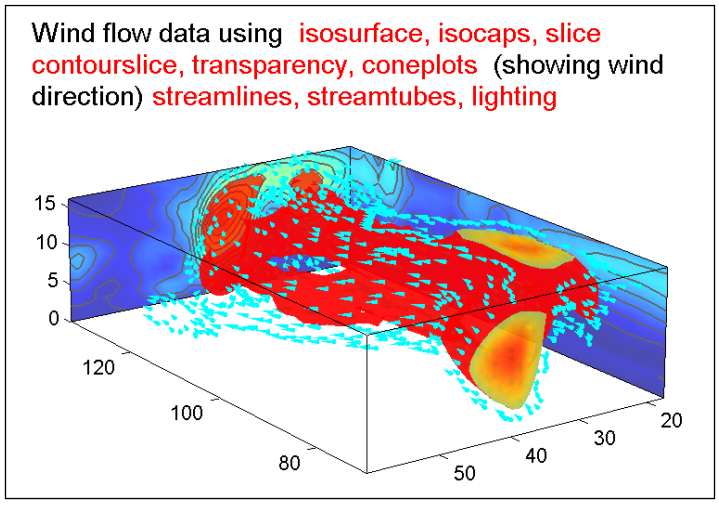
The output is as follows:



4. Show the wind direction with cone plots:

```
% calculate direction
[f verts] = reducepatch(isosurface(...
  x,y,z,wind_speed,30),.2);

% plot with cones
h=coneplot(x,y,z,u,v,w,verts(:,1),verts(:,2),...
  verts(:,3),2);
set(h, 'Facecolor', 'cyan','EdgeColor', 'none');
```

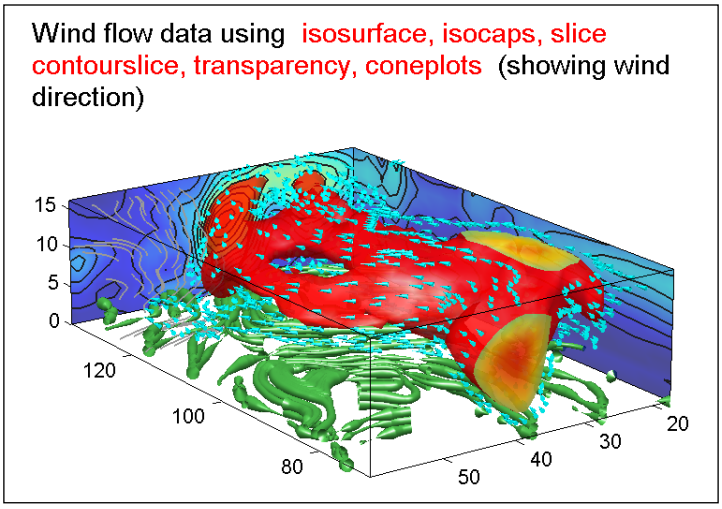
The output is as follows:



5. Add streamlines to one side in grey and stream tube to the bottom plane. Note that additional lighting parameters bring the scene to life.

```
% calculate path for stream lines
[sx sy sz] = meshgrid(120:6:130, 50:2:60, 0:5:15);

% add streamlines
h2=streamline(x,y,z,u,v,w,sx,sy,sz);
set(h2, 'Color', [.6 .6 .6],'Linewidth',1);

% calculate path for stream lines
[sx,sy,sz] = meshgrid(xmin:10:xmax,ymin:10:ymax,0:2);

% add streamtubes
htubes = streamtube(x,y,z,u,v,w,sx,sy,sz,[.5 20]);
set(htubes,'EdgeColor','none','facecolor',[.2 .5 .2],...
  'AmbientStrength',.5)

%set lighting params
camlight left;
camlight right;
lighting Gouraud;
```

The output is as follows:



How it works...

A number of volume visualization techniques have been brought together to create an enhanced visualization for the wind data. In this case, you started with `isosurface` and used the `isonormals` command for computing the isonormals using vertex data, so that you obtain a smooth finish to the surface. You added `isocaps` to create the cut off effect. You changed `alpha` to impact the transparency of the surface to see the `contourslices` that you constructed as the back walls to this visual. You added `coneplots` to show the divergence, and `streamlines` and `streamtube` for visualizing the flow in the parts of the data block not covered by the isosurface. You used `lighting` and camera angle positioning to set the view. The combination of these techniques creates an effective way to explore 3D datasets.

Takeaways from this recipe:

- Use a combination of techniques to explore your 3D dataset

See also

Look up **MATLAB help** on the `slice`, `contourslice`, `alpha`, `camera`, `streamline`, `stream3`, `streamtube`, and `volumebounds` commands.

Explore with camera motion

MATLAB offers a plethora of camera control tools. This enables you to move the view position around the body of the data, along the body of the data, or directly inside the body of the data, for exploration. This recipe demonstrates programming the movement of the view point to create data exploration movies for 3D datasets.

Getting started

You will use a flow dataset which comes as part of MATLAB installation. This is generated at the command line as follows:

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

How to do it...

Perform the following steps:

1. Create an isosurface. Use the `patch` command so that you can retrieve the handle and alter some of its attributes:

```
p = patch(isosurface(x,y,z,v,-2));
set(p,'FaceColor','red','EdgeColor','none');
```

2. Set the camera projection to `perspective` (which is the normal way we see things, with distant objects appearing smaller in size):

```
camproj perspective
```

3. Set up the lighting parameters by adding both, a scene light as well as altering the reflectance properties of your surface. The headlight setting ensures that the light will move with the camera position. For lighting, MATLAB must use either the `zbuffer` or, if available, `OpenGL` renderer settings. The OpenGL renderer is likely to be much faster; you need to use Gouraud lighting with OpenGL.

```
hlight = camlight('headlight');
set(p,'AmbientStrength',.1,...
  'SpecularStrength',1,...
  'DiffuseStrength',1);
lighting gouraud
set(gcf,'Renderer','OpenGL')
```

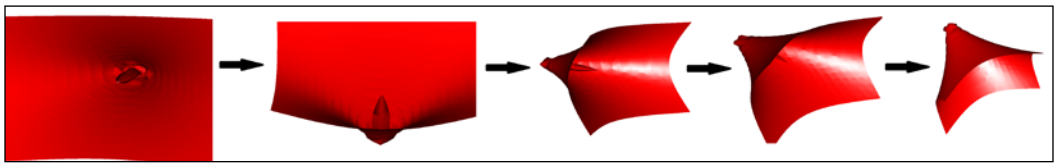
4. Define a custom path for the camera movement. Select a few pivot points for the camera using the camera toolbar (described in the next section). Generate a linear path connecting these positions as follows:

```
xp = [30*ones(1,50) linspace(30,55,50) ...
  linspace(55,45,50) linspace(45,-45,50) linspace(45,15,50)];
yp = [linspace(-12,5,50) linspace(5,-0.5,50) ...
  linspace(-0.5,-30,50) linspace(-30,-25,50)...
  linspace(-25,-15,50)];
zp = [5.4*ones(1,50) linspace(5.4,-25,50) ...
  linspace(-25,10,50) linspace(10,30,50) linspace(30,80,50)];
```

5. Move the camera along the defined path. The `drawnow` command is used to redraw and show the effects of each move.

```
for i=1:length(yp)
  campos([30,yp(i),zp(i)]);
  camtarget([3.5,-0.1666,0]);
  camlight(hlight,'headlight');
  drawnow;
end
```

A set of snapshots taken at intervals of 50 steps is as follows:



How it works...

In this recipe, you used camera tools to explore your dataset. The view can be controlled using camera tools in several ways. Here, you moved the position of the camera by the `campos` command. You also simultaneously moved the positions of the camera target via the `camtarget` command. You added a headlight to your camera to enhance the realism of the views.

There are several ways to choose the path for moving the camera positions. One option is to use a set of predefined options such as setting the direction for the `cameraupvector` and using either the `camorbit`, `camdolly`, or `camroll` functions to move your camera with respect to the `up` vector in predefined ways. Alternatively, you can define a custom path, such as the one you used here using the camera toolbar to select your pivots and then connecting them with linear interpolation points. To select your pivot points, enable camera movement after the object is displayed in 3D by clicking the camera icon (highlighted in yellow) on the figure toolbar, as shown in the following screenshot:

Move the object as desired by selecting your 3D object directly from the graph. To select a configuration as a pivot, access the associated camera position and the camera target position parameter values as follows:

```
get(gca,'CameraTarget');
get(gca,'CameraPosition');
```

If using vector data, you could also use the `stream3` function to create a streamline and extract the `x`, `y`, `z` values along that stream to move your camera along the path so defined.

There's more...

When the camera target and the camera position are moved at the same time with a slight constant offset, you can get the effect of a fly through (as if you are flying through the geometry of your object). Perform the following steps:

1. Set the view angle. The `camva` command helps to set the view angle. A small view angle has the effect of a zoom in to the image.

```
camva(5);
```

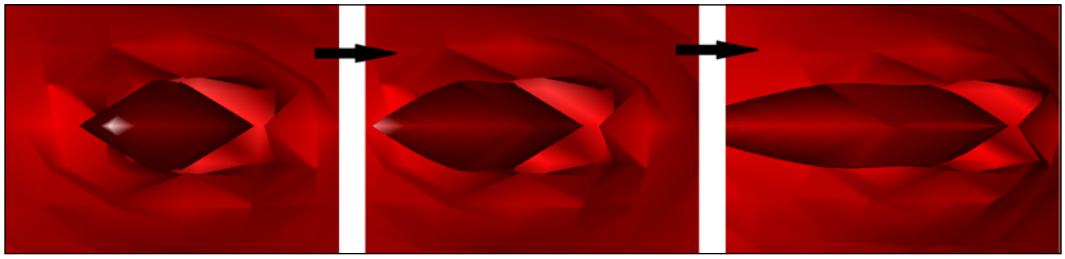
2. Set the path:

```
xt = [linspace(1,.2,25) linspace(.2,.1,75)];
yp = [linspace(0,.8,25) .4*linspace(2,4,25) ...
  1.6+.4*linspace(0,10,50)];
yt = [linspace(-.4/3,-.4/3,50) ...
  -(.4/3)*linspace(1,10,50) ];
zt = [linspace(0,0,25) linspace(0,0,75) ];
```

3. Move the camera target and position along that path:

```
for i=1:length(xt)
  campos([xt(i)+10,yp(i),zt(i)]);
  camtarget([xt(i) yt(i) zt(i) ]);
  camlight(hlight,'headlight');
  drawnow
end
```

A set of snapshots taken at certain intervals along the path is as follows:



Takeaways from this recipe:

- Use programmatic view control as a tool to explore your 3D dataset with animation

See also

Look up **MATLAB help** on the `camdolly`, `camlookat`, `camorbit`, `campan`, `campos`, `camproj`, `camroll`, `camtarget`, `camup`, `camva`, and `camzoom` camera manipulation commands.

6
Designing for Higher Data Dimensions

In this chapter, we will cover:

- Fusing hyperspectral data
- Survey plots
- Glyphs
- Parallel coordinates
- Tree maps
- Andrews' curves
- Down-sampling for fast graphs
- Principal Component Analysis
- Radial Coordinate Visualization

Introduction

This chapter discusses visualization options for beyond the three dimensions. For data in high dimensions, the volume increases so fast that all objects appear to be sparse and a lot more data becomes necessary to establish statistical significance in calling two objects dissimilar, so that common data organization strategies become inefficient. This problem is often described as the **curse of dimensionality**. The first set of recipes in this chapter present strategies to visualize high dimensional data without any reduction in the data dimensions. The last three recipes address data reduction strategies. Specifically, the reduction of higher dimensional data to lower dimensions without significant information lossis addressed with the **Principal Component Analysis** (a data reduction technique) and the **Radial Coordinate Visualization** (a projection technique) in this chapter.

There are other advanced methods for dimensionality reduction such as the **projection pursuit** and their interactive/guided versions known as the **grand tour**, which are commonly used in the **exploratory data analysis** community to deal with this problem. The **projection pursuit** algorithm finds the most interesting projections of the multidimensional data, using a search algorithm that optimizes some fixed criterion of interestingness – such as deviation from a normal distribution (negative entropy) or measure of variation (principal components), class separability (using some distance metric) – all using a linear mapping. These advanced methods are not included as recipes. The *Appendix* includes references to these techniques for further investigation.

Fusing hyperspectral data

Spectral imaging collects information from across the electromagnetic spectrum including and extending beyond the visible ranges. Objects leave patterns across the spectrum (their spectral signatures), that enable their identification. For example, a spectral signature for oil helps mineralogists find new oil fields. The reason for imaging across multiple bands is that not all features are visible in all bands.

These large volumes of data collected demand appropriate methods for scanning and interpretation. Commonly, data is first transformed using a method such as the Principal Component Analysis and then the three most significant components are used as the RGB channel data to create **false color** images.

Getting ready

In this recipe, you will visualize a hyperspectral dataset. The data was downloaded from the free datasets made available by SpecTIR™ on their website. The data covers a part of the Gulf of Mexico region, showing the 2010 oil spill images from 360 different spectral bands. Download the data from this location: `http://www.spectir.com/free-data-samples/request-deepwater-radiance/`. Providing the dataset in the code repository is avoided because of its significant file size. Once downloaded, import the data as follows:

```
X = multibandread(...
  '0612-1615_rad_sub.dat',...Filename
  [1160, 320, 360],...size ([lines, samples, bands])
  'uint16', ...precision data type in the ENVI header file
  0,...offset parameter from ENVI header file
  'bil',...interleave parameter 'bsq', 'bil' or 'bip'
  'ieee-le',...byteorder
  {'Band','Range',[1,5,360]},...subset 1
  {'Row','Range',[150,1,1000]},...subset 2
  {'Column','Range',[60,1,300]}...subset 3
  );
```

Note that every input parameter in the command is explained with an accompanying comment on the same line (the part after the ellipses is treated as comments). Each subset is defined as a triplet of `{DIM, METHOD, INDEX}`, where `DIM = 'Row', 'Column'` or `'Band'`, `METHOD = 'Direct'` or `'Range'` and `INDEX` gives the indices to use to extract the data based upon the `METHOD` definition. For more explanation look up MATLAB help on `multibandread`.

Note that only a subset of the data was loaded for memory efficiency purposes.

How to do it...

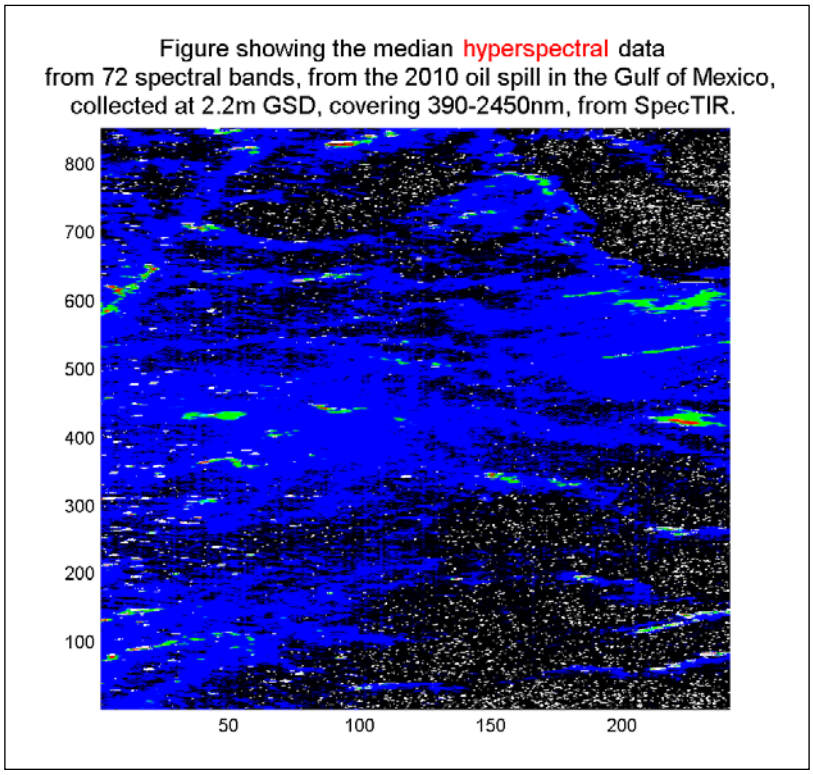
Perform the following steps:

1. Normalize and visualize the median signal across 72 bands:

```
X = X./max(X(:));

% visualize median signal across several bands
surf(log(median(X,3))); view(2);
shading interp;
axis tight;
brighten(-.8);

% annotate
```

```
title({['\color{red}False Color Image \color{black}' ...
  'showing the median' ...
  '\color{red} hyperspectral \color{black}data'],...
  ['from 72 spectral bands, from the 2010 oil ' ...
  'spill in the Gulf of Mexico,',...
  'collected at 2.2mGSD, covering 390-2450nm,' ...
  'from SpecTIR.']},'Fontsize',14);
```

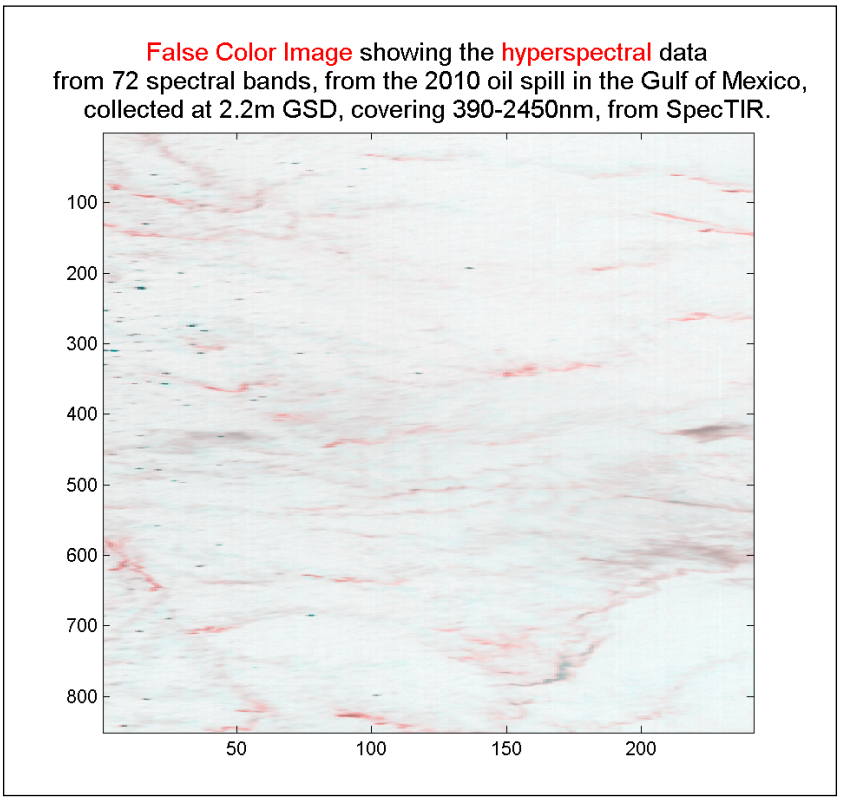
The output is as follows:



2. Alternately, use the data to generate RGB values and create a false color image:

```
R = log(median(X,3));R = abs(R./max(abs(R(:))));
G = log(range(X.^2,3));G = abs(G./max(abs(G(:))));
B = log(median(X,3));B = abs(B./max(abs(B(:))));
d(:,:,1) = G;d(:,:,2) = B;d(:,:,3) = R;
image(d);
title('\color{red}False Color Image');
```

The output is as follows:



How it works...

The **ENVI data** comes in pairs with a `.hdr` header file and the `.dat` binary datafile. The data file has to be read per the parameters specified in the header file. The MATLAB function `multibandread` takes several input parameters that you need to obtain from the header file of your dataset in order to use it to read the binary dataset as was shown in the *Getting Ready* section.

While most of the parameters needed for the function are easily identifiable in the header because they use the same name, for two parameters in particular, you have to convert the ENVI values to a corresponding value used by MATLAB.

Correspondence between ENVI data types and MATLAB data types is given as follows:

ENVI data type	MATLAB precision
1	1 byte signed integer 'int8'
2	2 byte signed integer 'int16'
3	4 byte signed integer 'int32'
4	4 byte float 'float'
12	2 byte unsigned integer 'uint16'

Correspondence between ENVI byte order and MATLAB byte order is given as follows:

ENVI byte order	MATLAB byte order
0	'ieee-le'
1	'ieee-be'

In this recipe, you first used the median data across 72 spectral bands to visualize the oil spill radiance measurements. Next, you generated RGB values using some transformations on the data (you could also split available data into groups to get your RGB values). Using the RGB values, you generated a false color image.

Takeaways from this recipe:

- Assemble spectral data from multiple bands into a single visualization scheme for detecting features of interest visible in one or more of the large number of spectral bands

See also

Look up **MATLAB help** on the `multibandread.m` command.

Survey plots

The idea of **survey plots** is to present heterogeneous data that correspond to a single entity in a way that facilitates both an intra-dimensional comparison and comparison across the entities. The heterogeneous data dimensions can be a mix of spatial, temporal, numeric, and alphanumeric values.

Getting ready

The data in this recipe represents the SAT scores and crime rates reported from three counties across five locations. A county is allowed two data points to allow approximately equal areas to be covered by each point. The data is grouped by the income levels. Additionally, population demographic data is presented in the graph for identifying possible correlations. Load the data as follows:

```
load customCountyData
```

How to do it...

