gnuplot
IN ACTION

Understanding data with graphs

SECOND EDITION

Philipp K. Janert

MANNING

Praise for the First Edition

Knee-deep in data? This is your guidebook to exploring it with gnuplot.

—Austin King
Mozilla

Sparkles with insight about visualization, image perception, and data exploration.

—Richard B. Kreckel
GiNaC.de

Incredibly useful for beginners—indispensable for advanced users.

—Mark Pruett
Dominion

Bridges the gap between gnuplot's reference manual and real-world problems.

—Mitchell Johnson
Border Stylo

A Swiss Army knife for plotting data.

—Nishanth Sastry
University of Cambridge / IBM

Plain and simple: if you use Gnuplot and would like to understand it better, this book is for you. If you are looking for an excellent plotting tool—one that is highly configurable and can easily handle millions of data points, then download Gnuplot and get this book.

—Amazon reviewer

Gnuplot in Action

SECOND EDITION

PHILIPP K. JANERT

MANNING
SHELTER ISLAND

The purpose of computing is insight, not numbers.

—R. W. Hamming

The purpose of computing is insight, not pictures.

—L. N. Trefethen

brief contents

contents

ix

preface

On New Year's Day, 2015, the gnuplot development team released version 5.0—the first major new gnuplot release in over 10 years! I decided to take this opportunity to bring *Gnuplot in Action* up to date and to cover all the new features gnuplot has acquired since the first edition of this book was written (in 2007).

It quickly became apparent it wouldn't be sufficient to just add a couple of chapters explaining the new features. In fact, the book you're reading now has been almost entirely rewritten from scratch. Most of the material from the first edition has been retained, but it's been heavily rearranged to accommodate the addition of new topics and to reflect the changes in my own understanding and priorities.

Gnuplot 5 is largely backward compatible with previous versions, and hence most of the first edition remains valid. At the same time, new features have been added to all parts of gnuplot, either to add new functionality or to streamline and improve the existing usage. Although many of the new features are small by themselves, when taken together, their cumulative effect leads to a significantly different, more sophisticated user experience.

In the process, the book's page count has increased substantially from the first edition. To keep the physical dimensions of the printed book in check without having to sacrifice important and useful material, some topics of a more specialized nature have been relegated to the electronic (e-book) version. Access to the e-book is included in the purchase of a print copy of the book.

Today, gnuplot is still going strong. Despite increased competition, gnuplot's two most attractive features are still largely unmet by other tools:

- The ability to explore data graphically, with an absolute minimum of effort, protocol, overhead, or boilerplate

- The ability to create immaculate, very high-quality graphs, with text labels and other decorations, for presentation purposes

What's new is that gnuplot has arrived in the 21st century. Color is now the standard, font handling is up to date, and the graphing backend makes use of all contemporary technologies to create the best-looking graphs possible.

In the first edition, I wrote that gnuplot was "an indispensable part of my toolbox: one of the handful of programs I can't do without." Several years on, this is still true.

acknowledgments

about this book

This book is intended to be a comprehensive introduction to gnuplot: from the basics to the power features and beyond. In addition to providing a tutorial on gnuplot itself, it demonstrates how to apply and use gnuplot to extract insight from data.

The gnuplot program has always had complete and detailed reference documentation, but what was often missing was a continuous presentation that tied all the different bits and pieces of gnuplot together and demonstrated how to use them to achieve certain tasks. This book attempts to fill that gap. It should also serve as a handy reference for more advanced gnuplot users and as an introduction to graphical ways of knowledge discovery.

And finally, this book tries to show you how to use gnuplot to achieve some surprisingly nifty effects that will make everyone say, "How did you do that?"

Contents of this book

This book is divided into four parts. Part 1 consists of chapters 1 through 3 and is intended as a tutorial introduction to get you started with gnuplot. These three chapters cover all the truly essential material so that by the end of chapter 3, you should be able to handle most basic plotting tasks in gnuplot.

Whereas part 1 only skims the surface, part 2 goes into depth. First, chapters 4 and 5 lay more groundwork by talking about the ins and outs of file formats, string handling, and other practical matters. Then, chapters 6 through 8 discuss the various ways to change the appearance of a plot: using different plotting styles; adding labels, arrows, or other decorations; and changing the axes and their subdivisions. These chapters cover the tactical aspects of working with gnuplot in detail.

Part 3 turns its attention away from individual graphs and addresses a variety of more technical aspects. First, in chapter 9, you'll learn more about color specification, point and line types, and other relatively low-level graph elements. Chapter 10 explains how to export plots to common graphics file formats. Finally, chapters 11 and 12 address ways to improve the overall workflow through scripting and configuration changes.

In the last part, I'll mostly take gnuplot's features for granted and concentrate on the things you can do with them. Chapter 13 presents various fundamental types of graphs and explains when and how to use them. Chapter 14 is more advanced and offers solutions to some recurring topics in graphical analysis, before we end the book with a reminder of what it's all about in chapter 15.

The book has several appendixes. Appendix A explains how to obtain, build, and install gnuplot. Appendix B provides pointers to some relevant resources.

Finally, some topics of a more specialized character have been relegated to a set of supplemental appendixes: appendixes C and D discuss three-dimensional surface plots and false-color plots (heatmaps), appendix E treats some special types of graphs, and appendix F covers more mathematical topics. To reduce the physical dimensions of the print book, these four appendixes are only available in the electronic (e-book) version of this book. The purchase of a hard copy includes access to the e-book as well—you can find instructions in the front of the print book.

> **TIP** Appendixes C through F are only available in the e-book version of this book, which is included with the purchase of the hard-copy version. Check the front of the print book for instructions on how to obtain the e-book.

How to read this book

This book was written as if readers were going to read it sequentially, cover to cover. New material is presented in order, with later chapters relying only on topics introduced earlier and avoiding forward references as much as possible. I realize that this is not a realistic picture and that the need for technical information tends to arise in a much more disjointed manner. In this spirit, I offer a few different "trail maps" to the material presented here:

- If you're new to gnuplot, begin with chapters 2 and 3 and then dive into chapters 4–8 as required to pick up the skills you need to complete whatever task you want to accomplish.
- If you're comfortable creating day-to-day graphs with gnuplot, then the material in chapters 9–12 should help you achieve greater efficiency in your work and fine-tune the results.
- If you've been using gnuplot for a long time already, then make sure you read up on the new features in gnuplot 5. Chapters 5, 9, and 10, as well as parts of chapters 11 and 12 will probably be of the most immediate interest to you.

- If you're new to graphical analysis, you may want to begin with chapter 13 to learn some of the basic methods and concepts.

Finally, keep in mind that some interesting and useful material is only available in the e-book. Three-dimensional surface and contour plots are discussed in appendix C. False-color plots (heatmaps) are treated in appendix D, together with guidelines for how to construct effective color gradients for data visualization. Appendix E explains how to combine individual graphs into composites and also discusses some other specialized types of graphs. Appendix F treats topics of a more mathematical nature.

Whom this book is for

This book is intended for anyone who wants to plot and visualize data, either to explore data sets graphically, or to create attractive, high-quality graphs for presentation and publication purposes. I had two kinds of people in mind when writing this book—those who already know gnuplot, and those who don't:

- If you already know gnuplot, I hope you'll still find it a useful reference, in particular in regard to some of the more advanced topics later in the book. I've tried to provide exactly the big-picture explanations and examples that have always been missing from the standard gnuplot reference documentation.
- If you're new to gnuplot, I think you'll find it easy enough to pick up—in fact, I can promise you that by the end of chapter 2, you'll be productive with gnuplot; and by the end of chapter 3, you'll be well equipped for most day-to-day data graphing tasks that may come your way.

This book doesn't require a strong background in mathematical methods or any in statistics, but I occasionally do expect you to have at least a fleeting familiarity with simple programming concepts. A few sections naturally require some special preliminaries (for instance, some of the discussions in chapter 10 require knowledge of LaTeX, and some sections in chapter 11 use Perl or Python code), but you can safely skip those sections if their material doesn't apply to you.

Conventions

I spell the name of the program in all lowercase (gnuplot), except at the beginning of a sentence, when I capitalize it normally. This is in accordance with the usage recommended in the gnuplot FAQ.

The gnuplot documentation is extensive, and I refer to it occasionally for additional details on topics covered only briefly or not at all here. Traditionally, the gnuplot documentation has been called the online help or online documentation, owing to the fact that it's available online during a gnuplot session. But since the advent of the internet, the word *online* seems to suggest network connectivity—falsely, in this context. To avoid confusion, I'll always refer to it as the *standard gnuplot reference documentation*.

Code examples

Gnuplot commands are shown using a monospace font, like this: `plot sin(x)`. Gnuplot commands can be entered at the gnuplot command prompt as shown in the text; the prompt itself has been suppressed to save space.

Single command lines can be long; to make them fit on a page, I occasionally had to break them across multiple lines. If so, a gray arrow (➡) has been placed at the beginning of the next line, to indicate that it is the continuation of the previous one:

```
plot "data" using 1:2 smooth csplines title "data" with lines,
➡     sin(x) title "model"
```

The break in the original line isn't indicated separately. When using gnuplot in an interactive session, your terminal program should automatically wrap a line that's too long. Alternatively, you can break lines by escaping the newline with a backslash as usual. This is useful in command files for batch processing (and you'll see some examples in chapter 12 in the context of string macros).

Some code snippets are only intended to demonstrate the syntax and don't have a graph associated with them. In this case, I use the generic name "data" as a placeholder for the actual filename. No file named *data* exists in the downloads (in the same way that no key named *any* can be found on a computer keyboard). It's just a generic placeholder.

Occasionally, I show Unix commands that need to be entered in a Unix shell; to emphasize that these aren't gnuplot commands, I prefix them with a generic shell prompt, like this: `shell>`. Similarly, Python commands to be entered in a Python session are prefixed with `python>>>`.

Downloads

The code for all numbered listings is available for download from www.manning.com/books/gnuplot-in-action-second-edition, and so are the data sets. The only exception to this are publicly available data sets: for these, I provide the URL where they can be found.

Gnuplot searches for data files in the current directory, so the easiest way to run the supplied command files is as follows:

1. Change into the data directory of the downloaded bundle.
2. Start gnuplot.
3. Issue `plot` commands at the gnuplot prompt the way they're shown in the text (for example, `plot "marathon" using 1:2`), or give the full pathname to the gnuplot command file that you wish to run (for example, `load "../gnuplot/shapes.gp"`).

Command synopses

Gnuplot has a large number of options, and keeping all of them and their sub-options and optional parameters straight is a major theme running through this book. Frequently, I'll display all available options to a command in a command synopsis before discussing the options in detail. To distinguish a synopsis of available options from actual gnuplot code, a synopsis uses an italic font, like so:

```
set datafile commentschar ["{str:chars}"]
```

Within these summaries, I use a few syntactic conventions. My intent here is to stay close to the usage familiar from the standard gnuplot reference documentation, but also to follow more general conventions (such as those used for Unix man pages):

```
[ ... ]              ⟵———— Square brackets for optional parts
[ | ]                ⟵———— Vertical bars to separate alternatives
{ ... }              ⟵———— Curly braces for user-supplied input
```

For parameters supplied by the user, it's often not clear from the context what kind of information the command expects: is it a string or a number? If it's a number, is it a value selected from a fixed range of integers or a numerical factor? And so on. I've tried to clarify this situation by prefixing each user-supplied input parameter with a type indicator, terminated by a colon. I summarize the prefixes and their meanings in table 1.

Table 1   Type indicators for user-supplied parameters

Prefix	Description
str:	A string
int:	An integer number
flt:	A floating-point number
idx:	An integer number, which is interpreted as a selection from a fixed range of values
clr:	A color specification—for example, rgbcolor "red" or rgb "#FFFF00"
pos:	A pair of comma-separated coordinates, optionally containing coordinate system specifiers—for example, 0,0 or first 1.1, screen 0.9
enum:	A gnuplot keyword as unquoted string

Abbreviations

Many gnuplot commands have abbreviated forms, which I use frequently. The essential plot command, in particular, takes a large number of keyword directives, which I usually abbreviate to save space and keystrokes. I strongly recommend that you quickly become familiar with these shorthands and use them yourself. Table 2 lists both the abbreviated and the full forms. The plot command also understands a large number of appearance options (controlling aspects such as line width, style, and color), which are generally also abbreviated. A comprehensive summary of appearance options, together with their shorthands, can be found in table 6.1.

Table 2   Abbreviations for frequently used directives to
the `plot` command

Abbreviation	Full
`i`	`index`
`ev`	`every`
`u`	`using`
`s`	`smooth`
`s acs`	`smooth acsplines`
`s f`	`smooth frequency`
`s kdens`	`smooth kdensity`
`t`	`title`
`w`	`with`
`w l`	`with lines`
`w linesp` or `w lp`	`with linespoints`
`w p`	`with points`
`w vec`	`with vectors`

Table 3 lists three frequently occurring commands that are also usually abbreviated.

Table 3   Abbreviations for frequently occurring commands

Abbreviation	Full
`set t`	`set terminal`
`set o`	`set output`
`set logsc`	`set logscale`

The figures in this book

The graphs in this book were generated with gnuplot; some special cases were handled using pic. All graphs were originally prepared in color, using my own set of preferred colors instead of one of gnuplot's default color schemes. The color versions of the graphs are used in the electronic (e-book) version of this book. For the print book, I prepared black-and-white versions through the application of an appropriate stylesheet (see chapter 12). A handful of graphs required manual touch-ups in addition to the monochrome stylesheet to yield an optimal appearance.

You'll find the line-type definitions of both the color and the black-and-white stylesheets in table 4. The same colors and dash patterns are discussed in listings 12.7 and 12.9.

In particular in the latter part of the book, I frequently use point types (point shapes) that aren't the default, because the visual appearance of the graphs can often be improved greatly this way. If so, the point type is usually chosen explicitly in the appropriate code examples and listings.

The final version of each figure was generated using the `pdfcairo` terminal, using a (non-default) aspect ratio of √2 to 1 and Helvetica as the requested font.

Table 4 Colors and dash patterns used for the color and monochrome figures in this book

Color	Monochrome
`set linetype 1 lc rgb '0xee0000'`	`set linetype 1 lc black dt solid`
`set linetype 2 lc rgb '0x008b00'`	`set linetype 2 lc black dt (8,6)`
`set linetype 3 lc rgb '0x0000cd'`	`set linetype 3 lc black dt (4,3)`
`set linetype 4 lc rgb '0xff3fb3'`	`set linetype 4 lc black dt (3,6)`
`set linetype 5 lc rgb '0x00cdcd'`	`set linetype 5 lc black dt (12,5,2,5,2,5)`
`set linetype 6 lc rgb '0xcd9b1d'`	`set linetype 6 lc black dt (16,8)`
`set linetype 7 lc rgb '0x8968cd'`	`set linetype 7 lc black dt (20,6,2,6)`
`set linetype 8 lc rgb '0x8b8b83'`	`set linetype 8 lc black dt (30,10)`

Hardware and software requirements

This book describes gnuplot version 5.0 or higher, which was initially released in early 2015. Not all examples in this book will work with earlier gnuplot versions. If you have an earlier version of gnuplot, you should upgrade to a more current version—appendix A tells you how.

I assume you have access to a reasonably modern computer running any flavor of Unix/Linux, a recent release of MS Windows, or Mac OS X. Gnuplot has been ported to many other platforms but is actively supported primarily on the three operating systems just mentioned, and so I concentrate on them in this book.

Reference materials

Command and option references are distributed throughout the book, wherever the material is first introduced. The following pointers are intended to help you find these summaries more easily.

Graphical styles and specifications

Appearance specifiers and line options	Table 6.1	page 104
Graph locations	Figure 7.2	page 128
Graph layers	Figure 7.3	page 129
Explicit color-specification formats	Table 9.1	page 184
Point types	Figure 9.7	page 196
Dash patterns	Table 9.2	page 197

File access

Column-access methods and functions	Table 4.2	page 67
Pseudofiles	Table 4.3	page 71
Metadata in data files	Table 4.4	page 76

String handling and formatting

String functions	Table 5.1	page 81
General conversion specifiers	Table 8.2	page 160
Accuracy specifiers	Table 8.3	page 160
Time-series conversions, sorted alphabetically	Table 8.4	page 172
Time-series conversions, sorted by topic	Table 8.5	page 173
Time functions	Table 8.6	page 176
Enhanced text mode	Table 10.1	page 218

Operators and mathematical functions

Unary operators	Table 3.1	page 35
Binary operators	Table 3.2	page 35
Mathematical functions	Table F.1	page F11
Complex numbers	Table F.2	page F13

Programming constructs

Inline loops	Listing 5.3	page 92
General loops and conditionals	Table 11.1	page 238

About the author

PHILIPP K. JANERT was born and raised in Germany. He obtained a Ph.D. in theoretical physics from the University of Washington in 1997 and has been working in the tech industry ever since, including four years at Amazon.com, where he initiated and led several projects to improve Amazon's order-fulfillment process. He's the author of several books on data analysis and applied math, including the best-selling *Data Analysis with Open Source Tools* (O'Reilly, 2010). He has contributed to CPAN and is an occasional committer on the gnuplot project. Visit his company website at www.principal-value.com.

Author Online

Purchase of *Gnuplot in Action, Second Edition* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the lead author and from other users. To access

the forum and subscribe to it, point your web browser to www.manning.com/books/gnuplot-in-action-second-edition. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to Author Online remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the cover

The figure on the cover of *Gnuplot in Action, Second Edition* is captioned "A peer of France." The title of Peer in France was held by the highest-ranking members of the French nobility. It was an extraordinary honor granted only to a few dukes, counts, and princes of the church. The illustration is taken from a 19th-century edition of Sylvain Maréchal's four-volume compendium of regional dress customs published in France. Each illustration is finely drawn and colored by hand.

The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

Dress codes have changed since then, and the diversity by region, so rich at the time, has faded away. It's now hard to tell apart the inhabitants of different continents, let along different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it's hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Getting started

Gnuplot is a tool for visualizing data and mathematical functions. The chapters in this first part will give a first introduction to gnuplot and its most important features. Chapter 1 introduces gnuplot and describes the kinds of problems it's designed to solve. Chapter 2 provides a quick tutorial to gnuplot. By the end of this chapter, you'll be able to prepare simple plots with gnuplot and to save and export your work. Chapter 3 takes a detailed look at the all-important `plot` command, which is used to generate most graphs in gnuplot. You'll also learn about inline transformations and built-in smoothing methods.

Prelude: understanding data with gnuplot

1

This chapter covers

- Warmup examples
- What is graphical analysis?
- What is gnuplot?

NOTE TO PRINT BOOK READERS   Some material of a more specialized nature is only available in the e-book version of this book. The e-book also shows all the graphs in color. To get your free e-book in PDF, ePub, or Kindle format, go to www.manning.com/books/gnuplot-in-action-second-edition to register your print book.

Gnuplot has long been one of the most popular open source programs for plotting and visualizing data. In this book, I want to show you how to use gnuplot to make plots and graphs of your data: both quick and easy graphs for your own use and highly polished graphs for presentations and publications.

But I also want to show you something else: how to solve data-analysis problems using graphical methods. The art of discovering relationships in data and extracting information from it by visual means is called *graphical analysis*, and I believe gnuplot to be an excellent tool for it.

As a teaser, let's look at some problems and how you might be able to approach them using graphical methods. The graphs here and in the rest of the book (with very few exceptions) have been, of course, generated with gnuplot.

1.1   A busy weekend

To get a feeling for the kinds of problems you may be dealing with and for the kinds of solutions gnuplot can help you find, let's look at two examples. Both take place during a long, busy weekend.

1.1.1   Planning a marathon

Imagine you're in charge of organizing the local city marathon. There will be more than 2,000 starters, traffic closed around the city, plenty of spectators—and a major Finish Line Festival to celebrate the victors. The big question is: when should the Finish Line crew be ready to deal with the majority of runners? At what point do you expect the big influx of the masses?

You have the results from last year's event. Assuming that the starters haven't improved dramatically over the last year (probably a safe assumption), you do a quick average of the completion times and find that last year's average was 282 minutes. To be on the safe side, you calculate the standard deviation as well, which comes out to about 50 minutes. So you tell your crew to be ready for the big rush starting three and a half hours (210 minutes) after the start, and you feel reasonably well prepared for the event.

So it comes as a surprise when on the big day, plenty of runners start showing up at the finish line after only two hours—a good 90 minutes earlier than the expected onset of the rush. In terms of event management, the number of runners who show up early isn't overwhelming, but it's a bit strange. The next day you wonder: what went wrong?

Let's look at the data to see what you can learn about it. So far, all you know are the mean and the standard deviation.

The mean is convenient: it's easy to calculate, and it summarizes the entire data set in a single number. But in forming the mean, you lost a lot of information. To understand the entire data set, you have to *look* at it. And because you can't understand data by looking at more than 2,000 individual finish times, this means you have to *plot* it.

It will be convenient to group the runners by completion time and to count the number of participants who finished during each five-minute interval. The resulting file might start like this:

```
# Minutes Runners
135      1
140      2
145      4
150      7
155      11
160      13
165      35
170      29
...
```
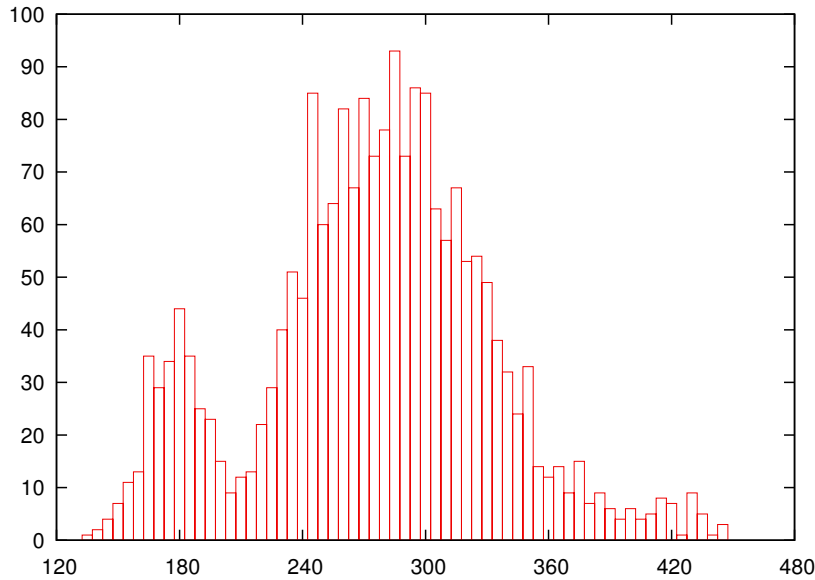
Figure 1.1   Number of finishers vs. time to complete (in minutes)

Now you plot the number of runners against the completion time (see figure 1.1). It's immediately obvious where you went wrong: the data is *bimodal*, meaning it has two peaks. There is an early peak at around 180 minutes and a later main peak at 300 minutes.

Actually, this makes sense: a major sporting event such as a city marathon attracts two very different groups of people: athletes, who train and compete throughout the year and are in it to win, and a much larger group of amateurs, who come out once a year for a big event and are mostly there to participate. The problem is that for such data, the mean and standard deviation are obviously bad representations—so much so. that at the time when you expected the big rush (200 minutes), there's a lull at the finish line!

The take-home message here is that it's usually not a good idea to rely on summary statistics (such as the mean) for unknown data sets. You *always* should investigate what the data looks like. Once you've confirmed the basic shape, you can choose how to summarize your findings best.

And of course, there is always more to learn. In this example, for instance, you see that after about 400 minutes, almost everybody has made it, and you can start winding down the operation. The actual "tail" of the distribution is quite small—surprisingly so. (I would've expected to see a greater number of stragglers, but possibly many runners who are *really* slow drop out of the race when they realize they'll place badly.)

USING GNUPLOT

Let's look at the gnuplot command that was used to generate figure 1.1. Gnuplot is command-line oriented: after you start gnuplot, it drops you into an interactive command session, and all commands are typed at the interactive gnuplot prompt.

Gnuplot reads data from simple text files, with the data arranged in columns as shown previously. To plot a data file takes only a single command, `plot`, like this:[1]

```
plot "marathon" using 1:2 with boxes
```

The `plot` command requires the name of the data file as argument in quotes. By default, gnuplot looks for the data file in the current working directory—normally the directory from which you started gnuplot. The filename provided to the `plot` command may contain path information to refer to a file that doesn't reside in the current directory.

The rest of the command line specifies which columns to use for the plot and in which way to represent the data. The `using 1:2` declaration tells gnuplot to use the first and second columns in the file called marathon. The final part of the command, `with boxes`, selects a box style, which is often suitable to display counts of events.

Gnuplot handles most everything else by itself: it sizes the graph and selects the most interesting plot range, it draws the border, and it draws the tic marks and their labels. All these details can be customized, but gnuplot typically does a good job at anticipating what the user wants.

