Chapter No. 1
"Customizing Elements of MATLAB Graphics— the Basics"
In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.1 "Customizing Elements of MATLAB Graphics— the Basics"

A synopsis of the book’s content

Information on where to buy this book

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.

For More Information:
MATLAB Graphics and Data Visualization Cookbook

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.

For More Information:
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.
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.

For More Information:
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:

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

The Plot Edit button is circled in the following screenshot:
Chapter 1

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:

For More Information:
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:

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

2. Label the x and y axis:
   
   ```matlab
   xlabel(['Independent Variable: ' headerLabels{1}]);
   ylabel(['Dependent Variable: ' headerLabels{2}]);
   ```

3. Add a title:
   
   ```matlab
   title('Scatter plot view of sorted data');
   ```

For More Information:

The output at this point should be as follows:

4. Estimate the trend (using a linear least squares fit):
   \[
   p = \text{polyfit}(\text{numericData}(I,1),\text{numericData}(I,2),1);
   y = \text{polyval}(p,\text{numericData}(I,1));
   \]

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.
   \[
   \text{hold on;}
   \text{plot(\text{numericData}(I,1),y,'r--');}
   \]

6. Add a legend:
   \[
   \text{legend(\{'Data','Fit'\},'Location','NorthWest');}
   \]
Customizing Elements of MATLAB Graphics—the Basics

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.

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

The effect of step 7 is as follows:
Chapter 1

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.

For More Information:
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:

   ```matlab
   % 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:
   
   ```matlab
   rotateXLabels(gca, 45);
   ```

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

   ![Graph showing gene expression levels for various cancers](image)

   **How it works...**

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

   For More Information:
   
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

For More Information:

**Customizing Elements of MATLAB Graphics—the Basics**

**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.

```matlab
load stdNormalDistribution;
```

### How to do it...

Perform the following steps:

1. Plot the data. Add the line at the mean position:
   ```matlab
   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:
   ```matlab
   [xmeannfu ymeannfu] = dsxy2figxy(gca,[.5,0],[.15,.05]);
   annotation('textarrow',xmeannfu,ymeannfu,'String','Mean',...)
   ```

For More Information:

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.

*For More Information:*

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.

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

---

For More Information:

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:

![Image](image.png)

**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.

**For More Information:**

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
   ```

For More Information:

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

```matlab
% 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:

   ![Graph showing distribution of ethnicities in each income group of SF bay area residents](image)

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'});
5. Turn the automated grid off:
   ```matlab
   grid off;
   ```

6. Add custom grid lines:
   ```matlab
   % 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,'Color',[.7 .7 .7],...
        'LineStyle','-');
   ```

For More Information:
Customizing Elements of MATLAB Graphics—the Basics

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(2)]',[YgridPos(end),...]
    YgridPos(end)],'Color',[1 1 1],'LineStyle','-');
line([xLimits(1) xLimits(end)]',[YgridPos(1),...]
    YgridPos(1)],'Color',[1 1 1],'LineStyle','-');
line([xLimits(end) xLimits(end)]',[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]);
anotation('textarrow',xmeannfu,ymeanfuu,...
    'String',{"Grid lines','to separate categories','with one missing bound'});
[xmeannfu ymeannfu]=...
    dsxy2figxy(gca,[6.5,8.5],[1600,1600]);
anotation('arrow',xmeannfu,ymeanfuu);
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;

For More Information:
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.

For More Information:
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 \( c_{\text{min}} \) and the last row corresponds to the color with which to represent \( c_{\text{max}} \). \( c_{\text{min}} \) and \( c_{\text{max}} \) correspond to the minimum and maximum values in your data. When you make a plot, MATLAB automatically sets the \( c_{\text{min}} \) and \( c_{\text{max}} \) 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.

```matlab
load 10NormalDistributions
```

For More Information:

**How to do it...**

Perform the following steps:

1. Plot the data with basic labeling:
   ```matlab
   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:

```matlab
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.

```matlab
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:

![Image of ten different normal distributions](image.png)

**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:

```matlab
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'],...```

For More Information:

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:

```matlab
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
```

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:

```matlab
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
```
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.)
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).

\[ y_1 = r \cdot (1+E)^x; \]

How to do it...

Perform the following steps:

1. Plot the original and transformed data with the `plotyy` command:
   
   ```matlab
   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',':');
   
   [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:
      
      ```matlab
      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');
      ```

For More Information:

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:

```matlab
% 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));
```

For More Information:
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:

```matlab
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:

![Graphs showing data transformations](image)

**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.

For More Information:
Getting ready

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

```matlab
% 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.

   ```matlab
   % 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
   ```

For More Information:

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

For More Information:
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',datetick('linspace(dateAAPL(idx(1)),...
   dateAAPL(idx(end)),15),'dmmmyy'));
   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):

```matlab
% Layout the axis
Panel2 = axes('Position', [0.0570 .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), 'ddmmyy')));
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):

```matlab
% Layout the axis
Panel3 = axes('Position', [0.0570 .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
```
Customizing Elements of MATLAB Graphics—the Basics

% 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.

For More Information:
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 \( H = \text{subplot}(m,n,p) \), or \( \text{subplot}(mnp) \), 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:

For More Information:
Customizing Elements of MATLAB Graphics—the Basics

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'});
   categoryLabels = {'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
   For More Information:
Customizing Elements of MATLAB Graphics—the Basics

```matlab
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):

```matlab
% 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([categoryLabels{i} char(10) ...
     '#Samples' char(10) ' = ' num2str(ylimup)],...
     'Fontsize',11);
end

4. Add annotations:

```matlab
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 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.

---

**For More Information:**

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

For More Information:

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.