Perform the following steps:

1. Layout the figure:

```
figure('units','normalized','position',[.04 .12 .6 .8]);
axes('position',[0.0606    0.0515    0.9065    0.8747]);
set(gcf,'color',[1 1 1],'paperpositionmode','auto');
```

2. Create the spatial context with approximate outlines of the three counties, using a filled set of polygons:

```
% coordinate definitions
x1 = [2 3 4.5 3.5 2.5 1.5];y1 = [1 1.5 3 4 3.5 2];
x2 = [1.5 2.5 3.5 4 3 2];y2 = [2 3.5 4 6 5.5 4.5];
x3 = [4.5 3.5 4 5 5.5 5];y3 = [3 4 6 5.5 5 4.5];

% plot the counties using above cords
patch([x1; x2; x3]',[y1; y2; y3]',...
  0.8*ones(length(x1),3),[.9 .9 .9],...
  'edgecolor',[1 1 1]);
hold;alpha(.9);
```

3. Create the five data origin lines in the z direction and a grid connecting these lines for readability:

```
% x,y locations for the data positions
x = [  1.5 2   2.2 3.32   3.8 4.5 5.2];
y = [  2   1.5 3.8 2.5   4   5   4.5];

% draw vertical lines at the data positions
for i = 2:6
  line([x(i) x(i)],[y(i) y(i)],[1 11],...
    'Color',[.8 .8 .8]);
```

```
end

% draw connecting grid lines
for i = 2:11
  line([x(1:end-1); x(2:end)],[y(1:end-1); y(2:end)],...
    i*ones(2,6),'Color',[.8 .8 .8]);
end
```

4. Declare the following color matrix definitions:

```
% cd defines the colors used for the ethnicity
cd = [141 211 199; 255 255 179; 190 186 218; ...
  251 128 114; 128 177 211]/255;

% ci defines the colors used for the crime rate
ci = [205 207 150; 254 178 76; 240 59 32]/255;
```

5. For a given year (indexed by `j`), at a given location (indexed by `i`), plot a set of lines showing the income groups and a set of lines showing the ethnic makeup at that position. For example, `for i=2, j=2`.

```
i=2; j= 2;
% the line representing the ethnic distribution
% extends along the y axis
dt =  [0 demgraphics{i-1}{j-1}/100];
for r = 1:5
  line([x(i) x(i)],...
    [y(i)-1+sum(dt(1:r)) y(i)-1+sum(dt(1:r+1))], ...
    [j j],'Color',cd(r,:),'Linewidth',5);
end

% the line representing the income groups
% extends along the x axis
dt =  [0 incomeGroups{i-1}{j-1}];
for r = 1:3
  xx1 = x(i)+sum(dt(1:r))/2;
  xx2 = x(i)+sum(dt(1:r+1))/2;
  line([xx1xx2],[y(i) y(i)], [j j],...
    'Color',ci(r,:),'Linewidth',2);
end
```

6. On the lines for each income group, draw circles of area proportional to the average SAT scores (above the lines) and crime rates (below the lines):

```
for r = 1:3
  xx1 = x(i)+sum(dt(1:r))/2;
```

```
    xx2 = x(i)+sum(dt(1:r+1))/2;

    switch(r)
    case 1
    scatter3((xx1+xx2)/2,y(i),...
      j+.2,100*AverageSATScoresLI(i-1,j-1),...
      ci(r,:),'filled');
    scatter3((xx1+xx2)/2,...
      y(i),j-.2,100*crimeRateLI(i-1,j-1),...
      ci(r,:),'filled');
    case 2
    scatter3((xx1+xx2)/2,y(i),j+.2,...
      100*AverageSATScoresMI(i-1,j-1),...
      ci(r,:),'filled');
    scatter3((xx1+xx2)/2,y(i),j-.2,...
      100*crimeRateMI(i-1,j-1),ci(r,:),'filled');
    case 3
    scatter3((xx1+xx2)/2,y(i),j+.2,...
      100*AverageSATScoresHI(i-1,j-1),...
      ci(r,:),'filled');
    scatter3((xx1+xx2)/2,y(i),j-.2,...
      100*crimeRateHI(i-1,j-1),ci(r,:),'filled');
    end
  end
```

7. Set the view:

```
view(3);
campos([ 14.7118  -42.0473   45.3905]);
axis tight off
```

8. Add annotations:

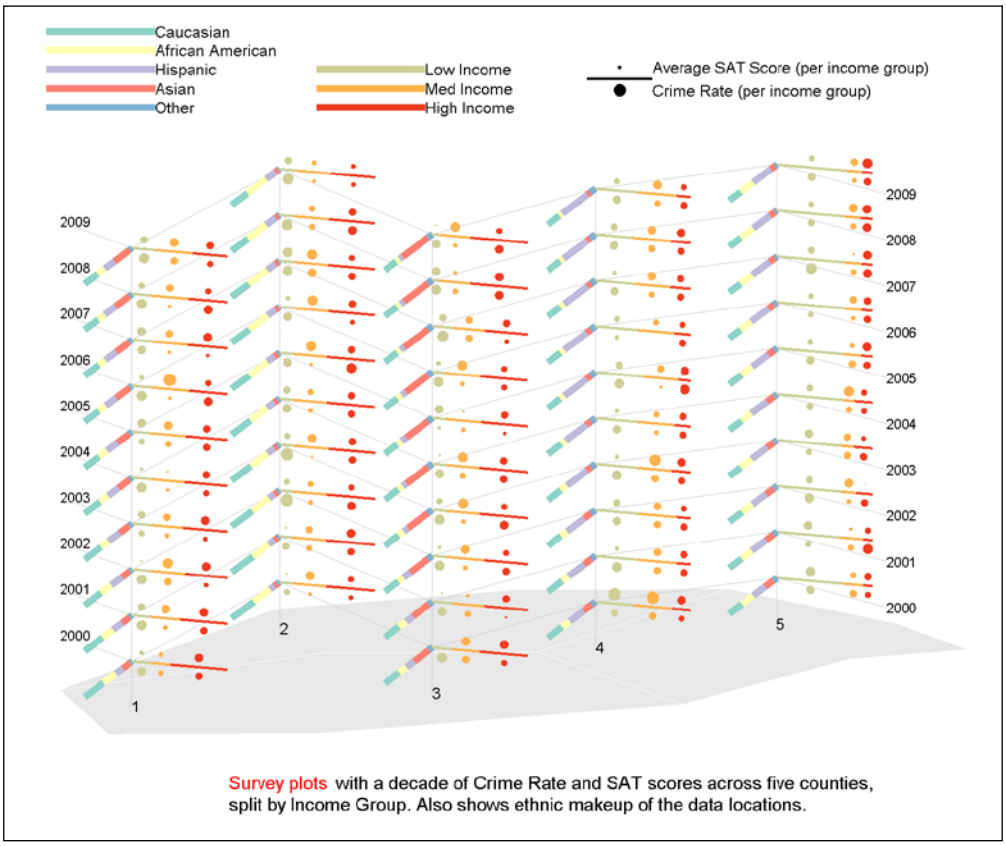
```
annotation('textbox','position',[.27 .07 .68 .04],...
  'string',{['\color{red}Survey plots \color{black}'...
  'with a decade of Crime Rate and SAT '...
  'scores across five counties,'],...
  ['split by Income Group. Also shows'...
  'ethnic makeup of the data locations.']},...
  'Fontsize',14,'linestyle','none');

% Add the year labels
for i = 2:11
  text(x(1),y(1),i,years{i-1});
  text(x(7),y(7),i,years{i-1});
```

```
end

% Add the location labels
for i = 2:6
  text(x(i),y(i),1,num2str(i-1),'fontsize',12);
end
```

9. Add the extensive set of legends using hidden axis, lines, and associated text labels:

```
axes('position',[.1107 .8478 .2881 .1347],...
  'Visible','off');
axis([0 .6 -.5 5.5]);

for i = 1:5
  line([0 .2],[5-(i-1) 5-(i-1)],'color',cd(i,:),...
    'Linewidth',5);
  text(.2,5 - (i-1),lege{i},'Fontsize',12);
end
axes('position',[.3495 .8478 .2881  .1347],...
  'Visible','off');
axis([0 .6 -.5 5.5]);
for i = 1:3
 line([0 .2],[3-(i-1) 3-(i-1)],'color',ci(i,:),...
    'Linewidth',5);
 text(.2,3 - (i-1),icleg{i},'Fontsize',12);
end
axes('position',[.5886 .8478 .2881 .1347]);hold on;
line([0 .2],[.5 .5],'color',[0 0 0],'Linewidth',2);
scatter(.1,.5+.1,10, [0 0 0],'filled');

text(.2,.5+.1,'Average SAT Score (per income group)',...
  'Fontsize',12);
scatter(.1,.5-.1,100,[0 0 0],'filled');
text(.2,.5-.1,'Crime Rate (per income group)',...
  'Fontsize',12);
axis([0 1 0 1]);
set(gca,'Visible','off');
```

The output is as follows:



Survey plots with a decade of Crime Rate and SAT scores across five counties, split by Income Group. Also shows ethnic makeup of the data locations.

How it works...

The graph shows that over the decade there has been an increase in the Asian population at locations 1 and 3. The mid and high income groups have consistently had a higher SAT score than the lower income groups across all areas and times. The crime rates are higher in the lower income groups. Other details emerge from this representation with careful review, which is left for the user to discover.

In this recipe, you used the `patch` element to layout the county map outline to provide a spatial context. You used the vertical dimension to represent the time series at each location. The demographics and income group data were presented as fractions contributing to the length of a line with color coding a specific category. Two additional pieces of numerical data from each income group category were presented using area of circles above and below the lines in corresponding colors.

Takeaways from this recipe:

- Use the concept of **survey plots** in designing your displays with set of heterogeneous information about an entity affording intra-dimensional comparisons and comparison across the entities

See also

Look up **MATLAB help** on the `patch`, `line`, and `legend` commands.

Glyphs

A **glyph** is a marker that has the features of the marker encoded with meaning. For example, an arrow can be used as a glyph in the context of data visualization. The length of the arrow can depict one parameter; the orientation of the arrow, the thickness of the lines, the thickness of the arrow head, and the color in which it is drawn are all legitimate features that can be encoded with meaning to visualize a multi-dimensional data point.

Getting ready

You will use a car dataset that comes with the MATLAB installation. It has attributes such as acceleration, number of cylinders, displacement, mileage, horse power, and weight information for cars with different models and from different manufacturers.

```
% load the data
load carsmall
dat = [Acceleration Cylinders Displacement Horsepower ...
  MPG Weight];
```

How to do it...

In this recipe, you will first visualize the car data using an arrow as the glyph. In later sections, you will use the MATLAB function `glyphplot` that comes with the Statistical Toolbox™ to visualize this same data with two other types of glyphs, namely the star and Chernoff faces.

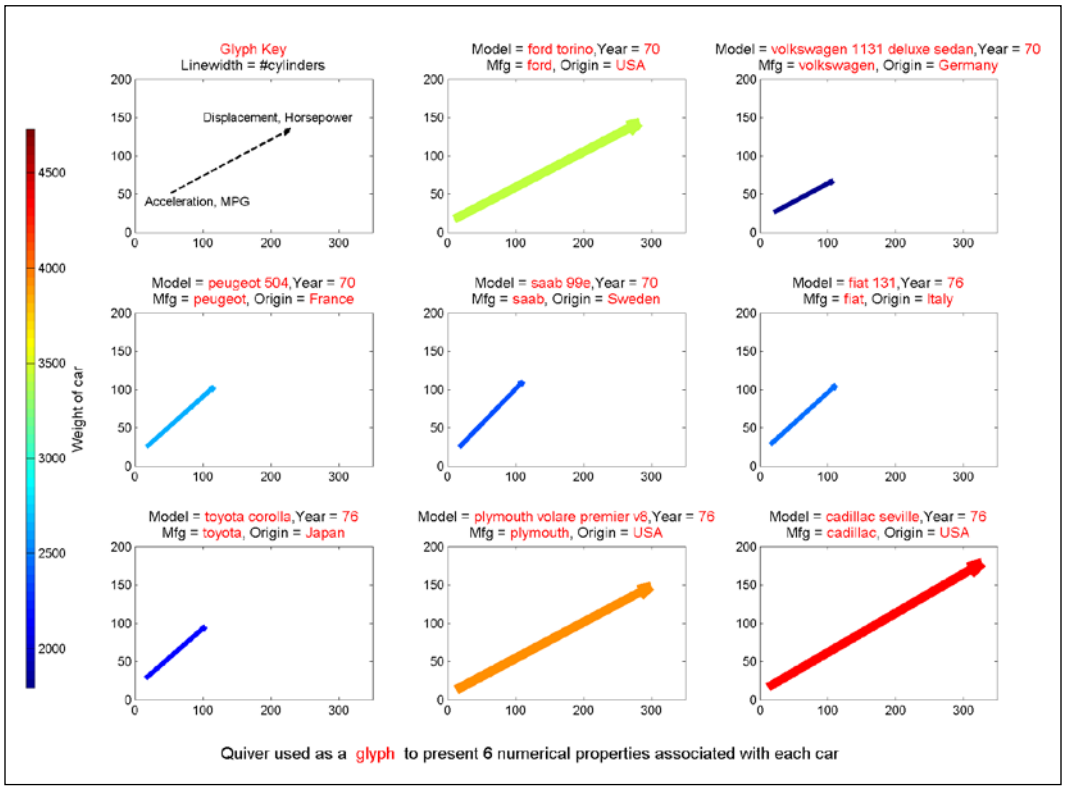
Perform the following steps:

1. Construct each arrow with the MATLAB command `quiver`, and change the appearance of the arrow based upon the following six numerical attribute values of the car data records: acceleration, displacement, MPG, cylinders, horsepower, and weight:

```
% layout the figure
figure('units','normalized','position',...
  [0.0010    0.1028    0.6443    0.8028]);

% Because you will use color as one of the graphical
% attributes to code a data dimension, set the climits
% to the min and max value of that data dimension
m = colormap;
climMat = [min(Weight) max(Weight)];

% choose which data points to display using the glyph
tryThese = [5 26 27 29 36 59 62 66];

% generate a graphic to act as the key to interpret the
% rest of the glyphs. Label the meaning of the start and
% end points of the glyph, the color, line width and
% title; specially mark it gray to make it stand out
subplot(3,3,1);
quiver(50,50,200,95,'Linewidth',2,'Color',...
  [0 0 0],'linestyle','--');
text(15,40,'Acceleration, MPG');
text(100,150,'Displacement, Horsepower');
xlim([0 350]);ylim([0 200]);
title({'\color{red}Glyph Key',['\color{black}'...
'Linewidth = #cylinders']},'fontsize',12);
  set(gca,'color',[.9 .9 .9]);

% plot the 8 chosen 6 dimensional data with the glyph
for ii = 1:8
  subplot(3,3,ii+1);set(gca,'clim',climMat); hold on;
  i = tryThese(ii); box on;
  index = fix((Weight(i)-climMat(1))/...
    (climMat(2)-climMat(1))*63)+1;
  quiver(Acceleration(i),MPG(i),Displacement(i),...
    Horsepower(i),'Linewidth',Cylinders(i),...
    'Color',m(index,:));
  title({['\color{black}Model = \color{red}']...
    [deblank(Model(i,:)) '\color{black},']...
```

```
    ['Year = \color{red}' num2str(Model_Year(i,:)) ],...
    ['\color{black}Mfg = \color{red}' deblank(Mfg(i,:))]...
    ['\color{black}, Origin = \color{red}']...
    [deblank(Origin(i,:))]},'fontsize',12);
  xlim([0 350]);ylim([0 200]);
end

% add a color bar and annotation
h=colorbar;
set(h,'position',[0.0323    0.1252    0.0073    0.7172]);
ylabel(h,'Weight of car','fontsize',12);
annotation('textbox','position',...
  [0.2013    0.0256    0.5756    0.0323],...
  'string',{['Quiver used as a  \color{red}glyph ']...
  ['\color{black} to present 6 numerical properties']...
  [' associated with each car']},...
  'fontsize',14,'linestyle','none');
```

The output is as follows:



How it works...

The command `quiver` was used to construct arrows that you used as a **glyph** to represent the six dimensions of the car data. The coordinates of the start and end of the arrow, the line thickness, and line color were used to encode information. Look at the glyph key in shaded plot position to read the corresponding glyphs. Clearly, the cars originating in USA have high horsepower and displacement values; they also have a heavier weight.

Glyphs are complicated to understand and a clear communication of the glyph key is essential in guiding the user to read a glyph. Also, note that while it allows you to see multiple dimensions at once, you are only looking at one data record at a time with a glyph. This makes high numbers of records difficult to see together. A careful choice of marker or glyph style may mitigate some of the perception challenge evident from the use of glyphs. Use techniques such as the Principal Component Analysis to rank factors or transform data into a more relevant space, so that the features with the greatest discriminatory power can be utilized to represent the samples.

There's more...

The `glyphplot` function is part of the MATLAB Statistics Toolbox. There are two types of markers that you can use with this command. One is a star and the other is a face.

A star plot represents each observation as a star whose ith spoke is proportional in length to the ith coordinate of that observation.

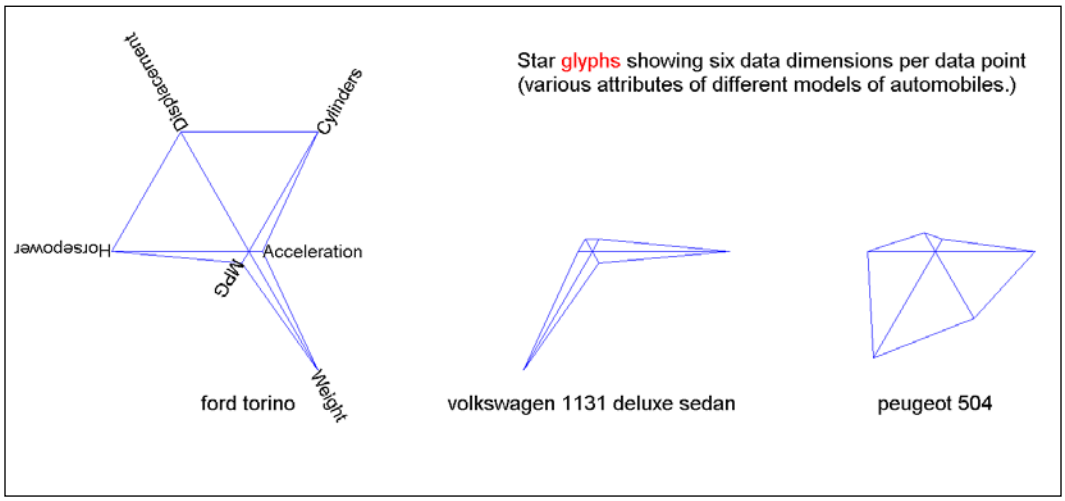
The command standardizes the range of values of the variables by shifting and scaling each column separately onto the interval [0, 1] before making the plot. It centers the glyphs on a rectangular grid that is as close to square as possible. `glyphplot` treats NaNs in the data vector as missing values and does not plot the corresponding rows.

Use `glyphplot` to make star plots:

```
%set up the figure
set(gcf,'units','normalized','position',...
  [ 0.3547    0.1306    0.5510    0.4041]);
set(gca,'position',...
  [ 0.0320    0.0035    0.9257    0.9965],'fontsize',12);

% ready the data
x = [Acceleration Cylinders Displacement Horsepower ...
  MPG Weight];
varLabels = {'Acceleration','Cylinders','Displacement',...
  'Horsepower','MPG','Weight'};

% the default glyph used by glybphplot is of type star
```

```
h = glyphplot(x(tryThese(1:3),:),'standardize','column',...
  'obslabels',deblank(Model(tryThese(1:3),:)),...
  'grid',[1 3]);

% get the xdata, ydata out from the set of handles returned
% by glyph plot
xdata=get(h(1,2),'XData');
ydata=get(h(1,2),'YData');

% to construct the glyph key, rotate each var label to
% align with the spokes of the star
r = [0 60 120 180 -120 -60];
for i = 1:6
  text(xdata(1,2+(i-1)*3),ydata(1,2+(i-1)*3),...
    varLabels{i},'rotation',r(i),'fontsize',12);
end

% set font size for the axis of each star
set(h(1,3),'Fontsize',14);
set(h(2,3),'Fontsize',14);
set(h(3,3),'Fontsize',14);

% set the view
axis off
```

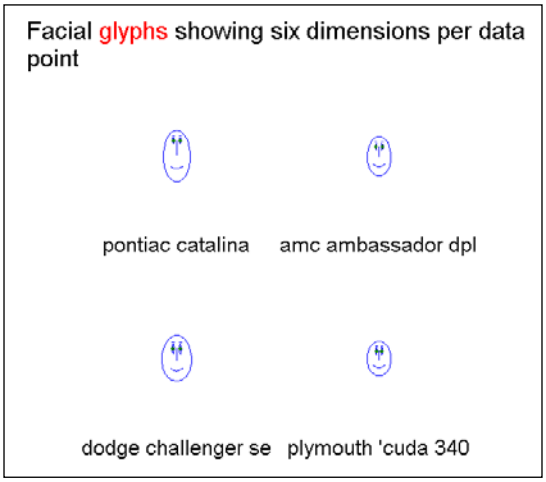
The output is as follows:



Next, use `glyphplot` to make face plots. A face plot represents each observation as a face, whose i^{th} facial feature is drawn with a characteristic proportional to the i^{th} coordinate of that observation. A total of 17 features of the face (described in the manual pages) can be used to encode 17 different dimensions of a data point. You will use the same data as before. This time, use a 2 x 3 grid layout and choose a specific page (such as page 3 here) for display:

```
glyphplot(x,'glyph','face',...
  'obslabels',Model,...
  'grid',[2 2],...
  'page',3);
```

The output is as follows (use table for the facial feature to data attribute mapping):



Facial feature	Data attribute
Size of face	Acceleration
Forehead/jaw relative arc length	Cylinders
Shape of forehead	Displacement
Shape of jaw	Horsepower
Width between eyes	MPG
Vertical position of eyes	Weight

Takeaways from this recipe:

- Use **glyph** plots to visualize high dimensional data. Choose dimensions for maximum discrimination by using techniques such as the PCA to transform the data ranked by their relative importance.

See also

Look up **MATLAB help** on the `glyphplot` command.

Parallel coordinates

Parallel coordinate plots are a popular way to visualize multi-dimensional numerical data. For n dimensions, consider n points along the xaxis. The y axis corresponds to the normalized range of values for each dimension. An n-dimensional data record is represented by joining the points (xi, yi) where xi = 1: n for the n dimensions, and yi is the normalized value of the ith dimension of that data point. If the range of values for all dimensions are similar, normalizing across all the records for each dimension is not necessary.

Getting ready

You will use the car dataset that comes with the MATLAB installation. It has attributes such as acceleration, number of cylinders, displacement, mileage, horsepower, and weight information for cars with different models and from different manufacturers:

```
% load the data
load carsmall
dat = [Acceleration Cylinders Displacement Horsepower ...
  MPG Weight];
```

How to do it...

You will construct the parallel coordinate plot using the function `parallelCoordPlot.m` that comes with this book. Perform the following steps:

1. Layout the figure:

```
figure('units','normalized','position',...
   [0.1214    0.2065    0.5234    0.6991]);
axes('position',[ .0587   .1755   .6913   .7530]);
```

2. Format the input parameters for the parallel coordinate plotting function:

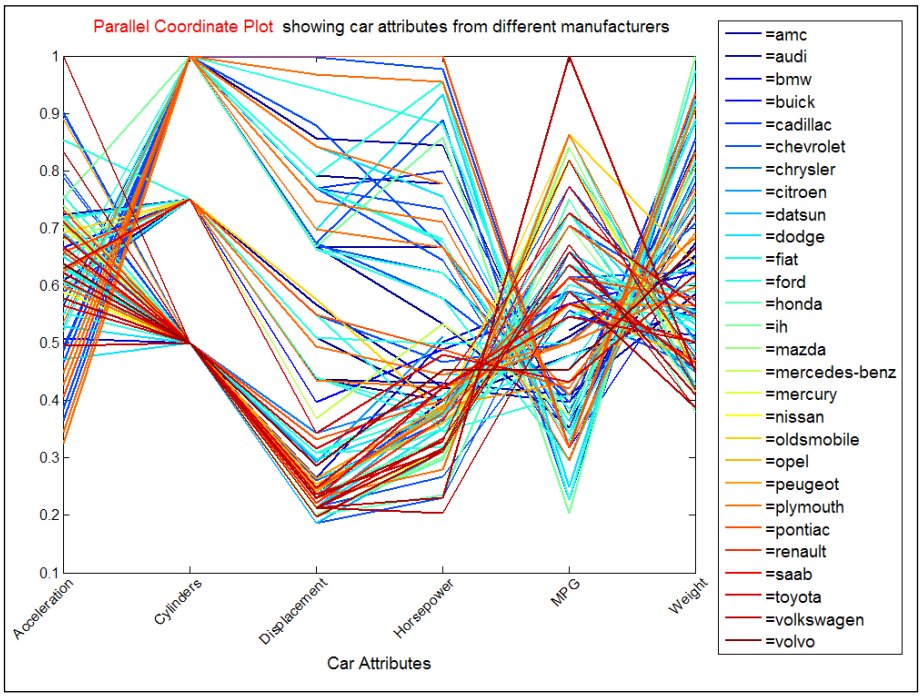
```
% need to normalize each data dimension
normalize=1;

% create the group Index array with the group labels
% of same size as the number of datapoints
for iiii=1:100
  hh{iiii} = deblank(Mfg(iiii,:));
end
```

```
yy=unique(hh);
for iiii=1:100
  groupIndex(iiii) = find(strcmp(yy,hh(iiii)));
end

% define x tick labels
labl = {'Acceleration','Cylinders','Displacement',...
  'Horsepower','MPG','Weight'};

% define grouping attribute labels
legendLabel=yy;
```

3. Make the function call and add annotations:

```
parallelCoordPlot(dat,normalize,groupIndex ,...
  labl,legendLabel);box on;
xlabel('Car Attributes','Fontsize',14);
title({' \color{red}Parallel Coordinate Plot '},...
  'Fontsize',14);
```

The output is as follows:



How it works...