> **NOTE**   The little markers along the edge that define the *scale* of the corresponding axis are called *tick marks* (or *tic marks*). The gnuplot standard reference documentation uses the spelling *tic* mark; the relevant commands are called `set xtics`, `set ytics`, and so on. In order to avoid confusion, I use the same spelling (*tic*) throughout this book.

1.1.2   Determining the future

The same weekend when 2,000 runners are running through the city, a diligent graduate student is working on his research topic. He studies diffusion limited aggregation (DLA), a process wherein a particle performs a random walk until it comes into contact with a growing cluster of particles. At the moment of contact, the particle sticks to the cluster at the location where the contact occurred and becomes part of the cluster. Then a new random walker is released to perform a random walk, until *it* sticks to the cluster. And so on. Clusters grown through this process have a remarkably open, tenuous structure (as shown in figure 1.2): they're *fractals*.[2]

The DLA process is simple, so it seems straightforward to write a program to grow such clusters in a computer, and this is what the busy graduate student has done. Initially, all seems well; but as the simulation progresses, the cluster appears to grow more and more slowly—excruciatingly slowly, in fact. The goal was to grow a DLA cluster in excess of 100,000 particles. Will the program ever finish?

---

[1] Depending on your gnuplot setup and initialization, your graphs may look slightly different from the figures shown in this chapter. We'll discuss user-defined appearance options starting with chapter 6.

[2] The original paper on DLA was "Diffusion Limited Aggregation, A Kinetic Critical Phenomenon" by T. A. Witten and L. M. Sander, *Physical Review Letters* 41 (1981): 1400. It's one of the most-quoted papers from that journal of all time. If you want to learn more about DLA and similar processes, check out *Fractals, Scaling, and Growth Far From Equilibrium* by Paul Meakin (Cambridge University Press, 1998).
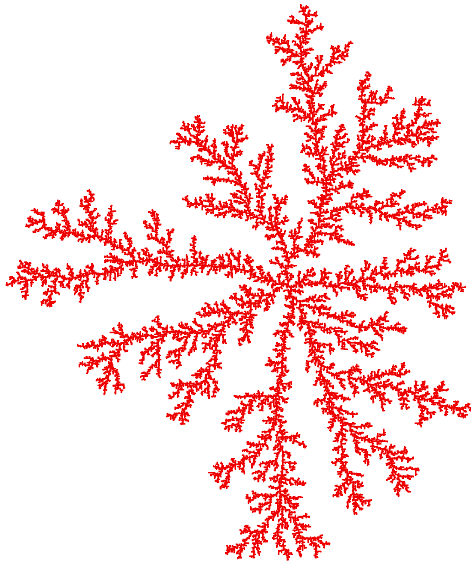
Figure 1.2   A DLA cluster of *N*=50,000 particles, drawn with gnuplot

Luckily, the simulation program periodically writes information about its progress to a log file: for each new particle added to the cluster, the time (in seconds) since the start of the simulation is recorded. The grad student should be able to predict the completion time from this data, but an initial plot (figure 1.3) isn't helpful; there are too many ways this curve can be extrapolated to larger cluster sizes.
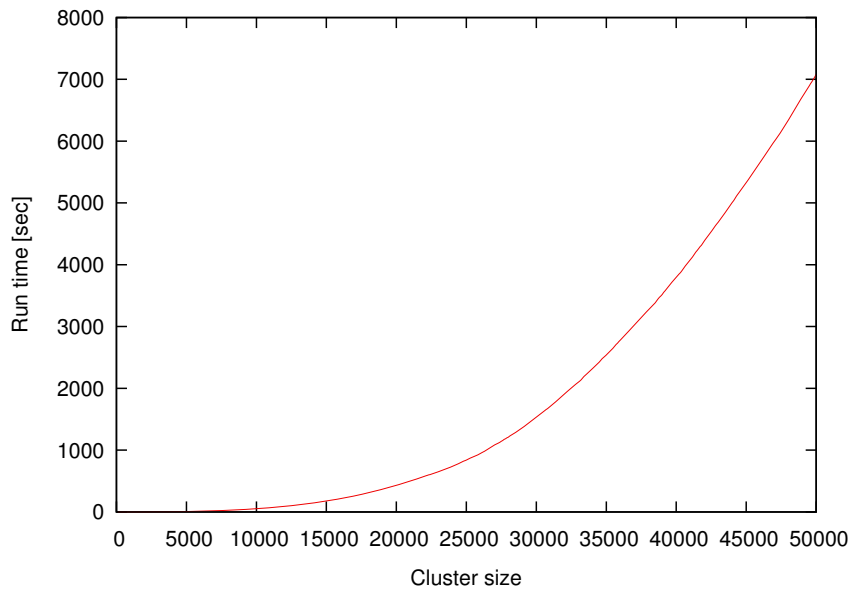
Figure 1.3   Time required to grow a DLA cluster

The time consumed by many computer algorithms grows as a simple power of the size of the problem. In this case, this would be the number N of particles in the cluster $T \sim N^k$, for some value of k. The research student therefore plots the running time of his simulation program on a double-logarithmic plot versus the cluster size (see figure 1.4). The data points fall on a straight line, indicating a power law. (I'll explain later how and why this works.) Through a little trial and error, he also finds an equation that approximates the data quite well. The equation can be extended to any cluster size desired and will give the time required. For N=100,000 (which was the original goal), he can read off almost T=100,000 seconds (or more), corresponding to more than 24 hours, so there is no point in your friend spending the weekend in the lab—he should go out (maybe run a marathon) and come back on Monday, or perhaps work on a better algorithm. (For simulations of DLA cluster growth, dramatic speedups over the naive implementation are possible. Try it if you like.)
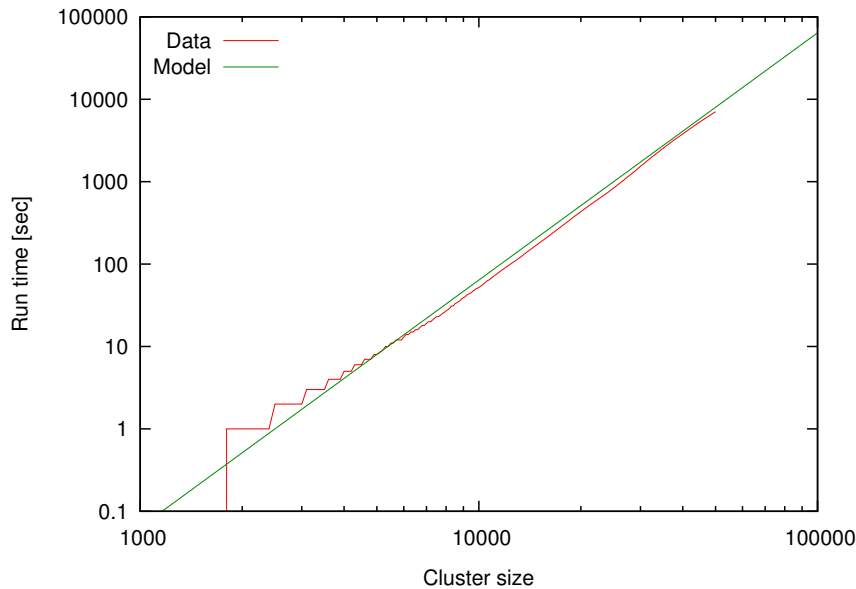
Figure 1.4   Time required to grow a DLA cluster in a double-logarithmic plot, together with an approximate mathematical model

USING GNUPLOT

Again, let's see how the graphs in this section were created. The easiest to understand is figure 1.3. Given a file containing two columns, one listing the cluster size and the other listing the completion time, the command is just

```
plot "runtime" using 1:2 with lines
```

The only difference compared to figure 1.1 is the style: rather than boxes, I use line segments to connect consecutive data points: with lines.

Did you notice that figure 1.3 and figure 1.4 contain more than just data? Both axes are now labelled! Details such as labels and other helpful decorations often make the difference between a mediocre and a high-quality graph, because they provide the observer with the necessary context to fully understand the graph.

In gnuplot, all details of a graph's appearance are handled by setting the appropriate options. To place the labels on the x and y axes in figure 1.3, I used

```
set xlabel "Cluster size"
set ylabel "Run time [sec]"
```

Figure 1.4 is drawn using double-logarithmic axes. This is another option, which is set as follows:

```
set logscale
```

Figure 1.4 shows two curves: the data together with a best "fit." Plotting several data sets or mathematical functions together in one plot is easy—you list them one after another on the command line for the plot command:

```
plot "runtime" using 1:2 title "Data" with lines,
➡     (x/2500)**3 title "Model"
```

This command introduces a further gnuplot feature: the title directive. It takes a string as argument, which is displayed together with a line sample in the plot's key or legend (visible at upper left in figure 1.4).

Finally, we come to figure 1.2. It's a somewhat different beast. Notice that the border and the tic marks are missing. The aspect ratio (the ratio of the graph's width to its height) has been constrained to 1, and a single dot has been placed at the position of each particle in the cluster. Here are the most important commands that I used:

```
unset border
unset xtics
unset ytics

set size square

plot "cluster" using 1:2 with dots
```

You can see that gnuplot is simple to use. In the next section, I talk more about using graphical methods to understand a data set, before coming back to gnuplot and discussing why it's my favorite tool for this kind of activity.

1.2   What is graphical analysis?

The previous two examples should have given you an idea of what graphical analysis is and how it works. The basic steps are always the same:

1 Plot the data.
2 Inspect it, trying to find some recognizable behavior.
3 Compare the actual data to data that represents the hypothesis from the previous step (as in the second example earlier, when our grad student plotted the running time of the simulation program together with a power-law function).
4 Repeat.

You may try more sophisticated things, but this is the basic idea. If the hypothesis in the second step seems reasonably justified, you can try to *remove* its effect—for instance, by subtracting a formula from the data—to see whether there is any recognizable pattern in the residual. And so on.

Iteration is a crucial aspect of graphical analysis: plotting the data this way and that way; comparing it to mathematical functions or to other data sets; zooming in on interesting regions or zooming out to detect the overall trend; applying logarithms or other data transformations to change its shape; using a smoothing algorithm to tame a noisy data set; and so on. During an intense analysis session using a new but promising data set, it's not uncommon to produce literally *dozens* of graphs.

None of these graphs will be around for long. They're transient, persisting just long enough for you to form a new hypothesis, which you'll try to justify in the next graph you draw. This also means these graphs aren't polished in any way, because they're the graphical equivalent of scratch paper: notes of work in progress, not intended for anyone but ourselves.

This isn't to say that polishing doesn't have its place. But it comes later in the process: once you know the results of your analysis, you need to communicate them to others. At this point, you'll create permanent graphs, which will be around for a long time—maybe until the next departmental presentation, or (if the graph will be part of a scientific publication, for instance) possibly forever!

Such permanent graphs have different requirements: other people must be able to understand them, possibly years later, and most likely without you there to explain them. Therefore, graph elements such as labels, captions, and other contextual information become very important. Presentation graphs must be able to stand by themselves.

Presentation graphs also should make their point clearly. Now that you know the results of your analysis, you should find the clearest and most easily understood way of presenting your findings. A presentation graph should make one point and make it well.

Finally, some would argue that a presentation graph should look good. Maybe. If it makes its point well, there is no reason it shouldn't be visually pleasing, too. But that's an afterthought. Even a presentation graph is about the content, not the packaging.

1.2.1 *Why graphical analysis?*

Graphical analysis is a discovery tool. You can use it to reveal as-yet-unknown information in data. In comparison to statistical methods, it helps you discover *new* and possibly unexpected behavior.

Moreover, it helps you develop an intuitive understanding of the data and the information it contains. Because it doesn't require particular math skills, it's accessible to anyone with an interest and a certain amount of intuition.

Even if rigorous model building is your ultimate goal, graphical methods still need to be the first step so that you can develop a sense for the data, its behavior, and its quality. Knowing this, you can then select the most appropriate formal methods.

1.2.2 *Limitations of graphical analysis*

Of course, graphical analysis has limitations and its own share of problems:

- *Graphical analysis doesn't scale.* It's a manual process that can't easily be automated. Each data set is treated as a separate special case, which isn't feasible if there are thousands of data sets.

  But this problem is sometimes more apparent than real. It can be remarkably effective to generate a large number of graphs and browse them without studying each one in great depth. It's totally possible to scan a few hundred graphs visually, and doing so may already lead to a high-level hypothesis regarding the classification of the graphs into a few subgroups, which can then be investigated in detail. (Thank goodness gnuplot is scriptable, so preparing a few hundred graphs poses no problem.)

- *Graphical analysis yields qualitative—not quantitative—results.* Whether you regard this as a strength or a weakness depends on your situation. If you're looking for new behavior, graphical analysis is your friend. But if you're trying to determine the percentage by which a new fertilizer treatment increases crop production, quantitative methods are the way to go.

- *It takes skill and experience.* Graphical analysis is a creative process, using inductive logic to move from observations to hypothesis. There is no prescribed set of steps to move from a data set to conclusions about the underlying phenomena, and not much can be taught in a conventional classroom format.

  But by the same token, it doesn't require formal training, either. Ingenuity, intuition, and curiosity are the most important character traits. Everyone can play this game, if they're interested in finding out what the data tries to tell them.

1.3 *What is gnuplot?*

Gnuplot is a program for exploring data graphically. Its purpose is to generate plots and graphs from data or functions. It can produce highly polished graphs, suitable for publication, or simple throw-away graphs when you're merely playing with an idea.

Gnuplot is command-line–driven: you issue commands at a prompt, and gnuplot redraws the current plot in response. Gnuplot is also interactive: the output is generated and displayed immediately in an output window. Although gnuplot can be used as a background process in batch mode, typical use is highly interactive. On the other hand, its primary user interaction is through a command language, not through a point-and-click GUI interface.

Don't let the notion of a command language throw you: gnuplot is easy to use—*really* easy to use! It takes only one line to read and plot a data file, and most of the command syntax is straightforward and intuitive. Gnuplot doesn't require programming or any deeper understanding of its command syntax to get started.

This is the fundamental workflow of all work with gnuplot: plot, examine, repeat—until you have found out whatever you wanted to learn from the data. Gnuplot perfectly supports the iterative process model required for exploratory work.

1.3.1   Gnuplot isn't GNU

To dispel one common point of confusion right away, gnuplot isn't GNU software, has nothing to do with the GNU project, and isn't released under the GNU Public License (GPL). Gnuplot is released under a permissive open source license.

Gnuplot has been around a long time—a very long time! It was started by Thomas Williams and Colin Kelley in 1986. On the gnuplot FAQ, Thomas has this to say about how gnuplot was started and why it's named the way it is:

> I was taking a differential equation class and Colin was taking Electromagnetics, we both thought it'd be helpful to visualize the mathematics behind them. We were both working as sys admin for an EE VLSI lab, so we had the graphics terminals and the time to do some coding. The posting was better received than we expected, and prompted us to add some, albeit lame, support for file data. Any reference to GNUplot is incorrect. The real name of the program is "gnuplot." You see people use "Gnuplot" quite a bit because many of us have an aversion to starting a sentence with a lower case letter, even in the case of proper nouns and titles. gnuplot isn't related to the GNU project or the FSF in any but the most peripheral sense. Our software was designed completely independently and the name "gnuplot" was actually a compromise. I wanted to call it "llamaplot" and Colin wanted to call it "nplot." We agreed that "newplot" was acceptable but, we then discovered that there was an absolutely ghastly pascal program of that name that the Computer Science Dept. occasionally used. I decided that "gnuplot" would make a nice pun and after a fashion Colin agreed.

1.3.2   Why gnuplot?

I have already indicated why I like gnuplot, but three primary reasons stand out to me:

- Gnuplot lends itself to ad hoc, *iterative work* with a minimum of effort or overhead. Plotting a data file involves only a single command, without the need (for example) to load the file into an internal data structure first. Common tasks can be accomplished simply. The absence of most programming features helps to keep the workflow focused on creating and analyzing graphs and figures.

- Gnuplot is *mature* and *robust*. Edge cases are generally detected and handled properly. Gnuplot isn't picky about input formats and is tolerant of messy or poorly formatted data sets.
- Gnuplot makes it easy to produce polished, *high-quality figures*, using good colors, fonts, and symbols. The default appearance of most graph elements is generally well-chosen or can be customized easily. Manipulating low-level graph elements, although possible, is rarely necessary.

In addition, gnuplot is scriptable and can handle large data sets. Its reliance on plain text files for data sets makes it easy to use gnuplot in combination with other tools or programs. Finally, gnuplot is free and open source, actively supported, and available for all major platforms (Linux, Windows, and Mac OS X).

1.3.3 Limitations

It's important to remember that gnuplot is a data-plotting tool, nothing more, nothing less. In particular, it's neither a numeric nor a symbolic workbench, nor a statistics package. It can therefore only perform simple calculations on the data. On the other hand, it has a flat learning curve, requiring no programming knowledge and only the most basic math skills.

Gnuplot is also not a drawing tool. All of its graphs are depictions of some data set or mathematical function. It has only very limited support for arbitrary box-and-line diagrams, and none at all for freehand graphics.

Finally, gnuplot isn't a tool to create dashboards, infographics, or virtual-reality visualizations. It's a tool for quantitative analysis, and therefore its bread and butter are dot and line plots. It has only rudimentary support for three-dimensional solid-body imaging and none whatsoever for ray-tracing, fisheye functionality, and similar techniques.

Overall, I regard these limitations more as strengths in disguise: in the Unix tradition, gnuplot is a simple tool, doing (mostly) one thing, and doing it very, very well.

1.3.4 Gnuplot 5: the best gnuplot there ever was!

On New Year's Day, 2015, the gnuplot development team released gnuplot 5.0—the first *major* new release in over 10 years! It goes without saying that gnuplot 5 is the best, most modern, and most complete gnuplot there ever was.

Most of the new features and changes to previous versions are relatively small and incremental, but in aggregate their effect is pronounced. A few major themes stand out that make gnuplot 5 distinct from the previous major release:

- *Color*—The greatest change that I perceive is the universal use of color throughout. Gnuplot assumes that you're working on a color terminal and are producing graphs for a color device. Producing black-and-white graphs is now the exception case.

If you're familiar with previous gnuplot versions, the most visible change is probably gnuplot's new *default color sequence*, which replaces the familiar red/green/blue sequence of colors. (Hint: you can restore the previous behavior through the command `set colorsequence classic`. More on that topic in chapter 9.)

Support for color has been enhanced in various ways. Most styles and facilities support partial transparency (alpha shading). You can generally choose the color of a plot element based on the numerical value of the data (data-dependent coloring).

- *Terminals*—Handling of terminals and output file formats has been greatly improved and unified. A major breakthrough is the development of a *family* of terminals that are all based on a single set of contemporary libraries for graphics and text rendering. Because these terminals all share the same back end, you can now create bitmaps (PNG), vector images (PDF), and graphs for interactive viewing (wxWidgets) with consistent appearance across all formats. Gnuplot takes advantage of the contemporary features of the underlying libraries to produce high-quality graphs through the use of over-sampling, anti-aliasing, and so on.

- *Fonts*—Another huge improvement compared to previous gnuplot versions is the *unified font handling*. Current terminals all rely on the same system libraries for font handling, which means generally all locally available fonts can be used for text labels in gnuplot graphs with a minimum of fuss. By default, *enhanced text mode*, which allows the inclusion of sub- and superscripts, is now enabled in all current terminals. Enhanced text mode has also been extended to include support for characters in boldface and italics.

    In the past, gnuplot's support for PDF was relatively weak, particularly in comparison to the very full-featured PostScript terminal. This situation has now been reversed, and the Cairo-based PDF terminal is probably to be preferred over the classic PostScript terminal.

- *Styles for points and lines*—You have a greater deal of control over the appearance of points and lines than ever before. In addition to line color, line width, point size, and point shape, you can now control and customize the *dash pattern* of lines. The syntax has been unified, and the appearance of points and lines can be customized locally wherever lines and points are used. Moreover, there is now a universal set of (at least) eight standard point and line styles, which are consistent across all contemporary terminals.

- *Unicode*—Strings and string expressions in gnuplot support Unicode. This is a big deal, because it makes the entire universe of Unicode characters and symbols seamlessly available within gnuplot. Greek letters, mathematical symbols, and dingbats are now as easy to use as ASCII characters. (You still need a font that provides glyphs for all these characters, but several freely available fonts exist that provide extensive coverage of Unicode characters.)

- *Loops and programming constructs*—Gnuplot has acquired support for some programming constructs, most notably `for` loops. This promises to simplify certain repetitive tasks significantly; but because gnuplot doesn't (yet) include an iterable data type (that is, an array type), the use of loops often leads to somewhat haphazard constructs and ad hoc data manipulations.[1]

In addition, there of course are many smaller improvements and enhancements, such as an improved multiplot mode, more consistent commands to control tic marks on coordinate axes, additional plotting styles, and more.

1.4 Summary

In this chapter, I showed you a couple of examples that demonstrate the power of graphical methods for understanding data. I also suggested a suitable method for dealing with data-analysis problems. Start with a plot of the data, and use it to identify the essential features of the data set. Then iterate the process to bring out the behavior you're most interested in. And finally (not always, but often), develop a mathematical description for the data, which can then be used to make predictions (which, by their nature, go beyond the information contained in the actual data set).

Your tool in doing this kind of analysis will be gnuplot. And it's gnuplot, and how to use it, that we'll turn to next. Once you've developed the skills to use gnuplot well, we'll return to graphical analysis and discuss useful techniques for extracting the most information possible from a data set, using graphs.

---

[1] This may be changing. Merely days before this book went to the printer, Ethan Merritt published an experimental patch that adds an array-type to gnuplot. Check it out: http://sourceforge.net/p/gnuplot/patches/724/.

Tutorial: essential gnuplot

This chapter covers

- Invoking gnuplot
- Plotting functions and data
- Saving and exporting
- Managing options
- Getting help

This chapter introduces gnuplot's most important features: generating plots, saving them to a file, and exporting graphs to common graphics file formats. I'll also explain briefly how to set options and how to access gnuplot's built-in documentation. This chapter covers those commands that you'll find yourself using almost every time you start up gnuplot. In the next couple of chapters, you'll learn about further features for graphical analysis and how to manage data sets and files. By the end of chapter 4, you'll know most of the commands you'll use on a day-to-day basis.

Are you surprised that just a few chapters are sufficient to get you this far? Congratulations! You just discovered why gnuplot is cool: it makes easy things easy, and hard things possible. This chapter and the next two cover the easy parts. As to the hard parts … well, that's what the rest of this book is about.

16

2.1 Simple plots

Because gnuplot is a plotting program, it should come as no surprise that the most important gnuplot command is `plot`. It can be used to plot both functions (such as $\sin(x)$) and data (typically from a file). The `plot` command has a variety of options and subcommands, through which you can control the appearance of the graph as well as the interpretation of the data in the file. The `plot` command can even perform arbitrary transformations on the data as you plot it.

2.1.1 Invoking gnuplot and first plots

Gnuplot is a *text-based* plotting program: you interact with it through command-line-like syntax, as opposed to manipulating graphs using the mouse in a WYSIWYG fashion.[2] Gnuplot is also *interactive*: it provides a prompt at which you type commands. When you enter a complete command, the resulting graph immediately pops up in a separate window. This is in contrast to a graphics *programming language* (such as PIC), where writing the command, generating the graph, and viewing the result are separate activities, requiring separate tools. Gnuplot has a history feature, making it easy to recall, modify, and reissue previous commands. The entire setup encourages you to play with the data: making a simple plot, changing some parameters to hone in on the interesting sections, eventually adding decorations and labels for final presentation, and in the end exporting the finished graph in a standard graphics format.

If gnuplot is installed on your system, it can usually be invoked by issuing the command

```
shell> gnuplot
```

at the shell prompt. (Check appendix A for instructions on obtaining and installing gnuplot, if your system doesn't have it already.) Once launched, gnuplot displays a welcome message and then replaces the shell prompt with a `gnuplot>` prompt. Anything entered at this prompt is interpreted as gnuplot commands until you issue an `exit` or `quit` command, or type an end-of-file (EOF) character, usually by pressing Ctrl-D on Unix.

Probably the simplest plotting command you can issue is

```
plot sin(x)
```

(Here and in the following, the `gnuplot>` prompt is suppressed to save space. Any code shown should be understood as having been entered at the `gnuplot>` prompt, unless otherwise stated.)

On Unix running a GUI (an arbitrary window manager running on top of X11), this command opens a new window showing the resulting graph, something like figure 2.1. Note how gnuplot has selected a "reasonable" range for the x values automatically (by default, from -10 to +10) and adjusted the y range according to the values of the function.