Parallel coordinate plots allowed you to visualize the six numerical dimensions of each car record at the same time. The previous plot shows that more cylinders means worse mileage and the variance caused by the different models cannot make up for more cylinders. Also Toyota makes more fuel efficient cars with extra cylinder engines than Ford.

The function `parallelCoordPlot` used to build the parallel coordinate plots takes an m x n data matrix, with m records of n dimensions each. A `Boolean` input parameter `normalize` controls the normalization of each data dimension between 0 and 1. The parameter `groupIndex` can be the index of the variable you want to group by, for coloring purposes (the value ranging between 1 to n), or you can provide an array of size m x 1 with the group number for each of the m entries (in this case, unique values of group number labels will range from 1 to k for k groups in the data). In this example, you provided `groupIndex` as an array of the same size as the number of data points, containing the labels for the grouping attribute, a unique number representing the name of the manufacturer. This group index drives the line coloring. The input parameter `labl` is an array of strings with variable names (data dimension names) to be used as x tick labels. The input parameter `legendLabel` is an array of strings with labels to use for legend entries corresponding to the desired grouping. The last argument, `colormapCustom` is optional and takes a custom color map definition.

To enable grouping of legend entries, such that one legend entry corresponds to multiple individually constructed lines, you do the following:

1. For each group, you set the `IconDisplayStyle` property to `off` for all sets of lines drawn, except for those between the $(n-1)^{th}$ and n^{th} variable, for which you set the `IconDisplayStyle` property to `on`.
2. Each such set of lines with the`IconDisplayStyle` property `on` is then assigned to an `hggroup` object.
3. The `IconDisplayStyle` property is set to `on` for these `hggroup` objects.
4. The `legend` command would then just add one entry per `hggroup` object.

Takeaways from this recipe:

- Use **parallel coordinate plots** to visualize high dimensional data. Normalize each data dimension between 0 and 1 (if they vary largely in magnitude) in order to make the comparison easy

See also

Look up **MATLAB help** on the `hggroup` and `legend` commands. Check out `parallelcoord`, part of the statistics toolbox, for constructing parallel coordinate plots with a single MATLAB command.

Tree maps

Although area-based visualizations are perceptually more challenging than those based on comparing lines, they are a compact way to look at high dimensional data. **Tree maps**, which is the type of display you will use in this recipe, displays hierarchical (tree-structured) data as a set of nested rectangles. Each branch is a rectangle, which is then tiled with smaller rectangles representing subbranches. Many tiling algorithms exist as options for doing this. Often the color of the leaf node is used to code a separate dimension of the data. Several algorithms exist to organize the data for this type of a presentation and for ensuring that the area units are approximately square. But a trade-off must be made between the aspect ratio and the order of placement of the records. If the order is emphasized, the aspect ratio is degraded. This recipe is adopted from the submission by Joe Hicklin on MATLAB Central File Exchange.

Getting ready

This recipe shows the Gross Domestic Product (in millions of US dollars) across 100 countries. The data was obtained from OpenData at `www.socrata.com`. Load the data:

```
gdp =xlsread('GDP_By_Country_And_Continent_Treemap.csv',...
  'C2:C101');
[blah, labels] = ...
  xlsread('GDP_By_Country_And_Continent_Treemap.csv',...
  'a2:a101');
```

How to do it...

Perform the following steps:

1. Set the color scale:

```
m = colormap;

%set the color limits
climMat=[min(log(gdp)) max(log(gdp))];
set(gca,'clim',climMat);

% query the indices into the color map corresponding to
% the data values
index = fix((log(gdp)-climMat(1))/...
  (climMat(2)-climMat(1))*63)+1;
```

2. Compute the tree map:

```
r = treemap(gdp);
```

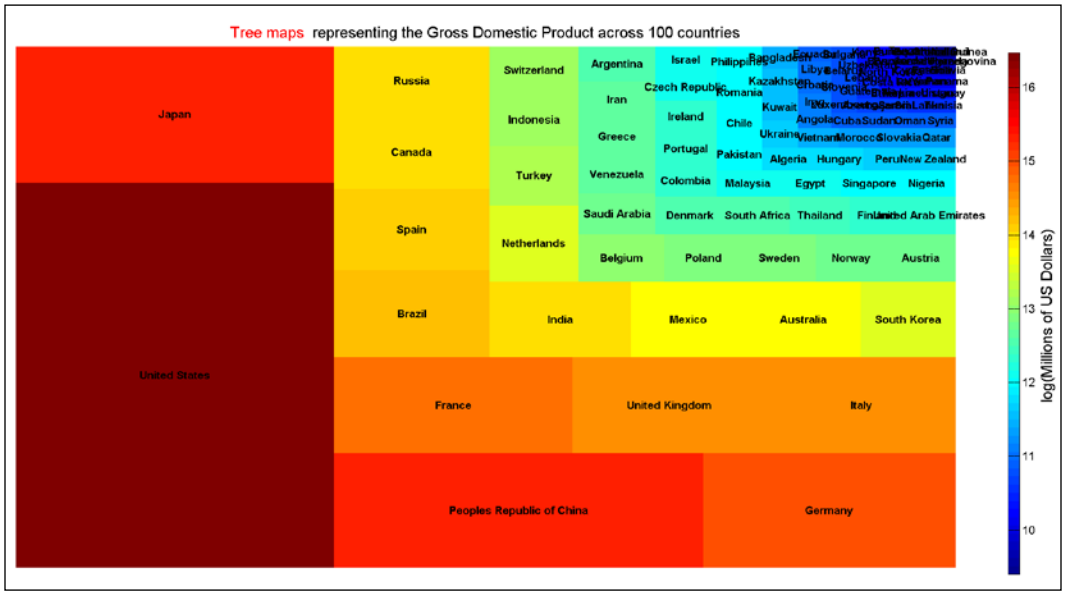
3. Plot the results:

```
plotRectangles(r,labels,m(index,:));
```

4. Add annotations:

```
h=colorbar('position',[.95 .03 .01 .89]);
ylabel(h,'log(Millions of US Dollars)','fontsize',14);
title('\color{red}Tree maps','fontsize',14);
```

The output is as follows:



How it works...

Tree maps offer a space constrained method of visualizing large volumes of data. The figure shows 100 countries with the area representing the GDP in millions of US dollars and the color representing the log of the GDP values in millions of US dollars.

The tree map algorithm is adopted from Hicklin's submission on MATLAB File Exchange. It works as follows:

1. The data is sorted from the largest to the smallest.
2. Then a row or column is laid out along the shortest side (w or h) of the remaining rectangle.

3. Blocks are added to this new row or column until adding a new block would worsen the average aspect ratio of the blocks in the row or column.
4. Once this row or column is laid out, they recurse on the remaining blocks and the remaining rectangle.

Takeaways from this recipe:

- Use tree plots as a compact option to look at high volume of data. Be aware that area and color based visualizations pose an increased perceptual challenge

See also

Look up **MATLAB help** on the `treemap` and `plotrectangles` commands.

Andrews' curves

The idea of coding and representing multivariate data by curves was suggested by Andrews in 1972. Each multivariate observation *Xi = (Xi,1... Xi,p)* is transformed into a curve, such that the observations represent the coefficients of a Fourier series for *t Є [0, 1]*.

$$f_i(t) = \frac{X_{i,1}}{\sqrt{2}} + X_{i,2}\sin(2\pi t) + X_{i,3}\cos(2\pi t) + \cdots$$

Getting ready

You will use the Fisher iris dataset that is part of the MATLAB installation. Load the data:

```
load fisheriris
```

How to do it...

Perform the following steps:

1. Call the function to generate the curves:

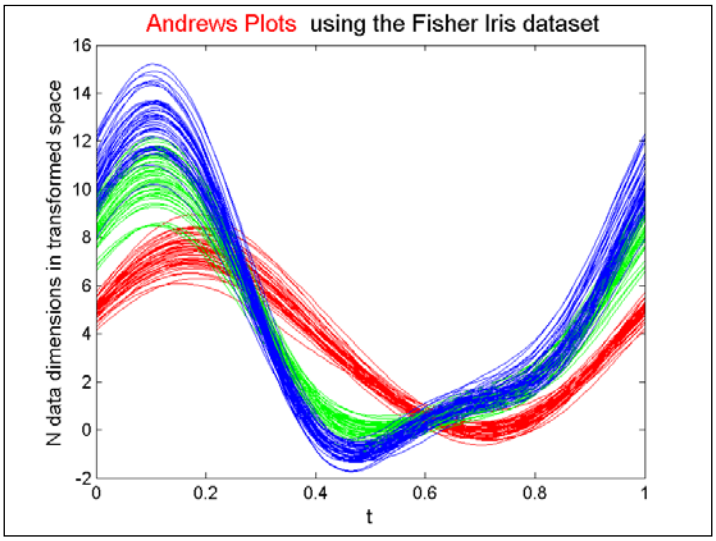
```
[t curves] = andrewsCurves(meas);
```

2. Group by the class labels in your plot to demonstrate that different class labels correspond to different curve clusters:

```
groupIndex = unique(species);
colors = [1 0 0; 0 1 0; 0 0 1];
hold on;
```

```
for i = 1:length(groupIndex)
  idx = find(strcmp(species,groupIndex{i}));
  plot(t,curves(idx,:)','color',colors(i,:));
end
```

The output is as follows:



How it works...

The Fisher iris dataset has three classes, one of which is more easily separated than the other two. This property clearly holds true in the Andrews' curve visualization space. Note that the order of variables is important. A different order may not render the classes as distinct in this visualization space. It is recommended that you use a technique such as the **Principal Component Analysis** (**PCA**) to understand how to order the variables for maximal impact (in terms of bringing out the similarities and differences) between the data points.

Takeaways from this recipe:

- Use **Andrews' curve** plots to transform higher dimensional data into the Fourier series space for visualization
- Use techniques such as the Principal Component Analysis (PCA) to understand how to order the variables for constructing the Andrews' curves for maximal differentiation

See also

Check out `andrewsplot.m`, part of the statistics toolbox, as an alternative method for constructing Andrews' curves.

Downsampling for fast graphs

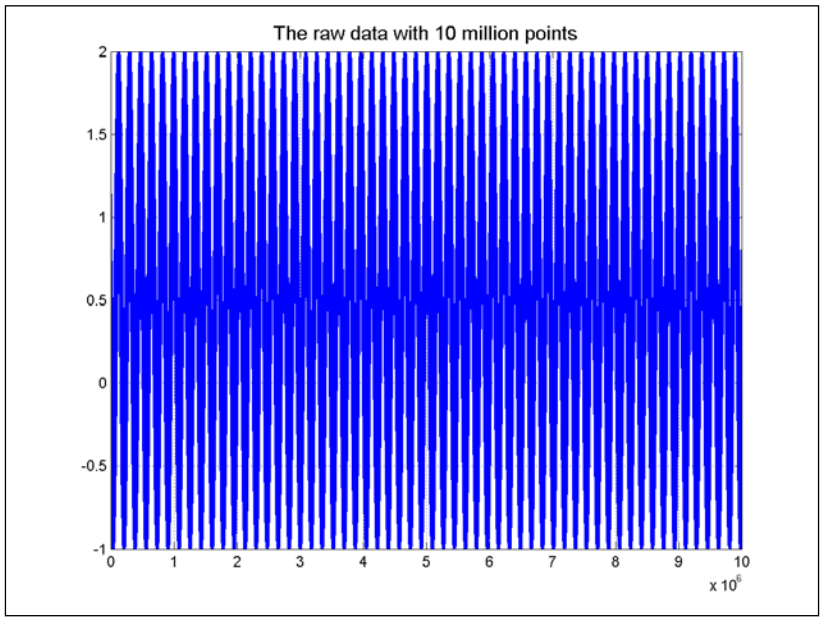
Visually, it is difficult to perceive beyond a certain data density. This recipe illustrates this point and recommends you to consider **downsampling** your data before you visualize to obtain faster rendering.

Getting ready

In this recipe, you will visualize an enormous time series dataset by subsampling or downsampling the data. The data will be binned and an average or median of the data points in the bin will be used to represent the series. Load the raw data and plot it:

```
dat = rand(10e6,1) + cos(linspace(0,360,10e6)' ...
  + exp(rand(10e6,1)));
plot(dat); box on; grid on;
title({'The raw data with 10 million points'},...
  'Fontsize',14);
```

Here is how the original looks, and takes circa .5 seconds to render:

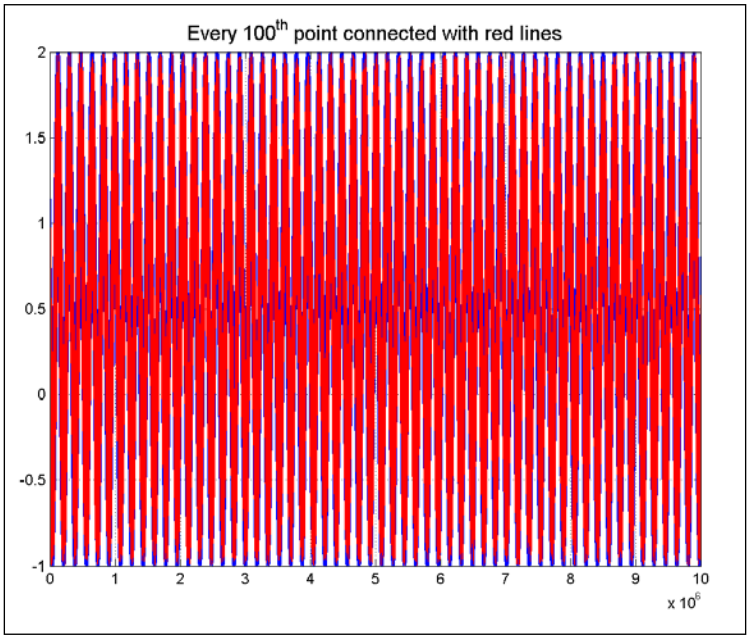


How to do it...

1. Down-sample and draw the new series with connecting lines:

```
plot(1:100:10e6,dat(1:100:end),'r-');
```

   Here is how the plot looks and takes circa .01 seconds to render:



2. Bin the data and use a central representative from each bin to draw the data. (Note that the average as the representative point may not always be a suitable strategy. Here, the point of the recipe is to illustrate that a representative point is desirable, rather than the high number of points that make the graphic slow but does not add to the information content of the graph.):

```
plot(dat); box on; grid on;cnt = 1;cnt1=1;
while cnt+100<10e6
  datbin(cnt1) = mean(dat(cnt:cnt+100));
  cnt = cnt + 100;
  cnt1 = cnt1 + 1;
end
plot(1:cnt1-1,datbin,'r'); box on; grid on
```

Here is how the plot looks and takes circa .02 seconds to render:

How it works...

This recipe shows using a representative point in a high data density situation captures most of the features of the dataset. All 10 million points are not really necessary to observe the variation present in this dataset. Note that subsampling at steps higher than the inverse of the highest frequency component in the data will result in a loss of information (as it will violate the Nyquist sampling theory). However, for most practical purposes, a sufficient downsampling interval should be investigated to make the visualization faster and more responsive to interaction with the graphics.

Takeaways from this recipe:

- Use the minimum number of sample points to convey the information contained in your data to obtain the fastest rendering

See also

Look up **Nyquist Sampling Theory** for more information on allowed limits on downsampling.

Principal Component Analysis

Principal Component Analysis (**PCA**) uses an orthogonal transformation to convert a set of observations of one plus variables into a set of values of linearly uncorrelatedvariables called principal components. It offers an excellent way to select the most relevant data dimension for further analysis and visualization.

Getting ready

You will be looking at a dataset with records of the metallic composition of glass from a variety of sources. This was downloaded from the UCI Machine learning database. Prepare the data as follows:

```
fid = fopen('GLASS.txt');
C = textscan(fid, '%d,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f');
fclose(fid);

% arrange relevant data into a matrix format,
% first position ignored as it is a positional index
data = reshape(cell2mat({C{2:11}}'),214,10);
```

How to do it...

Perform the following steps:

1. Layout the figure:

```
set(gcf,'units','normalized','position',...
[.21 .22 .43 .69]); hold on;
```

2. Perform PCA:

```
[coeff, score, latent, tsquared] = ...
  princomp(data(:,1:9));
```

3. Prepare the labeling for your legend:

```
glassTypes = [1 2 3 5 6 7];
glassTypesStr = {'building windows _{float processed}',...
  'building windows_{non-float processed}',...
  'vehicle windows_{float processed}',...
  'vehicle_{windows_non_float_processed}',...
  'containers','tableware', 'headlamps'};
```

4. Select a color palette to color each marker by glass type:

```
colors = colormap;
colors = ...
  colors(round(linspace(1,64,length(glassTypes))),:);
```

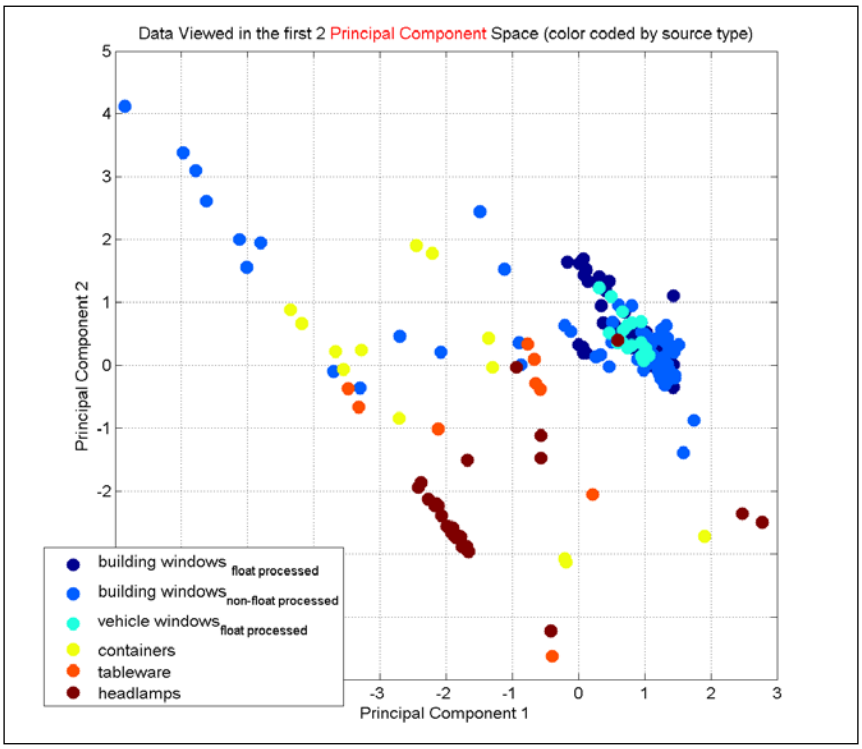
5. Investigate the data in the space of the first two components:

```
for i = 1:length(glassTypes)
  idx=find(data(:,10)==glassTypes(i));
  plot(score(idx,1),score(idx,2),'.',...
    'Markersize',30,'Color',colors(i,:));
end
box on; grid on;
```

6. Add annotation:

```
legend(glassTypesStr(glassTypes),'position',...
  [0.0506    0.0756    0.3429    0.2066]);
set(gca,'Fontsize',12);
xlabel('Principal Component 1','Fontsize',12);
ylabel('Principal Component 2','Fontsize',12);
title('Data Viewed in the first 2 PC Space','Fontsize',12);
```

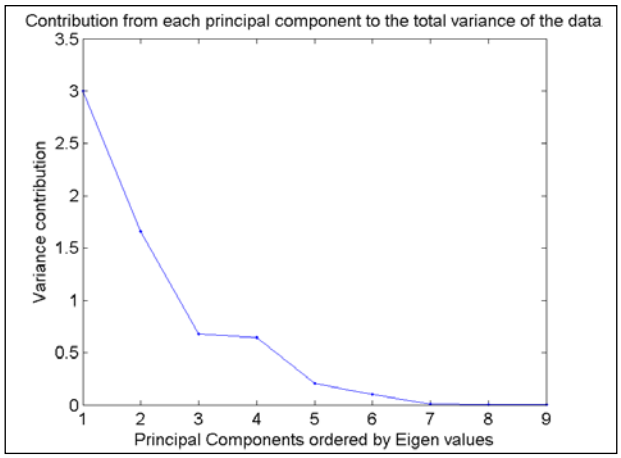
The output is as follows:



7. Assess the contribution from each principal component to the total variance in the dataset:

```
plot(latent,'.-');
title(['Contribution from each principal component to' ...
  'the total variance of the data'],'Fontsize', 12);
xlabel(['Principal Components ordered by Eigen'...
  'values'],'Fontsize',12);
ylabel('Variance contribution','Fontsize',12);
```

   The output is as follows:



How it works...

The previous output shows that at least the first four components of this dataset contribute significantly to the total variance in the dataset. This implies that the prior output you constructed with the first two components is not really sufficient to segregate the data into the self-contained groups. This is borne out by the mixed population of the clusters shown in that output.

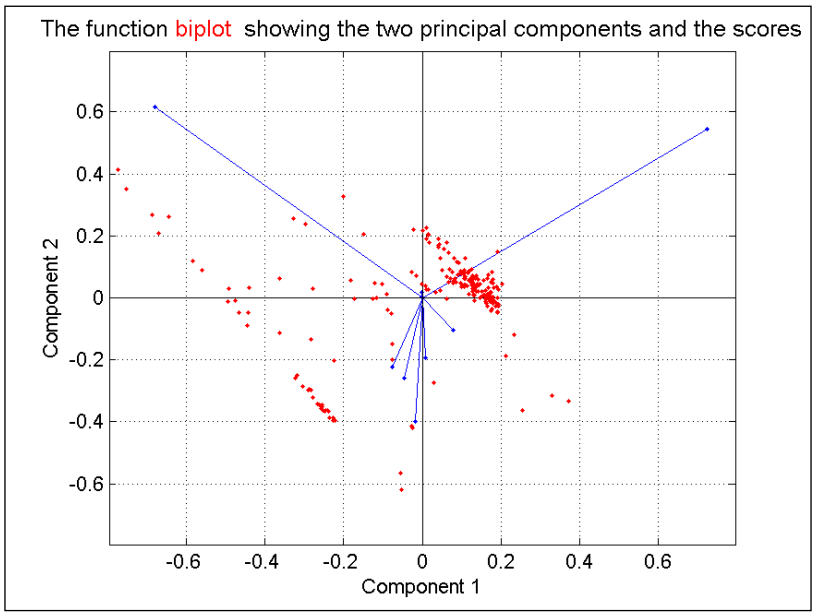
Principal Component Analysis (PCA) uses an orthogonal transformation to convert a set of observations of one plus variables into a set of values of linearly uncorrelated variables called principal components. The number of principal components is less than or equal to the number of original variables. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component in turn has the highest variance possible under the constraint that it be orthogonal to the preceding components. Principal components are guaranteed to be independent only if the dataset is jointly, normally distributed. PCA is sensitive to the relative scaling of the variables. The main advantage of using a PCA is that it offers a way to select the most relevant and economical set of dimensions to represent your data for further analysis and visualization.

There's more...

MATLAB provides the function `biplot` for visualizing the scores along with the principal component coefficients as follows:

```
biplot(coeff(:,1:2),'Scores', score(:,1:2));
```

The output is as follows:



`biplot` allows you to visualize the magnitude and sign of each variable's contribution to the first two (or three) principal components, and how each observation is represented in terms of those components. `biplot` imposes a sign convention, forcing the element with the largest magnitude in each column of `coeffs` to be positive. This flips some of the vectors in `coeffs` to the opposite direction, but often makes the plot easier to read. Interpretation of the plot is unaffected, because changing the sign of a coefficient vector does not change its meaning.

Takeaways from this recipe:

- Use Principal Component Analysis to transform your data to facilitate visualization in terms of the most impactful few variables

See also

Look up **MATLAB help** on the `princomp` and `biplot` commands.