---

[2] The Windows version of gnuplot contains a menu you can use to build up command strings using the mouse.
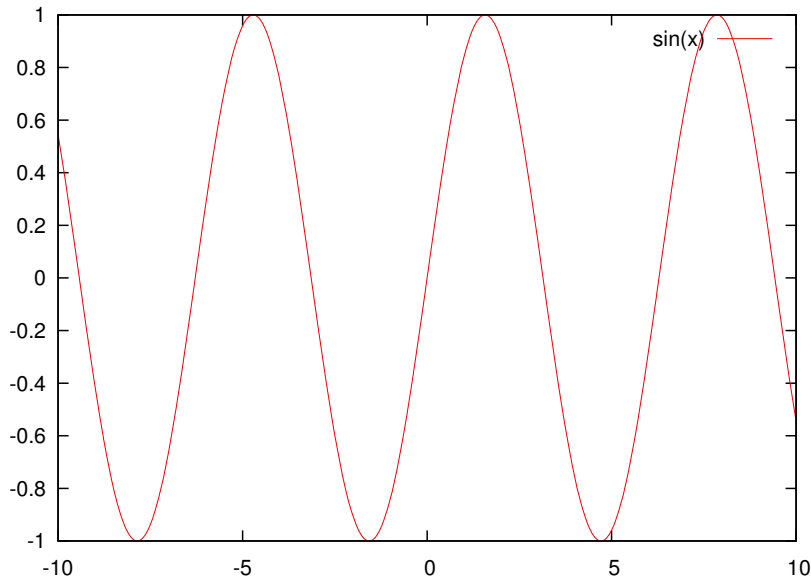
Figure 2.1   Your first plot: `plot sin(x)`

Let's say you want to add more functions to plot together with the sine. You recall the last command (using the up-arrow key or Ctrl-P for "previous") and edit it to give

```
plot sin(x), x, x-(x**3)/6
```

This will plot the sine together with the linear function x and the third-order polynomial $x - \frac{1}{6}x^3$, which are the first few terms in the Taylor expansion of the sine.[3] (Gnuplot's syntax for mathematical expressions is straightforward and similar to that found in almost any other programming language. Note the `**` exponentiation operator, familiar from Fortran or Perl. Section 3.2 contains tables of all available operators and their precedences.) The resulting plot (see figure 2.2) is probably *not* what you expected.

The range of y values is far too large, compared to the previous graph. You can't even see the wiggles of the original function (the sine wave) anymore. Gnuplot adjusts the y range to fit in all function values, but for this plot, you're only interested in points with small y values. So, you recall the last command again (using the up-arrow key) and define the plot range you're interested in:

```
plot [][-2:2] sin(x), x, x-(x**3)/6
```

---

[3]   A Taylor expansion is a local approximation of an arbitrary, possibly complicated, function in terms of powers of *x*. You won't need this concept in the rest of this book. Check your favorite calculus book if you want to know more.
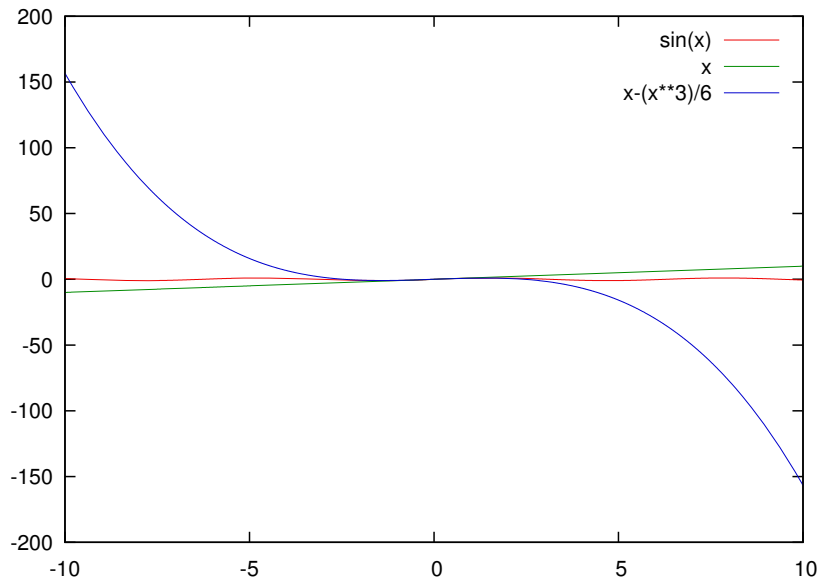
Figure 2.2   An unsuitable default plot range: `plot sin(x), x, x-(x**3)/6`

The range is given in square brackets *immediately after* the `plot` command. The first pair of brackets defines the range of x values (leave it empty, because you're happy with the defaults in this case); the second restricts the range of y values shown. This results in the graph shown in figure 2.3.
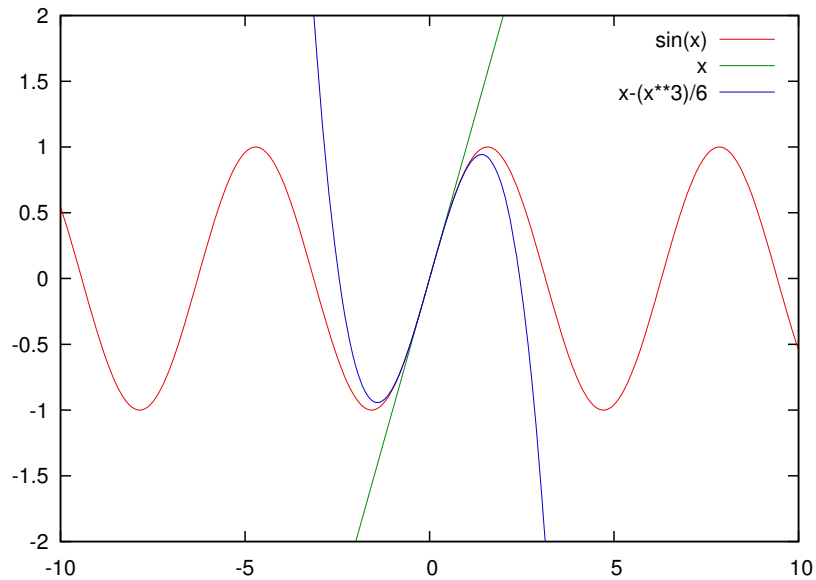
Figure 2.3   Using explicit plot ranges: `plot [][-2:2] sin(x), x, x-(x**3)/6`

You can play much longer with function plots, zoning in on different regions of interest and trying different functions (section F.4 contains a table of all built-in mathematical functions). But let's move on and discuss what gnuplot is most useful for: plotting data from a file.

2.1.2   *Plotting data from a file*

Gnuplot reads data from text files. The data is expected to be *numerical* and to be stored in the file in *whitespace-separated columns*. Lines beginning with a hash mark (#) are considered to be comment lines and are ignored. The following listing shows a typical data file containing the share prices of two fictitious companies, with the equally fictitious ticker symbols PQR and XYZ, over a number of years.

Listing 2.1   A typical data file: stock prices over time (file: prices)

```
# Average PQR and XYZ stock price (in dollars per share) per calendar year
1975      49     162
1976      52     144
1977      67     140
1978      53     122
1979      67     125
1980      46     117
1981      60     116
1982      50     113
1983      66      96
1984      70     101
1985      91      93
1986     133      92
1987     127      95
1988     136      79
1989     154      78
1990     127      85
1991     147      71
1992     146      54
1993     133      51
1994     144      49
1995     158      43
```

The canonical way to think about this is that the x value is in column 1 and the y value is in column 2. If there are additional y values corresponding to each x value, they're listed in subsequent columns. (You'll see later that there's nothing special about the first column. In fact, any column can be plotted along either the x or the y axis.)

    This format, simple as it is, has proven to be extremely useful—so much so that long-time gnuplot users usually generate data this way to begin with. In particular, the ability to keep related data sets in the same file is a big help (so you don't need to keep PQR's stock price in a separate file from XYZ's—although you could if you wanted to).

    Although whitespace-separated numerical data is what gnuplot expects natively, gnuplot can parse and interpret significant deviations from this norm, including text

columns (with embedded whitespace if enclosed in double quotes), missing data, and a variety of textual representations for calendar dates, as well as binary data. (See chapter 4 for a more detailed discussion of input file formats, chapter 5 for strings, and chapter 8 for the special case when one of the columns represents date/time information.)

Plotting data from a file is simple. Assuming that the file shown in listing 2.1 is called prices and is in the current working directory (typically, the directory from which gnuplot was started), you can type

```
plot "prices"
```

Because data files typically contain many different data sets, you'll usually want to *select the columns* to be used as x and y values. This is done through the `using` directive to the `plot` command:

```
plot "prices" using 1:2
```

This plots the price of PQR shares as a function of time: the first argument to the `using` directive specifies the column in the input file to be plotted along the horizontal (x) axis, and the second argument specifies the column for the vertical (y) axis. If you want to plot the price of XYZ shares in the same plot, you can do so easily (as shown in figure 2.4):

```
plot "prices" using 1:2, "prices" using 1:3
```
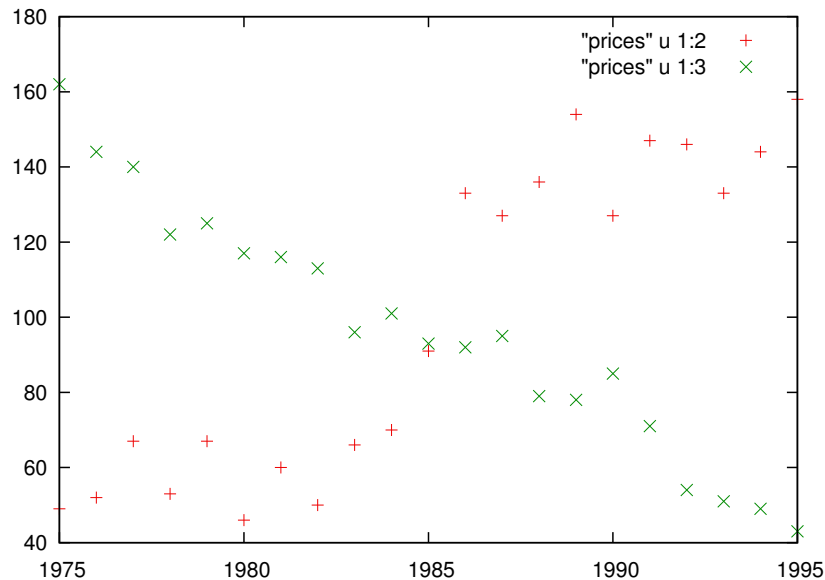

Figure 2.4 Plotting from a file: `plot "prices" using 1:2, "prices" using 1:3`

> **TIP**   The `using` directive tells the `plot` command which columns to use. `plot "data" using 1:2` selects the first column for the x axis and the second column for the y axis.

By default, data points from a file are plotted using unconnected symbols. Often this isn't what you want, so you need to tell gnuplot what *style* to use for the data. You do so using the `with` directive. Many different styles are available. Among the most useful are `with linespoints`, which plots each data point as a symbol and also connects subsequent points, and `with lines`, which just plots the connecting lines, omitting the individual symbols:

```
plot "prices" using 1:2 with lines,
➥     "prices" using 1:3 with linespoints
```

> **TIP**   The `with` directive to the `plot` command selects the plotting style. The most frequently used styles include `with points`, `with lines`, and `with linespoints`.

This looks good, but it's not clear from the graph which line is which. Gnuplot automatically provides a *key*, which shows a sample of the line or symbol type used for each data set together with a text description, but the default description isn't very meaningful in this case. You can do much better by including a `title` for each data set as part of the `plot` command:

```
plot "prices" using 1:2 title "PQR" with lines,
➥     "prices" using 1:3 title "XYZ" with linespoints
```

This changes the text in the key to the string given as the title (see figure 2.5). The `title` has to come after the `using` directive in the `plot` command. A good way to memorize this order is to remember that you must specify the data set to plot *first* and provide the description *second*: define it first, then describe what you defined.

> **TIP**   You can use the `title` directive to place a descriptive string in the graph's legend.

Want to see how PQR's price correlates with XYZ's? No problem; plot one against the other, using PQR's share price for x values and XYZ's for y values, like so:

```
plot "prices" using 2:3 with points
```

You see here that there's nothing special about the first column. Any column can be plotted against either the x or the y axis; you pick whichever combination you need through the `using` directive. Because it makes no sense to connect the data points in the last plot, I chose the style `with points`, which plots a symbol for each data point but no connecting lines (see figure 2.6).

A graph like figure 2.6 is known as a *scatter plot* and can show correlations between two data sets. In this graph, you can see a clear negative correlation: as the stock price of PQR is going up, the price of XYZ is going down. We'll revisit scatter plots and their uses in chapter 13.
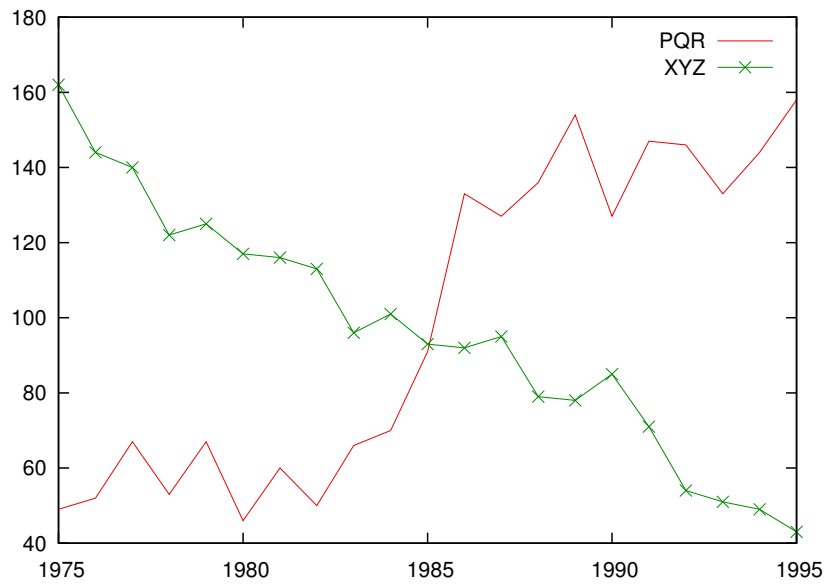
Figure 2.5   Introducing styles and the `title` keyword: `plot "prices" using 1:2 title "PQR" with lines, "prices" using 1:3 title "XYZ" with linespoints`
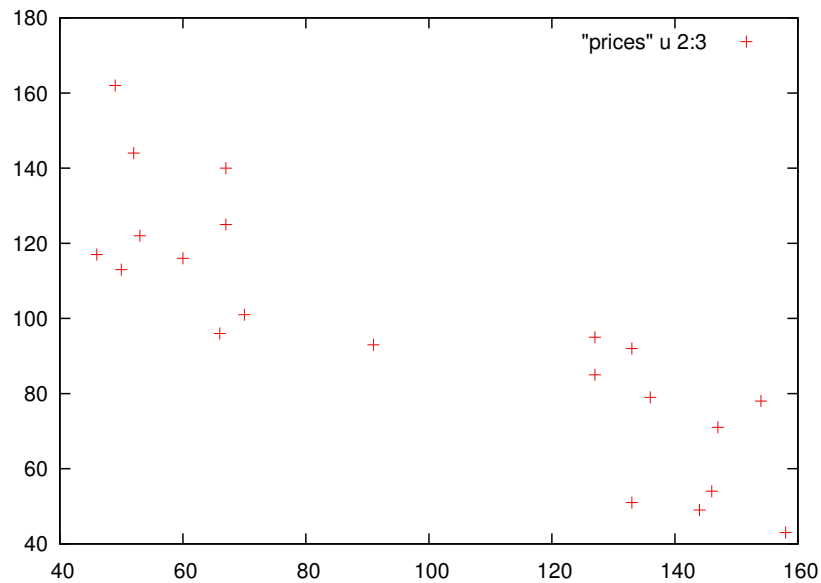
Figure 2.6   Any column can be used for either the x or y axis: `plot "prices" using 2:3 with points.`

Now that you've seen the most important basic commands, let's step back for a moment and quickly introduce some creature comforts that gnuplot provides to the more experienced user.

2.1.3 *Abbreviations and defaults*

Gnuplot is good at encouraging iterative, exploratory data analysis. Whenever you complete a command, the resulting graph is shown immediately, and all changes take effect at once. Writing commands isn't a different activity from generating graphs, and there's no need for a separate viewer program. (Graphs are also created almost instantaneously; only for data sets including millions of points is there any noticeable delay.) Previous commands can be recalled, modified, and reissued, making it easy to keep playing with the data.

Gnuplot offers two more features to the more proficient user: *abbreviations* and *sensible defaults*. Any command and subcommand or option can be abbreviated to the shortest, non-ambiguous form. So the command

```
plot "prices" using 1:2 with lines,
     "prices" using 1:3 with linespoints
```

is more likely to be issued as

```
plot "prices" u 1:2 w l, "prices" u 1:3 w linesp
```

You can shorten the `linespoints` style description even further to `lp`, so the command becomes

```
plot "prices" u 1:2 w l, "prices" u 1:3 w lp
```

This compact style is useful when you're doing interactive work, and you should master it. From here on, I'll increasingly use it. (You can find a table with the most frequently used abbreviations at the beginning of this book in the "About this book" section.)

But this is still not the most compact form possible. Whenever part of a command isn't given explicitly, gnuplot first tries to interpolate the missing values with values the user has provided; failing that, it falls back to sensible defaults. You've already seen how gnuplot defaults the range of x values to `[-10:10]` but adjusts the y range to include all data points.

> **TIP** Abbreviations for common keywords facilitate quick, interactive, iterative work. You can find a table of common abbreviations in the "About this book" section at the start of the book.

Whenever a filename is missing, the most recent filename is interpolated. You can use this to abbreviate the previous command even further:

```
plot "prices" u 1:2 w l, "" u 1:3 w lp
```

Note that the second set of quotation marks *must* be there.

In general, any user input (or part of user input) remains unaffected until explicitly overridden by subsequent input. The way the filename is interpolated in the preceding example is a good example of this behavior. In later chapters, you'll see how options can be built up step by step, by subsequently providing values for different suboptions. Gnuplot helps to keep commands short by remembering previous commands as much as possible.

One last example concerns the `using` directive. If it's missing entirely and the data file contains multiple columns, gnuplot plots the second column versus the first (this is equivalent to `using 1:2`). If a `using` directive is given but lists only a single column, gnuplot uses this column for y values and provides x values as integers starting at zero. This is also what happens when no `using` is given and the data file contains only a single column.

2.2   Saving commands and exporting graphics

There are two ways to save your work in gnuplot: you can *save* the gnuplot commands used to generate a plot, so that you can regenerate the plot at a later time. Or you can *export* the graph to a file in a standard graphics file format, so that you can print it or include it in web pages, documents, or presentations.

> **TIP**   *Saving* a graph is the act of storing the gnuplot commands used to create the graph. *Exporting* a graph means creating a version of the graph in a commonly used graphics file format (such as PNG, PDF, or SVG).

2.2.1   Saving and loading commands

If you save to a file the commands you used to generate a plot, you can later load them again and regenerate the plot where you left off. Gnuplot commands can be saved to a file using the `save` command:

```
save "graph.gp"
```

This saves the current values of all options, as well as the most recent `plot` command, to the specified file. This file can later be loaded again using the `load` command:

```
load "graph.gp"
```

The effect of loading a file is the same as issuing all the contained commands (including the `plot` command) at the gnuplot prompt.

> **TIP**   Never forget to save your commands using `save` so that you can re-create your graph later using `load`.

An alternative to `load` is the `call` command, which is similar to `load` but takes up to nine additional parameters after the filename to load. The parameters are available in the loaded file in the variables `ARG1` through `ARG9`. The special variable `ARG0` is set to

the filename, and the variable ARGC holds the number of parameters supplied to call.[4] You can use call to write simple scripts for gnuplot.

Command files are plain text files, usually containing exactly one command per line. Several commands can be combined on a single line by separating them with a semicolon (;)—this also works at the interactive command line. The hash mark (#) is interpreted as a comment character: the rest of the line following a hash mark is ignored. The hash mark isn't interpreted as a comment character when it appears inside quoted strings.

The recommended file extension for gnuplot command files is .gp, but you may also find people using .plt instead. Because command files are plain text files, they can be edited using a regular text editor. It's sometimes useful to author them manually and load them into gnuplot (see section 12.2 for further discussion of that point), and they're also used for batch operations (chapter 11) and configurations (chapter 12).

2.2.2   *Exporting graphs*

The save command I just introduced saves the commands required to generate (or regenerate) the graph into a text file, but it doesn't save the graph itself. How do you get the graph that you see in the plot window into a file? It turns out there are three ways to do it: easy, roundabout, and complicated.

USING THE GUI BUTTON

In recent gnuplot versions, most contemporary terminals (including the wxt and qt terminals) include a GUI button you can use to export a graph to a file. The button opens a standard file-selection dialog where you select the desired name of the target file and the file format (PNG, PDF, or SVG). Easy.

USING THE CLIPBOARD

A roundabout way to save the graph to a file is to take a screenshot of the graph, using your favorite screenshot utility, and save the result as a GIF or PNG file. After all, the graphics window on the screen is nothing more than a bitmap image, and a screenshot saves it to a file.

Gnuplot facilitates this process by offering a GUI button that copies the current content of the plot window to the window manager's clipboard, from which the graph can be pasted into a paint program (such as the Gimp or Microsoft Paint) or saved directly to a file. Only the canvas is copied, without the tool and title bars, and without any of the decorations put on the window by the window manager. Don't be disturbed if nothing appears to be happening if you click the GUI button: there is no visual feedback, but the window's content is copied to the clipboard, from which it can be pasted into other applications. (The details of how to do that depend on your choice of window manager and the way the clipboard is configured on your local computer.) This method may appear a bit unsophisticated, but it has two significant advantages: the

---

[4] Earlier versions of gnuplot used the special tokens $0 through $9. That syntax is now deprecated.

workflow is simple, and the resulting graph corresponds *exactly* to the graph that was on the screen.

USING TERMINALS

Both the GUI button and the clipboard are relatively new methods for exporting a graph. They're easy to use but don't allow for much flexibility in the appearance of the generated graph. To get full control over the export process, you need to become familiar with gnuplot's *terminal* facility.

In gnuplot parlance, a terminal is a graphics-capable output device. Traditionally, this may have been a specific piece of hardware (such as a pen-and-ink plotter), but today a gnuplot terminal is merely a reference to the underlying graphics library.

> **TIP**   A terminal is gnuplot's abstraction for a graphics-capable end point. Terminals can be interactive (screen oriented) or file-based. All graphics output must be directed to a terminal.

Contemporary terminals come in two flavors: interactive and file-based. Interactive terminals create the graphs that you see on the screen. On Linux, you have a choice among three interactive terminals, each built using a different widget set: the `qt` terminal uses Qt, the `wxt` terminal uses wxWidgets, and the old `x11` terminal is a pure Xlib application. (Both the `qt` and the `wxt` terminals also exist on Windows and Mac OS X, together with platform-specific terminals.)

But to export a graph to a file, you need to employ a file-based terminal that can generate output in the desired output format (PNG, PDF, **SVG**, and so on). Most file-based terminals accept a large number of options through which you can control various aspects (such as the size) of the resulting graph. These options are covered in detail in chapter 10.

All of this is straightforward. But there is one stumbling block: exporting a graph via a file-based terminal *requires multiple steps.* In order to export a graph with a gnuplot terminal, *several commands* must be executed in proper sequence. This may come as a surprise, because "exporting to a file" is a single, atomic operation in most other applications these days. Gnuplot is different.

The full sequence of steps is shown in listing 2.2. Let's step through it:

1. Begin with an arbitrary `plot` command.
2. Select the desired file-based terminal, using the `set terminal` command.
3. Specify the name of the output file, using the `set output` command.
4. Regenerate the last plot, this time sending it to the file-based terminal and the named file.
5. Restore the interactive terminal again, using both `set terminal` and the `set output` command.

Listing 2.2   Complete workflow to export to file

```
plot exp(-x**2)                          A plot command

set terminal pngcairo                    Selects the file format
set output "graph.png"                   Specifies the output filename
replot                                   Repeats the most recent plot command, with
set terminal wxt                         the output now going to the specified file
set output
                                         Sends output to the screen again by
                                         using an empty filename
```

Restores the terminal settings
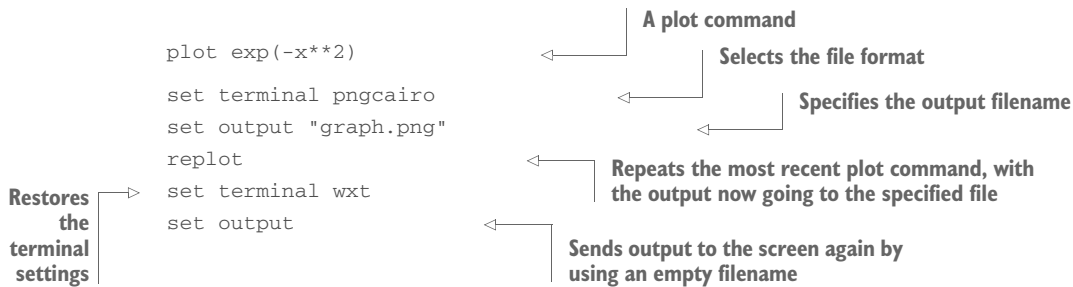
This is the first time you've encountered gnuplot's `set` command—I'll talk about it in more detail in the next section (and subsequently in almost every chapter of this book!). For now, it's enough to understand that it sets a parameter (such as `terminal`) to a value. But—and this is often forgotten—it does *not* generate a plot! The only commands that do so are `plot`, `splot` (which is used for three-dimensional graphs, discussed in appendix C), and `replot` and `refresh` (both of which repeat the most recent `plot` or `splot` command).