Radial Coordinate Visualization

Radial Coordinate Visualization maps *m* dimensional points to a 2D space using a nonlinear mapping. The idea is to consider a point in 2D space connected to *m* equally spaced points on a circle with springs. Now, each of the *m* dimensions of the data point is considered to be the spring constant for the corresponding spring. If the central data point is allowed to move and reach equilibrium position, this would then be the mapping of the *m* dimensional data points onto 2D space. In order to determine the location of the data point, the sum of the spring forces needs to equal zero.

Getting ready

You will use the cancer gene expression dataset that is provided as part of this book that constitutes 198 samples with gene expression levels from 16063 genes. This data was downloaded from the machine learning data repository maintained by the Department of Statistics at Stanford University. Load the dataset, and put the test and train data together into one dataset:

```
load 14cancer.mat
data = [Xtest; Xtrain];
```

How to do it...

Perform the following steps:

1. Layout the figure:

```
set(gcf,'units','normalized','position',...
  [.30 .35 .35 .55]);
```

2. Normalize the data per variable between 0 and 1:

```
for i = 1:size(data,2)
  data(:,i) = data(:,i)./range(data(:,i));
end
```

3. Calculate the Radial Coordinate Visualization:

```
[uxuy] = radviz(data);
```

4. Select a color palette to color each marker by cancer type:

```
colors = colormap;
colors = colors(round(linspace(1,64,14)),:);
```

5. Plot the 2D projection data with marker color determined by cancer type:

```
hold on;
for i = 1:14
  mm=find([ytestLabelsytestLabels]==i);
  plot(ux(mm),uy(mm),'.','Markersize',30,'color',...
    colors(i,:));
end
```

6. Plot a radial border:

```
plot(cos((pi/180)*linspace(0,360,361)),...
  sin((pi/180)*linspace(0,360,361)),'.','markersize',2);
```

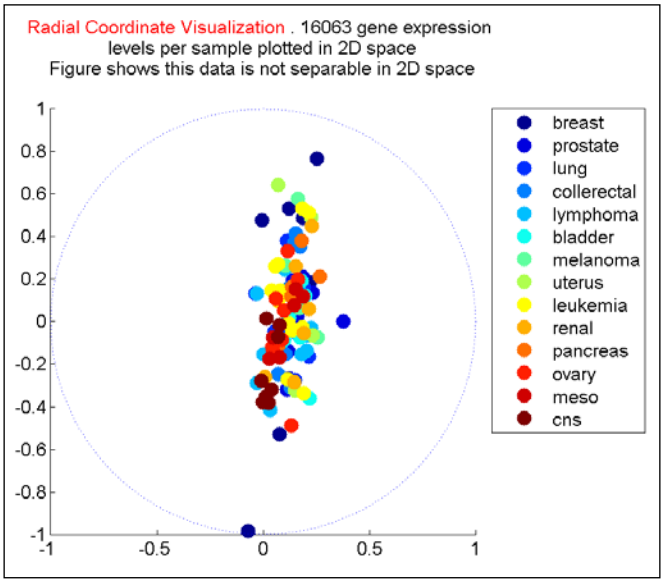
7. Add annotation:

```
legend(strtrim(classLabels),'location','bestoutside');
set(gca,'Fontsize',12);
title({['\color{red}Radial Coordinate Visualization'...
  '\color{black}. 16063 gene expression '],...
  'levels per sample plotted in 2D space',...
  ['Figure shows this data is not separable in'...
  '2D space'],''},'Fontsize',12);
```

8. Set the view:

```
axis equal square
set(gcf,'color',[1 1 1],'paperpositionmode','auto');
```

The output is as follows:



How it works...

The figure shows that the 198 samples with 16063 dimensions are not separable in 2D space. Note that for m variables, there are effectively *(m-1)! / 2* possible projections and not just one. The choice of a projection can be made dependent on maximizing the variance of the data points in 2D space (such that there is maximal separation between objects). Or, one can use one of the nearest-neighbor distance based criteria. Note that the number of options to search for optimization increases very rapidly with m. Clearly investigation into optimization heuristics for this problem is necessary for applying to cases with large m. The drawback of this visualization is the over plotting problem. However, as indicated above, a suitable criterion for selecting one projection over the other can help this scheme to become more revealing.

The function `radviz.m` is provided for you to adopt this visualization for your dataset.

Takeaways from this recipe:

- Use Radial Coordinate Visualization to project high dimensional data onto 2D (or 3D) space

See also

Look up projection pursuit, independent component analysis, targeted projection pursuit, and grand tour as alternate data reduction strategies for visualization.

7

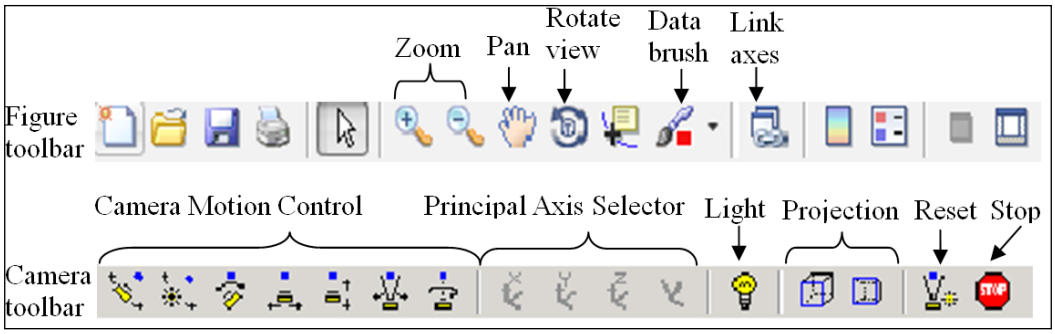
Creating Interactive Graphics and Animation

In this chapter, we will cover:

- Callback functions
- Obtaining user input from the graph
- Linked axes and data brushing
- The magnifying glass demo
- Animation with playback of frame captures
- Stream particle animation
- Animation by incremental changes to chart elements

Introduction

This chapter showcases MATLAB's capabilities for creating **interactive graphics** and **animations**. A static graphic is essentially two dimensional. The ability to rotate the axes and change the view, add annotations in real time, delete data, and zoom in or zoom out adds significantly to the user experience, as the brain is able to process and see more from that interaction (see *Appendix*, *References* for related literature). MATLAB supports interactivity with the standard zoom, pan features, a powerful set of camera tools to change the data view, data brushing, and axes linking. The set of functionalities accessible from the figure and camera toolbars are outlined briefly as follows:



The steps of interactive exploration can also be recorded and presented as an animation. This is very useful to demonstrate the evolution of the data in time or space or along any dimension where sequence has meaning. In *Chapter 5*, *Playing in the Big Leagues with Three-dimensional Data Displays*, you programmatically moved the figure camera in a series of steps creating an animation with a sequence of different data views which proved a very effective method for data exploration. Note that some recipes in this chapter may require you to run the code from the source code files as a whole unit because they were developed as functions. As functions, they are not independently interpretable using the separate code blocks corresponding to each step.

Callback functions

A mouse drag movement from the top-left corner to bottom-right corner is commonly used for zooming in or selecting a group of objects. You can also program a custom behavior to such an interaction event, by using a **callback function**. When a specific event occurs (for example, you click on a push button or double-click with your mouse), the corresponding callback function executes. Many event properties of graphics handle objects can be used to define callback functions.

In *Chapter 5*, *Playing in the Big Leagues with Three-dimensional Data Displays*, in two recipes, namely, *Slice (cross sectional views)* and *Isosurface, Isonormals, and Isocaps*, you used a slider element to get input from the user on where to create the slice or an isosurface for 3D exploration. In this recipe, you will write callback functions which are essential to implement this kind of behavior. You will also see options available to share data between the calling and callback functions.

Getting started

In *Chapter 6*, *Designing for Higher Data Dimensions*, a graphic was designed that displayed numerical data within their spatial context. That graphic is extended in this example by allowing the user to interact with the data by selecting the data to be displayed from a custom menu item.

Load the dataset. Split the data into two main sets—`userdataA` is a structure with variables related to the demographics and `userdataB` is a structure with variables related to the Income Groups. Now create a nested structure with these two data structures as shown in the following code snippet:

```
load customCountyData
userdataA.demgraphics = demgraphics;
userdataA.lege = lege;
userdataB.incomeGroups = incomeGroups;
userdataB.crimeRateLI = crimeRateLI;
userdataB.crimeRateHI = crimeRateHI;
userdataB.crimeRateMI = crimeRateMI;
userdataB.AverageSATScoresLI = AverageSATScoresLI;
userdataB.AverageSATScoresMI = AverageSATScoresMI;
userdataB.AverageSATScoresHI = AverageSATScoresHI;
userdataB.icleg = icleg;
userdataAB.years = years;
userdataAB.userdataA = userdataA;
userdataAB.userdataB = userdataB;
```

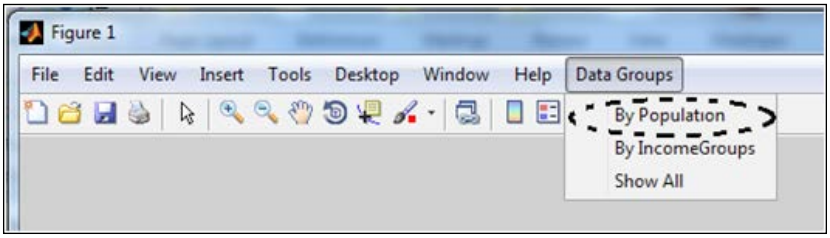
How to do it...

Perform the following steps:

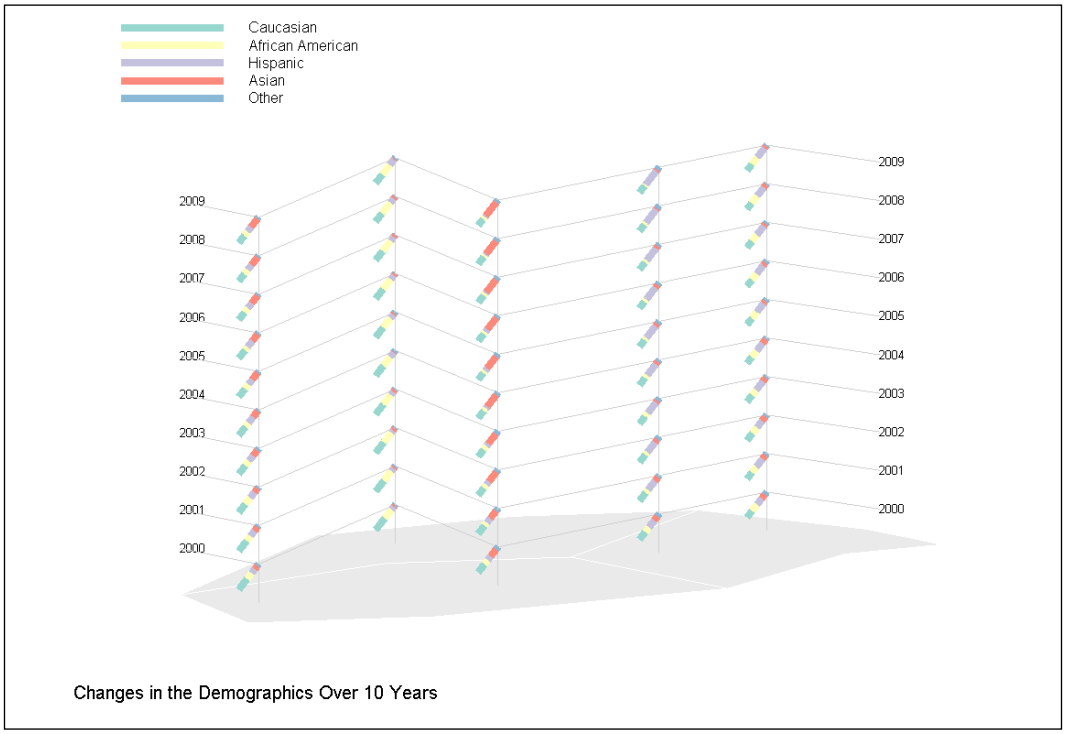
1. Run this as a function at the console:

   `c3165_07_01_callback_functions`

2. A figure is brought up with a non-standard menu item as highlighted in the following screenshot. Select the **By Population** item:



Here is the resultant figure:



3. Continue to explore the other options to fully exercise the interactivity built into this graphic.

How it works...

The function `c3165_07_01_callback_functions` works as follows:

A custom menu item **Data Groups** is created, with additional submenu items—**By population**, **By Income Groups**, or **Show all**.

```
% add main menu item
f = uimenu('Label','Data Groups');

% add sub menu items with additional parameters
uimenu(f,'Label','By  Population','Callback','showData',...
  'tag','demographics','userdata',userdataAB);
uimenu(f,'Label','By IncomeGroups',...
```

```
    'Callback','showData','tag','IncomeGroups',...
    'userdata',userdataAB);
uimenu(f,'Label','ShowAll','Callback','showData',...
    'tag','together','userdata',userdataAB);
```

You defined the `tag` name and the `callback` function for each submenu item above. Having a tag name makes it easier to use the same callback function with multiple objects because you can query the tag name to find out which object initiated the call to the callback function (if you need that information). In this example, the callback function behavior is dependent upon which submenu item was selected. So the tag property allowed you to use the single function `showData` as callback for all three submenu items and still implement submenu item specific behavior. Alternately, you could also register three different callback functions and use no tag names.

You can specify the value of a callback property in three ways. Here, you gave it a function handle. Alternately, you can supply a string that is a MATLAB command that executes when the callback is invoked. Or, a cell array with the function handle and additional arguments as you will see in the next section.

For passing data between the calling and callback function, you also have three options. Here, you set the `userdata` property to the variable name that has the data needed by the `callback` function. Note that the `userdata` is just one variable and you passed a complicated data structure as `userdata` to effectively pass multiple values. The user data can be extracted from within the callback function of the object or menu item whose callback is executing as follows:

```
userdata = get(gcbo,'userdata');
```

The second alternative to pass data to callback functions is by means of the `application data`. This does not require you to build a complicated data structure. Depending on how much data you need to pass, this later option may be the faster mechanism. It also has the advantage that the `userdata` space cannot inadvertently get overwritten by some other function. Use the `setappdata` function to pass multiple variables. In this recipe, you maintained the main drawing area axis handles and the custom legend axis handles as application data.

```
setappdata(gcf,'mainAxes',[]);
setappdata(gcf,'labelAxes',[]);
```

This was retrieved each time within the executing callback functions, to clear the graphic as new choices are selected by the user from the custom menu.

```
mainAxesHandle = getappdata(gcf,'mainAxes');
labelAxesHandles = getappdata(gcf,'labelAxes');
if ~isempty(mainAxesHandle),
  cla(mainAxesHandle);
  [mainAxesHandle, x, y, ci, cd] = ...
```

```
      redrawGrid(userdata.years, mainAxesHandle);
else
  [mainAxesHandle, x, y, ci, cd] = ...
    redrawGrid(userdata.years);
end

if ~isempty(labelAxesHandles)
  for ij = 1:length(labelAxesHandles)
    cla(labelAxesHandles(ij));
  end
end
```

The third option to pass data to callback functions is at the time of defining the callback property, where you can supply a cell array with the function handle and additional arguments as you will see in the next section. These are local copies of data passed onto the function and will not affect the global values of the variables.

The `callback` function `showData` is given below. Functions that you want to use as function handle callbacks must define at least two input arguments in the function definition: the handle of the object generating the callback (the source of the event), the event data structure (can be empty for some callbacks).

```
function showData(src, evt)

userdata = get(gcbo,'userdata');

if strcmp(get(gcbo,'tag'),'demographics')
  % Call grid  f drawing code block
  % Call showDemographics with relevant inputs
elseif strcmp(get(gcbo,'tag'),'IncomeGroups')
  % Call grid drawing code block
  % Call showIncomeGroups with relevant inputs
else
  % Call grid drawing code block
  % Call showDemographics with relevant inputs
  % Call showIncomeGroups with relevant inputs
end

function labelAxesHandle = ...
    showDemographics(userdata, mainAxesHandle, x, y, cd)
  % Function specific code
end

function labelAxesHandle = ...
    showIncomeGroups(userdata, mainAxesHandle, x, y, ci)
```

```
  % Function specific code
end


function [mainAxesHandle x y ci cd] = ...
    redrawGrid(years, mainAxesHandle)
  % Grid drawing function specific code
end

end
```

Explanations on the details of the actual plotting of the data are the same as in recipe in *Chapter 6*, *Survey Plots*, where the static version of this recipe is presented.

There's more...

This section demonstrates the third option to pass data to callback functions by supplying a cell array with the function handle and additional arguments at the time of defining the callback property. Add a fourth submenu item as follows (uncomment line 45 of the source code):

```
uimenu(f,'Label',...
  'Alternative way to pass data to callback',...
  'Callback',{@showData1,userdataAB},'tag','blah');
```

Define the `showData1` function as follows (uncomment lines 49 to 51 of the source code):

```
function showData1(src, evt, arg1)
  disp(arg1.years);
end
```

Execute the function and see that the value of the `years` variable are displayed at the MATLAB console when you select the last submenu **Alternative way to pass data to callback** option.

Takeaways from this recipe:

- Use callback functions to **define custom responses for each user interaction** with your graphic
- Use one of the three options for sharing data between calling and callback functions—pass data as arguments with the callback definition, or via the user data space, or via the application data space, as appropriate

See also

Look up **MATLAB help** on the `setappdata`, `getappdata`, `userdata` property, `callback` property, and `uimenu` commands.

Obtaining user input from the graph

User input may be desired for annotating data in terms of adding a label to one or more data points, or allowing user settable boundary definitions on the graphic. This recipe illustrates how to use MATLAB to support these needs.

Getting started

The recipe shows a two-dimensional dataset of intensity values obtained from two different dye fluorescence readings. There are some clearly identifiable clusters of points in this 2D space. The user is allowed to draw boundaries to group points and identify these clusters. Load the data:

```
load clusterInteractivData
```

The `imellipse` function from the **MATLAB image processing toolbox™** is used in this recipe. Trial downloads are available from their website.

How to do it...

The function constitutes the following steps:

1. Set up the user data variables to share the data between the callback functions of the push button elements in this graph:

```
userdata.symbChoice = {'+','x','o','s','^'};
userdata.boundDef = [];
userdata.X = X;
userdata.Y = Y;
userdata.Calls = ones(size(X));
set(gcf,'userdata',userdata);
```

2. Make the initial plot of the data:

```
plot(userdata.X,userdata.Y,'k.','Markersize',18);
hold on;
```

3. Add the push button elements to the graphic:

```
uicontrol('style','pushbutton',...
  'string','Add cluster boundaries?', ...
```

```
  'Callback',@addBound, ...
  'Position', [10 21 250 20],'fontsize',12);
uicontrol('style','pushbutton', ...
  'string','Classify', ...
  'Callback',@classifyPts, ...
  'Position', [270 21 100 20],'fontsize',12);
uicontrol('style','pushbutton', ...
  'string','Clear Boundaries', ...
  'Callback',@clearBounds, ...
  'Position', [380 21 150 20],'fontsize',12);
```

4. Define `callback` for each of the pushbutton elements. The `addBound` function is for defining the cluster boundaries. The steps are as follows:

```
% Retrieve the userdata data
userdata = get(gcf,'userdata');

% Allow a maximum of four cluster boundary definitions
if length(userdata.boundDef)>4
  msgbox('A maximum of four clusters allowed!');
  return;
end

% Allow user to define a bounding curve
h=imellipse(gca);

% The boundary definition is added to a cell array with
% each element of the array storing the boundary def.
userdata.boundDef{length(userdata.boundDef)+1} = ...
  h.getPosition;
set(gcf,'userdata',userdata);
```

5. The `classifyPts` function draws points enclosed in a given boundary with a unique symbol per boundary definition. The logic used in this classification function is simple and will run into difficulties with complex boundary definitions. However, that is ignored as that is not the focus of this recipe. Here, first find points whose coordinates lie in the range defined by the coordinates of the boundary definition. Then, assign a unique symbol to all points within that boundary:

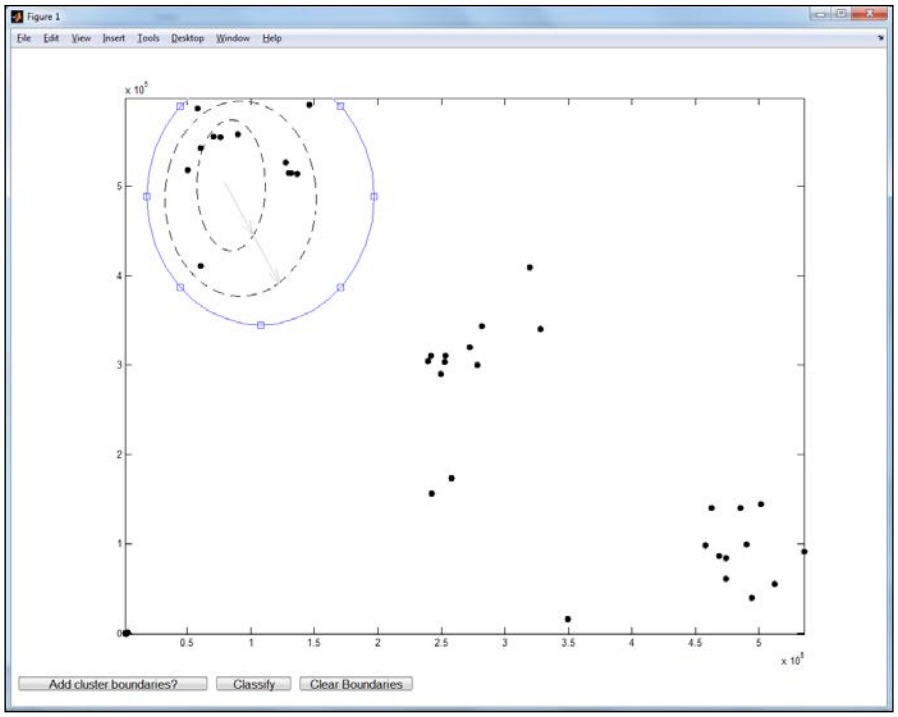
```
for i = 1:length(userdata.boundDef)
  pts = ...
  find( (userdata.X>(userdata.boundDef{i}(:,1)))& ...
    (userdata.X<(userdata.boundDef{i}(:,1)+ ...
    userdata.boundDef{i}(:,3))) &...
    (userdata.Y>(userdata.boundDef{i}(:,2)))& ...
    (userdata.Y<(userdata.boundDef{i}(:,2)+ ...
```

```
    userdata.boundDef{i}(:,4))));
  userdata.Calls(pts) = i;
  plot(userdata.X(pts),userdata.Y(pts), ...
    [userdata.colorChoice{i} '.'], ...
    'Markersize',18); hold on;
end
```

6. The `clearBounds` function clears the drawn boundaries and removes the clustering based upon those boundary definitions.

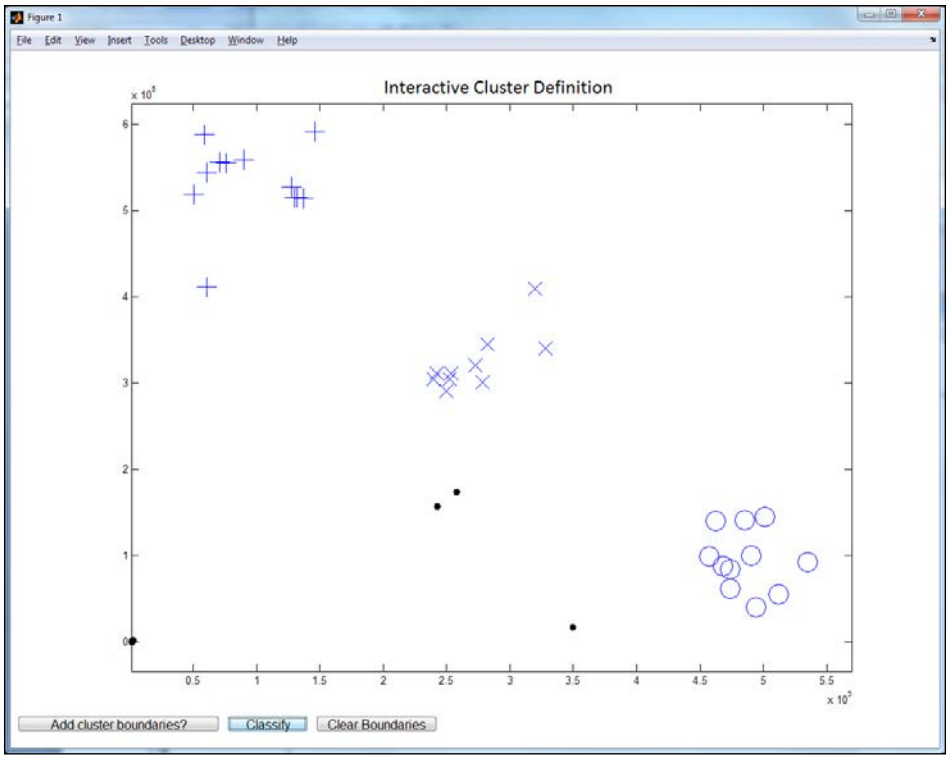
```
function clearBounds(src, evt)
  cla;
  userdata = get(gcf,'userdata');
  userdata.boundDef = [];
  set(gcf,'userdata',userdata);
  plot(userdata.X,userdata.Y,'k.','Markersize',18);
  hold on;
end
```

7. Run the code and define cluster boundaries using the mouse. Note that until you click the on the **Classify** button, classification does not occur. Here is a snapshot of how it looks (the arrow and dashed boundary is used to depict the cursor movement from user interaction):



8. Initiate a classification by clicking on **Classify**.

   The graph will respond by re-drawing all points inside the constructed boundary with a specific symbol:



How it works...

This recipe illustrates how user input is obtained from the graphical display in order to impact the results produced. The image processing toolbox has several such functions that allow user to provide input by mouse clicks on the graphical display—such as `imellipse` for drawing elliptical boundaries, and `imrect` for drawing rectangular boundaries. You can refer to the product pages for more information.

Takeaways from this recipe:

- Obtain user input directly via the graph in terms of data point level annotations and/or user settable boundary definitions

See also

Look up **MATLAB help** on the `imlineimpoly`, `imfreehandimrect`, and `imellipseginput` commands.

Linked axes and data brushing

MATLAB allows creation of programmatic links between the plot and the data sources and linking different plots together. This feature is augmented by support for data brushing, which is a way to select data and mark it up to distinguish from others. Linking plots to their data source allows you to manipulate the values in the variables and have the plot automatically get updated to reflect the changes. Linking between axes enables actions such as zoom or pan to simultaneously affect the view in all linked axes. Data brushing allows you to directly manipulate the data on the plot and have the linked views reflect the effect of that manipulation and/or selection. These features can provide a live and synchronized view of different aspects of your data.

Getting ready

You will use the same cluster data as the previous recipe. Each point is denoted by an x and y value pair. The angle of each point can be computed as the inverse tangent of the ratio of the y value to the x value. The amplitude of each point can be computed as the square root of the sum of squares of the x and y values. The main panel in row 1 show the data in a scatter plot. The two plots in the second row have the angle and amplitude values of each point respectively. The fourth and fifth panels in the third row are histograms of the x and y values respectively. Load the data and calculate the angle and amplitude data as described earlier:

```
load clusterInteractivData
data(:,1) = X;
data(:,2) = Y;
data(:,3) = atan(Y./X);
data(:,4) = sqrt(X.^2 + Y.^2);
clear X Y
```

How to do it...

Perform the following steps:

1. Plot the raw data:

```
axes('position',[.3196 .6191 .3537 .3211], ...
  'Fontsize',12);
scatter(data(:,1), data(:,2),'ks', ...
  'XDataSource','data(:,1)','YDataSource','data(:,2)');
box on;
xlabel('Dye 1 Intensity');
ylabel('Dye 1 Intensity');title('Cluster Plot');
```

2. Plot the angle data:

```
axes('position',[.0682 .3009 .4051 .2240], ...
  'Fontsize',12);
scatter(1:length(data),data(:,3),'ks',...
  'YDataSource','data(:,3)');
box on;
xlabel('Serial Number of Points');
title('Angle made by each point to the x axis');
ylabel('tan^{-1}(Y/X)');
```

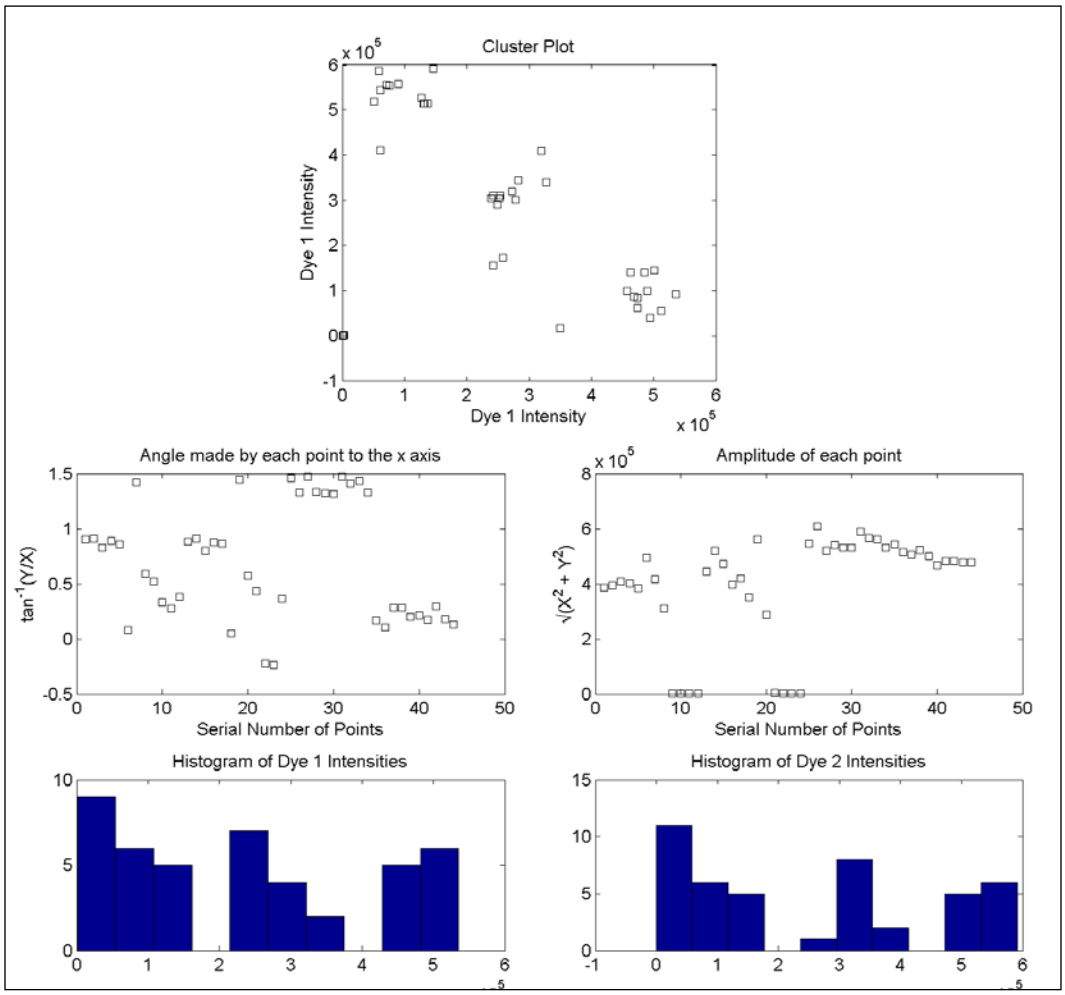
3. Plot the amplitude data:

```
axes('position',[.5588 .3009 .4051 .2240], ...
  'Fontsize',12);
scatter(1:length(data),data(:,4),'ks', ...
  'YDataSource','data(:,4)');
box on;
xlabel('Serial Number of Points');
title('Amplitude of each point');
ylabel('{\surd(X^2 + Y^2)}');
```

4. Plot the two histograms:

```
axes('position',[.0682 .0407 .4051 .1730], ...
  'Fontsize',12);
hist(data(:,1)); title('Histogram of Dye 1 Intensities');

axes('position',[.5588 .0407 .4051 .1730], ...
  'Fontsize',12);
hist(data(:,2));
title('Histogram of Dye 2 Intensities');
```

The output is as follows:



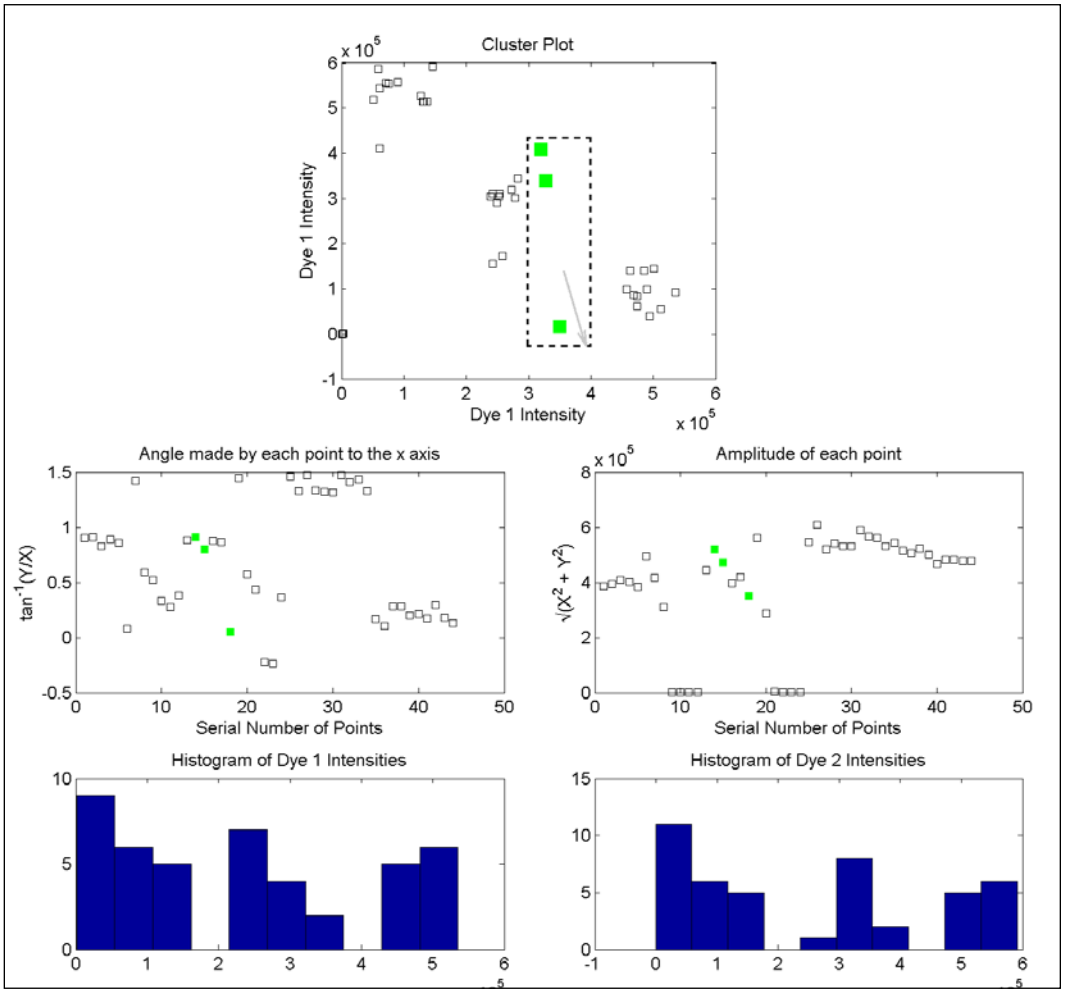
5. Programmatically, link the data to their source:

```
linkdata;
```

Programmatically, turn brushing on and set the brush color to green:

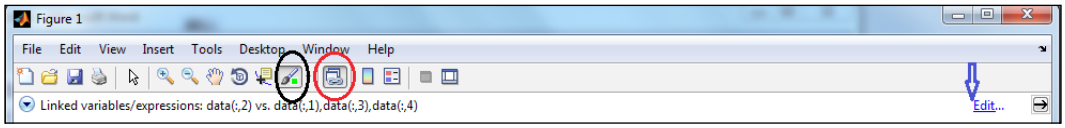
```
h = brush;
set(h,'Color',[0 1 0],'Enable','on');
```

Use mouse movements to brush a set of points. You could do this on any one of the first three panels and observe the impact on corresponding points in the other graphs by its turning green. (The arrow and dashed boundary is used to depict the cursor movement from user interaction in the following figure):



How it works...

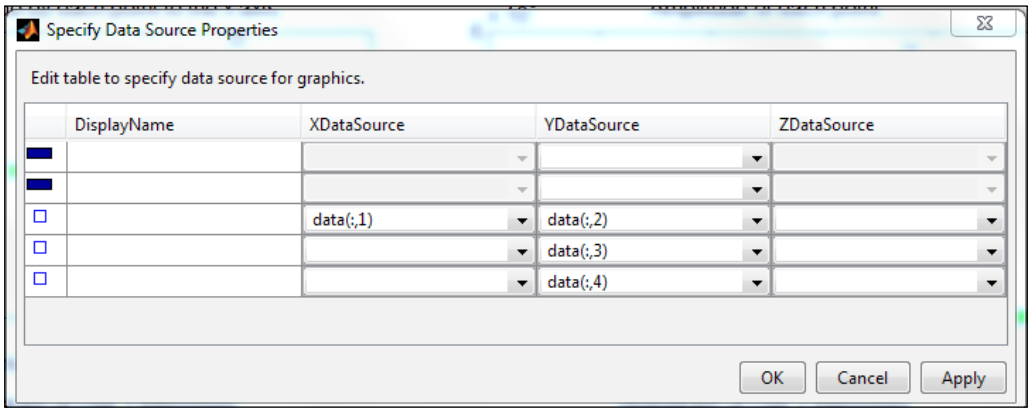
Because brushing is turned on, when you focus the mouse on any of the graph areas, a cross hair shows up at the cursor. You can drag to select an area of the graph. Points falling within the selected area are brushed to the color green, for the graphs on rows 1 and 2. Note that nothing is highlighted on the histograms at this point. This is because the x and y data source for the histograms is not correctly linked to the data source variables yet. For the other graphs, you programmatically set their x and y data source via the `XDataSource` and the `YDataSource` properties. You can also define the source data variables to link to a graphic and turn brushing on by using the icons from the figure toolbar as shown in the following screenshot. The first circle highlights the brush button; the second circle highlights the link data button. You can click on the **Edit** link pointed by the arrow to exactly define the x and y sources:



There's more...

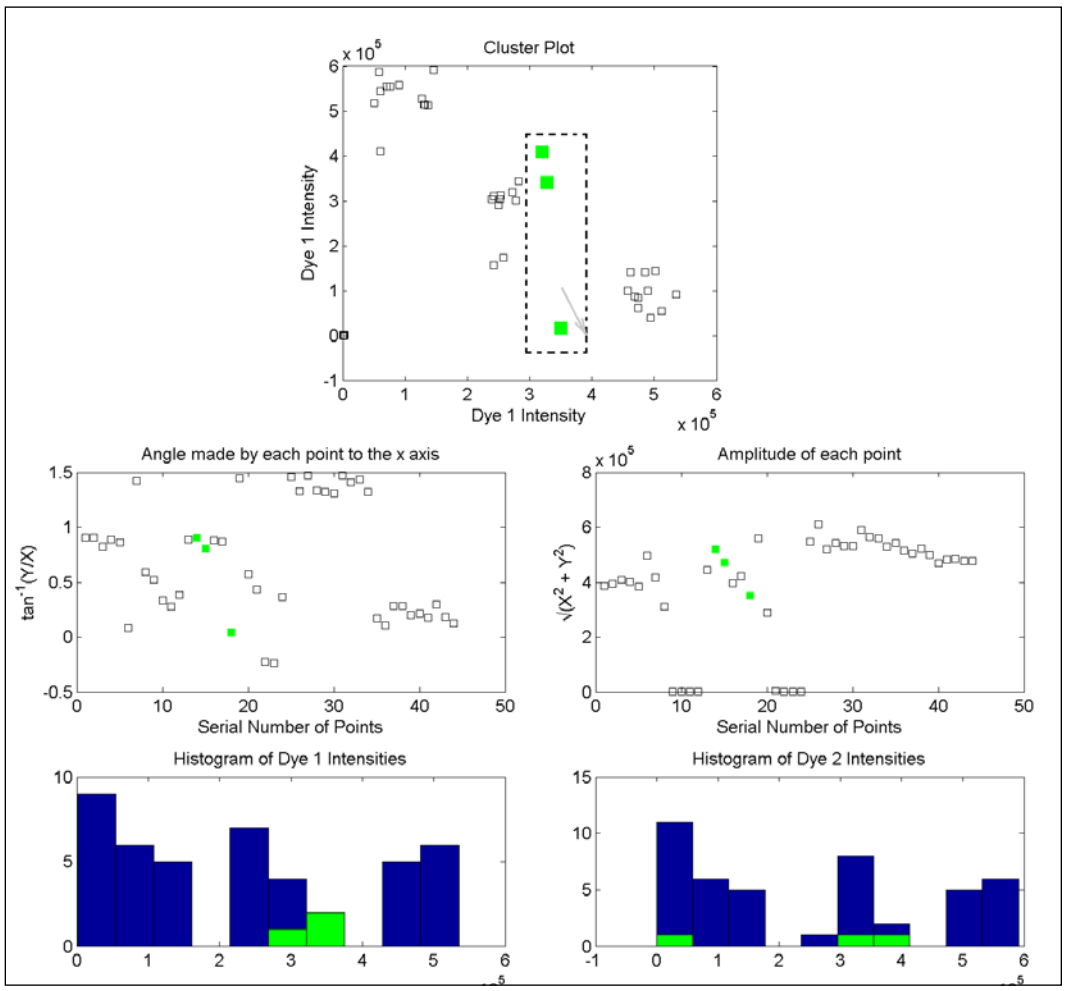
To define the source data variables to link to a graphic and turn brushing on by using the icons from the Figure toolbar, do as follows:

1. Clicking on **Edit** (pointed to in preceding figure) will bring up the following window:



2. Enter `data(:,1)` in the **YDataSource** column for row 1 and `data(:,2)` in the **YDataSource** column for row 2.

3. Now try brushing again. Observe that bins of the histogram get highlights in a bottom up order as corresponding points get selected (again, the arrow and dashed boundary is used to depict the cursor movement from user interaction):

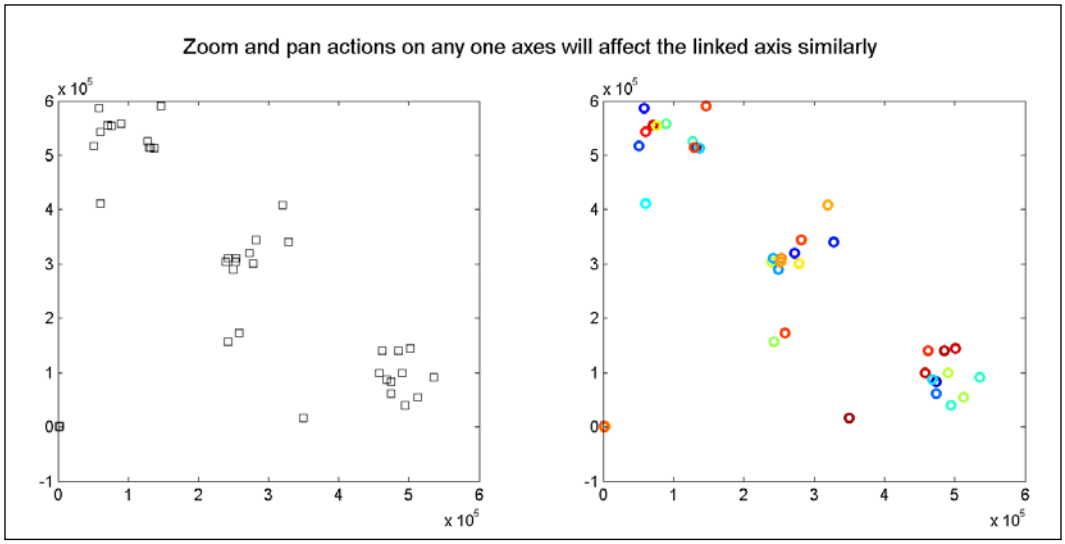


4. Link axes together to simultaneously investigate multiple aspects of the same data point. For example, in this step you plot the cluster data alongside a random quality value for each point of the data. Link the axes such that zoom and pan functions on either will impact the axes of the other linked axes:

```
axes('position',[.13 .11 .34 .71]);
scatter(data(:,1), data(:,2),'ks');box on;
axes('position',[.57 .11 .34 .71]);
```

```
scatter(data(:,1), data(:,2),[],rand(size(data,1),1), ...
  'marker','o', 'LineWidth',2);box on;
linkaxes;
```

The output is as follows. Experiment with zoom and pan functionalities on this graph.



Takeaways from this recipe:

- Use **data brushing and linked axes** features to provide a live and synchronized view of different aspects of your data.

See also

Look up **MATLAB help** on the `linkdata`, `linkaxes`, and `brush` commands.

The magnifying glass demo

A **virtual magnifying glass** function is developed in this recipe. Using this, you can load any image and run the virtual magnifying glass over it for closer inspection at your cursor tip. This recipe was adapted from Mingjing Zhang's submission on MATLAB Central File Exchange.

Getting ready

Load the image of a microtiter plate with a distribution of filled and empty wells:

```
userdata.img_rgb = imread('sampleImage.png');
```

How to do it...