I'd like to emphasize three things about listing 2.2:

- The file format and the output filename must be specified separately, using `set terminal` and `set output`, respectively.
- The graph must be sent to the newly created file in a separate command, using `plot`, `replot`, or an equivalent command.
- Afterward, the interactive terminal must be restored explicitly. (This is easy to forget.)

As long as you keep these three items in mind, creating files via terminals won't pose any difficulties. (In chapter 11, you'll see how to use scripts and macros to facilitate this process.)

> **TIP**   Be sure to complete *all* the required steps, and in the proper order. Typical mistakes are forgetting to issue a `replot` command (resulting in an empty file) and forgetting to restore the original terminal at the end.

Before leaving this section, one last word of advice: always save the commands used to generate a plot to a command file *before* exporting to a printable format. Always. It's almost guaranteed that you'll want to regenerate the plot to make a minor modification later (such as fixing a typo in a label, adding one more data set, or adjusting the plot range slightly). This can only be done from the commands saved to file using `save`, not from plots exported to a graphics file. We'll come back to this topic several times.

> **TIP**   Don't just store the finished graph. Always save the commands required to generate it as well, so you can come back and modify it later.

2.3   *Managing options with set and show*

Gnuplot has relatively few commands (such as the `plot`, `save`, and `load` commands you've already encountered) but a large number of *options*. These options are used to control everything from the format of the decimal point to the name of the output file. More than 100 such options are available, along with countless sub-options for each. Check the entries for `set` and `show` in the gnuplot standard reference manual for the complete list.

> **TIP**   Many aspects of gnuplot's behavior can be controlled through options. Use the `set` command to change their value; use `show` to see their current value.

The three commands used to manipulate individual options are as follows:

- `show`—Displays the current value of an option
- `set`—Changes the value of an option
- `unset`—Disables a specific option or returns it to its default value

There's also a fourth command, `reset`, which returns *all* options to their default values. The only options not affected by `reset` are those directly influencing output generation: `terminal` and `output`.

The syntax of all three commands is straightforward. To assign a new value to an option, use the `set` keyword followed by the name of the option and the new value. The `show` command takes only a single argument: the name of the option you want to inspect.

The `show` command is also used more generally to display all kinds of information about gnuplot's internal state, not just options that can be changed using `set`. For example, `show variables` displays all variables that are defined in the current session together with their values, and `show functions` lists all user-defined functions.

Another useful command is

```
show version long
```

This prints the current version of gnuplot, together with a copyright notice and pointers to the online documentation. More important, it also shows the compile-time flags that gnuplot was compiled with. This is particularly relevant because some features have only recently been added to gnuplot and may not be enabled on all platforms. You can use `show version long` to see which flags your version of gnuplot was built with.

Finally, you can use `show all` to see a listing of all possible options and their values. Be prepared for a long listing!

> **TIP**   You can use the `show` command to learn about your gnuplot installation and to inspect the current state of a gnuplot session.

2.4   *Getting help*

Gnuplot has an extensive built-in, online help system (online in the sense that it's accessible from within the gnuplot session; it has nothing to do with network connectivity to the internet). To get started, enter `help` at the gnuplot prompt. Alternatively, you can go directly to the reference page for a specific command or option by entering the name of the command or option as an argument to the `help` command. For example, to learn about the `plot` command, you'd use

```
help plot
```

The online help is detailed and comprehensive, so you should become familiar with it. But keep in mind that it's a *reference*, not a tutorial. If you know the name of the command or option you're looking for, it's great. But if you want to find all relevant options for a specific task, navigating the online help can be frustrating.

> **TIP** Type `help`, followed by a command or option name, to access gnuplot's built-in reference documentation.

2.5   *Summary*

In this chapter, you learned how to do the most important things with gnuplot: plotting, saving, and exporting. You also learned how to set and inspect options and how to access the built-in reference documentation.

This means you can already do the three most important things for day-to-day work: generate a plot, save it to file, and export it. In the next chapter, you'll begin to learn how to use gnuplot to analyze data and understand what it's telling you.

The heart of the matter:
the plot command

This chapter covers

- Plotting functions and data
- Math with gnuplot
- Inline transformations
- Logarithmic plots
- Smoothing data

In gnuplot, you use the `plot` command to create plots of functions and data. It is, hands down, the most frequently used and most important of all gnuplot commands. In this chapter, we'll take a systematic look at the `plot` command and the ways it can be used.

You'll begin by making simple plots of functions and data. Then I'll introduce *inline transformations*: gnuplot can apply arbitrary transformations to the data as it's being plotted. You'll learn how to use this useful feature after we take a quick detour to describe how to do math with gnuplot. Logarithmic plots are another indispensable technique in graphical analysis, and I'll show you how and when to use them. (If you've never seen them before, you'll find a sidebar giving you more

background on them.) Finally, we'll be ready to discuss gnuplot's built-in features to simplify, smooth, and summarize data.

3.1   *Plotting functions and data*

You use the `plot` command to plot both functions and data. Let's look at plotting functions first and then move on to plotting data.

3.1.1   *Plotting functions*

To plot a function, you use the `plot` command followed by the function to plot and (optionally) any style directives you wish to use (see chapters 6 and 9 to learn about available styles and how to specify them):

```
plot sin(x)                         ⟵——— Plots in the default style
plot cos(x) with linespoints            ⟵——— Plots with lines and points
```

Including two or more functions in the same graph is easy. You simply list them all after the initial `plot` command, separated by commas:

```
plot sin(x), cos(x) w lp
```

If you don't specify a style, gnuplot chooses a different line color or dash pattern for each new function in the plot. If you include more functions than there are currently defined styles, gnuplot cycles through them again from the beginning. (This is explained in more detail in chapters 6 and 9.)

Gnuplot comes with a large number of built-in functions, and it's also possible to define your own. I'll talk about this in section 3.2 later in this chapter.

THE SAMPLING RATE FOR FUNCTIONS

The `set samples` option controls the number of points at which a function is evaluated to generate a plot. It defaults to 100 points, but this isn't sufficient for curves including cusps or singularities. A value of 1,000–3,000 works well—the sampling rate should be slightly higher than the horizontal number of pixels of the output device (which is usually around 600 for interactive and bitmap terminals). This option has no effect when plotting data, unless you use one of the smoothing algorithms (see section 3.5 later in this chapter); the smooth curve is sampled according to the value of the `set samples` option.

Changing the number of sampling points is easy. Give an integer argument to `set samples`:

```
set samples 1000            ⟵——— Now uses 1,000 points
```

You may want to include that line in your personal gnuplot initialization file, overriding gnuplot's (ridiculously) small default value (see chapter 12 to learn how to set up a gnuplot initialization file).

TIP   Choose an appropriate value (such as 1,000 or even 3,000) for the num-
ber of points gnuplot uses to plot a function. The default value of 100 is
unreasonably low given the capabilities of today's computers and displays.

3.1.2   Plotting data

To plot data, you again use the `plot` command, but in place of the function to plot,
you specify the name of the file containing the data set. There is one additional wrin-
kle: you also need to specify which *columns* from the file to select. And finally, you can
again choose which style gnuplot should use for your plot.

I'll talk about data files and their permissible formats in more detail in chapter 4.
For now, suffice to say that gnuplot expects data files to be whitespace-separated, col-
umn-oriented text files, with one line per record. Listing 3.1 shows an example: the
world records (as of late 2014) for both men and women running events at a variety of
distances. (Distances are in meters, and times are in seconds—the last two entries with
the "odd" distances are for half-marathon and full-marathon races.)

Listing 3.1   Data for figure 3.1 (file: records)

```
# Distance     Men       Women    Men          Women
100            9.58      10.49    2009-08-16   1988-07-16
200           19.19      21.34    2009-08-20   1988-09-29
400           43.18      47.60    1999-08-26   1985-10-06
800          100.91     113.28    2012-08-09   1983-07-26
1000         131.96     148.98    1999-09-05   1996-08-23
1500         206.00     230.46    1998-07-14   1993-09-11
3000         440.67     486.11    1996-09-01   1993-09-13
5000         757.35     851.15    2005-05-31   2008-06-06
10000       1577.53    1771.78    2005-08-26   1993-09-08
21098       3503.00    3912.00    2010-03-21   2014-02-16    ◁——— Half marathon
42195       7377.00    8262.00    2014-09-28   2003-04-13    ◁——— Full marathon
```

Figure 3.1 displays a plot of the finishing times for events up to 10,000 meters as a
function of the distance, together with two function plots for comparison (the `using`,
`title`, and `w p` directives were introduced in section 2.1.2):

```
plot [:12000] "records" u 1:2 title "Men" w p, "" u 1:3 title "Women" w p,
➥             0.14*x, 0.16*x
```

Not surprisingly, the time required increases roughly in proportion to the distance,
but not precisely so. A comparison with the straight lines included in the graph shows
that for longer distances, the required time is greater than you'd expect given the
short-distance results. Again, that's no surprise: as the distance gets longer, the run-
ners slow down. (We'll come back to this point later.)

This example plots two (simple) functions together with the data, so that you can
compare the data against a known behavior (as expressed by the function). This kind
of comparison constitutes an important part of graphical analysis, and gnuplot makes
it particularly easy to combine multiple data sets *and* functions in a single graph.
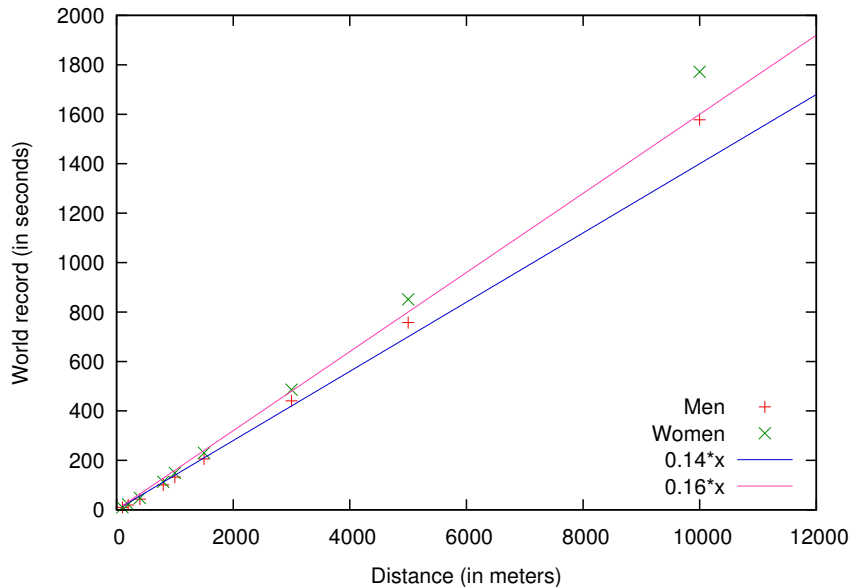
Figure 3.1   Current athletic world-record running times as a function of the competition distance, for both men and women. The data is shown in listing 3.1.

> **TIP**   Comparing data against a known function can tell you a lot about the data set. Including one or several suitable functions in the graph brings out the behavior of the data.

3.2   *Math with gnuplot*

Not surprisingly, gnuplot can evaluate mathematical expressions and includes support for many mathematical functions. Gnuplot's syntax for mathematical expressions is straightforward and similar to the conventions found in other programming languages.

Gnuplot ships with its own math library, which these days is used only when the system math library is found to be insufficient. (The most notable extension that gnuplot's math library provides is support for complex numbers: see appendix F.)

3.2.1   *Mathematical expressions*

Gnuplot uses standard infix syntax for mathematical expressions, including the normal operators for the four basic arithmetical operations, as in most C-like languages. Parentheses can be used to change the order of evaluation. Gnuplot has the exponentiation operator (**) found in Fortran and Perl. All of these operate either on floating-point values or on integers.

It's important to understand that *integer division truncates*, so 3/2 yields 1. Integers are promoted to floating-point values in mixed expressions, but in some situations

truncation occurs in expressions involving variables. If you want nontruncating division, it's always safer to force the use of floating-point arithmetic by writing $x/2.$ or $x/2.0$. You can also force a mixed expression by adding a floating-point zero: $x + 0.0$.

> **TIP** Keep in mind that integer division truncates. Hence an expression like 3/2 evaluates to 1, not to 1.5. To force floating-point division, include a decimal point: 3/2. evaluates to 1.5.

Gnuplot also offers a range of operators for integer arithmetic, such as the familiar modulo operator %, and a unary factorial operator: n! = 1 * 2 * ... * (n-1) * n. You can also find bit-shift and bitwise logical operators.

All the usual relational and logical operators are available as well, including the ternary conditional operator (?:). A recent addition is the *comma operator*, which can be used to evaluate several statements as part of a single expression. (This is a relatively advanced feature; you'll see an application in section 14.1.2.) Tables 3.1 and 3.2 list all the unary and binary operators in order of their precedence.

Table 3.1 Unary operators in order of decreasing precedence

Operator	Example	Argument type	Description
!	!a	Integer	Logical NOT
~	~a	Integer	One's complement
!	a!	Integer	Factorial
–	–a	Any numerical	Unary minus
+	+a	Any numerical	Unary plus

Table 3.2 Binary operators in order of decreasing precedence

Operator	Example	Argument type	Description
**	a**b	Any numerical	Exponentiation
*	a*b	Any numerical	Multiplication
/	a/b	Any numerical	Division
%	a%b	Integer	Modulo
+	a+b	Any numerical	Addition
–	a–b	Any numerical	Subtraction
.	a.b	String	String concatenation
<<	a<<b	Integer	Unsigned bitwise left shift
>>	a>>b	Integer	Unsigned bitwise right shift
<	a<b	Any numerical	Less than

Table 3.2   Binary operators in order of decreasing precedence *(continued)*

Operator	Example	Argument type	Description
`<=`	`a<=b`	Any numerical	Less than or equal to
`>`	`a>b`	Any numerical	Greater than
`>=`	`a>=b`	Any numerical	Greater than or equal to
`==`	`a==b`	Any numerical	Numerical equality
`!=`	`a!=b`	Any numerical	Numerical inequality
`eq`	`a eq b`	String	String equality
`ne`	`a ne b`	String	String inequality
`&`	`a&b`	Integer	Bitwise AND
`^`	`a^b`	Integer	Bitwise XOR
`|`	`a|b`	Integer	Bitwise OR
`&&`	`a&&b`	Integer	Logical AND
`||`	`a||b`	Integer	Logical OR
`=`	`a=b`	Any (numerical or string)	Assignment
`(,)`	`(a,b)`	Any (numerical or string)	Sequential evaluation

3.2.2   *Built-in functions*

Gnuplot provides all the mathematical functions you've come to expect on any scientific calculator: square root, exponential function and logarithms, and trigonometric and hyperbolic functions. It also supports some less-standard functions that are relevant in physics, applied math, and statistics (such as Bessel functions, the error integral, and the gamma function). You can execute `help expressions functions` from within gnuplot or check table F.1 in appendix F for the full list.[1]

Gnuplot also includes a built-in random number generator, which can be quite useful at times. You access it through the function `rand(x)`. If called with an argument that is equal to zero, the function returns a pseudo-random number between zero and one.

The algorithm used to generate the pseudo-random numbers requires two seeds to be set; given the same values for the seeds, exactly the same sequence of numbers is returned. Gnuplot doesn't have a separate function to set the seeds; instead, this is accomplished by providing special values as arguments to `rand(x)`. Check table 3.3 for details. Keep in mind that `rand(x)` is a function and therefore must be called as an expression, even if you're only setting the seeds. The command `print rand(-1)`, for example, can be used to reset the seeds to their default values.

---

[1]   A summary of built-in functions for string and time handling can be found in tables 5.1 and 8.6, respectively.

Table 3.3   The `rand(x)` function is used both to access the random number generator and to set its seeds, depending on the value of the argument.

Invocation	Description
`rand(0)`	Returns the next pseudo-random number, using the current values of both seeds
`rand(-1)`	Resets both seeds to a standard value
`rand(x)`	For `x>0`, sets both seeds to a value based on the value of `x`
`rand({x,y})`	For `x,y>0`, sets the first seed to `x` and the other seed to `y`

The standard gnuplot distribution contains a file called stat.inc in the demo/ folder that defines many additional functions, most of them related to various probability distributions. You can load this file into your current gnuplot session using `load` and then use these functions the same way you would use built-ins.

> **TIP**   Check the file stat.inc in gnuplot's demo/ folder for many useful functions related to probability distributions.

3.2.3   *User-defined variables and functions*

It's easy to define new variables by assigning an expression to a name. For instance, you might want to define some useful mathematical constants. (You don't need to define `pi`, because it's already defined by gnuplot.) The following listing shows several useful constants you may want to define.

Listing 3.2   Examples of user-defined variables and constants

```
e = 2.71828182845905          ◁——————  The base of the natural logarithm
sqrt2 = sqrt(2.0)             ◁——————  The square root of 2
euler = 0.57721566490153      ◁——————  The Euler-Mascheroni constant
```

Here, `euler` is the Euler-Mascheroni constant (usually represented using the Greek letter gamma), which is "the most important number that you've never heard of before" (J. Stopple).[2]

Functions can be defined in a similar fashion, as shown next.

Listing 3.3   Examples of user-defined functions

```
f(x) = -x * log(x)

gauss( x, m, s ) = exp( -0.5*( (x-m)/s )**2 )/sqrt(2*pi*s**2)
                                                                   Uses the factorial
binom( n, k ) = n!/(k!*(n-k)!)    ◁——————                          operator to generate
                                                                   binomial coefficients

min( a, b ) = ( a < b ) ? a : b   ◁——————  Uses the ternary ? operator
step( x ) = ( x < 0 ) ? 0 : 1     ◁——————  A piece-wise defined function
```

---

[2] Read his wonderful *A Primer of Analytic Number Theory* (Cambridge University Press, 2003) to find out why you should care. The quote can be found on page 49.

Functions can have up to 12 variables and can contain other functions and operators. You use them as you would use any other function. Assuming you had issued the definitions in listing 3.3, you could then write `plot sin(x), f(x)`.

By default, gnuplot assumes that the independent dummy variable, which is automatically replaced by a range of x values when plotting a function, is labeled x (as in: `plot sin(x)`), but you can change this using the `set dummy` command. For example, `set dummy t` makes t the independent variable, so now you would say `plot sin(t)`.

All other variables occurring in a function definition (parameters) must have been assigned values before you can plot (that is, evaluate) the function. For convenience, you can assign values to parameters as part of the `plot` command. For example, you could do the following (notice that no comma is required between the variable assignment and the function):

```
g(x) = exp(-0.5*(x/s)**2)/s
plot s=1 g(x), s=2 g(x), s=3 g(x)
```

But in general, I think it's better to make all parameters explicit *arguments* in the function definition. The previous example could (and probably should) be written as follows (except for the entries in the graph's legend or *key*, the resulting figures are identical—see figure 3.2):

```
g(x, s) = exp(-0.5*(x/s)**2)/s
plot g(x,1), g(x,2), g(x,3)
```
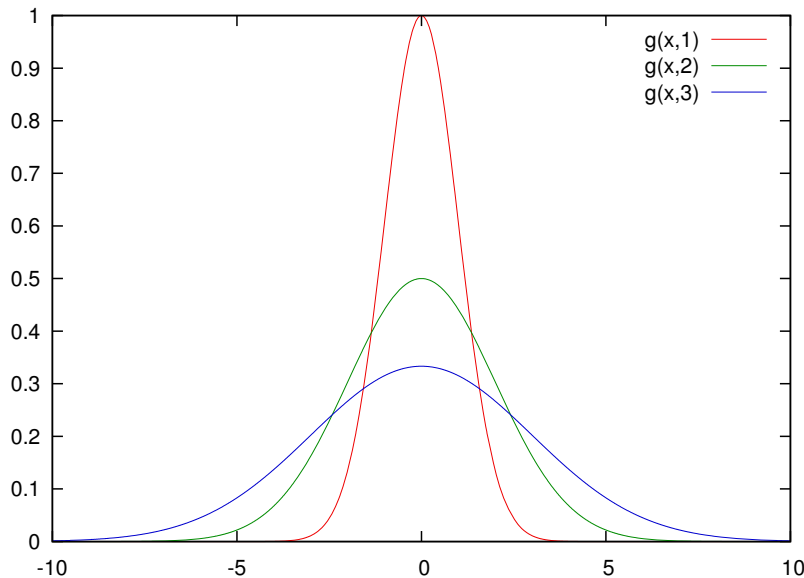

Figure 3.2 Plotting a function for different values of a parameter: `plot g(x,1), g(x,2), g(x,3)` where `g(x, s) = exp(-0.5*(x/s)**2)/s`

All functions and variables have global scope. There's no such thing as a private variable or local scope!

You can obtain lists of all user-defined variables and functions using the following two commands (remember to use `help expressions functions` to list built-in functions):

```
show variables
show functions
```

3.2.4 *Mathematically undefined values and NaN (not a number)*

Gnuplot tends to be pretty tolerant when encountering undefined values: rather than failing, it doesn't produce any graphical output for data points with undefined values. For instance, you may safely plot the square root (which is only defined for positive values) over any plot range: if the argument is negative, gnuplot won't fail—it just won't show anything on the graph for $x < 0$. Similarly, you never have to worry whether the denominator of a function might equal zero for any value in the plot range: if it does, gnuplot will silently skip those points.

This behavior can be used to suppress data points or define piecewise functions, usually in conjunction with the ternary conditional operator. For example, consider the following function:

```
f(x) = abs(x) < 1 ? 1 : NaN
```

It's only defined on the interval `[-1:1]`. For values with an absolute value equal to 1 or greater, it returns `NaN`, which stands for *not a number* and indicates a mathematically undefined value.[3] A plot of this function shows data points only on the interval `[-1:1]` and is empty otherwise. This `NaN` technique is a good trick that's frequently useful, in particular in conjunction with the ternary operator, as in these examples.

> **TIP** Gnuplot doesn't fail when it encounters a mathematically undefined value during a plot; it just doesn't generate graphical output for such points. You can use this effect to exclude intervals or data points from plots.

3.3 *Data transformations*

As stated before, gnuplot is first and foremost a *plotting* tool: a program to generate straightforward plots of raw data in a simple and efficient manner. Specifically, it's *not* a statistics package or a workbench for numerical analysis. Large-scale data transformations aren't what gnuplot is designed for. Properly understood, this is one of gnuplot's main strengths: it does a simple task and does it well, and it doesn't require learning an entire toolset or programming language.

Nevertheless, gnuplot has the ability to perform arbitrary data transformations on the fly as part of the `plot` command. Thus you can apply filters to the data from within gnuplot without having to take recourse in external tools or programming languages.

---

[3] Using the `NaN` special value from the IEEE 754 standard for floating-point computations is new with gnuplot 5; in previous releases, it was customary to achieve the same effect using the expression `1/0`.

3.3.1   *Simple data transformations*

An arbitrary function can be applied to each data point as part of the `using` directive in the `plot` command. If an argument to `using` is enclosed in parentheses, it's treated not as a column number, but as an expression to be evaluated. Inside the parentheses, you can access the column values for the current record by preceding the column number with a dollar sign (`$`) (as in shell or awk programming). Some examples will help to clarify:

- To plot the square root of the values found in the second column versus the values in the first column, use
  ```
  plot "data" using 1:( sqrt($2) ) with lines
  ```
- To plot the average of the second and third columns, use
  ```
  plot "data" using 1:( ($2+$3)/2 ) with lines
  ```
- To generate a log/log plot, you can use the following command (although the `logscale` style, discussed in section 3.4, is the preferred way to achieve the same effect):
  ```
  plot "data" using ( log($1) ):( log($2) ) with lines
  ```

> **TIP**   Inline transformations allow you to apply an arbitrary transformation to each point as it's being plotted.

A fundamental limitation to all these transforms is that they can only be applied to a *single record at a time.* If you need aggregate functions over several records (sums or averages, for example), or across different data sets, you'll usually have to perform them externally to gnuplot. Nevertheless, the ability to apply an arbitrary filter to each data point and to combine different data points for the same x value is often tremendously useful.