Perform the following steps:

1. Define parameters for passing to the callback function to be invoked in response to cursor movement:

```
% The image size
userdata.size_img = size(userdata.img_rgb);
% the start time (to be used along with the frame rate
% info to determine how often image should be updated)
userdata.start_time = tic;
% The frame rate
userdata.FPS = 20;
% Magnifying Power
userdata.MagPower = 2;
% The Radius of the Magnifier
userdata.MagRadius = 100;
% The radius of the image to be magnified
userdata.PreMagRadius = userdata.MagRadius./userdata.MagPower;
userdata.alreadyDrawn = 0;
```

2. Set up the figure and axes:

```
MainFigureHdl = figure('Name', 'Magnifier Demo', ...
  'NumberTitle' ,'off', ...
  'Units', 'normalized', ...
  'Position', [.1854    .0963    .4599    .8083], ...
  'MenuBar', 'figure', ...
  'Renderer', 'opengl');
MainAxesHdl = axes('Parent', MainFigureHdl, ...
  'Units', 'normalized',...
  'Position', [0 0 1 1], ...
  'color', [0 0 0], ...
  'YDir', 'reverse', ...
  'NextPlot', 'add', ...
  'Visible', 'on');
```

3. Plot the initial image and initial magnified image at the initial cursor tip position:

```
userdata.img_hdl = image(0,0,userdata.img_rgb);
axis tight
% The magnified image object
userdata.mag_img_hdl = image(0,0,[]);
userdata.mag_img = ...
  userdata.img_rgb(1:userdata.PreMagRadius*2+1,...
  1:userdata.PreMagRadius*2+1,:);
```

4. Create a circular mask for the magnified image:

```
[x y] = ...
  meshgrid(-userdata.PreMagRadius:userdata.PreMagRadius);
dist = double(sqrt(x.^2+y.^2)); % dist pixel to center
in_circle_log = dist<userdata.PreMagRadius-1;
out_circle_log = dist>userdata.PreMagRadius+1;
dist(in_circle_log) = 0;
dist(out_circle_log) = 1;
dist(~(in_circle_log|out_circle_log)) = ...
  (dist(~(in_circle_log|out_circle_log)) - ...
  (userdata.PreMagRadius-1))./2;
userdata.mask_img = 1 - dist;
```

5. Initialize the image object:

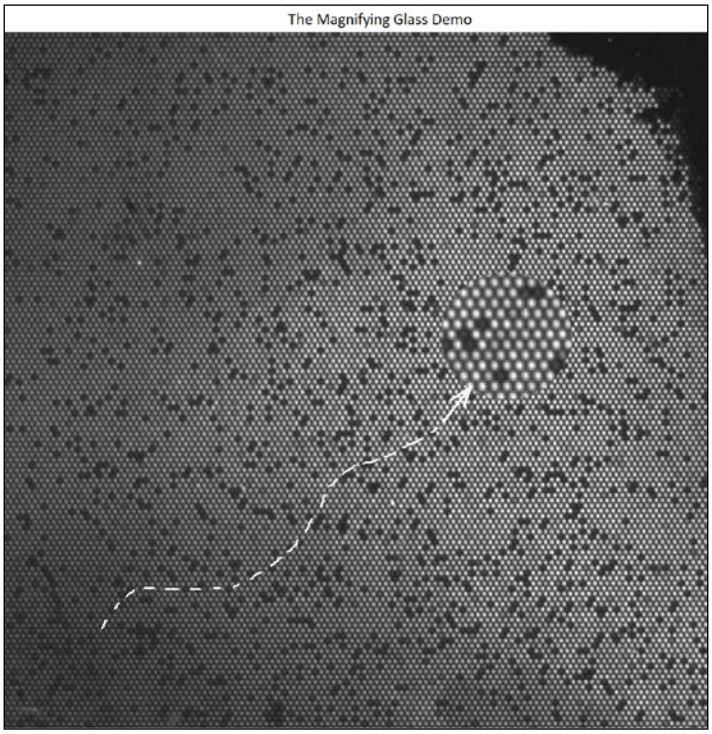
```
set(userdata.mag_img_hdl, 'CData',userdata.mag_img, ...
  'AlphaData', userdata.mask_img);
%designate the call back function
set(MainFigureHdl,'userdata',userdata,...
  'WindowButtonMotionFcn',...
  @stl_magnifier_WindowButtonMotionFcn);
```

6. The next task is to define this callback function—this function should select the image pixels within the mask defined at the cursor tip, and display a magnified version of it at that location.

```
function stl_magnifier_WindowButtonMotionFcn(obj,event)

  %extract parameters
  userdata = get(gcbo,'userdata');

  % determine if image needs to be updated per frame rate
  % definitions
  cur_time = toc(userdata.start_time);
  curFrameNo = floor(cur_time.*userdata.FPS);

  % If this frame has not been drawn yet
```

```
  if userdata.alreadyDrawn < curFrameNo
    mag_pos = get(obj,'CurrentPoint');
    mag_pos(1) = ...
      round(size(userdata.img_rgb,2)*mag_pos(1));
    mag_pos(2) = size(userdata.img_rgb,2) - ...
      round(size(userdata.img_rgb,2)*mag_pos(2))+1;

    % The size of the part of the image to be magnified
    % The range has to be cropped in case it is outside the
    % image.
    mag_x = mag_pos(1)+[-userdata.PreMagRadius ...
    userdata.PreMagRadius];
    mag_x_cropped = min(max(mag_x, 1),...
    userdata.size_img(2));
    mag_y = mag_pos(2)+[-userdata.PreMagRadius ...
    userdata.PreMagRadius];
    mag_y_cropped = min(max(mag_y, 1),...
    userdata.size_img(1));

    % Take the magnified part of the image
    userdata.mag_img([mag_y_cropped(1):...
    mag_y_cropped(2)]-mag_pos(2)+...
    userdata.PreMagRadius+1,...
    [mag_x_cropped(1):mag_x_cropped(2)]-...
    mag_pos(1)+userdata.PreMagRadius+1,:) = ...
    userdata.img_rgb(mag_y_cropped(1):...
    mag_y_cropped(2),...
    mag_x_cropped(1):mag_x_cropped(2),:);

    % Show the image as twice its actual size
    set(userdata.mag_img_hdl, 'CData', ...
    userdata.mag_img,'XData', mag_pos(1)+...
    [-userdata.MagRadius userdata.MagRadius], ...
    'YData', mag_pos(2)+[-userdata.MagRadius ...
    userdata.MagRadius]);

    % Update the object
    drawnow;
    userdata.alreadyDrawn = curFrameNo;
  end
  set(gcf,'userdata',userdata);
end
```

Here is a view of the final software in action (dashed line depicts cursor movement; arrow depicts the position of the cursor):



How it works...

The callback function accesses the coordinates of the cursor position and calculates the center coordinates of the magnifier as the current position ± the radius of the circular magnifying glass mask. The coordinates are then cropped to ensure that they do not access a location beyond the coordinates of the main image. This image is then plotted at twice the original size.

Takeaways from this recipe:

- Use the virtual magnifying glass to load any image and subject it to closer inspection at your cursor tip.

See also

Look up **MATLAB help** on the `image` and `WindowButtonMotionFcn` commands.

Animation with playback of frame captures

Animation is a sequence of images telling a story. For a sequence that is changing a lot from frame to frame, the best approach to animation is to play it back at some meaningful rate as a movie, the key point being that the rendering is not in real time; rather, it is the playback of a pre-generated series of images. In this recipe, you will explore the concept of playing back a series of pre-rendered images.

Getting ready

MATLAB supplies a dataset of brain MRI slices which you used extensively in *Chapter 5*, *Playing in the Big Leagues with Three-dimensional Data Displays*. First, load the data:

```
load MRI
```

How to do it...

Perform the following steps:

1. The first approach will be to use the `getframe` command to collect each frame of the display and then playback using the `movie` command. Note that as you call the `image` command to make the frames, the live rendering of the different frames will be visible. The command `movie` does the actual off line playback of the images.

```
figure;
for image_num = 1:27;
  image(D(:,:,image_num));colormap(map);
  f(image_num) = getframe;
end
close;
movie(f);
```

2. Another alternative is to record the frames as part of an **AVI** (**Audio Video Interleave**) object and then save that as an avi file that you can play with any media player:

```
aviobj = avifile('example.avi','fps',3,...
  'quality',100,'compression','none');
for image_num = 1:27;
  image(D(:,:,image_num));
  colormap(map);
  aviobj = addframe(aviobj,getframe);
end
aviobj = close(aviobj);
close;
```

3. Yet another alternative is to use the command `imwrite` and save in the gif format. You can set a frame rate and other parameters for the animation as follows:

```
imwrite(D,map,'letsTry.gif','gif', 'DelayTime',2,...
  'Location', [503   289], 'LoopCount',7);
```

How it works...

In this recipe, you have essentially taken snapshots of the image as it is displayed and stored it, ready to be played back at a later time. `getframe` returns a snapshot (pixmap) of the current axes or figure. Note that if the screensaver is started up while waiting for an image to render or you open up a browser that plots over the figure, then the pixel values from the screensaver or whatever else is displayed in the same location as the figure gets captured. When running remotely on virtual desktops, the virtual desktop window needs to be active on your current desktop for successful capture.

The `avifile` command creates an **avifile** object. As you saw, a number of parameters such as the **frame rate**, **quality**, and **compression** codec in use can be specified as a parameter and value pairing with this command.

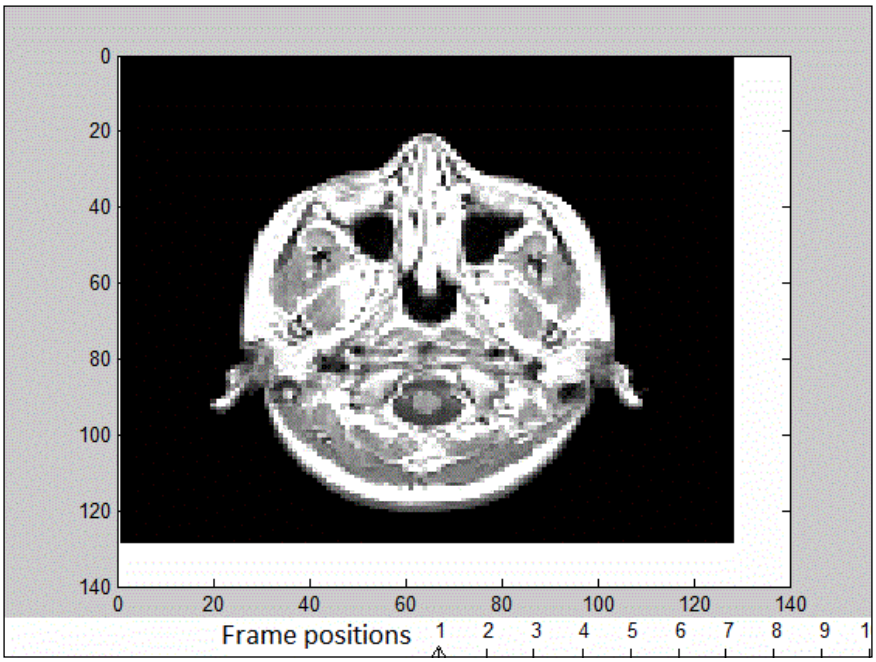
The `imwrite` command writes the image `D` to the file specified by filename in a specified format. There are format specific parameters that can be supplied as property name and value pairs. In this recipe, you have set the delay time, the location of the plot and the loop count for the `.gif` format. The delay time refers to how long each frame is displayed. The location indicates where on the browser window the image will be created. The loop count specifies how many times the animation is run before it is stopped.

There's more...

One possibility to generate a **smooth playback** would be to generate the intermediate image between two snapshots. Here, this is generated using the `anymate` function, a submission by Jerker Wågberg on MATLAB File Exchange. The `anymate` function analyzes the changes in the properties of Handle Graphics objects and can interpolate between these changes to generate a smooth transition between each given state, here called a break. To generate an animation, `anymate` collects the values of all Handle Graphics objects for each break and then estimates the true changes between these breaks for presentation. Perform the following to generate an animation that smoothly interpolates between the given frames:

```
D_less = D(:,:,1,1:2:end);
for image_num = 1:size(D_less,4)-1;
  figure; image(D_less(:,:,image_num));
  colormap(map);
end

anymate;
```

The output is as follows:



Takeaways from this recipe:

- Use **play back of pre-rendered frame captures** to show evolution of data in time or space
- Use **image interpolation techniques to generate a smooth playback** from few available time lapse snapshots

Stream particle animation

A **stream particle animation** is useful for visualizing the flow direction and speed of a vector field. The "particles", represented by a line marker, 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.

Getting ready

This recipe builds a stream particle animation to trace the streamline path of a certain section of the wind flow data that comes as part of the MATLAB installation. Start with loading the dataset:

```
load wind
```

How to do it...

Perform the following steps:

1. Investigate range of data and experiment at different values of x, y, and z to decide the cross sections you want to use in this animation:

```
disp([min(x(:)) max(x(:)) min(y(:)) max(y(:)) ...
  min(z(:)) max(z(:))]);
```

2. Define the mesh over which stream lines will be constructed:

```
[sx sy sz] = meshgrid(85:20:100, 20:2:50, 6);
```

3. Define the stream lines:

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

4. Define the view:

```
axis tight; box on; grid on;
daspect([19.9445   15.1002    1.0000]);
campos([ -165.7946 -223.0056   11.0223]);
```

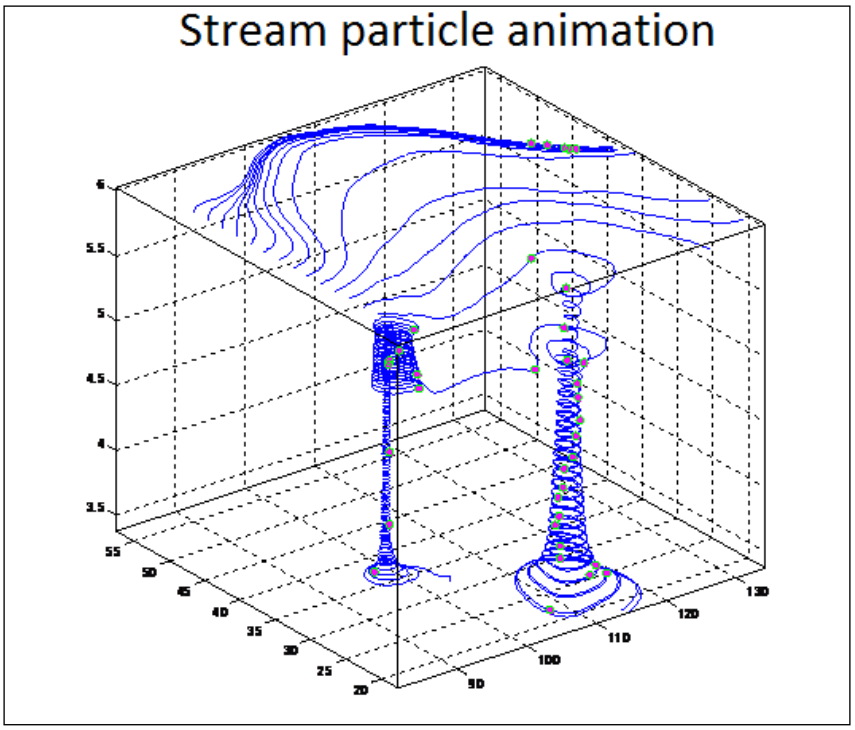
5. Pick the vertices at which to place the particles:

```
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.08);
```

6. Start the animation:

```
set(gca,'drawmode','fast');
streamparticles(iverts,15,...
  'FrameRate',5,...
  'Animate',10,...
  'ParticleAlignment','on',...
  'MarkerEdgeColor','green',...
  'MarkerFaceColor','green',...
  'Marker','o');
```

The output is as follows:



How it works...

You have already encountered the `stream3` and `streamline` commands in *Chapter 5*, *Playing in the Big Leagues with Three-dimensional Data Displays*. These commands help to create the paths for the particles to travel enabling a visual context for the animation. All the animations start at the plane z = 6. The `campos` function sets the position of the camera from where this animation will be observed. Setting the data aspect provides greater resolution along the x and y axis.

The `interpstreamspeed` function returns the vertices at which the particles should be drawn. As the path is provided, it is not required to draw every vertex. Making it appear at a certain step will be adequate to create the desired effect. The velocity with which the particles move may be scaled to increase or decrease the number of interpolated vertices. In this example, the velocities are scaled by 0.08 to increase the number of interpolated vertices.

Setting the axis property `DrawMode` to `fast` makes the animation run faster.

The `streamlineparticles` function allows a number of its properties to be set to impact the visualization. In addition to the 3D vertices, the command takes an integer argument *n*, set to `15` here, that determines how many particles to draw. Since subsequently, the `ParticleAlignment` property is set to `On`, *n* is interpreted as the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value.

Additionally, you set properties such as the `FrameRate` to 5 frames to be displayed per second, impacting how fast the animation is changing; the property `Animate` to 10 times indicating that the entire sequence is to be run 10 times before it stops; and the properties `MarkerEdgeColor`, `MarkerFaceColor`, and `Marker` impacting the way the actual particles look. Animations run faster when marker edges are not drawn, so the `none` value is recommended for the `MarkerEdgeColor` property.

Note that the particles on the z plane travel much faster than the particles along the spiral.

Takeaways from this recipe:

- Use **stream particle animation** along any path of your choice (including stream lines generated by MATLAB command) to observe the flow of particles in time.

See also

Look up **MATLAB help** on the `interpstreamspeed` and `streamlineparticles` commands.

Animation by incremental changes to chart elements

An alternative approach to animation is to continually **erase and redraw** the objects on the screen, making incremental changes with each redraw. One of the ways to do this is to redefine the XData, YData, ZData, and/or CData plot object properties for every change, then make calls to `refreshdata` followed by `drawnow`; or equivalently linking the plot to the data sources, which will cause the plot to be automatically updated each time the source data is changed by implicitly calling `refreshdata` and `drawnow`. This alternative approach allows for faster rendering at a finite cost to the rendering accuracy. In this recipe, you will create an animation using this erase and redraw strategy.

Getting ready

This recipe illustrates the process of convolution between two functions. It was adapted from a File Exchange submission on convolution by Laine Berhane Kahsay. Convolution is a mathematical operation on two functions f and g, producing a third function that is a modified version of one of the original function, giving the area overlap between the two functions as a function of the amount by which the second function is translated. This recipe uses a parabolic and a square function. Generate the data. This involves defining a range for the x values of each function and defining the y values for each function.

```
%% data generation
s_int = 0.1;                % sampling interval constant
t = -10:s_int:10;           % interval for function 'f(t)'
f = 0.1*(t.^2);             % definition of function 'f(t)'
t1 = -7:s_int:7;            % interval for function 'g(t1)'
go = [-1*ones(1,40) ones(1, 61) -1*ones(1, 40)];
                            % definition of function 'g(t1)'
c = s_int * conv(f, go);    % convolve: note the
                            % multiplication by the
                            % sampling interval

% flip 'go(t1)' for the graphical convolutions g = go(-t1)
g = fliplr(go);
tf = fliplr(-t1);

% slide range of 'g' to discard non-overlapping areas with
% 'f' in the convolution
tf = tf + ( min(t)-max(tf) );

% get the range of function 'c' which is the convolution of
% 'f(t)' and 'go(t1)'
tc = [ tf t(2:end)];
tc = tc+max(t1);
```

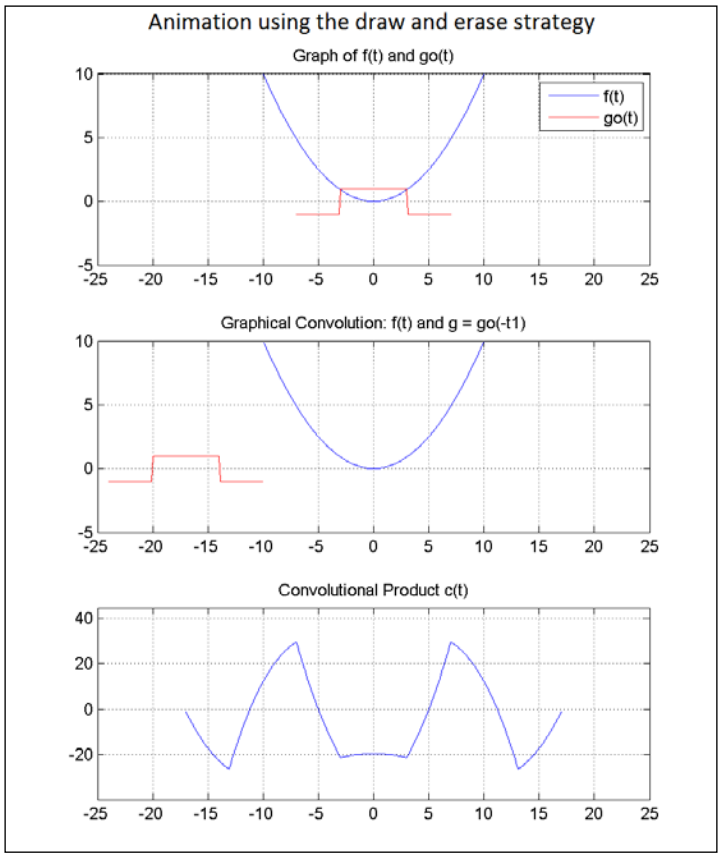
How to do it...

Perform the following steps:

1. Plot the static part of the data into a set of three panel graphics:

```
figure('units','normalized','Position', [.16 .14 .31 .69]);

% Plot f(t) and g0(t1) in panel 1
subplot(3,1,1);
op = plot(t,f, 'b'); hold on;
plot(t1, go, 'r');grid on;
xlim( [ ( min(t)-abs(max(tf)-min(tf)) - 1 ) ...
  ( max(t)+abs(max(tf)-min(tf)) + 1 ) ] );
title('Graph of f(t) and go(t)');
legend({'f(t)' 'go(t)'});

% Plot f and g in panel 2. And two vertical lines showing
% the overlapped region between the two functions.
% Add yellow on the overlapped region.
subplot(3,1,2);
plot(t, f);hold on; grid on;
title('Graphical Convolution: f(t) and g = go(-t1)');
q = plot(tf, g, 'r');
xlim( [ ( min(t)-abs(max(tf)-min(tf))-1 ) ...
  ( max(t)+abs(max(tf)-min(tf))+1 ) ] );
u_ym = get(gca, 'ylim');
% bound and shade the overlapped region
s_l = line( [min(t) min(t)], ...
  [u_ym(1) u_ym(2)], 'color', 'g'  );
e_l = line( [min(t) min(t)], ...
  [u_ym(1) u_ym(2)], 'color', 'g'  );
sg = rectangle('Position', ...
  [min(t) u_ym(1) 0.0001 u_ym(2)-u_ym(1)], ...
  'EdgeColor', 'w', 'FaceColor', 'y', ...
  'EraseMode', 'xor');

% convolution result in panel 3
subplot(3,1,3);
r = plot(tc, c);grid on; hold on;
xlim( [ ( min(t)-abs(max(tf)-min(tf)) - 1 ) ...
  ( max(t)+abs(max(tf)-min(tf)) + 1 ) ] );
title('Convolutional Product c(t)');
```

The output is as follows:

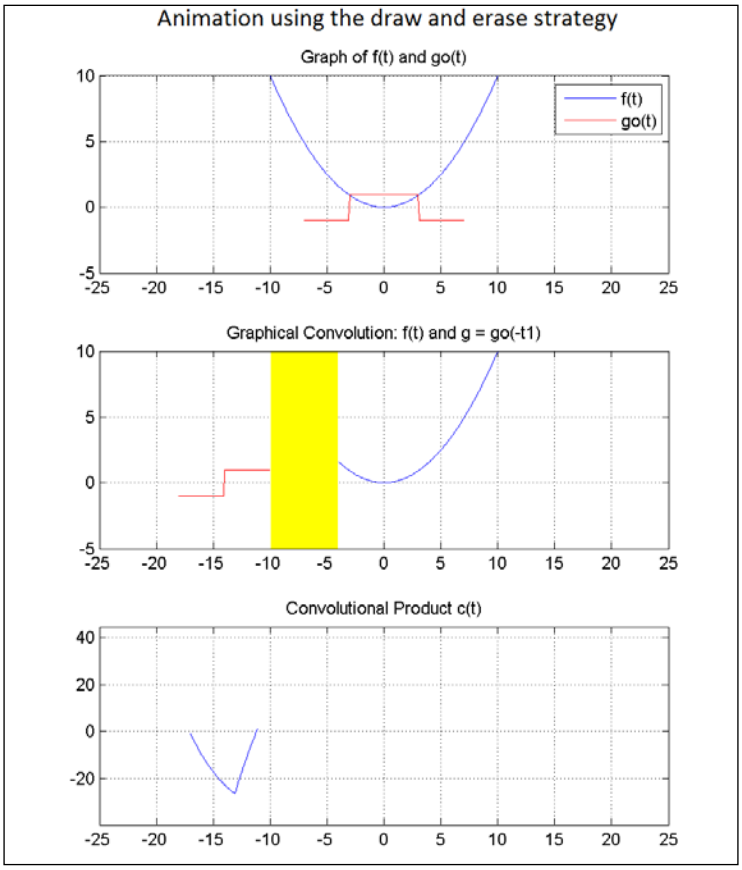


2. Animation block—perform these steps for the range of time over which the convolution product is going to show effect. Pause a certain time at each step to allow user to observe the effects:

```
for i=1:length(tc)

  pause(0.1);

  % Update the position of sliding function 'g',
  % its handle is 'q'
  tf=tf+s_int;
  set(q,'EraseMode','xor','XData',tf,'YData',g);

  % Show a vertical line for a left boundary of
  % overlapping region
  sx = min( max( tf(1), min(t) ), max(t) );
  sx_a = [sx sx];
```

```
    set(s_l,'EraseMode','xor', 'XData', sx_a);

    % Show a second vertical line for the right
    % boundary of overlapping region
    ex = min( tf(end), max(t) );
    ex_a = [ex ex];
    set(e_l,'EraseMode','xor', 'XData', ex_a);

    % Update shading on overlapped region
    rpos = [sx u_ym(1) max(0.0001, ex-sx) u_ym(2)-u_ym(1)];
    set(sg, 'Position', rpos);

    % Update the plot of convolution product 'c',
    % its handle is r
    set(r,'EraseMode','xor','XData',tc(1:i),'YData',c(1:i) );

end
```

The output is as follows:



How it works...