A WORKED EXAMPLE

Remember the running records from listing 3.1? Based on the basic plot of time versus distance, you were led to suspect that athletes run more slowly over longer distances. The best way to confirm this would be a plot of the average *speed* for the different events. The data set doesn't contain the speed, but you can calculate it easily as distance (column 1) divided by time (in column 2 or 3, respectively). This is precisely the kind of simple combination of column values that inline transformations are intended for (remember that an empty filename is a placeholder for the most recently accessed file):

```
plot [9:][5:11] "records" u 2:($1/$2) t "Men" w lp,
➥              "" u 3:($1/$3) t "Women" w lp
```

Figure 3.3 shows the speed as a function of the time (not distance), in a semi-logarithmic plot (in the *horizontal* axis: `set logsc x`). The figure confirms the expectation that average speed decreases for longer distances, but it shows you more: the rate of decrease isn't constant. We can distinguish three regimes: a sprint regime (up to 20
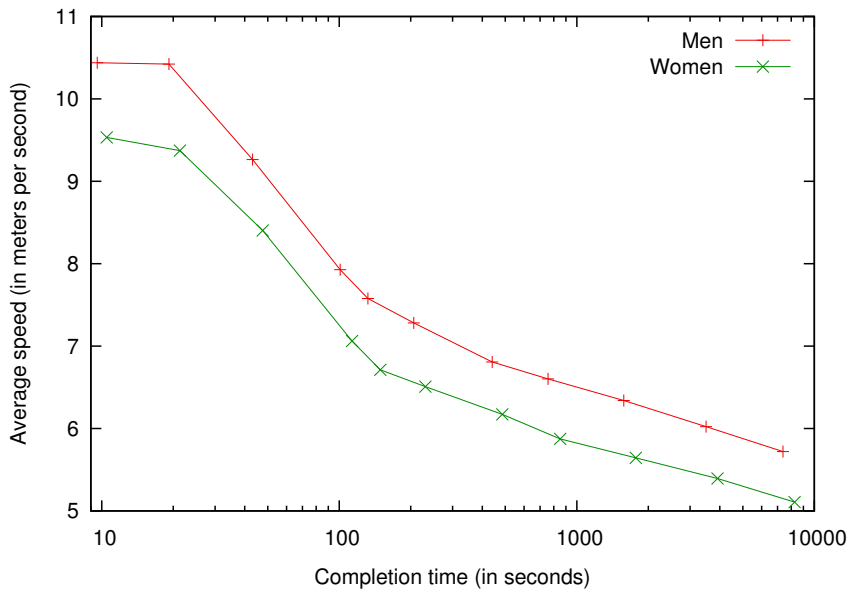
Figure 3.3 **The average speed of the current world record holder, as a function of the distance. The speed has been calculated from the distance and the finishing time as part of the `plot` command, using an inline transformation: `plot "records" u 2:($1/$2) w lp`. (See listing 3.1 for the data.)**

seconds); a long-duration regime (longer than about 2 minutes); and a transition regime in between, where the decrease in speed is particularly drastic.

MORE IDEAS FOR INLINE TRANSFORMATIONS

Here are some more creative uses for inline transformations. To plot two data sets of different magnitudes on a similar scale, use this (assuming that the data in column 3 is typically greater by a factor of 100 than the data in column 2):

```
plot "data" using 1:2 with lines, "" using 1:( $3/100.0 ) with lines
```

If the data file contains the x value in the first column, the mean in the second, and the variance in the third, you can plot the band in which to expect 68% of all data to fall:

```
plot "data" using 1:( $2+sqrt($3) ) with lines,
➡    "" using 1:( $2-sqrt($3) ) with lines
```

All expressions involving operators or functions can be part of using expressions, including the conditional operator:

```
plot "data" using 1:( $2 > 0 ? log($2) : 0 ) with lines
```

You can use a similar method to exclude certain data points when plotting data from a file. For example, the following command only plots data points for which the y value is less than 10:

```
plot "data" using 1:( $2 < 10 ? $2 : NaN ) with points
```

Finally, keep in mind that the expression supplied in parentheses can be a *constant*. You'll see some examples for this usage in section 3.5 later in this chapter.

3.4   *Logarithmic plots*

Let's see how you can generate logarithmic plots. Logarithmic plots are a crucial technique in graphical analysis. In gnuplot, it's easy to switch to and from logarithmic plots:

```
set logscale              ◁——————  Turns on double logarithmic plotting
set logscale x            ◁——————  Turns on logarithmic plotting for the x axis only …
set logscale y            ◁——————  … for the y axis only

unset logscale            ◁——————  Turns off logarithmic plotting for all axes
unset logscale x          ◁——————  … for the x axis only
unset logscale y          ◁——————  … for the y axis only
```

You can provide a base as a second argument: `set logscale y 2` turns on binary logarithms for the y axis. (The default is to use base 10.) We'll talk more about uses for `set logscale` in chapter 13.

How do logarithmic plots work?

Logarithmic plots are a truly indispensable tool in graphical analysis. It's possible to understand what they do even without detailed understanding of the mathematics behind them, but the math isn't all that hard; therefore, I'll explain in this sidebar how logarithmic plots work and how they're used.

Let's quickly recapitulate the most relevant properties of logarithms. The defining relation for the log function is

$$n = \log_b b^n$$

There isn't a single log function, but infinitely many; you always need to specify the *base b* with respect to which the logarithm needs to be taken.

Logarithms with respect to different bases are related to one another in a simple way:

$$\log_a x = \log_a b \cdot \log_b x$$

Logarithms to two specific bases are used most often: the logarithm to base 10 (typically abbreviated $\log$) and the so-called *natural logarithm* to base e=2.7818... (abbreviated $\ln$). Be aware that the notation just introduced ($\log(x)$ and $\ln(x)$) is common in mathematical contexts, but when you're using gnuplot's built-in functions, `log(x)` refers to the natural logarithm; you must use `log10(x)` for the logarithm to base 10.

Logarithms have many interesting properties, two of which are of fundamental importance:

$$\log(xy) \;=\; \log(x) + \log(y)$$

and

$$\log(x^k) \;=\; k\,\log(x)$$

You can see that the logarithm of a product equals the sum of the logarithms, so these two relations can be summarized by saying that logarithms turn products into sums and powers into products. You'll see how these relationships are at the core of logarithmic plots.

In a logarithmic plot, you don't plot the actual values; instead, you plot their logarithms. But tic marks on the axes are usually labeled with the actual values (not their logarithms). Here's an example: suppose you want to plot the values 1, 10, 100, and 1,000. Their logarithms (to base 10) are 0, 1, 2, and 3, respectively (because $10^3 =$ 1,000, and so on). In a logarithmic plot, you place tic marks at equal distances from each other (at the locations 0, 1, 2, and 3) but *label* them 1, 10, 100, and 1,000.

The following figure illustrates these aspects. The y axis on the left is scaled logarithmically. Comparing the values on the left vertical axis to the units of the vertical axis on the right shows how a logarithmic axis is linear in the exponent to which the base must be raised to obtain the actual value (that is, it's linear in the logarithm). Also note how the same *relative* change results in the same visual length when using logarithmic scales: both scale arrows are the same length and represent a 100% change,
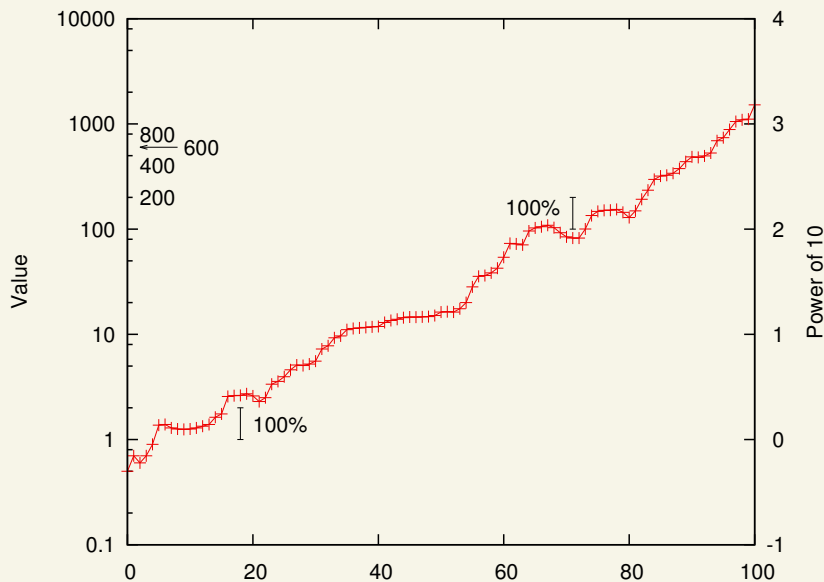
A semi-log plot. See the text for details. (Labels for some of the minor tic marks on the left axis are provided.)

(continued)

although the absolute change is different for both arrows (1 to 2 on the left, 100 to 200 on the right).

You can switch on logarithmic plotting either on a single axis (usually the y axis) or on both, yielding a single- or double-logarithmic plot, respectively. Single-logarithmic plots are sometimes referred to as *semi-log plots*, and double-logarithmic plots are also known as *log-log plots*. Both plots are used for different purposes.

There are two uses for single-logarithmic plots. First, if you have data with exponential behavior, showing it on a single-logarithmic plot lets the data fall onto a straight line:

$$y(x) = e^{ax} \implies \log(y(x)) = ax$$

In other words, if $y(x)$ is an exponential function of x, then $\log(y(x))$ is a linear function (straight line) of x. You can use single-logarithmic plots in this way to determine, for instance, the half-life of a radioactive substance from measured decays.

The other use for single-logarithmic plots is common in finance: on a single-logarithmic plot, all *relative* changes have the same size, no matter how large the underlying absolute changes. Here's an example: if a stock you own falls by \$10 in a single day, your reaction will be different depending on whether you paid \$20 or \$100 initially. In the first case, you lost 50% of your money; but in the second, you lost only 10%. You care about relative changes, not absolute ones. Therefore you want a plot in which all changes of the same relative magnitude result in the same distance on the graph. That's what semi-log plots do for you. If the value before the change is y_1 and the value after the change is $y_2 = g\,y_1$, where g equals 1.1 (corresponding to a 10% gain), then the change in absolute terms is

$$y_1 - y_2 = y_1 - gy_1 = (1 - g)y_1$$

which depends on the overall size of the value y_1. But if you consider logarithms, you find

$$\log(y_1) - \log(y_2) = \log(y_1) - \log(gy_1) = \log(y_1) - \log(g) - \log(y_1) = -\log(g)$$

which is *independent* of the overall value and depends only on the relative size of the change, g. This is why long-term charts of, for example, the Dow Jones index are usually shown as semi-log plots: a 10% drop today or in 1929 results in the same distance covered on the graph, despite the fact that the Dow Jones is around 17,000 today but stood at about 300 then.

Double-logarithmic plots serve a different purpose. They help you identify *power-law behavior*—that is, data that follows an equation such as the following (C is a constant):

$$y(x) = Cx^k$$

The analysis goes through as before, but you end up with logarithms now on *both* sides of the equation:

$$\log(y(x)) = k\log(x) + \log(C)$$

The resulting graph is a straight line with a slope that depends on the exponent k. You saw an example of this in chapter 1, when estimating the completion time of a long-running computer program.

Double-logarithmic plots are very important. Power laws occur in many different contexts in the real world but aren't always easy to spot. Go back to figure 1.3 in chapter 1: many different curves will seem to fit the data about equally well. But once plotted on a double-log plot (see figure 1.4), the linear shape of the data stands out and provides a strong and easily recognizable indicator of the underlying power-law behavior.

TIP Log and log-log plots are part of the standard toolset. They reveal exponential and power-law behavior, and they're generally useful when dealing with data that changes over many orders of magnitude. Learn how to use them!

3.5 Smooth interpolation and approximation

Gnuplot can construct a variety of smooth curves as representations of a data set. You can invoke this feature using the `smooth` directive to the `plot` command, followed by an additional keyword that selects the specific interpolation method. Table 3.4 lists the available smoothing methods, together with brief descriptions.

Table 3.4 Smoothing methods available for `plot ... smooth ...`

	Keyword	Description
1	`csplines`	Cubic spline interpolation
2	`mcsplines`	Monotone cubic spline interpolation
3	`acsplines`	Approximate cubic spline
4	`bezier`	Bézier curve
5	`sbezier`	Sorted Bézier curve (constructed after sorting the data points on their x values)
6	`kdensity`	Kernel density estimate
7	`cumulative`	Cumulative distribution function (constructed after sorting the data points on the x values)
8	`cnormal`	Normalized cumulative distribution function (independent of the number of data points—see the text for details)
9	`unique`	Sorts data points on their x values, dedupes repeated entries, and replaces the y value for repeated x values with the average of the y values
10	`frequency`	Sorts data points on their x values, dedupes repeated entries, and replaces the y value for repeated x values with the sum of the y values

Although all of these smoothing methods are invoked using the smooth keyword, they fall into three broad groups that do different things and apply to different situations. Methods 1–5 construct a smooth curve through a set of (x,y) coordinate pairs. Methods 6–8 treat the data set as a one-dimensional (or *univariate*) point distribution and generate visual representations of this distribution. Methods 9 and 10 remove multiple entries from a data set. As it turns out, method 10 (frequency) can also be used to count data points in order to construct a histogram.

The remainder of this chapter describes each group of options. In chapter 13, I'll show you how these methods can be applied to solve common problems in the graphical analysis of data sets.

3.5.1   Interpolation curves

Gnuplot provides two sets of methods to draw a smooth curve through a set of (x,y) points: those based on cubic splines, and those based on Bézier curves.

SPLINE INTERPOLATIONS

Splines[4] are smooth curves constructed from piecewise-polynomial functions that are smoothly joined together. If the polynomials used are third order, then the resulting spline is called a *cubic* spline. Splines can be used to give a smooth approximation to a set of points; *interpolating* splines pass through each of the points exactly, whereas *approximating* splines only pass the vicinity of each point.

Both csplines and mcsplines construct interpolating cubic splines, meaning the resulting curve passes through every point in the data set. The second form (mcsplines)[5] also guarantees that the curve has the same monotonicity and convexity properties as the original data. Look at figure 3.4: the curve drawn using csplines occasionally "overshoots" near locations of greatest curvature, whereas the line drawn using mcsplines doesn't. The second curve therefore may give a better, more faithful representation of the data.

The third keyword (acsplines) creates an *approximating* spline, meaning the curve doesn't pass exactly through the data points. To control how close the curve passes to each point, you need a new parameter (in addition to the x and y coordinates of each point), which is supplied to gnuplot using a *third* "column" in the using directive, like so:

```
plot "data" u 1:2:(0.01) smooth acsplines
```

This third parameter is a measure for the weight with which each point contributes to the spline: the greater it is, the more the point influences the curve, and the closer the curve passes to each point; the smaller it is, the less each point influences the curve. The two extreme cases are instructive: as the weight parameter approaches infinity,

---

[4] I'll have more to say about splines in section 13.1.2.
[5] F. N. Fritsch and R. E. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM Journal on Numerical Analysis* 17 (1980): 238–246.
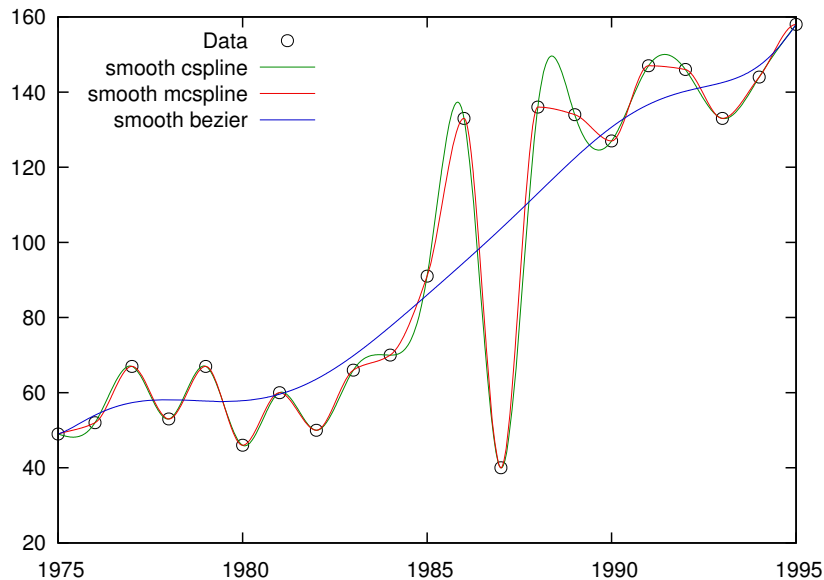
Figure 3.4 Interpolating splines: both `csplines` and `mcsplines` interpolate the data points smoothly, but `mcsplines` doesn't "overshoot" the data. In contrast to splines, which are locally defined, the Bézier curve constitutes a global approximation. `plot "splines" u 1:2 w p pt 6, "" u 1:2 smooth csplines, "" u 1:2 smooth mcsplines, "" u 1:2 smooth bezier`. (The directive `pt 6` selects the shape of point symbols—see section 9.2.1.)

the curve approaches the interpolating spline (which passes through every point exactly); but as the weight parameter approaches zero, the resulting spline approaches the linear least-squares fit to the data. (See figure 3.5.)

I'd like to draw attention to the way I handled the third column specifier in the `using` directive: I supplied an expression in parentheses, which means it was interpreted (because of the parentheses) as a *transformation* to be applied to the data. The expression itself, however, is a constant, so the value of this third column is the value of that constant. Because the third column gives the weight with which the point should contribute to the spline approximation, every point received the same weight. Instead of supplying a numerical constant, you could specify an actual column from which to take the weights, like so: `plot "data" u 1:2:3 smooth acsplines` (note the missing parentheses, which means the `3` is interpreted as a column number, not as a numerical constant). If the third column in the data file contains some measure of the reliability or accuracy with which each point is known, you can have each point contribute with a *different* weight, according to the certainty with which it's known.

> **TIP** It's frequently useful to supply a constant expression in inline transformations. For example, to have every point contribute equally to a spline approximation, you can use `plot "data" u 1:2:(0.1) smooth acsplines`.
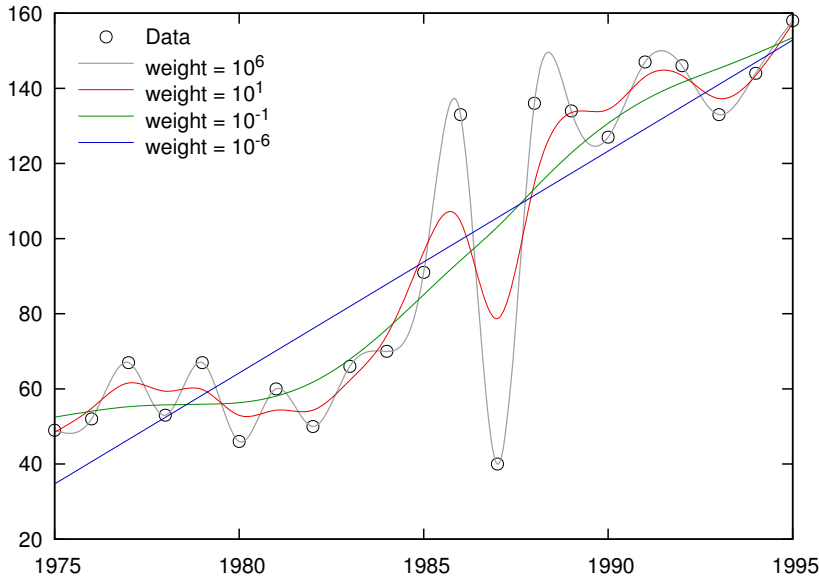
Figure 3.5   Influence of the weight parameter on approximate splines. As the weight increases, the curve approximates an interpolating spline; as the weight goes to zero, the curve approaches a straight line: `plot "splines" u 1:2 w p pt 6, "" u 1:2:(10e6) s acs, "" u 1:2:(10) s acs, "" u 1:2:(1./10) s acs, "" u 1:2:(10e-6) s acs`.

Approximating splines are an important tool for graphical analysis. In chapter 13, I'll show you how to use them to visualize trends and features in noisy data sets.

BÉZIER APPROXIMATIONS

Bézier curves[6] are constructed as the path that would be traced by a particle that, starting at one point, steers toward each of the subsequent points in turn. By construction, Bézier curves pass through the first and last point exactly but in general don't pass through any of the internal points. Whereas splines are defined locally (between neighboring points), Bézier curves are defined for the entire data set. As a result, Bézier curves usually appear particularly smooth—which is also why they're used so frequently in computer graphics and design.

Gnuplot creates a Bézier curve of order n (where n is the number of data points) using `smooth bezier`. (See figure 3.4.)

Because Bézier curves are traced out by a particle heading toward a set of waypoints in turn, not only the positions of the waypoints matter, but also the order in which they're traversed. When you use `smooth sbezier`, the data points are first sorted by their x coordinates, and a Bézier curve is constructed for the resulting data set.

---

[6]   The Wikipedia entry on Bézier curves is a good place to start.

3.5.2 *Point distributions*

Gnuplot includes a few different ways to visualize univariate point distributions, some involving the `smooth` facility, and some not. Here we're primarily interested in the technicalities of using the `smooth` feature—consult chapter 13 for additional information about the math involved and to learn about the kinds of questions these methods can help you answer.

KERNEL DENSITY ESTIMATES

Assume that you have a number of measurements of a *single* quantity, like the ones in the following listing. These numbers may be weights of different shipments, or the time it took a bunch of queries to finish, or the temperatures in various parts of the house. It doesn't matter: in every case, you want to know how the values are distributed, and that means how much the points scatter about.

Listing 3.4 Data for figure 3.6 (file: measurements)

```
2.14
1.25
1.10
1.00
1.60
1.31
0.95
0.90
4.01
1.95
2.53
0.65
```

You can just plot all the points along a horizontal line (see figure 3.6) and take a look, but if the number of points becomes large, it can be difficult to recognize the details of the distribution. In a *kernel density estimate*, you replace each point with a strongly peaked but smooth curve (the *kernel*) centered at the location of each point. You then sum the contributions from all these curves and plot the result.

Gnuplot's `smooth kdens` feature uses Gaussian curves as kernels and takes two columns in the `using` specifier: the first column gives the position of each point, the second the weight of each point. If you want to weight all points equally, you may again supply a constant. To obtain a *normalized* density estimate, the weight of each point should be $1/n$, where n is the number of points in the data set.

With a third parameter, you control the *width* of each kernel. If you choose a large value for this parameter, each point will appear more smeared out, and the resulting curve will show few details; if you choose a small bandwidth, more features will show, but the resulting curve may be noisy. (This is explained in greater detail in section 13.3.3.) The bandwidth parameter isn't passed as a pseudo-column; instead, it follows the `bandwidth` keyword:

```
plot "data" u 1:(1) s kdens, "" u 1:(1) s kdens bandwidth 1
```
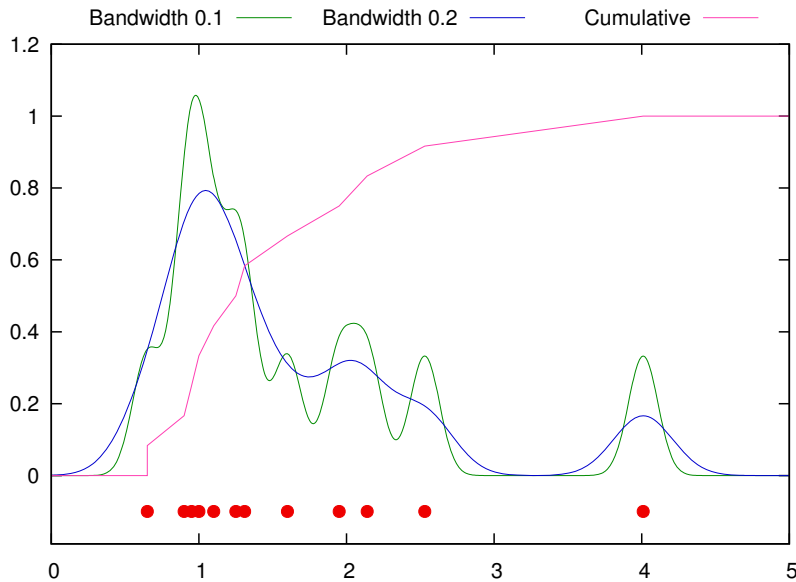
Figure 3.6   Visualizing point distributions using kernel density estimates and the cumulative distribution function: `plot "measurements" u 1:(-0.1) w p pt 7, "" u 1:(1/12.) s kdens bandwidth .1, "" u 1:(1/12.) s kdens bandwidth 0.2, "" u 1:(1/12.) s cumul`. The calculated default bandwidth is 0.58. See listing 3.4 for the data.