The main idea is to extract handles for those items that need to be changed at every step of the animation. This includes the two boundary lines (handles `s_l` and `e_l` respectively) and the shaded yellow rectangle (with handle `sg`) demarking the overlapped region between the two functions; the sliding function `g` (with handle `q`) and the convolution product `c` (with handle `r`). For each of these objects, the new data to be displayed is calculated at each step and the `XData` and `YData` properties of these objects are set to the new values to be displayed (except for the rectangle, whose position coordinates are updated).

At each step, data is updated and a redraw effort is necessary. If you updated the data by means of using the `xdatasource` or `ydatasource` object property, you would need to explicitly make a call to the `refreshdata` function to actually update the data being graphed. Here, the data properties are directly manipulated and therefore you are not making explicit calls to `refreshdata`. Also, you are not making explicit calls to the function `drawnow` to reflect the changes. The reason `drawnow` need not be called is because `pause` is called at each iteration. It implicitly causes all changed handles to be re-drawn.

For all handles, the `EraseMode` property is set to `xor`. `xor` enables the draw and erase of these elements by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the line.

Takeaways from this recipe:

- Use the **erase and redraw** strategy making incremental changes at each step for creating animations.

See also

Look up **MATLAB help** on the `EraseMode`, `XData`, `YData`, `refreshdata`, and `drawnow` properties.

8

Finalizing Graphics for Publication and Presentations

In this chapter, we will cover:

- Export formats and resolution
- Vector graphics for inclusion into documents
- Preserving onscreen font size and aspect ratios
- Publishing code and graphics to a webpage

Introduction

Graphics may be presented either in an electronic format or printed to hard copies for the consumption of your target audience. Image quality and formatting requirements vary depending upon your final presentation goal. In this chapter, recipes illustrate MATLAB's capabilities to appropriately finish your graphics.

Some general considerations to keep in mind while designing graphics for presentation or publication are as follows (reference: *Society for Imaging Sciences and Technology*).

- Time available for viewing should be proportional to the amount of information you pack in a display.
- Use patterns as well as color where possible in your graphs because color may not be available in the printed format.
- Use a horizontal format for images to be projected, to ensure that it will be visible to the entire audience. Avoid placing critical data on the edges.
- Use an aspect ratio of 4:3 for CRT and 3:2 for slides, where possible.

Export formats and resolution

When it comes to exporting your graphics, there is a range of available choices for formats and resolutions. Different formats serve different needs such as onscreen viewing or online publishing to print publishing with specific strengths and limitations. This recipe demonstrates options in MATLAB for export formats and resolutions.

Getting ready

Create a graphic in MATLAB:

```
peaks; view(2); shading interp;
```

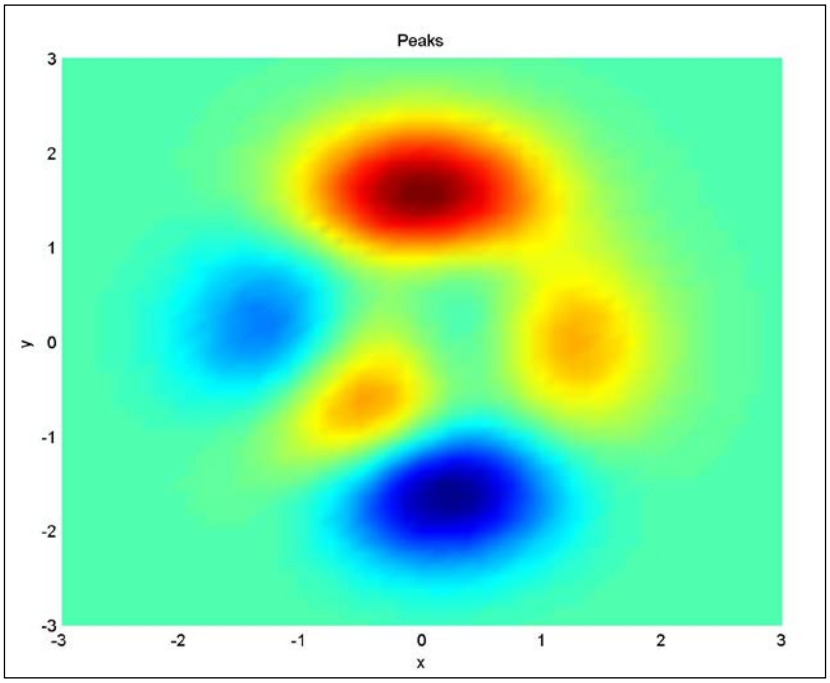
How to do it...

Perform the following steps:

1. Use the `print` command to export current graphic to a JPEG image file at a desired compression level and resolution:

```
print(gcf,'-djpeg100','-r200','3165_08_01_1.jpeg');
```

Compression level determines the extent of compression for the JPEG file format. The output is as follows:



How it works...

The `print` command was used to write your graphics to an image file in a specific format. You printed a sample graph using the MATLAB handle to the figure as a **JPEG** file.

You also used the `-rX` option to set a **resolution** for your output figure. The resolution determines how accurately your figure is printed. A higher resolution produces higher quality output. A higher resolution also means slower rendering and higher memory consumption. `X` is the **DPI** or dots per inch value. The DPI value is an instruction on how large an image is to be printed because a digitally stored image has no inherent physical dimensions. DPI may be interpreted as the physical density of ink dots for a printer, or more commonly, for a digital image it is equivalent to the **PPI** (**pixels per inch**) specification. The larger the PPI, the more clearly defined the image looks when printed.

The specific definition of resolution also depends on whether your figure is output as a **bitmap** or as a **vector graphic**. Bitmap stores graphics as 2D arrays of pixels. Vector formats store graphics as geometric objects and render them using drawing commands. Bitmaps are preferable for high-complexity plots. An example of a high-complexity plot is a surface plot that uses interpolated shading. Vector formats are preferable for most 2D plots and for some low-complexity surface plots. Vector formats create better lines and text than bitmap formats. Note that lighting and transparency are only supported by bitmap formats. One advantage with vector graphics is that you can resize it after importing it into most software applications without losing quality. Resizing after import is a problem with bitmap formats as it causes round-off errors that result in degradation of picture quality.

The format is also connected to the chosen rendering mechanism. For example, vector graphics can only be created by the `Painters` renderer.

Each format also has a supported **bit depth** which is the number of bits a format uses to store each pixel. This determines the number of colors the exported figure can contain. Bit depth applies mostly to bitmap graphics. An 8-bit image uses eight bits per pixel, enabling it to define 28 or 256 unique colors. In vector files that don't normally have a bit depth, the color of objects is specified by drawing commands stored in the file.

Ghostscript formats support a limited number of fonts. If you use an unsupported font, Courier is substituted. You cannot change the resolution of a Ghostscript format. The resolution is low (72 DPI) and might not be appropriate for publications.

Note that you can calculate the size of a figure exported to an uncompressed bitmap by multiplying the figure size by its resolution and the bit depth of the chosen format.

Takeaways from this recipe:

Use the following guidelines for choosing your image resolution:

- For printing, the default resolution of 150 DPI could be sufficient for typical laser-printer output
- For higher quality, you might want to use 200 or 300 DPI
- For exporting, base your decision on the resolution supported by the final output device

Use the following guidelines for choosing your image export format:

- BMP: Screen display under Windows
- EPS: Printing to PostScript printers / Image setters
- GIF: Screen display, especially the Web
- JPEG, JPG: Screen display, especially the Web, particularly photographs
- PNG: Replacement for GIF and to a lesser extent, JPG and TIF
- TIFF, TIF: Printing to PostScript printers

See also

Look up **MATLAB help** on the `print` command.

Vector graphics for inclusion into documents

Vector graphics formats create high fidelity publication-quality graphics. MATLAB supports the **Enhanced Metafiles** (**EMF**), which are vector files, capable of producing near publication-quality graphics. EMF is an excellent format to use if you plan to import your image into a Microsoft application and want the flexibility to edit and resize your image once it has been imported. It is the only supported MATLAB vector format you can edit from within a Microsoft application. The other alternatives are the EPS, SVG, ILL, and PDF formats, which work on both Windows and UNIX systems.

Getting ready

For this recipe, you will take a cross section of the flow data that comes with the MATLAB installation and that you used in *Chapter 4*, *Customizing Elements of MATLAB Graphics—Advanced* and *Chapter 5*, *Playing in the Big Leagues with Three-dimensional Data Displays*.

```
A = flow; surf(A(:,:,14)); shading interp;
```

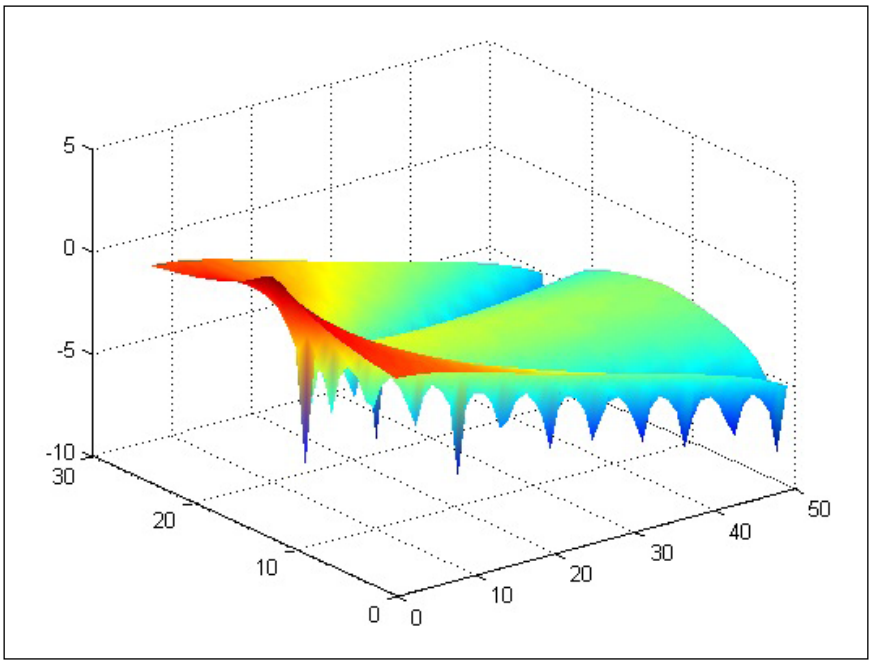
How to do it...

Perform the following steps:

1. Print the file as a meta file that now becomes available in your clip board:

```
print(gcf,'-dmeta');
```

Now if you execute a paste in a Microsoft Word or PowerPoint file, you will see the following output:



How it works...

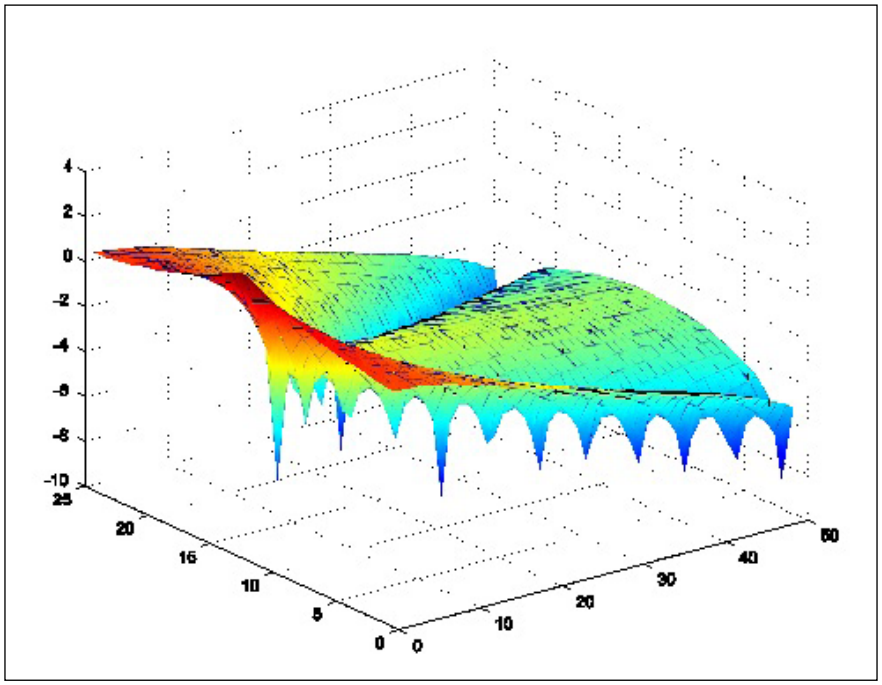
The EMF is a vector format and this affords the increased editability within your Word processing or PowerPoint environment. EMF is only applicable to Windows systems. The `print` command as used here copies the figure to the memory of your clipboard. Other vector formats supported by MATLAB include EPS, SVG, ILL, and PDF, which work on both Windows and UNIX systems.

There's more...

For other vector formats such as the EPS, you can choose to include a low resolution TIFF preview for viewing electronically. The following command allows you to save your figure as a colored Encapsulated Postscript file at 300 DPI and preview it in most word processors at 72 DPI using a TIFF:

```
print('-depsc','-tiff','-r300','filename');
```

Drag-and-drop the EPF file just created into this Word document to see the TIFF preview (which is of degraded quality due to the poor resolution of the preview). When you print this document it is printed at the improved resolution:



Takeaways from this recipe:

- Use vector export formats to obtain high quality images for use with printers
- Use the EMF format for figures to be included into Microsoft applications
- Use the TIFF preview options when using export in the other vector formats

See also

Look up **MATLAB help** on the `print` command.

Preserving onscreen font size and aspect ratios

Font sizes often need adjustment before export because they may appear too small in the figure when it is exported and resized afterward. Another problem is that the aspect ratio of your onscreen figures may not get reflected into your exported graphic. This recipe addresses how to manage these requirements.

Getting ready

For this recipe, you will use the function `peaks` again to generate the data:

```
peaks;view(2);shading interp;
```

How to do it...

Perform the following steps:

1. Resizethe image to render at a desired aspect ratio:

```
set(gcf,'units','normalized','position',...
  [0.0547    0.5000    0.5906    0.4046]);
```

2. Print with a landscape orientation (orients the longest page dimension horizontally):

```
set(gcf,'PaperOrientation','landscape');
```

3. Set where to print on paper in inches using the `paperposition` property:

```
set(gcf,'paperposition', [.25 2.5 8 6],...
  'Papernunits','inches');
```

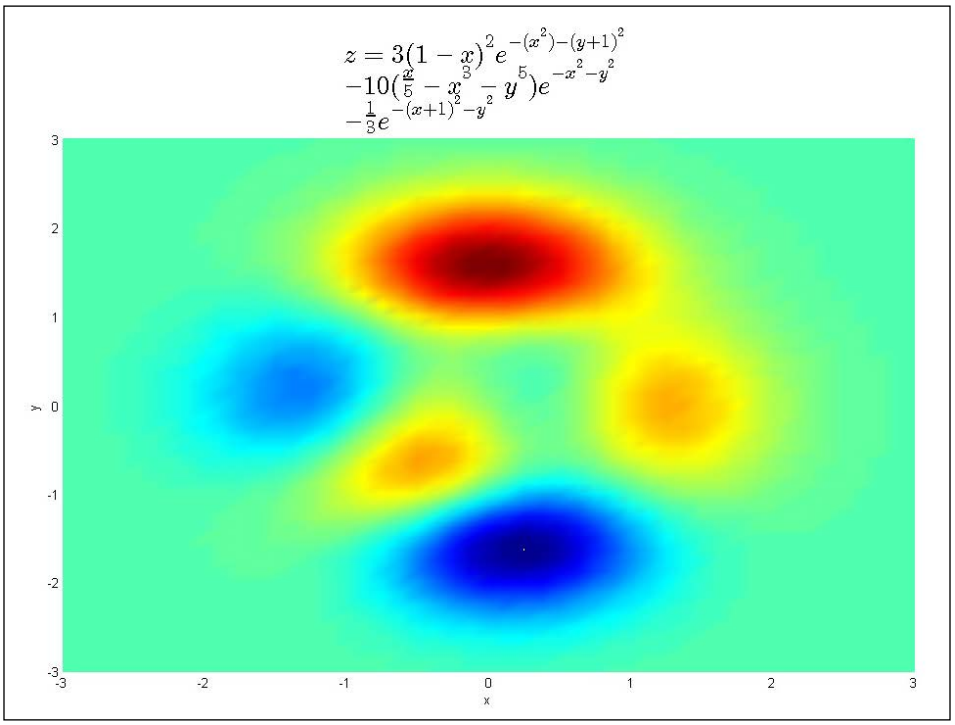
4. Make sure MATLAB exports using onscreen settings by setting `PaperPositionMode` to `auto`:

```
set(gcf,'PaperPositionMode','auto');
```

5. Add the source equation to the data as the title to your graphic using **LaTeX syntax** to format the string definition. Reformat the equation to meet minimum font size requirements (set at `20` for this case):

```
title({'$z =  3(1-x)^2e^{-(x^2) - (y+1)^2} $ ',...
  '$- 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} $',...
  '$- \frac{1}{3}e^{-(x+1)^2 - y^2}$'},...
  'interpreter','latex','Fontsize',20);
```

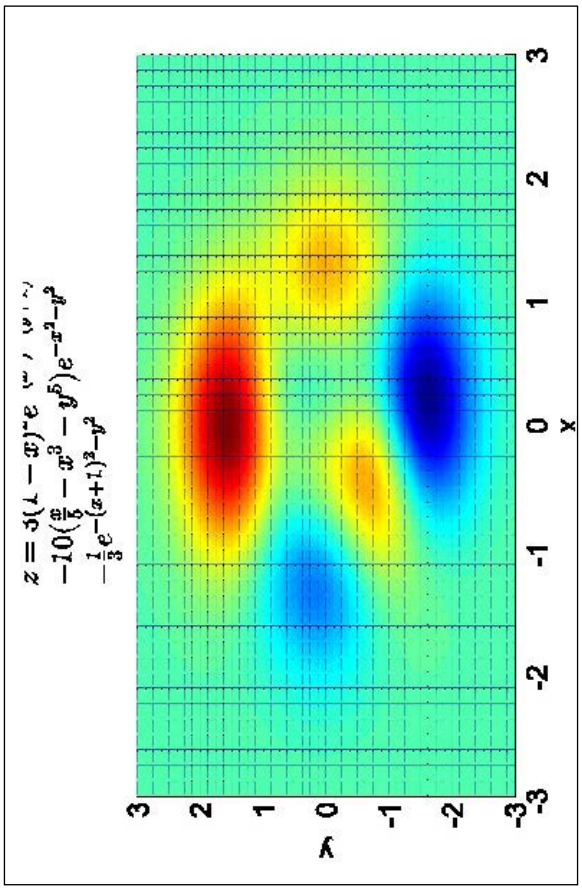
The output is as follows:



6. Use the `exportfig` function submitted by Ben Hinkle on File Exchange to output to EPS with customized font size, dimensions, orientation, and preview options:

```
exportfig(gcf,...
  '3165_08_03_2','format','eps', 'preview',...
  'tiff','height',4,'width',7,'color','cmyk',...
  'Fontmode','fixed','fontsize',15);
```

The output is as follows:



How it works...

This recipe demonstrates strategies to control font size, aspect ratio, figure orientation, and position of final graphic on paper.

See also

Look up **MATLAB help** on the `PaperOrientation`, `PaperPosition`, `PaperUnit`, and `PaperPositionMode` axes properties.

Publishing code and graphics to a webpage

MATLAB makes it really easy to share code with the document generation feature. You will have to break up your code into sections called as **Code Cells** which are evaluated in sequence. Using text markup features you can add commentary to these code blocks. Commentary can be stylized including bulleted and numbered items, and bold and monospace fonts and it also includes LaTeX equations. You could direct some of the code to be included with the results it generates. Finally, you can directly publish to various formats, including HTML, XML, and LaTeX. If Microsoft Word or Microsoft PowerPoint applications are on your system, you can publish to their formats as well.

Getting ready

You will markup the following code snippet to make it ready to publish using MATLAB's `publish` feature:

```
figure;
[x y z] = peaks;
surf(z);view(2);shading interp;
set(gcf,'Color',[1 1 1]);axis tight;
print(gcf,'-djpeg','-r400','3165_08_02_1.jpeg');

zN{1} =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2);
zN{2} = - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2);
zN{3} =  - 1/3*exp(-(x+1).^2 - y.^2);

for i = 1:3
  figure;
  surf(zN{i});view(2);shading interp;
  set(gcf,'Color',[1 1 1]);axis tight;
  print(gcf,'-djpeg','-r400',...
    ['3165_08_05_' num2str(i+1) '.jpeg']);
end
```

How to do it...

Make a `.m` file with the following steps:

1. Partition the code into code cell blocks. Each block presumably performs some logical steps that belong together. Insert the `%%` sign to define code cells:

   ```
   %% Declare Code Cells
   ```

2. Make a cell with no code to turn it into a heading. Note that the single line comments turn into body text:

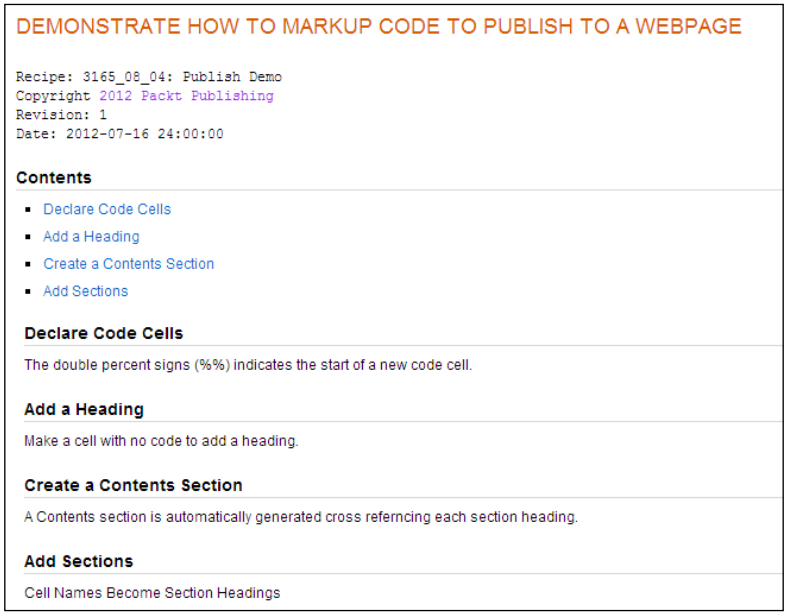
```
%% DEMONSTRATE HOW TO MARKUP CODE TO PUBLISH TO A WEBPAGE
%
%   Recipe: 3165_08_02: Publish Demo
%   Copyright 2012 Packt Publishing
%   Revision: 1
%   Date: 2012-07-16 24:00:00
%
%%
```

3. Add cell names to translate into section headings:

```
%% Add Sections
% Cell Names Become Section Headings
```

   If you execute the `publish` command at this stage, it translates the previous set of statements to the following output in a browser. Do this by clicking on the icon in the toolbar:

   A snapshot of the browser output is shown as follows:



DEMONSTRATE HOW TO MARKUP CODE TO PUBLISH TO A WEBPAGE

Recipe: 3165_08_04: Publish Demo
Copyright 2012 Packt Publishing
Revision: 1
Date: 2012-07-16 24:00:00

Contents

- Declare Code Cells
- Add a Heading
- Create a Contents Section
- Add Sections

Declare Code Cells

The double percent signs (%%) indicates the start of a new code cell.