If you don't supply a bandwidth explicitly, gnuplot calculates one based on the standard deviation of the data set and stores it in a variable called GPVAL_KDENSITY_ BANDWIDTH. In practice, this default bandwidth often appears too broad. You can inspect its value using print GPVAL_KDENSITY_BANDWIDTH. (I'll formally introduce gnuplot's internal variables in section 5.5.)

CUMULATIVE DISTRIBUTION FUNCTIONS

Another way to visualize a point distribution is to count the fraction of points that lie to the left of the current position: this is the *cumulative distribution function* of the data set. By construction, the cumulative distribution function is monotonically increasing: it's 0 to the left of the point with the smallest value, and it's 1 to the right of the point with the greatest value. (See figure 3.6.)

In gnuplot, you use smooth cumul to create a cumulative distribution function. It takes two columns in the using specifier: the first holds the position of each measurement, the second the associated weight. (It isn't necessary for the data set to be sorted: gnuplot takes care of that.) If you supply a weight of 1, then the value of the resulting curve at any position is equal to the *number* of data points to the left of the current position; if you supply a weight of $1/n$ to cumul, where n is the number of points in the data set, then the curve is equal to the *fraction* of data points to the left.

Gnuplot offers a second way to draw a cumulative distribution function, which doesn't explicitly depend on the number of data points: smooth cnormal. Like smooth

cumul, you need to supply two columns in the using directive: the first for the position of each measurement, the second for the weight of each point. With smooth cnormal, gnuplot sums the weights and then divides the sum by the total number of data points in the current data set. The following two commands are therefore equivalent (assuming that the variable n holds the number of records):

```
plot "data" u 1:(1.0/n) smooth cumul
plot "data" u 1:(1.0) smooth cnormal
```

The advantage of smooth cnormal is that you don't need to know and supply the number of data points explicitly. Nevertheless, the second parameter in the using directive must be there!

3.5.3   *Deduping repeated entries*

The last two smoothing methods available are less sophisticated. When using smooth unique on a data set consisting of (x,y) coordinate pairs, gnuplot first sorts the points in ascending order on their x coordinates. It then replaces all points that have the same x coordinate with a single point that has the same x coordinate, but whose y coordinate is the *average* over all the y coordinates of such points.

The smooth frequency method works the same way, except that it replaces the y coordinate with the *sum* of the y coordinates of all the affected points. This isn't very useful by itself, but it can be used to construct histograms of point distributions—I'll show you how to do that when we revisit this topic in chapter 13.

3.6   *Summary*

The plot command is the most important gnuplot command. Not only does it generate plots of functions and data, but it also can apply arbitrary transformations to the data as it's being plotted, and it can perform a variety of operations to interpolate or approximate the data. We also looked at the way gnuplot handles mathematical expressions and at its built-in functions.

Next, we'll take a closer look at data files and data file formats. You'll see how gnuplot reads data from files and learn more about accessing column values in conjunction with inline transformations.

Part 2

Creating graphs

This part explains in detail gnuplot's various options and features that you'll use all the time when preparing graphs and figures. Chapter 4 discusses file formats of data files and the different pieces of information that gnuplot can extract from them. Chapter 5 covers a number of supplemental topics. Very important is an introduction to gnuplot's string data type, which not only is essential when you want to add labels and explanations to a graph, but also is used in gnuplot's macro facility and as gnuplot's only complex data type. This chapter also explains how you can facilitate repetitive tasks through inline loops. Chapter 6 provides an overview and catalog of all the different styles that are available to represent data. Chapter 7 introduces all kinds of decorations that can make a graph more informative. You'll learn about arrows, text labels, and ways to customize a graph's legend. Chapter 8 describes how to control the way axes and tic marks are drawn. The chapter also explains how to handle time-series data and how to use calendar dates as tic labels.

Managing data sets and files

This chapter covers

- Data-file formats and options
- Managing structured data sets
- Accessing columns and pseudocolumns
- Pseudofiles and input/output redirection
- Metadata in data files

Working with data is what gnuplot is all about. Data is usually presented to gnuplot as files, and in this chapter you'll learn a range of useful tricks that can make you more productive when working with gnuplot.

We'll begin with a review of gnuplot's standard file format and discuss options to modify its defaults. We'll then look at some special gnuplot syntax to pick out only parts from larger files. This is required when dealing with data sets that are larger or more complicated than the straightforward ones you've encountered so far, and it's often useful.

Next, we'll systematically consider all the ways you can access columns—in particular, in conjunction with inline transformations. In the process, you'll also

55

encounter *pseudocolumns*, which are synthetic columns that gnuplot automatically provides when reading a file. Pseudocolumns hold useful information (such as line numbers) and often come in handy when you're constructing inline transformations. Then we'll turn to files that aren't: *pseudofiles* are data sources that don't live on the disk but that you can treat almost as if they were regular files.

But data files often contain information beyond the actual data that goes onto a plot. Such *metadata* typically consists of textual information that you may want to place onto a plot in the form of text labels. In this chapter, where we talk about data files in general, we'll merely summarize the kind of metadata that gnuplot can take from a file and use in a plot. The details will follow in their proper context when we discuss how to handle text labels and similar auxiliary information in a plot in chapters 6 through 8.

4.1   *Quickstart: the standard data-file format*

Data is usually fed to gnuplot as plain text in a simple, whitespace-separated format: gnuplot neither uses nor requires special proprietary binary formats or weird markup. Nor do you need to use special tools or libraries to generate data that constitutes suitable input for gnuplot.

Input files may also contain comments, text strings, and other information, in addition to the data. Furthermore, many aspects of this format can be changed through options. We'll consider each of these topics in turn.

By default, gnuplot data files are text files that contain data in whitespace-separated columns, with one line per record. This is a natural tabular format—basically, the way you would lay out information in a spreadsheet or the way it would result from a database query. Listing 4.1 shows the beginning of a data set with information on customer orders as received by an online retailer. There's a lot of information in such a file; in particular, the relationships between different columns are often interesting. Figure 4.1 shows four different plots, using different plot *types*, based on this file. It includes a time-series plot of the number of orders received per day, a histogram of the number of items per order, and a scatter plot investigating the relationship between the weight of an order and its value. The last panel in figure 4.1 consists of a parallel-coordinates plot: a special graphical technique to identify relationships in multivariate data sets. (For the sake of completeness, the commands to generate figure 4.1 are given in listing 4.2. Don't worry if you don't understand all this yet; we'll cover these commands in the chapters to come.)

> ### Listing 4.1   Incomplete generic data file—see figure 4.1 (file: orders)

```
# ID      Date        CustomerID    ItemCount    Weight    Value    ShipMethod
oid0012   2014-12-01  cid59         2            2.13      11.39    1-Regular
oid0014   2014-12-01  cid61         4            7.11      15.44    0-Ground
oid0015   2014-12-01  cid30         2            2.33      11.11    0-Ground
oid0025   2014-12-01  cid87         4            4.48      16.55    2-Express
oid0034   2014-12-01  cid08         4            4.35      16.95    1-Regular
...
```
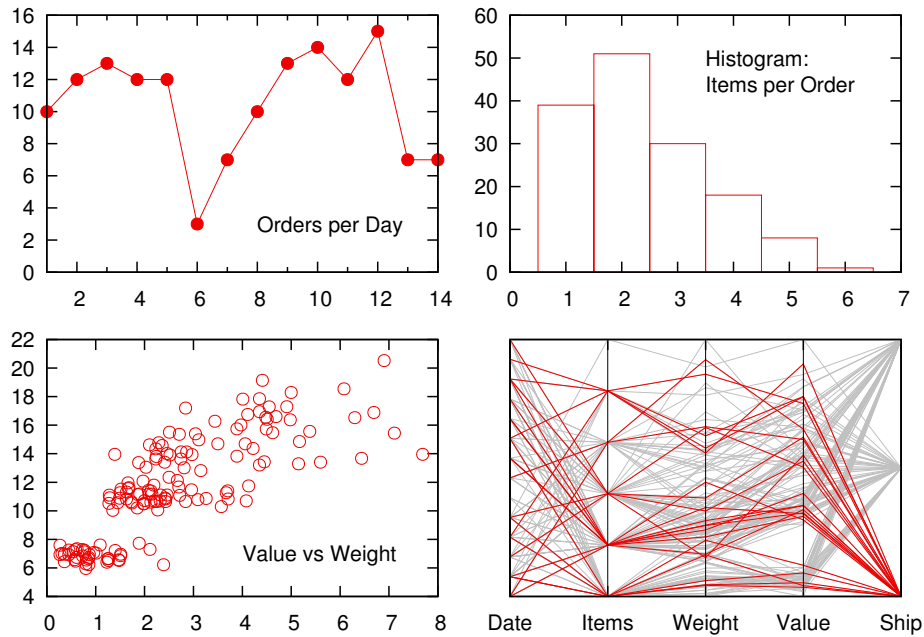
Figure 4.1 Different types of graphs, all based on a single data file. Clockwise from top left: a time-series plot of the number of orders per day, a histogram showing the distribution of the number of items per order, a parallel-coordinates plot of several order attributes (with ground-shipping orders highlighted), and a scatter plot of order value versus order weight.

Listing 4.2 Commands for figure 4.1 (file: orders.gp)

```
unset key

set multiplot layout 2,2 margins 0.06,0.95,0.085,0.95 spacing 0.075,0.1

# --------------------
set xtics 2,2
set mxtics 2

set label 1 "Orders per Day" at 8,3
plot [1:14] "orders" u (strcol(2)[9:10]+0):(1) s f w lp pt 7

set xtics 1
unset mxtics

# --------------------
set label 1 "Histogram:\nItems per Order" at 3.5,50
plot [0:7] "orders" u 4:(1) s f w boxes

# --------------------
set label 1 "Value vs Weight" at 4,7
plot "orders" u 5:6 w p pt 6
```

```
# --------------------
set xtics ( "Date" 1, "Items" 2, "Weight" 3, "Value" 4, "Ship" 5 )
unset ytics

set style parallel lw 1
set linetype 1000 lc "0xff000000"
                                                Returns the shipping
                                                method as an integer
ship(c) = 0+strcol(c)[1:1]          ◁───────┘

unset label 1
plot [1:5] "orders" u (strcol(2)[9:10]+0):4:5:6:(ship(7)) w parallel lt 10,
➡          "" u (strcol(2)[9:10]+0):4:5:6:(ship(7)):(ship(7)==0?1:1000)
➡               w parallel lc var

unset multi
```

4.1.1   *Comments and header lines*

Lines beginning with a # are considered comment lines and are ignored. You can also instruct gnuplot to ignore the first *n* lines of a file by using the skip directive in the plot command:

```
plot "data" skip 2 using 1:2
```

Now gnuplot will discard the first two lines of the file, *regardless of whether they're commented out.* This is a convenient way to get rid of header lines without having to comment them out explicitly.

> **TIP**   You can use the new skip keyword to discard the first *n* lines of a file as it's being plotted.

4.1.2   *Selecting columns*

You can pick out any column and plot its values against any other column by means of the using directive. If no using directive is found, gnuplot plots the first column against the line number; if only a single column is specified, gnuplot plots that column against the line number.

For the most part, input-file parsing with gnuplot is very robust and works without much tinkering. One good piece of advice is to always specify all required columns explicitly via the using directive. If this is done, gnuplot silently skips any garbage (fields it can't parse) in the file, treating them as missing values. If you rely on the default columns (without the using directive), gnuplot will instead silently bail when it encounters an unparsable field. This is most likely to happen when you're doing casual work with small files containing only two columns, and it can sometimes lead to mysterious failures. My advice: make it a habit *always* to specify all columns with using.

> **TIP**   If you specify all columns to plot explicitly in the using directive, gnuplot will correctly parse even files that are partially garbled or that contain garbage.

Time and date information in a data file constitutes a special case. Gnuplot provides special facilities to parse timestamps, which we'll discuss in detail in section 8.4.2. Do *not* attempt to parse time and date information yourself—use the appropriate options![1]

Plain text as data format

Using text files as primary data stores has a number of advantages: you don't need special tools to generate or read them, and it's easy to write scripts to pre- or post-process them. If they're large, it's simple to compress them. And if necessary, you can even load them into any text editor (or a spreadsheet!) and manipulate them by hand.

Furthermore, text files are largely portable across most current computer architectures (so that files generated on Unix, say, can be opened and read on Windows)—except for the choice of the linebreak indicator (newline character). Gnuplot expects lines in input files to be terminated by whatever the local operating system (or rather, the local C library) considers the "native" newline character. If you encounter problems reading files generated on a different platform, try converting newlines to the local format. (The same argument applies to the files gnuplot writes: lines are terminated with the native newline character on the given system.)

Gnuplot string handling is based around Unicode and UTF-8. We'll have more to say about both topics in the sidebar in section 5.1 in the next chapter.

Gnuplot currently does *not* have built-in support for data files in a hierarchical or markup format, such as XML, JSON, or HDF5. If you want to plot such data sets, you'll have to extract them to plain, column-oriented text files first.

4.2 Managing structured data sets

Sometimes data sets have more internal structure than the simple one-record-per-line model you've seen so far. Two cases in particular are reasonably common and must be dealt with: files containing data blocks—that is, different data sets one after another in a single file—and files containing records that span several lines each. Let's look at how to handle each case in turn.

4.2.1 Multiple data sets per file: index

Here's a common scenario: monitoring web traffic across multiple hosts. A script logs in to each host, pulls the log files for the last few days, summarizes the hits per day, and writes the results to an output file before moving on to the next host. You want to use gnuplot to look at the contents of the resulting file.

---

[1] In listing 4.2, I chose to manipulate the date value explicitly using string functions, but only because in the crowded multipanel figure, I needed to exert finer control over the appearance of the tic marks.

The important issue here is that the resulting file doesn't contain individual data *points*, but entire data *sets*: one for each host. Each data set spans several rows: one for each day. The file might look something like the one shown in the following listing.[2]

Listing 4.3   Data file containing several sets (file: traffic1)

```
# Day    Hits
0        101
1        100             host=main
2        102
3        103

0         25
1         26             host=staging
2         27
3         28

0          5
1          4             host=test
2          3
3          2
```

This data file looks as if several distinct data files (for different hosts) had been appended to one another. Because this situation is sufficiently common, gnuplot provides a way to handle it. But first, we need to look at the meaning of *blank lines* in a data file.

For gnuplot, blank lines in a data file are significant. A *single blank line* indicates a *discontinuity* in the data. The data above and below the blank line is treated as belonging to the same data set (and is shown using the same line color and line style), but no connecting line is drawn between the records before and after the blank.

In contrast, *double blank lines* are used to distinguish *data sets* in the file. Each set can be addressed in the `plot` command as if it were in a separate file using the `index` directive to `plot`. Table 4.1 summarizes these differences.

Table 4.1   Meaning of blank lines in data files

	Meaning
Single blank line	Indicates a discontinuity in the data. No connecting line is drawn between points separated by a single blank line.
Double blank line	Indicates different data set (or data blocks) in the same file. Different data blocks can be selected using the `index` keyword with the `plot` command.

---

[2]   The data in this data set is dorky. This is intentional. I want to make sure you can distinguish data from each of the three hosts by looking at the data (the number of hits) itself. This will be important when we compare this data set with the one in listing 4.4.

The `index` directive follows immediately after the filename in the `plot` syntax and takes at least one argument specifying which data set to select from the file. The argument can be either numeric or a string.

SELECTING DATA SETS BY POSITION

A numeric argument is treated as an index into an array, following the C language convention of counting from 0 (zero). Therefore, to plot only the traffic for the staging site, you can use

```
plot "traffic1" index 1 using 1:2 w linespoints
```

This picks out only the data set with `index` 1 and shows it with `linespoints`.

The `index` directive can be abbreviated as `i`. It can take up to three arguments, separated by colons (similar to the syntax familiar from `using`):

```
index {int:start}[:{int:end}][:{int:step}]
```

If only a single argument is given, only the corresponding data set is plotted. If two arguments are present, they're treated as the index of the first and last data set (inclusive) to be shown: `plot "data" index 2:5` plots four data sets total. A third argument is interpreted as a step size. Accordingly, `plot "data" index 2:5:2` plots only the data in sets 2 and 4. Only the first argument is mandatory.

SELECTING DATA SETS BY NAME

Gnuplot 5 offers a new way to pick out a single data set from a file: if the data set is preceded by a comment line that includes the ID or name of the data set, you can use this identifier in the `index` directive. The comment character and any leading whitespace are removed; the following characters are treated as the "name" of the data set that follows the comment line in the file.

Referring again to the file shown in listing 4.3, the following two commands are equivalent:

```
plot "traffic1" index 1 using 1:2 w linespoints
plot "traffic1" index "host=staging" using 1:2 w linespoints
```

Only a single data set can be selected this way. If the "name" given is ambiguous, then gnuplot plots the *first* data set in the file that matches. (Matching is case-sensitive.) Referring again to listing 4.3, the following three commands are both unambiguous and will result in an identical plot:

```
plot "traffic1" index 2 using 1:2 w linespoints
plot "traffic1" index "host=t" using 1:2 w linespoints
plot "traffic1" index "host=test" using 1:2 w linespoints
```

In contrast, `plot "traffic1" index "host" u 1:2 w lp` is ambiguous, and gnuplot will plot the first matching data set in the file (which happens to be the one for the main host): that is, the first data set in the file.

4.2.2   *Records spanning multiple lines: the every directive*

Whereas the index directive lets you select consecutive sets of data from a file, the every option, which we discuss now, solves a different problem. Consider the same arrangement of hosts as in the previous section, together with an auxiliary script that gathers traffic data from all hosts and dumps it into a single file. But in contrast to the previous scenario, the script now retrieves only a *single* day's worth of traffic and appends it to the output file before moving on to the next host. After a few days, the output file might look like the one shown next.

Listing 4.4   Data file containing interleaved data sets (file: traffic2)

```
# Day      Hits
0          101          ◄──── host=main
0           25          ◄──── host=staging
0            5          ◄──── host=test
1          100
1           26
1            4
2          102
2           27
2            3
3          103
3           28
3            2
```

Here, each record for a single day spans three lines: one for each host. (Compare this file against the one in listing 4.3: the data in both is exactly the same!) If you want to plot the traffic for each host separately, you can use the every directive to pick up only the relevant subset of all lines. The following command, for instance, plots only the traffic for the staging host:

```
plot "traffic2" every 3::1 using 1:2 with lp
```

Using the every directive, you can control how you step through individual lines. The syntax looks similar to the syntax used for index, except that individual arguments are separated by two colons. Unfortunately, this similarity is somewhat deceiving, because the order of the arguments isn't the same for every as it is for index:

```
every {int:step}[::{int:start}[::{int:end}]]
```

The first argument is the increment, followed (optionally) by the first and last line numbers. Line numbers are counted from zero. Don't forget to use double colons with the every directive: single colons won't generate an error message but will lead to strange and hard-to-predict behavior.[3]

---

[3] I'm simplifying here. Gnuplot recognizes an additional concept known as a *data block* in a file: a set of consecutive lines delimited from each other using single blank lines. Data blocks are functionally redundant with data sets (delimited by double blank lines). Data blocks can be selected through additional arguments to the every directive, which are placed between the double colons. This is why it's not illegal to use single colons in this context. If you want to know more about data blocks, check the standard gnuplot reference documentation.

You can play other tricks with the `every` feature. For instance, the following command skips the first two lines of every data set in the file:

```
plot "data" every ::2 using 1:2
```

This is similar to, but not quite the same as, the `skip` facility: `skip` only discards lines at the beginning of the file, not at the beginning of every data set in the file. Moreover, `every` only applies to data lines, whereas `skip` also skips comments. Feel free to devise other creative uses for the `every` keyword!

> **TIP** I recommend that you use a file format suitable for the `index` facility when you need to combine several data sets in a single file. The `every` keyword is a bit of a legacy feature, and data handling based on `index` is more robust and transparent.

4.3   File format options in detail

You can control several aspects of the file format using options. For instance, you can choose additional characters to indicate comment lines in data files. In this section, we look at the fine points of input-file formats: numbers, missing values, comments, and strings.

4.3.1   Number formats

Gnuplot can read both integers and floating-point numbers, as well as numbers in *scientific notation* (strictly speaking, *scientific E notation*): a floating-point number, followed by an uppercase or lowercase character `e`, followed by an integer, which is interpreted as a power of 10. The numeric value of such a field is obtained by multiplying the floating-point part by 10 raised to the appropriate power. A couple of examples will make this clear: in scientific notation, the value 35,100 is encoded `3.51e4`; the value -0.0001 is written `-1e-4`.

You can also allow the letters `d` and `q` (both uppercase and lowercase) instead of `e` or `E` for Fortran D and Q constants, by setting

```
set datafile fortran
```

This option is off by default, because it requires additional parsing; it should be enabled only if needed.

4.3.2   Comments

You can include comments in a data file on lines starting with the comment character (#). The line must *start* with the comment character and is ignored entirely. If gnuplot encounters a # in any location other than the first one in the line, it isn't interpreted as a comment character, and any text following it is interpreted as additional data. This isn't a problem as long as only columns preceding it are specified in the `using` declaration of the `plot` command.

You can make gnuplot interpret additional characters as comment characters by using the `set datafile commentschars` command:

```
set datafile commentschars ["{str:chars}"]
```

For example, to tell gnuplot that the exclamation point indicates a comment line in a data file, you can say

```
set datafile commentschars "!"
```

The string can contain any number of characters, each of which is interpreted as a comment character if found at the beginning of a line. Resetting this option to a new value overrides all previous settings.

4.3.3   *Field separator*

By default, fields (columns) are separated from one another by whitespace, which means any number of space or tab characters. You can change the field separator using the `set datafile separator` command:

```
set datafile separator [ "{str:char}" | whitespace | tab | comma ]
```

You can use the symbolic names `whitespace`, `tab`, and `comma`, or specify a string of characters. If you provide an explicit string, then each character in the string will be treated as a separator character.

For example, to make both the colon and the pipe character individually be field separators, you'd use

```
set datafile separator ":|"
```

To let gnuplot parse comma-separated (CSV) files, either of the following two commands will work:

```
set datafile separator ","
set datafile separator comma
```

Separator characters aren't interpreted as separators when inside quoted strings: quoted strings are always interpreted as the entry of a single column.

You can specify several characters, each of which by itself is interpreted as a field separator. It isn't possible to specify a multicharacter pattern as a field separator. (This isn't true when using `whitespace`.) To reset to gnuplot's default behavior, you can issue `set datafile separator whitespace` or `set datafile separator` so that columns are split on whitespace again.

4.3.4 *Missing values*

You can use the `set datafile missing` command to specify a string to be used in a data file to denote missing data:

```
set datafile missing ["{str:str}"]
```

An example is `set datafile missing "NaN"`, which interprets the IEEE floating-point indicator `NaN` ("not a number") as a missing value. There's no default value for this parameter.

Having an indicator for missing values is important when you're using a whitespace-separated file format: if the missing value were left blank, gnuplot wouldn't recognize it as a column value and would use the value from the *next* column instead.

> **TIP** In a whitespace-separated format, you must use an explicit missing-value indicator. It isn't permissible to leave a missing value blank.

The interpretation of missing values in a data set depends on the precise syntax of the `using` directive. Let's look at two examples. The following listing shows a file containing a missing value.

Listing 4.5 Data file containing a missing value (file: missing)

```
1    10
2    10
3    11
4    12
5    NaN
6    11
7    11
8    10
9    10
```

First, let's assume that `datafile missing` has been set to `"NaN"`. If you now plot this file using

```
plot "missing" using 1:2 with linespoints
```

then the fifth record (the one containing the missing value) is ignored and the data is plotted with one continuous, unbroken line. In contrast, if you use the command

```
plot "missing" using 1:($2) with linespoints
```

or if the `datafile missing` option doesn't equal `"NaN"` (either because this option is undefined or because it's been set to a different value), then gnuplot also ignores the fifth record but treats it as a blank line and therefore doesn't draw a connecting line across the gap (see figure 4.2).
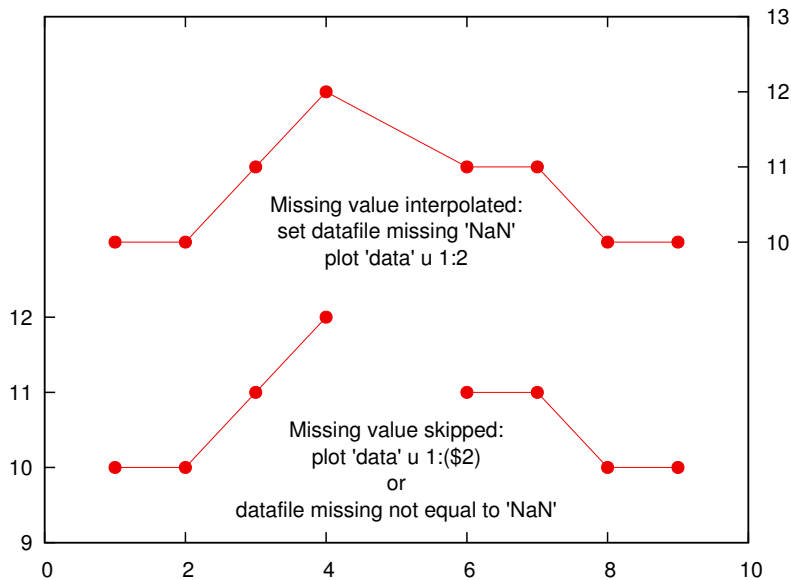
Figure 4.2   Gnuplot treats missing values differently, depending on the value of the `datafile missing` option and the `plot` syntax. The data file is shown in listing 4.5.

4.3.5   *Strings in data files*

Gnuplot can read and process text fields found in input files. A valid text field can be any string of printable characters that doesn't include blank spaces. If the string contains blanks, it must be enclosed in *double* quotes to prevent gnuplot from interpreting the blanks as column separators. (Single quotes don't work!) The enclosing double quotes are stripped off and aren't part of the field's value. If a field contains whitespace and is protected by enclosing double quotes, it must not contain double quotes as part of the string value. If you need to use quotation marks and blanks in the same string, you must use single quotes inside the string and double quotes to enclose the entire field. If you've designated a non-whitespace character as a column separator using `set datafile separator` (see the previous section), the same considerations apply: strings containing the separator must be protected with double quotes. Listing 4.6 shows some ways you can use strings in a data file.

> **TIP**   You can include strings in data files. If the strings contain whitespace, they must be enclosed in *double quotes*. Single quotes won't work.

Listing 4.6   Strings in data files need quotes only if they contain whitespace.

```
# Year   Title
1965     Yesterday                      ◁─────  Bare string without blanks
1966     "Yellow Submarine"             ◁─────  Blanks require double quotes.
1969     "Maxwell's Silver Hammer"      ◁─────  Single quote inside double quotes
```

For more information on string handling, check section 5.1 in the next chapter.

4.4   *Accessing columns and pseudocolumns*

In chapters 2 and 3, you saw how to access columns in `using` directives. Here, you'll review what you know already and learn some new features. Table 4.2 summarizes all methods of accessing columns in gnuplot.

Table 4.2   Column-access methods, pseudocolumns, and column-access functions. (An entry in the first column in *italics* is a placeholder for any matching value. Non-italic entries must be used verbatim.)

Specifier	Example	Description
number	`using 1:2`	Accesses a column by its horizontal position in the file.
string	`using "Height":"Weight"`	Accesses a column by its name as given in the first non-comment line in the file.
0	`using 0:2`	Pseudocolumn, containing the record number (starting from zero) in the current data set.
–1	`using -1:2`	Pseudocolumn, containing the line number (starting from zero). Reset by a single blank line.
–2	`using -2:2`	Pseudocolumn, containing the index (starting from zero) of the current data set. Reset by a double blank line.
(*expression*)	`using 1:($2+$3)`	Inline transformation. The expression in parentheses is evaluated, and the value of the expression is plotted. The column values for the current record are available through the shorthands `$1`, `$2`, and so on.
(*constant*)	`using 1:(1)`	Same as the previous, but the expression is a constant, independent of the values in the data file.
(column(*expression*))	`using 1:(column($3+1))`	The argument to the `column()` function should be an expression that evaluates to an integer. The value of this expression is interpreted as a column number, and the `column()` function returns the value of the current record for the desired column as a numeric value. (For example, `column(3)` returns the current value of the third column as a numeric value.)
(column(*string*))	`using 1:(column("Weight"))`	If the argument to the `column()` function is a string, it's interpreted as the name of a column (as given in the first non-comment line in the data file). The `column()` function returns the value of the current record for the desired column as a numeric value.

Table 4.2 Column-access methods, pseudocolumns, and column-access functions. (An entry in the first column in *italics* is a placeholder for any matching value. Non-italic entries must be used verbatim.) *(continued)*

Specifier	Example	Description
`(stringcolumn(`*expression*`))` or `(strcol(`*expression*`))`	`using 1:(stringcolumn($3+1))`	The argument to the `stringcolumn()` function should be an expression that evaluates to an integer. The `stringcolumn()` function returns the value of the current record for the desired column as a string value.
`(stringcolumn(`*string*`))` or `(strcol(`*string*`))`	`using 1:(stringcolumn("ID"))`	If the argument to the `stringcolumn()` function is a string, it's interpreted as the name of a column (as given in the first non-comment line in the data file). The `stringcolumn()` function returns the value of the current record for the desired column as a string value.
`(timecolumn(`*expression, formatstring*`))`	`using 1:(timecolumn(3, "%Y-%m-%d"))`	Parses a column entry as a timestamp and returns the corresponding Unix epoch second as numeric value. The first argument must evaluate to an integer, identifying a column. The second argument specifies a time format, according to table 8.4 or 8.5.
`(valid(`*expression*`))`	`using 1:(valid(2)?$2:$3)`	The argument to the `valid()` function *must* be an expression that evaluates to an integer (not a string). The `valid()` function returns 1 if the current record for the identified column is a valid number or zero otherwise.

4.4.1 Accessing columns by position or name

Columns are usually accessed by their position in the file (counting left to right, starting at one):

```
plot "data" using 1:2
```

This is convenient for files with only a few columns, but it can also work well for truly large files, because it's easy to iterate over numeric column specifiers with gnuplot's new *inline* looping feature. The following command plots columns 2 through 24 against the first column in a single plot (more on this kind of looping construct in section 5.4; we'll treat loops in greater detail in chapter 11):

```
plot for [j=2:24] "data" u 1:j
```

You can also specify a column by *name* if the data file contains a set of column labels in the first row of the file (or the data set, if the file contains multiple data sets addressed

using `index`). For example, for the file shown in listing 4.7, the following commands are equivalent:

```
plot "grains" skip 1 using 1:2
plot "grains" using 1:"Wheat"
plot "grains" using "Year":"Wheat"
```

Listing 4.7 Column names in a data file (file: grains)

```
Year    Wheat   Barley  Rye         ◁────── Not a comment line!
1990    8       6       14
1991    10      5       12
1992    10      7       15
1993    11      5       13
1994    9       6       12
```

Observe that the line containing the labels *must not be a comment line*. Furthermore, if any of the entries in the `using` phrase are strings, the entire first line is interpreted as labels and isn't included in the plot.

4.4.2 *Pseudocolumns*

Gnuplot supplies three pseudocolumns for each file it reads. They're numbered `0`, `-1`, and `-2`.

The pseudocolumn `0` contains the line number in the current file (or data set), starting at zero, without counting any comment, label, or skipped lines. You can also access this column in inline transformations using `$0`. (This counter is reset to zero when it encounters the double blank line that separates data sets.)

The pseudocolumn `-1` contains the line number, starting at zero, and is reset by a single blank line. (This is relevant if the data file is in grid format—see appendix C for details.)

The pseudocolumn `-2` contains the index of the current data set within the data file. When a double blank line is encountered in the file, the line number (corresponding to the value of pseudocolumn `0`) resets to zero, and the index is incremented.

You can use these pseudocolumns, for instance, like this:

```
plot "data" using 0:1        ◁────── Plots the first column against the line number
plot "data" using 1:-2        ◁────── Plots the data set index against the first column
```

4.4.3 *Column-access functions*

You can use the `column()` function whenever an expression has become too complicated for the `using` syntax, or in contexts where the `$` shorthand isn't available. The function evaluates its argument to a column identifier and returns the value of the identified column as a numeric value. (There is also a `stringcolumn()` [or `strcol()`] function that returns the column value as a string. See the next chapter for more information on strings.)

The argument to `column()` (or `stringcolumn()`) can be either numeric or a string. A numeric argument is evaluated into a column number. A string argument is only suitable if the file contains a non-comment header line with column identifiers (as in listing 4.7): if that's the case, the argument should evaluate into one of the column names. An error results if no column with the supplied identifier can be found.

The `column()` function and pseudocolumns work well together. For instance, this example plots all values from a file but adds a constant vertical offset of 1.5 to values from different data sets (to separate curves from different data sets from each other, so that you can distinguish them more easily):

```
plot "data" using 1:($2 + 1.5*column(-2)) with lines
```

The following example (found in the standard reference documentation) plots each data set in a different color (see chapter 9 to learn more about `linecolor variable`):

```
plot "data" u 1:2:(column(-2)) linecolor variable
```

FURTHER COLUMN-ACCESS FUNCTIONS

If a column contains date and time values, then the `timecolumn()` function may be useful. It takes two arguments: the first must be an integer identifying a column (it can't be a string!), and the second must be a string specifying the format of the time-stamp in the data file. The function returns the Unix epoch seconds corresponding to the parsed value as a number. This value may, for example, be formatted into a text label using the `strftime()` function.

The format specifiers are based on those used by the `strftime()` and `strptime()` functions from the C standard library. We'll discuss the handling of date and time information in section 8.4.2 in the context of time-series plotting; here we present only a brief preview. Imagine a file containing dates and values like this:

```
2014-12-01   3
2014-12-02   5
2014-12-05   1
```

The following command converts the dates in the first column to their corresponding epoch seconds and uses them as horizontal positions:

```
plot "data" u (timecolumn(1, "%Y-%m-%d")):2 w lp
```

One other function can be useful for inspecting column values: `valid()`. It returns true only if the value of the column with column number x is a valid number. You can use this function to test values from messy files and only plot those that are valid.

4.5   *Pseudofiles*

Typically, the data rendered by the `plot` command is read from a file, but in some situations it makes sense to accept data from a different source. Gnuplot includes a

number of special filenames that you can read from as if they were proper files, although they don't exist on any disk.

As of gnuplot 5, data can be embedded in command files and accessed through a variable-like identifier called a *heredoc*. Data can also be read from the command line interactively and stored in a heredoc—this can be useful when you want to add a few data points to an existing graph.

On platforms that support pipes, gnuplot can read data from standard input—this is particularly useful when you're scripting gnuplot to run in batch mode. When they're available, you can also send output to another process via pipes.

Finally, two special filenames let you apply the full flexibility of the using facility to function plots. Table 4.3 lists all these special filenames and describes their meaning.

Table 4.3   Pseudofiles

Pseudofile	Description
`""` (empty filename)	Reuses the most recently encountered filename.
`"-"`	Reads from standard input. In an interactive session, reads from the command window. (Terminate input with the EOF character or `e`.)
`"+"`	Generates `samples` equally spaced x values covering the entire current plot range.
`"++"`	Generates spaced x and y values covering the entire current plot range. (This is the two-dimensional equivalent to `"+"`.) The number of points in the x direction is controlled by `set samples`, and the number of points in the y direction is controlled by `set isosamples`.
`"< name"`	Reads from a subprocess via pipe.[a]
`"| name"`	Writes to a subprocess via pipe.[a]
`$heredoc`	Reads from a heredoc (no quotes around the name).

[a] Only available on Unix-like platforms.

4.5.1   *Reading data from standard input*

When given the special filename – (as in: plot `"-"` u 1:2), gnuplot attempts to read data from standard input, which in an interactive session is the command window. After you run this plot command, gnuplot shows a prompt at which you can type data. Finish each line by pressing Enter. Gnuplot will keep prompting for data until either an end-of-file (EOF) character (typically Ctrl-d) is encountered or the character e is entered on a line by itself.

You can even read data from standard input multiple times within the same plot command: plot `'-'`, `'-'` reads data until an EOF character is encountered and then expects to read *more* data (for the second "file") until it finds a second EOF character. Of course, the data entered at a prompt this way can have multiple columns, from

which you can select some with `using`; all the other features of the `plot` command can be used as well.

Although this feature can be used interactively, it's mostly intended for situations where gnuplot is used in batch mode as part of larger scripts (we'll talk more about that in chapter 11). When used interactively, this feature quickly becomes inconvenient, because (as explained earlier) gnuplot doesn't maintain data sets in memory and therefore all data has to be manually reentered every single time you want to plot or replot the graph.

4.5.2 *Heredocs*

Gnuplot 5 contains a new feature that makes it possible to embed data in a *command* file (as opposed to a data file) or to enter data in the command window and *store the data in a variable for the duration of the gnuplot session*. This is somewhat of a radical departure, because traditionally gnuplot never maintained data in memory—data was read from file every time the `plot` command was executed. This new feature is referred to as *heredocs* because its syntax is modeled after the eponymous feature in Perl and in Unix shells. (The gnuplot standard reference also calls them *data blocks*, but this term is already used in other contexts. I therefore use the unambiguous *heredoc* terminology for this feature, which seems justified given that these data structures are intended to be populated through a corresponding facility.)

DEFINING AND USING HEREDOCS

You define a heredoc by preceding a variable identifier with a `$` (dollar sign), followed by the `<<` redirection operator and an arbitrary sequence of characters that will mark the end of the data section:

```
$d << EOD
```

Now follows the actual data section, just as if it was a data file. The data is terminated by the end-of-data indicator (on a line by itself—the indicator must match exactly). The entire affair may look like the following listing.

Listing 4.8   Defining a heredoc

```
$d << EOD
1     0.5
2     0.75
3     0.99
EOD
```

You can now use the heredoc as you'd use a file. In particular, you can plot its contents (note that there are no quotes around the heredoc identifier):

```
plot $d using 1:2
```

Listing 4.8 could be part of a command file; in this case, it could be loaded (with `load`). Alternatively, it could be entered interactively during a gnuplot session.

Several gnuplot commands that usually write to a file can write to a heredoc, instead; both `set print` and `set table` accept a heredoc identifier (see section 5.3). This makes it possible to have gnuplot commands "write" to a heredoc from within a gnuplot session.

No specific commands exist to populate a heredoc from a generic data file or to persist the contents of a heredoc to disk. In contrast to other information that exists in the session (such as gnuplot variables), the contents of a heredoc are *not* saved by the `save` command. Using `set print` and `set table`, you can circumvent these hurdles (we'll come back to this in section 5.3).

A heredoc, once defined, occupies memory. To release these resources, use the `undefine` command: `undefine $d`. Using `undefine $*` drops all currently existing heredocs.

Understanding heredocs

Heredocs are a new and somewhat controversial feature, and some of their aspects may have changed by the time you read this. Heredocs break with several of gnuplot's longstanding and fundamental design principles. It's important to understand how they came about.

Heredocs arose from a desire to embed data in command files, in order to enable command files that are self-contained and don't require a separate data file. But because traditionally the contents of gnuplot command files can be issued without change at the gnuplot command prompt, the syntax chosen for gnuplot's heredocs can be issued interactively, as well. This gives the impression that heredocs are akin to session variables, but that isn't how they're *meant* to be used.

The result is strangely halfhearted and contradictory. Heredocs live in the session and occupy memory, just like gnuplot variables—yet they aren't persisted with `save`. Heredocs can stand in for files in the `plot` and `splot` commands, in `set print` and `set table`—yet they aren't intended to be populated from generic data files.

Heredocs break with the tradition that gnuplot doesn't maintain any memory of the data: data had to be read from file every time it was used so that gnuplot was stateless (with regard to data). With heredocs, data lives in the gnuplot session, giving the impression of a data set or data frame—yet there exist no operations to manipulate heredocs (in addition to the inability to populate them from a generic data file).

Heredocs are intended to be defined in command files, where they require a special (albeit simple) syntax, thus breaking with the principle that gnuplot data files are plain, unstructured, and generic text files.

Ultimately, the problem is that the heredoc design is too flexible and feature-rich for the intended purpose. What was intended was a static (read-only) data block as part of a command file—similar to the `DATA` filehandle in Perl. But the ability to define heredocs at runtime and to maintain them in the session, independent of file access, gives the appearance of a general-purpose data structure and invites to be used as such. But because heredocs were never intended as general-purpose data sets, some of the functionality you might expect is missing.

4.5.3   Reading data from a subprocess

On platforms that support input/output redirection, gnuplot can read from a subprocess via a pipe. Let's say that a.out is a program that prints data suitable for gnuplot to standard output. The following command runs a.out and captures and plots its output without accessing the file system:

```
plot "< a.out" using 1:2
```

The < character must always be the first character of the filename, even when reading from an entire pipeline. For instance, if the output of the first program a.out is piped through a second program b.out, the gnuplot command is `plot "< a.out | b.out"`.

You need to watch out for several things when reading from a subprocess like this. First, keep in mind data might be buffered by the operating system, so that gnuplot doesn't plot anything until the buffer is flushed or the process has finished (which is guaranteed to flush the buffer), at which point gnuplot plots all the data received at once. If you want to monitor a live stream with data trickling in over time, you need to use one of the methods from section 11.6. Furthermore, if you attempt to read from the same subprocess twice—for instance, `plot "< a.out" u 1:2, "" u 1:3`—gnuplot runs the program *twice* (and you'll have to wait until both instances complete). Repeating a `plot` command (or using the `replot` command) starts the subprocesses anew.

For these reasons, it's usually a better idea to write the process output to a file, which can then be plotted at leisure and as often as necessary. Alternatively, you might consider running gnuplot itself as a subprocess and piping data to it—we'll come back to this idea in chapter 11.

4.5.4   Writing to a pipe

If pipes are available, gnuplot can also write to them. Because gnuplot's output consists primarily of graphs, this feature is mostly of interest for post-processing image files. A single example will suffice.

Among gnuplot's terminals based on the cairo library (see section 10.3), there isn't one that can generate files in GIF format. The following sequence of commands uses gnuplot's `pngcairo` terminal to create a graph in the PNG format, which is then piped through the `convert` utility from ImageMagick (www.imagemagick.org) to create the desired GIF:

```
set terminal pngcairo
set output "| convert - graph.gif"
plot sin(x)
set output
```

The pipe character sends gnuplot's output to a pipe; the dash instructs `convert` to read from standard input. The final `set output` is necessary to flush gnuplot's output channel. (See chapter 10 for additional information.)

4.5.5 *Generating data*

When gnuplot plots a function (as in `plot sin(x)`), gnuplot evaluates the function on a set of equally spaced points, which are evenly distributed over the entire plot range. The number of points is controlled by the `set samples` option (see section 3.1), and the plot range is either given (in square brackets) as part of the `plot` command (see chapter 2) or specified using `set xrange` (see chapter 8).

Reading from the special filename `"+"` is like reading from a file that contains exactly those x positions at which the function would be evaluated, in a single column. In other words, the following two commands are equivalent:

```
plot sin(x)
plot "+" using 1:(sin($1)) w l
```

In the second command, you must specify the `with lines` style explicitly, because—as far as gnuplot is concerned—you're plotting *data* (not a function), and hence the default data style applies (see chapter 9).

The special file `"++"` is the two-dimensional equivalent of `"+"`. It generates regularly spaced positions on a two-dimensional grid as used by the `splot` command. (See appendix C for more information about `splot`.)

You probably won't need the `"+"` and `"++"` pseudofiles often—they're special-purpose tools for handling certain edge cases. Sections 6.3.4, 8.2.2, 9.1.5, and F.5.1 discuss potential applications.

4.6 *Metadata in data files*

Gnuplot can read a variety of (mostly textual) information from a data file in addition to the data points being plotted. We haven't discussed gnuplot's string-handling facilities in detail yet (we'll do so in the next chapter), nor have we introduced the context in which the need for metadata arises (we'll do so in chapters 7, on decorations that can be placed on a plot, and 8, on axes), so this section is mostly a teaser.

Strings can be read from a data file and placed directly onto a plot using the `with labels` style (see section 5.1.3 in the next chapter). With the `ticslabels()` family of functions (see chapter 8), you can use strings read from a file as labels on the axes of a plot (see section 8.3.5 and the examples in section 13.4). Special rules apply if the data in the file represents calendar dates and times (see the section on time series in 8.4.2). Finally, you can use the values of the first line in an input file (the column labels) as entries in the legend or *key* of a plot (see section 7.4.5). Table 4.4 summarizes all the metadata that can be found in a data file.

Table 4.4   Metadata in data files

	Section	Description
`#`	4.3.2	If found at the beginning of a line, treats the entire line as a comment.
`plot skip` *n*	4.1.1	Skips the first *n* lines.
`index` *name*	4.2.1	Identifies a data set in a file by a unique identifier in the comment line immediately preceding the data block.
Named column	4.4.1	Identifies a column by a name, which is found in the first non-comment line of the file.
Key entry explanations	7.4.5	Reads explanations to be used in the graph's key from the first non-comment line of the file.
`with labels`	6.3.5	Reads text labels from the data file and places them on the graph.
`xtics(`*col*`)`	8.3.5	Reads tic labels from column *col* in the data file.
`set xdata time`	8.4.2	In time-series mode, reads and parses appropriately formatted strings as timestamps.

4.7   *Other file formats*

In passing, let's mention some other file formats used by gnuplot that you haven't seen. It's possible to parse more complicated record formats than the ones we've discussed so far by passing a format string, which describes the format of each record, to `using`. The format string must be compatible with the `scanf()` family of functions, familiar from the standard C library. Check the standard gnuplot reference documentation if you believe this is relevant to you; but given the well-known fussiness of `scanf()`, this is rarely the best path forward. If a file has a format that can't be parsed normally by gnuplot, it's usually a better idea to convert it to a gnuplot-compatible format using a small conversion program in Perl, awk, or a similar tool.

Date and time strings are handled in a special way: don't attempt to parse them using a `scanf()`-like format string. Use the `timecolumn()` function (see table 4.2) or the special commands described in section 8.4, instead, to parse and process such data.

> **TIP**   Time and date information constitutes a special case. Use gnuplot's dedicated facilities for handling such data (see section 8.4.2).

In addition to the regular, column-oriented, tabular file format you've seen up to this point, gnuplot can also handle data on a two-dimensional grid. There are two formats to choose from: one that contains the data values together with the coordinates of the grid points, and a packed "matrix" format. We'll discuss them in appendix C on multi-dimensional plots.

Finally, gnuplot can read certain binary packed file formats. Again, if this is relevant to you, I suggest the standard gnuplot reference documentation. Unless you have particular needs, I recommend that you stick with text files.

4.8 Summary

Gnuplot is all about graphing data, and in this chapter we covered how to get data into gnuplot. You learned about data files and formats. Gnuplot reads data from *text* files, which are portable, easy to generate, and easy to modify.

You learned how to keep multiple data sets in a single data file and select from them with `index`, and how to select a subset of records with `every`. You also learned different ways to identify and access individual columns in a data file.

We discussed pseudofiles, which are data sources that don't live on the disk, and we also mentioned that gnuplot can read data from another process (if it's installed on a platform that permits it). Finally, we took a brief look at other information you can find in a data file and mentioned alternative file formats in addition to the whitespace-separated columns we've concerned ourselves with exclusively until now.

This concludes what you need to know to get started doing data analysis using gnuplot. In the next chapter, we'll turn away from gnuplot's core functionality of dealing with data and generating plots, and instead talk about a host of useful features that make your work with gnuplot easier.

<div align="right">

Practical matters:
strings, loops, and history

</div>

> **This chapter covers**
> - Strings and textual data
> - String operations and macros
> - Inline loops
> - Internal variables and the `stats` command
> - Command history

In the previous chapters, we looked at ways to manipulate and visualize data using gnuplot, which is the core task that gnuplot was designed for. In this chapter, we'll sidestep the presentation of gnuplot's graphic capabilities and introduce some foundational features that will be useful later.

First you'll learn about strings and textual information. Although gnuplot is primarily a tool for creating *graphics*, textual labels are nevertheless very important: it's only through information expressed in a human language, which tells the user what is displayed, that a graph receives meaning and *semantics*. In addition, strings are used in and by gnuplot as a general-purpose data structure—that's another reason to pay attention to them.

We'll then introduce some facilities that make it easier to work with gnuplot: internal variables allow you to inspect gnuplot's internal state and use this information to adjust options accordingly; and inline loops can help significantly reduce the number of keystrokes required when your commands contain repeated operations (for instance, when dealing with a large number of files). As a last topic, we'll introduce gnuplot's command history.

Although they don't produce graphs themselves, the features introduced in this chapter all help you be more productive with gnuplot. The effort to learn them will pay off!

5.1   Strings

In this section, we'll first look at the way strings are defined and then introduce operations that gnuplot can perform on strings, before moving on to some example applications that demonstrate what you just learned. This section concludes with a lengthy digression on Unicode and UTF-8: this may sound boring, but it is, in fact, extremely good news, because it greatly simplifies the handling of mathematical symbols and other special characters.

Although it may not be apparent yet, strings are tremendously important in gnuplot, because they serve three different purposes:

- *Strings as descriptions*—Strings have an obvious use as textual descriptions. This is no small matter. Only through textual description can you convey *semantic* information: what the data *means* that's shown in the graph. A graph without labels is mute.
- *Strings as data structures*—Strings are gnuplot's only complex data structure and are therefore used (or abused) as a general-purpose container type. Whitespace-separated strings of tokens frequently stand in for proper arrays, and functions exist to extract an individual token by its index (mimicking the access of an array element). In the absence of proper alternatives, strings are the only iterable data structure required by loops.
- *Strings as macros and functions*—Strings containing gnuplot commands can be executed at runtime. In the absence of subroutines, strings are one way to bundle several commands into a single callable unit. Moreover, because strings can be built up at runtime, it's possible to use them in a fashion resembling closures in certain dynamic programming languages.