Add a Heading

Make a cell with no code to add a heading.

Create a Contents Section

A Contents section is automatically generated cross referncing each section heading.

Add Sections

Cell Names Become Section Headings

Further, continue to add the following commands to your `.m` script and execute `publish` when you are done. The result is included as part of the code bundle with this book in the folder named `html`.

4. Direct some code and resultant figure for inclusion:

```
%% Add Code and Results
figure;
[x y z] = peaks;
surf(z);view(2);shading interp;
set(gcf,'Color',[1 1 1]);axis tight;
print(gcf,'-djpeg',...
  '-r400',['3165_08_02_1.jpeg']);
```

5. Add LaTeX equations:

```
%% Add source Equation in LaTeX format
% Enclose LaTeX expression using $ symbols:
%
% $$z = 3(1-x)^2e^{-(x^2) - (y+1)^2} - 10(\frac{x}{5} –
%   x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2 - y^2}$
```

6. Add some limited formatting to the textual output:

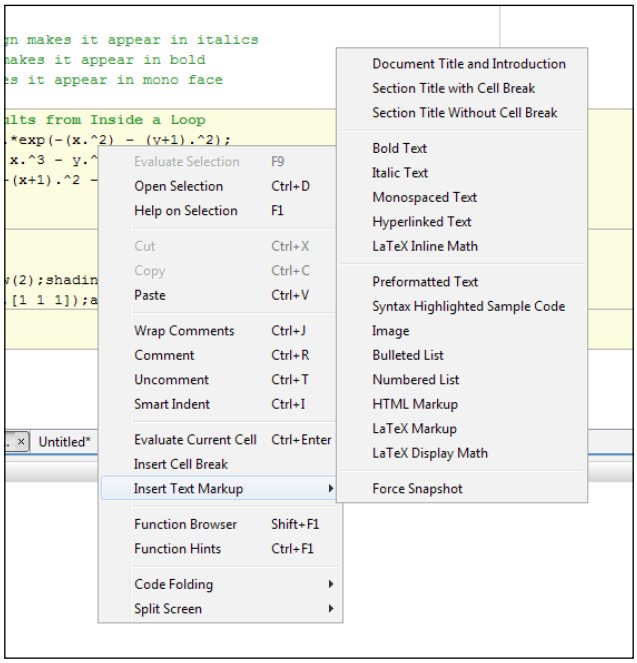
```
%% Add Basic Text Formatting
% Enclose text using the:
%
% * _underscore_ sign makes it appear in italics
% * *asterisk* sign makes it appear in bold
% * |pipe| sign makes it appear in mono face
```

7. Output code and graphics from inside a loop:

```
%% Add Code and Results from Inside a Loop
zN{1} =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2);
zN{2} = - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2);
zN{3} =  - 1/3*exp(-(x+1).^2 - y.^2);

for i = 1:3
  %%
  %
  figure;
  surf(zN{i});view(2);shading interp;
  set(gcf,'Color',[1 1 1]);axis tight;
  print(gcf,'-djpeg','-r400',...
    ['3165_08_02_' num2str(i+1) '.jpeg']);
end
```

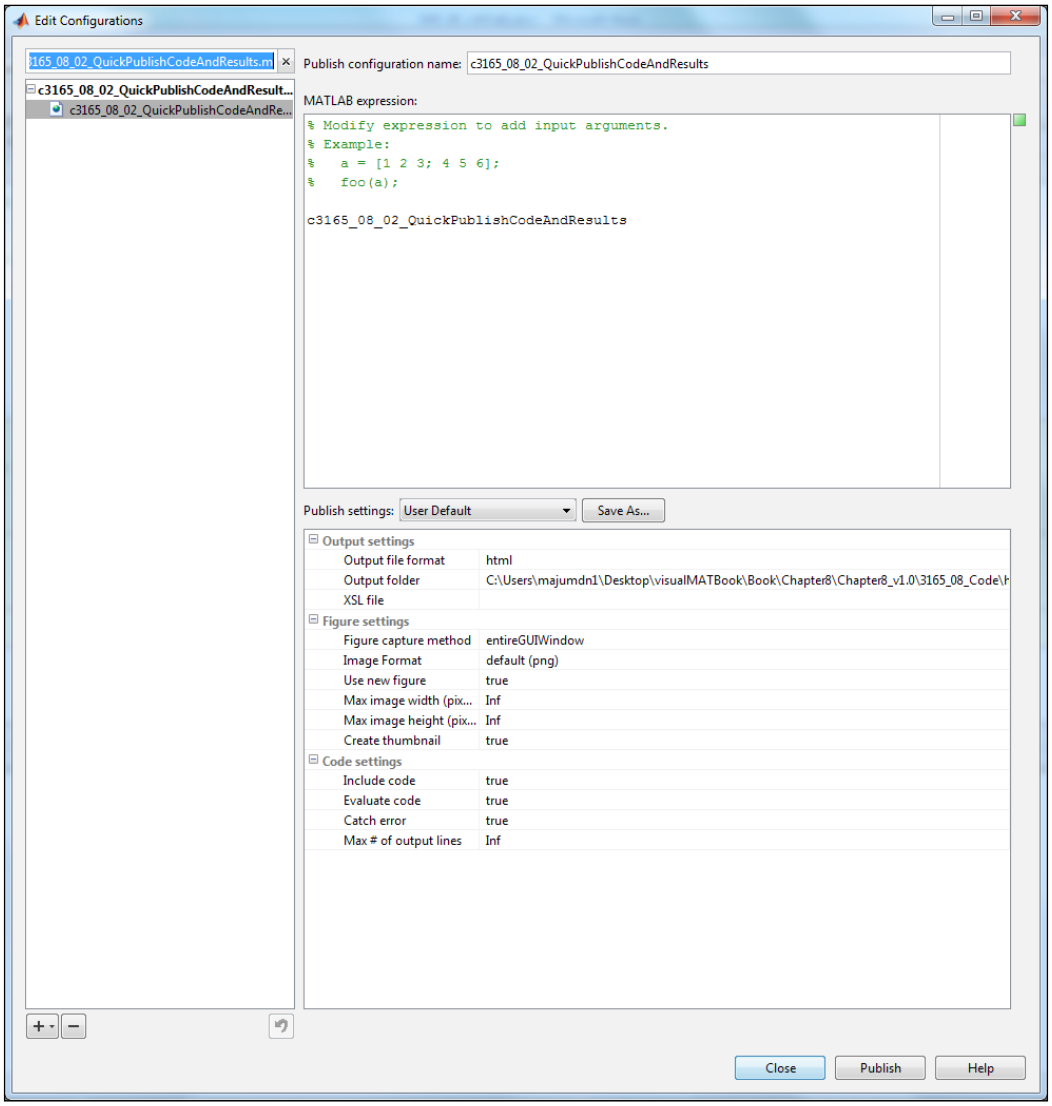
How it works...

The `publish` command makes it easy to generate annotated code with equations and textual formatting including output figures. You organized your code into cells that can be separately evaluated and the outputs from each is directed to a format of your choice (here, the default, which is HTML, was used). You learned how to markup the text to add headings, body text, equations, and textual formatting. An alternate way to do this is to right-click with your mouse on the text area of the code editor window, select the **Insert Text Markup** option, and select appropriately from the submenu items. Note that without any explicit action from you, all the section headings are brought together to make a contents item.



There's more...

It is possible to configure the publish parameters for a file, or even a function, and save it for future use. Click on the drop-down arrow next to the **publish** icon on the toolbar to access the **Edit Publish Configurations for...** option. It brings up the following window:



In the middle-right panel, where it says **MATLAB expression**, you can see that the name of the script or function whose parameters are being modified is mentioned. If the file is a function that takes certain parameters, they can be specified here, just as you would call it from the MATLAB prompt. Other settings can also be specified as shown in the lower right panel. You can save these settings with the **Publish configuration name** of your choice that you can specify in the upper-right textbox. You can save more than one setting for the same file and run the file with either.

Takeaways from this recipe:

- Use the **publish** option to share documented code and graphics with your audience

See also

Look up **MATLAB help** on the `publish` command.

References

- *Beautiful Evidence*, *Edward Tufte*
- *The Visual Display of Quantitative Information*, *Edward Tufte*
- *Envisioning Information*, *Edward Tufte*
- *Visual Explanations: Images and Quantities, Evidence and Narrative*, *Edward Tufte*
- *Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods*, *William S. Cleveland* and *Robert McGill*
- *The Structure of the Information Visualization Design Space*, *Stuart K. Card* and *Jock Mackinlay*
- *Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays*, *Christopher Ahlberg* and *Ben Shneiderman*
- *High-Speed Visual Estimation Using Preattentive Processing*, *C. G. Healey*, *K. S. Booth*, and *J. T. Enns*
- *Automating the Design of Graphical Presentations of Relational Information*, *Jock Mackinlay*
- *How NOT to Lie with Visualization*, *Bernice E. Rogowitz* and *Lloyd A. Treinish*
- *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*, *Ben Shneiderman*
- *A Tour Through the Visualization Zoo*, *Jeffrey Heer*, *Michael Bostock*, and *Vadim Ogievetsky*
- *3D Grand Tour for Multidimensional Data* and *Clusters*, *Li Yang.*
- *Trends in Interactive Visualization*, *Elena Zudilova-Seinstra*, *Tony Adriaansen*, and *Robert van Liere.*

- Course materials from *Scientific and Statistical Visualization*, *George Mason University*, *Fairfax*, *VA*. *Daniel B Carr*
- Course materials from *Statistical Graphics and Data Exploration*, *George Mason University*, *Fairfax*, *VA*. *Daniel B Carr*

Data repositories on the Web:

- University of California Irvine Machine Learning data depository (`http://archive.ics.uci.edu/ml/`)
- Datasets for *The Elements of Statistical Learning* maintained by Stanford University (`http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html`)
- KD Nuggets database (`http://www.kdnuggets.com/datasets/`)
- The Stanford 3D Scanning Repository (`http://graphics.stanford.edu/data/3Dscanrep/`)

Index

2D node link plots
about 97-99
working 99
2D scatter plots 88
3D scatter plots
about 144
steps 144-146
working 147
.csv files 60

A

adjacency matrix 67
algorithm test results
comparing 44-47
alpha values 127
ambient light 128
Andrews' curves 197-199
animation
about 231
creating 209
creating, erase and redraw strategy 236-241
with playback of frame captures 231, 232
annotations command
about 18
using 18
working 19-21
aspect ratio 137
issue 250
preserving 250-252
auto correlation function 81
AVI (Audio Video Interleave) object 231
Axes Properties 9
azimuth 137

B

backface lighting option 134
bar plots 50
bidirectional error bars
about 95
working 97
bin size 75
bit depth 246
bitmap 246
box-and-whisker diagram. *See* **box plots**
box plots
about 54
building, steps 55-58

C

calendar heat map
about 71
working 71-74
callback functions
about 155, 159, 210-213
data, passing 215
camera motion
exploring 170, 171
working 172, 173
child object 8
Choropleth maps
about 111, 112
working 113, 114
clustergram
about 100-102
working 103
Code Cells 253

color 123
compression 232
cone plots 144
contour plots
about 103, 104
working 104-106
contour slices 167
correlogram 81
cross-sectional views. *See* **slice**
Current Object Properties 9
curse of dimensionality 176
customized multigraph layout 43

D

data brushing
about 220-223
working 224
data transformations
details, visualizing 34-37
dendrogram
about 100-102
working 103
density plots
about 92
creating 94
detrend 80
distributional data
analyzing 75, 76
dots per inch value. *See* **DPI**
downsampling
for fast graphs 199-201
DPI 245
drawnow property 241

E

elevation 137
EMF 247
Enhanced Metafiles. *See* **EMF**
envelope 76
erase and redraw strategy
used, for creating animation 236-241
error bars 17, 95
excel 10
exploratory data analysis 176
export formats
options 244-247

F

false color 128
fast graphs
downsampling 199-201
fft 83
figure dimensions 28
figure legends 28
Figure Properties 9
flow maps
about 117-120
working 120
Fourier transform 82
frame rate 232
frequency 82

G

Ghostscript formats 246
global smoothing 83
glyph
about 186-188
working 189-191
grand tour 176
graphics
about 243
exporting 244
graphics object properties
programmatic manipulation 8
grid command 22

H

handle 7
Handle Graphics Objects 8
heat maps 88
histogram 75
human heart rate data
analyzing 79-85
Hyperspectral data
about 176
ENVI data, working 179, 180
fusing 177, 178

I

interactive graphics
creating 209

isocaps
about 157
working 158-160
isonormal
about 157
working 158-160
isosurface
about 156
working 158-160

J

JPEG 245

L

LaTeX syntax 250
legend
about 28
line specs, working 31-34
using 28, 30
lighting
about 128-130
back face lighting option 134
interacting, with transparency 138-141
interacting, with view command 138-141
vertex normals 132
working 131, 132
lighting parameters 166
line plots 50
line specification 11
Line specs 31
linked axes 220, 221

M

magnifying glass demo
about 226-228
working 230
magnifying glass function 226
MATLAB command
acf 85
alim 128
alpha 74, 117, 121, 128, 170
alphamap 128
andrewsplot.m 199
annotation 21
area 66
axis 44, 138, 141
bar 28, 54
biplot 206
boxplot 59
brush 226
bubbleplot3 148
calendar 74
callback property 216
camdolly 174
camera 174
cameraupvector 138, 141
camlight 134
camlookat 174
camorbit 174
campan 174
campos 138, 141, 148, 156, 172, 174
camproj 174
camroll 174
camtarget 174
camup 174
camva 174
camzoom 174
cat 95
clabel 107
clim 114
cluster 103
colorbar 74, 148
contour 107, 117
contourc 107
contourf 107
contourslice 156
contourslice 170
cumsum 66
daspect 138, 141
datenum 63
datestr 44
delaunay 111
DelaunayTri 111
dendrogram 103
dsxy2figxy 21
errorbar 17-18, 97
fft 85
fill 114
findobj 12, 14
flexlegend 34
get 14
getappdata 216

getframe 231, 232
glyphplot 192
glyphplotfunction 189
gplot 71
grid 28, 34
griddata 111
hggroup 194
hist 79
ifft 85
image 230
imagesc 74
imellipseginput 220
imfreehandimrect 220
imlineimpoly 220
imwrite 232
interp2 95
interpl 79
interpstreamspeed 236
intersect 95
isocap 160
isonormals 134, 160
isosurface 160
legend 14, 186, 194
light 134
line 21, 28, 34, 54, 121
linkage 103
linkaxes 226
linkdata 226
linspace 21, 186
loglog 37
material 134
mesh 111
meshgrid 95
meshgrid 111
multibandread.m 180
parallelcoord 194
patch 134, 160, 186
pcolor 111
pdist 103
peak 107
pie 50-54
plot 14, 54
plotmatrix 90-92
plotrectangles 197
plotyy 37
polyfil 14
polyval 14
princomp 206
print 247, 249
publish 258
qqplot 79
qqplot function 77
quiver 117
scatter 14, 54, 92
scatter3 147
scatterhist 92
semilogx 37
semilogy 37
set 14
setappdata 216
shading 114
slice 156, 170
smooth3 148
sort 14
squeeze 156
stairs 52, 54
stems 51, 54
stream3 165, 170
stream3 function 172
streamline 165, 170
streamlineparticles 236
streamlineparticles function 236
streamlines 162
streamslice 162
streamtube 165, 170
subplot 44
subplot function 38
surf 95, 100, 111, 114, 117, 124
treemap 197
treeplot 71
triplot 111
trisurf 111
uimenu 216
userdata property 216
view 138, 141
volumebounds 165, 170
WindowButtonMotionFcn 230
xlsread 14
zoom 141

MATLAB graphics objects 7

MATLAB graphics object property

FaceVertexAlphaData 127
findobj command 13
MarkerEdgeColor 236

MarkerFaceColor 236
MATLAB plot
creating 10-12
working 13
MATLAB property
Alim 128
AlimMode 128
AlphaData 127-128
AlphaDataMapping 127-128
Alphamap 128
drawnow 241
EdgeAlpha 128
EraseMode 241
FaceAlpha 128
FaceVertexAlphaData 128
PaperOrientation 252
PaperPosition 252
PaperPositionMode 252
PaperUnit 252
ParticleAlignment 236
refreshdata 241
Xdata 241
XDataSource 224
Ydata 241
YDataSource 224
MATLAB Property Editor 9
mesh 88
multigraph layouts
about 37
designing 38-42
working 43, 44
multi-tiered tick labels
about 59
working with 59

N

namevoyager.com 63
Natural Neighbor 109
node link plots
about 66
example 68-70
working 67
Normalized Figure Units 19
Nyquist Sampling Theory 201

P

Parallel coordinate plots
about 192
working 194
parent object 8
patch elements 88
PCA
about 202-204
working 204, 205
perception 137
pie charts
about 50, 54
plotting 50
pin annotation to axes 21
pixels per inch. *See* **PPI**
Plot Edit button 8
positional coordinates 50
power spectrum 82
PPI 245
Principal Component Analysis. *See* **PCA**
PCA 245
projection pursuit algorithm 176
Property 7
property editor 7
Property Editor
graphics object properties, altering 9, 10
Property Inspector Table 9
property settings 7
publish icon 257

Q

quality 232
quantile-quantile plots 77
quiver lines 116

R

Radial Coordinate Visualization
about 176, 206
working 208
references 260
refreshdata property 241
residuals 78

resolution
options 244-247

S

scattered data
gridding 107, 109
working 109, 110
scatter plot matrix 90
scatter plots 50, 88
scatter plot smoothing
about 92
working 94, 95
screen font size
preserving 250, 251
setappdata functions 213
shading algorithm 88
Show Property Editor 10
slice
visualizing 148-154
working 154-156
smooth playback 232
sparkline
about 59
creating, steps 60, 62
working 62
stacked line graphs
about 63
creating, steps 63, 64
working 65
stairs plots
about 52, 54
creating 52
static graphic 209
stem plots
about 51, 54
creating 51, 52
stream lines 144
stream particle animation
about 233
creating 234
working 235, 236
stream plots
about 162-165
working 165
stream ribbons 162
stream slice
about 160
steps 160
working 161, 162
stream tubes 162
string concatenation operator [] 13
surface elements 88
survey plots
about 180-184
working 185, 186

T

task
alternative ways to pass data to callback 215
break title into multiple lines 13
custom layout 40
define custom response to user interaction 215
linear least squares fit 13
map data value to a color value 70
multi-tiered tick label 59
natural neighbpr interpolation 109
rotate labels 16
string concatenation 13
turn off scientific notation of tick label 35-36
using non-uniform colors in contour plot 106
thematic maps with symbols technique
about 114, 115
working 116
tick labels
about 14
rotating 14, 15
working 16, 17
time series
about 79
analyzing 79-85
transparency 166
about 123-126
interacting, with lighting 138-141
interacting, with view command 138-141
working 127, 128
Tree maps
about 195
working 196, 197
tree plot 71

Tufte style gridding
about 22-27
working 27
two-dimensional scatter plots
about 88
using 89-91
working 91, 92

U

user input
obtaining, from graph 216-219

V

vector graphics
about 246-249
documents, including 247-249
VertexNormals
effects 132, 133
view command
about 137
interacting, with lighting 138-141
interacting, with transparency 138-141
view control
about 135, 136
aspect ratio 137
working 137
volumetric data
about 143
scalar 144
vector 144

W

webpage
code, publishing 253-258
graphics, publishing 253-258

Thank you for buying
MATLAB Graphics and Data Visualization Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

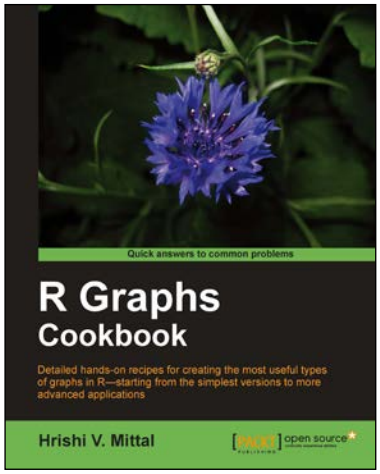
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING



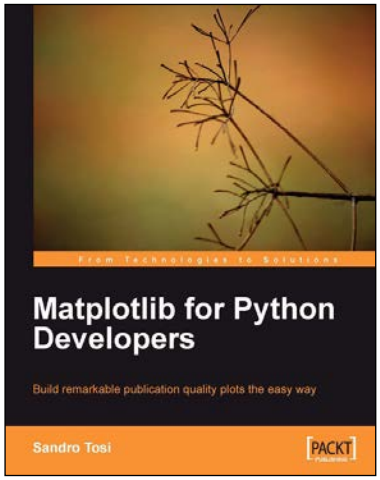
R Graphs Cookbook

ISBN: 978-1-84951-306-7 Paperback: 272 pages

Detailed hands-on recipes for creating the most useful types of graphs in R—starting from the simplest versions to more advanced applications

1. Learn to draw any type of graph or visual data representation in R
2. Filled with practical tips and techniques for creating any type of graph you need; not just theoretical explanations
3. All examples are accompanied with the corresponding graph images, so you know what the results look like
4. Each recipe is independent and contains the complete explanation and code to perform the task as efficiently as possible

---



Matplotlib for Python Developers

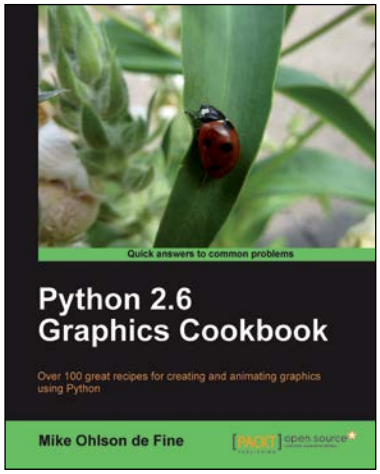
ISBN: 978-1-84719-790-0 Paperback: 308 pages

Build remarkable publication quality plots the easy way

1. Create high quality 2D plots by using Matplotlib productively
2. Incremental introduction to Matplotlib, from the ground up to advanced levels
3. Embed Matplotlib in GTK+, Qt, and wxWidgets applications as well as web sites to utilize them in Python applications
4. Deploy Matplotlib in web applications and expose it on the Web using popular web frameworks such as Pylons and Django

Please check **www.PacktPub.com** for information on our titles

[PACKT]
PUBLISHING



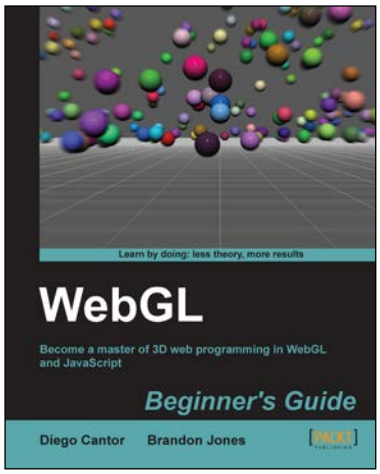
Python 2.6 Graphics Cookbook

ISBN: 978-1-84951-384-5 Paperback: 260 pages

Over 100 great recipes for creating and animating graphics using Python

1. Create captivating graphics with ease and bring them to life using Python
2. Apply effects to your graphics using powerful Python methods
3. Develop vector as well as raster graphics and combine them to create wonders in the animation world
4. Create interactive GUIs to make your creation of graphics simpler

---



WebGL Beginner's Guide

ISBN: 978-1-84969-172-7 Paperback: 376 pages

Become a master of 3D web programming in WebGL and JavaScript

1. Dive headfirst into 3D web application development using WebGL and JavaScript
2. Each chapter is loaded with code examples and exercises that allow the reader to quickly learn the various concepts associated with 3D web development
3. The only software that the reader needs to run the examples is an HTML5 enabled modern web browser. No additional tools needed

Please check **www.PacktPub.com** for information on our titles