We'll pursue some of the more creative uses for strings further in chapters 11 and 12. In this chapter, we'll lay the foundations by treating strings as holders of textual information. We'll also give a preview of their array-like application when we introduce inline loops in section 5.4.

5.1.1   *Quotes*

You can assign a string constant to a gnuplot variable. String constants must be enclosed in quotes—either single or double. The difference is that escaped control characters (such as \n) are interpreted as control characters within *double*-quoted strings but are treated literally (that is, not interpreted) inside *single*-quoted strings (reminiscent of the behavior of Perl strings, for instance).

A double-quoted string can contain single quotes; to obtain a double quote inside a double-quoted string, it must be escaped with a preceding backslash. Within single-quoted strings (which don't recognize the backslash as an escape operator), you can get a single quote by doubling it:

```
a = 'This is a string.'
b = "First Line\nSecond Line."

c = "Double quote\" escaped."
d = 'Single quote'' escaped.'
```

> **TIP**   Backslash-escaped control characters are interpreted as control characters inside *double*-quoted strings but are treated literally inside *single*-quoted strings.

5.1.2   *String operations*

Strings can be assigned to variables just as numbers can. Strings are converted to numbers silently, if possible. But only integers (not floating-point numbers) are promoted to strings:

```
x = '3.14'              ◁——————  String
y = 2 + x               ◁——————  Promotes to number: now y = 5.14

a = 4                   ◁——————  Number (integer!)
b = 'foo' . a           ◁——————  Promotes to string: now b = 'foo4'

z = 0.0 + x             ◁——————  Forces evaluation as number
c = '' . z              ◁——————  Forces evaluation as string
```

> **TIP**   Depending on the context, gnuplot automatically interprets the value of a parameter as number or as string. If necessary, you can *coerce* evaluation as number by adding 0.0 or as string by concatenating an empty string.

Three operators and a handful of functions act on strings. The first operator is the concatenation or *dot* operator:

```
a = 'baz'
b = "foo" . 'bar'
c = b . a               ◁——————  c is now "foobarbaz".
```

The other two operators are comparison operators for use in conditionals: eq (equal) and ne (not equal). The first returns true if both of its arguments are equal, and the second returns true if they differ:

```
a = 'foo'
b = 'bar'

c = a eq 'foo' ? 'equal' : 'different'
```

Strings can be indexed like arrays, using a syntax similar to the syntax used to indicate plot ranges:

```
a = "Gnuplot"
b = a[2:4]              ◁─────── b is now "nup".
c = a[4:]               ◁─────── c is now "plot".
```

The first character in the string has index 1 (not 0!). Also notice that *both* limits are inclusive. If you leave either the beginning or the end of the substring empty, it will default to the beginning or the end of the entire string, as shown.

> **TIP** The first character in a string has index 1 (not 0). Index ranges for substrings are interpreted inclusively at both ends.

A small set of functions is available to analyze and parse strings (see table 5.1). You can also define your own string functions, in the same way you define functions that operate on numbers:

```
head( s ) = s[1:3]
a = head( "January" )          ◁─────── a is now "Jan".
```

Table 5.1 String functions

Function	Description
strlen("*str*")	Takes a string and returns the number of characters in the string.
substr("*str*", *i*, *j*)	Takes a string and two integers. Returns the substring beginning at *i* and ending at *j*, *inclusive*. Equivalent to str[i:j].
strstrt("*str*", "*key*")	Takes two strings. Returns the index of the first character of the string *key* in the string *str*, or zero if not found.
words("*str*")	Takes a string. Strips the leading and trailing whitespace and then breaks the string into tokens on whitespace. Returns the number of tokens found. Newlines and tabs are interpreted as whitespace only if the string was double-quoted.
word("*str*", *n*)	Takes a string and an integer. Strips the leading and trailing whitespace and then breaks the string into tokens on whitespace. Returns the *n*th token found. (Tokens are counted starting at 1, not at 0.) Newlines and tabs are interpreted as whitespace only if the string was double-quoted.
sprintf("*format*", ...)	Takes a format string and an arbitrary number of arguments. Returns a formatted string. Equivalent to the sprintf() function in the C standard library.

Table 5.1   String functions *(continued)*

Function	Description
`gprintf( "format", ... )`	Takes a format string and a *single* argument to format. Returns a formatted string. Similar to `sprintf()` but uses gnuplot's format specifiers (compare table 8.2).
`value( "string" )`	Takes the name of a gnuplot variable as a string and returns the current value of that variable. (This is useful if the name of the variable is determined or constructed at runtime. See the examples in section 11.1.2.)
`system( "string" )`	Takes a shell command as string and executes it in a subshell. Returns the output of the shell command. (You can find more details in chapter 11.)

5.1.3   *Worked example: plotting the Unix password file*

As a crazy example of what's possible, let's plot a typical Unix password file with gnuplot! Listing 5.1 shows the file. (For non-Unix users: each line in the file describes a user. Each line consists of several fields, separated by colons. The first field is the username, the third field is a numeric user ID, and the fifth field is a textual description of the user. The other fields are of no relevance here.)

Listing 5.1   Data for figure 5.1 (file: passwd)

```
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
ftp:x:40:49:FTP account:/srv/ftp:/bin/bash
games:x:12:100:Games account:/var/games:/bin/bash
ldap:x:76:70:User for OpenLDAP:/var/lib/ldap:/bin/bash
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
mail:x:8:12:Mailer daemon:/var/spool/clientmqueue:/bin/false
man:x:13:62:Manual pages viewer:/var/cache/man:/bin/bash
mysql:x:60:108:MySQL database admin:/var/lib/mysql:/bin/false
news:x:9:13:News system:/etc/news:/bin/bash
ntp:x:74:103:NTP daemon:/var/lib/ntp:/bin/false
postfix:x:51:51:Postfix Daemon:/var/spool/postfix:/bin/false
sshd:x:71:65:SSH daemon:/var/lib/sshd:/bin/false
uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash
wwwrun:x:30:8:WWW daemon apache:/var/lib/wwwrun:/bin/false
```

To plot this file, you need to set the field separator to be the colon (`:`); you can then plot the file using the `with labels` style (see chapter 7). Just for fun, I'll also make the letter `'m'` the comment character. Verify how records starting with an `'m'` don't appear in the graph. (See section 4.3 for details of the file-format specifications.)

In the plot (see listing 5.2; the resulting graph is shown in figure 5.1), the numeric user ID is used as the x coordinate and the line number in the file as the y coordinate. The label, printed at the resulting position, consists of each user's login name, stacked (by virtue of a newline character) on top of the textual description of the user.

Listing 5.2 Commands for figure 5.1 using data from listing 5.1 (file: passwd.gp)

```
set datafile separator ':'
set datafile commentschar "m"
plot [-10:90][0:15] "passwd"
➥     u 3:($0+2):( stringcolumn(1) . "\n" . stringcolumn(5) ) w labels
```
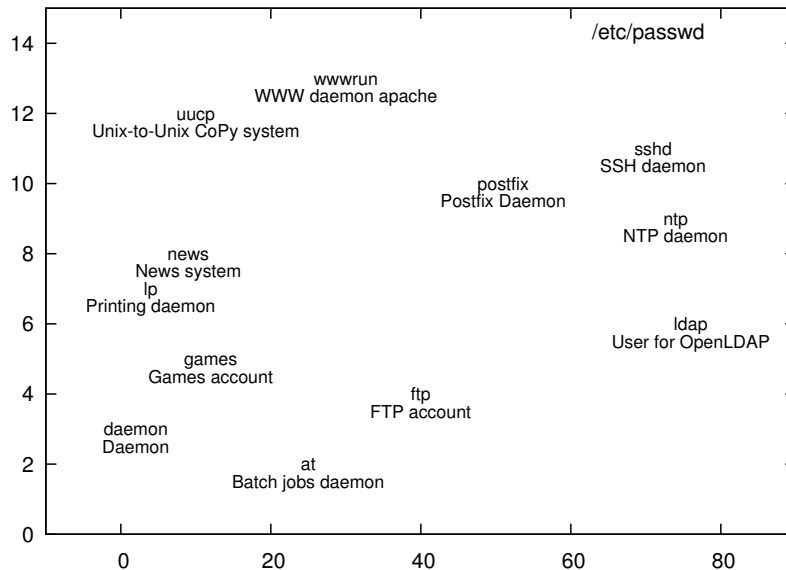

Figure 5.1   Demonstrating string functions and the `with labels` plot style: plotting the Unix password file with gnuplot. See listing 5.1 for the data and listing 5.2 for the commands.

Unicode and UTF-8

Gnuplot 5 can handle Unicode strings that are encoded using UTF-8. Although this may sound like the typical configuration and localization nightmare, it's actually really good news—once you understand why.

Unicode

Unicode is based on the concept of a *character*. A character is a logical abstraction, such as "the first capital letter in the Latin alphabet" (that is: the letter *A*), or the "question mark" character (?). In Unicode, *every* character is identified by a *unique* integer value, known as that character's *code point*. Unicode defines over 1 million code points, which are usually specified as U+xxxx, where each x stands for a hexadecimal digit. The capital letter *A*, for instance, can be found at code point U+0041.[a]

---

[a] You can find tables of Unicode characters and their code points on the Unicode website: www.unicode.org/charts.

(continued)

Separate from the notion of the character is the concept of the *glyph*. A glyph is the geometric shape or symbol used to *represent* a character visually. The same (logical) character can be represented by a different glyph, depending on the typeface or font.[b]

Here comes the first possible stumble: not every font provides a glyph for every possible code point—in fact, most fonts map only a small subset of characters. That shouldn't be surprising: a calligraphic font designed to print wedding invitations (in English!) need not include glyphs for mathematical symbols or for characters from Asian languages. Hence it's essential to use a font that provides glyphs for all the characters you want to display. Fortunately, several open source fonts exist that provide a wide variety of glyphs that may be useful in scientific or technical contexts. I recommend Nimbus Sans L, which is designed to be similar to Helvetica. A good font manager is helpful for browsing available fonts and glyphs—if you don't have a preference yet, you might want to check out Fontmatrix.[c] (I'll have more to say about font name resolution in gnuplot in chapter 10.)

It's important to remember that a Unicode code point *uniquely* identifies a character and that the only remaining task is to find a font that provides a glyph for that character. All this is very different from the old way of doing things, where the number of characters was limited to 255 and different *encodings* were used to map these 255 characters to different glyphs, depending on the language or symbols you wanted to use. In the first edition of this book, I explained how to switch from the main text font (such as Helvetica) to the Symbol font to get access to Greek letters and a (limited) choice of mathematical symbols. Although changing fonts is still possible, there is much less need for it today, because a single font can provide not only Latin letters but also a wide range of mathematical and geometric symbols, as well as letters from non-Latin alphabets.

UTF-8

I haven't yet explained how a Unicode string is represented internally. Fortunately, you (for the most part) don't need to know.

Clearly, one byte per character isn't enough to represent all the 1.1 million possible Unicode characters, and several encoding schemes have been developed to map Unicode characters to "bits and bytes." The good news is that one encoding came out as the clear winner: UTF-8.[d] It isn't necessary to know anything about the (very clever) details of UTF-8, as long as you stick with UTF-8 always and exclusively.

---

[b] Technically, the typeface is the abstract design, whereas the font is an implementation of that design. In the times of hot-lead typesetting, that distinction was a lot more important than it is today: you had to be in physical possession of the metal pieces that made up a "font" to be able to print something.

[c] Although its development seems to have been stagnant for several years, the currently available version works well and is available for a large number of Linux distributions.

[d] It's amusing to note that UTF-8 was designed, quite literally, on the back of a napkin: see Rob Pike's "UTF-8 History" at http://mng.bz/TuD3.

If you present your strings to gnuplot in UTF-8 (and use a font that provides glyphs for all characters in your strings), things will "just work."

This brings us to the second possible stumble: how to generate arbitrary strings encoded in UTF-8. The most common case (in my experience) is that I know the code point for a character I want to use (like the "white smiling face" character found at code point `U+263A`—but gnuplot isn't prepared to handle code points directly. I must translate the code point to a UTF-8 encoded string first.

There are several ways to do that. Some combination of keys on your keyboard may generate the desired character in the interactive terminal already. Your favorite editor may allow input of characters by code point or Unicode name:[e] you can then save the string to file and let gnuplot read it, or cut-and-paste the string from the editor window to the gnuplot session (yes, this works). Finally, current programming languages generally can handle both Unicode and UTF-8. In Python, for example, the following works:

```
python>>> print u'\u263a'.encode( 'utf-8' )        <———— Unicode literal
python>>> print unichr( 0x263a ).encode( 'utf-8' )  <———— Function call
```

Choosing an encoding

To take full advantage of Unicode, the encoding must be set to UTF-8:

```
set encoding utf8
```

This isn't necessarily selected by default, so you probably want to include this command in your gnuplot initialization file (see section 12.5.1). Not all terminals support all encodings, but contemporary terminals will usually work with both Unicode and UTF-8.

---

[e] In Emacs, for instance, typing `C-x 8 RET`, followed by the code point in hex or the Unicode name, will work—note that tab-completion of the name is available. You may also want to check out the TeX input method: you enable it using `C-x RET C-\`, followed by `TeX`. You can now input Greek letters and other symbols as in LaTeX: for instance, typing `\lambda` will produce the corresponding Greek letter.

TIP All strings in gnuplot (and the libraries it uses) can handle Unicode that's encoded using UTF-8. Together with an appropriate font, this should take much fear out of handling special characters in gnuplot. For best results, make sure to set the encoding explicitly to UTF-8 using `set encoding utf8`.

5.2 String expressions and string macros

The string-handling capabilities just introduced make it possible to construct command lines programmatically. Specifically,

- Wherever a command expects a string, a string expression can be substituted.
- The `eval` command takes a fixed string and executes it as if its contents were entered at the command line.

- By prefixing a variable containing a string with the @ character, you can create macros that can be used even in places where gnuplot doesn't expect to find a string expression.

Let's look at these options in a bit more detail.

5.2.1   *String expressions in commands*

As just pointed out, anywhere a command expects a string, you can substitute a *string expression*. For instance, this will work just fine:

```
file = "data.txt"        ◁───── Assigns the string "data.txt" to a variable
desc = "My Data"         ◁───── Assigns the description to a variable

plot file title desc     ◁───── Equivalent to "plot "data.txt" title "My Data""
```

You can also use functions that return a string. Here, a function provides the file extension for you:

```
f( x ) = x . '.txt'
plot f( 'data' )         ◁───── Plots the contents of "data.txt"
```

The example is whimsical, of course, but it's easy to come up with more useful applications. The string functions (see table 5.1) are often helpful in this context.

5.2.2   *Executing a string with eval*

The `eval` command takes a string (or string expression) and executes it as if its contents had been issued at the command prompt. This is occasionally useful when you want to use string functions to build up a command line dynamically at runtime.[1]

Here's an example. Exporting a graph to a file requires several steps (choosing the terminal and filename, regenerating the graph, and restoring the original state). You can facilitate this process by using a function that builds up a string containing *all* required steps:

```
export( file, terminal ) = sprintf(
➥    "set t push; set t %s; set o '%s'; replot; set o; set t pop",
➥    terminal, file )
```

When supplied with the desired filename and terminal type, this function returns a string consisting of all the required gnuplot commands to export the most recent graph. This string can be executed using `eval`. In fact, the function can be invoked simultaneously with the `eval` command:

```
eval export( "graph.pdf", "pdfcairo" )
```

---

[1]   At a deeper level, this reflects a fundamental limitation of gnuplot's functions. Gnuplot functions must be defined in terms of *expressions*. It isn't possible to define a function as a sequence of commands. Building up a complicated string and executing it is a way to group several commands together.

This works and is probably the most elegant way to export a graph to file. (The `export()` function as defined here uses the pseudoterminals `push` to store the most recently used terminal settings and `pop` to restore the original terminal. See section 10.1.2 for details.)

It isn't necessary for the string to be built up dynamically; you can use the same process to create convenient shorthands. For example, in appendix D, you'll learn to define color gradients. The definition of palettes is a bit wordy, so it may be convenient to define appropriately named string variables (possibly in your gnuplot configuration file—see chapter 12) and then execute them:

```
redblue = "set palette defined( 0 'blue', 1 'white', 2 'red' )"   <——  Define

eval redblue                 <——  Execute
```

5.2.3 String macros inside commands

When you want to replace part of a command line where gnuplot doesn't expect a string, you can't use a string variable (or string expression) directly. Instead, you must resort to a string *macro*.[2] String macros let you insert the contents of string variables at arbitrary positions in a gnuplot command. Here's an example:

```
cols  = "using 1:3"          <——  Assigns a string variable
style = "with lines"            <——  Assigns another one

plot "data" @cols @style            <——  Equivalent to "plot "data" using 1:3 w lines"
```

The symbol `@` in front of a variable name indicates that the value of the variable should be interpolated into the command line at the current location. The command line is evaluated after all string macros have been expanded.

Macros aren't evaluated within quotes (either single or double), but you can usually achieve the desired effect through simple string manipulations. This example shows both situations:

```
tool = "(made with gnuplot)"              <——  String variable

set title "My plot @tool"           <——  Macro expansion doesn't work …
set title "My plot " . tool            <——  … but string concatenation is fine!
```

Only string *variables* are evaluated when prefixed with an `@`—more general string *expressions* aren't. The return value from a string function must therefore always be assigned to a variable before it can be evaluated. Also keep in mind that string macros are evaluated only once: when the command line containing the macro is parsed. You also can't define and use a macro in the same command line.

---

[2] At the time of this writing, the future of string macros in gnuplot is a bit uncertain. You can run `show version long` and look for an entry of the form `+MACROS` to see whether macros are enabled in your gnuplot installation. In previous gnuplot versions, macro processing had to be explicitly enabled using `set macro`; this is now the default.

5.3   Generating textual output

Gnuplot creates graphs—after all, that's the whole point! Nevertheless, sometimes it's useful to have gnuplot create textual output. For example, you may want to export the results from gnuplot's spline-interpolation algorithm to a file so you can use them in another application. Or you may have applied an inline data transformation and want to get your hands on the resulting data.

Gnuplot has two facilities for generating text: the `print` command and the `set table` option. Although they may seem similar, they're intended for different uses:

- The `print` command evaluates expressions and displays their results as text.
- The `table` facility generates a textual description of the output of the `plot` or `splot` command. Instead of a graph of the data, the `table` style produces a representation of the data in textual format. In this way, it's akin to the `set terminal` command, which is used to choose the output format for the `plot` or `splot` command. (In previous versions of gnuplot, what is now the `set table` facility was exposed as a terminal.)

5.3.1   The print and set print commands

The `print` command evaluates one or more expressions (separated by commas) and prints them to the currently active printing channel—usually the screen:

```
print sin(1.5*pi)
print "The value of pi is: ", pi
```

This example demonstrates the two typical uses for the `print` command:

- Evaluating expressions (effectively, using gnuplot as a calculator)
- Inspecting the current value of variables and heredocs

THE SET PRINT OPTION

You can change the device to which `print` sends its output by using the `set print` option:

```
set print                            ◁── Prints to standard error
set print "-"                        ◁── Prints to standard output
set print "{str:filename}" [ append ]    ◁── Prints to file
set print $heredoc [append]          ◁── Prints to heredoc
```

By default, `print` sends its output to standard error (usually the current terminal if gnuplot is run interactively). But output can be redirected either to standard output (using the special filename -) or to a regular file (by giving the filename as argument to `set print`). Each invocation of `set print` creates a new file (clobbering any existing file of the same name), unless you specify the additional keyword `append`. The destination file may be a pseudofile (see section 4.5.1): on systems that support input/output redirection, you can specify a shell command as recipient of the output from `print` by starting the filename with the pipe symbol (|). You can also send the output from `print` to a heredoc so it can be used subsequently in the same gnuplot session.

Some gnuplot commands that generate textual output use the `set print` option internally to determine where to send their output. (An example is the `show palette palette` command; see section D.2.4.)

5.3.2 *The set table command and the with table style*

The `set table` facility gives you access to the data that makes up a graph *as text*. In other words, using `set table`, you can obtain the values of all points shown in a plot as numeric expressions.

The command to generate output as numbers, as opposed to graphics, is `set table`:

```
set table [ "{str:filename}" | $heredoc ]
```

As long as `set table` is active, all output is directed to the specified file (or to the current value of `set output` if no file was named). To return gnuplot to creating graphics, use `unset table`. (Don't forget!) You can also use a heredoc as the destination.

It's important to remember that `set table` doesn't generate output: it merely selects a channel to which the output is sent. Only the `plot`, `splot`, `replot`, and `refresh` commands generate output. One of them must be issued after `set table`; otherwise, the resulting file will be empty. You can use the `with table` style to suppress any graphics-related processing when doing so:

```
set table "output.txt"
plot "data" u 1:2 with table
unset table
```

Both x and y values are printed to the file as two separate columns. If the plot contains several curves, they're written to the file one after another, as separate data blocks. The data format for each column is equal to the format used for the tic labels of the corresponding axis (see section 8.3.3). The third column contains a marker indicating whether this data point was within the plot range or outside of it: `i` for "inside," `o` for "outside," or `u` if the point is undefined.

For data from a file, only the data points read from the input file are written to the output file. For functions, including Bézier and spline approximations, the number of data points depends on the value of the `set samples` option (see section 3.1).

You can also redirect the output to a heredoc using the `set table` facility. For example, if you want to use the results of one of gnuplot's smoothing algorithms (see section 3.5) in subsequent calculations, a heredoc is a convenient way to do so. In section F.6, I'll first use `plot ... smooth cumul` to calculate the cumulative distribution function, store the result in a heredoc, and later apply an inline transformation to generate a probability plot.

5.3.3   *Reading and writing heredocs*

No explicit commands exist to save the contents of a heredoc to a file or to populate a heredoc from a general data file, but both tasks can be accomplished through creative use of the techniques just discussed. Remember that the contents of a heredoc aren't included in the information persisted with save. Instead, you have to use print. Assuming that a heredoc called $d exists in the current session, these commands will persist its contents to a file called data:

```
set print "data"
print $d
unset print
```

To populate a heredoc from a data file, use set table:

```
set table $d
plot "data" with table
unset table
```

Writing heredocs with print isn't a problem, but reading heredocs from file using set table is subject to some limitations. First, no more than 14 columns can be read at a time. More important, the with table style is only suitable for numeric data. If you want to read strings into a heredoc, you must use the with labels style. But the with labels style requires exactly three columns in the using specification (typically x-position, y-position, and the string itself—see section 6.3.5). This means, among other things, that you can only read in a single string column per heredoc instance.

5.4   *Simplifying work with inline loops*

Recently, gnuplot has acquired features commonly associated with programming languages: loops and conditionals. We'll investigate them in detail in chapter 11. But the looping facility also has a shorthand version that is so simple and useful that I'll introduce it here.

   This shorthand is a form of for loop that is used to simplify plot and set commands that contain a repeated element. The loop is embedded in the plot or set command, which is why I call them *inline* loops. (This terminology isn't used in the standard gnuplot reference documentation.) A few examples will clarify.

5.4.1   *Loops over numbers*

Let's assume that you wish to plot the second, third, and fourth columns of a file against the first. You've learned how to do this:

```
plot "data" u 1:2 w l, "" u 1:3 w l, "" u 1:4 w l
```

For three columns, it isn't a big deal to type this out explicitly, but you can see how quickly it will become tedious if you include more columns. This is where the looping construct comes in handy. You can obtain the same plot as in the previous command as follows:

```
plot for [j=2:4] "data" u 1:j
```

The variable j is set to the values 2, 3, and 4, successively; and the data from all three columns is included in the graph. Notice that there are *no* parentheses around the second column specifier.

The body of the loop extends to the first comma, and multiple loops can be combined in the same plot command:

```
plot for [j=2:4] "data1" u 1:j, "data2" u 1:2
plot for [j=2:4] "data1" u 1:j, for [j=2:4] "data2" u 1:j
```

The first of these commands plots three columns from data1 and a single column from data2. The second command plots three columns from data1 together with three columns from data2.

The loop condition is specified in square brackets

```
for [ variable = start : end : increment ]
```

where the increment defaults to 1 if it isn't included explicitly. A negative increment is possible and will lead to a descending loop. An increment value that isn't an integer will be *truncated to the next integer.* (Gnuplot truncates the same way C does: (int)1.5 = 1, but (int)-1.5=-1.)

Both the start *and* end points of the loop are inclusive. In other words, gnuplot's for loop is equivalent to the following construct in a C-like language:

```
for( var = start; var <= end; var += increment ) {...}
```

All the examples so far have used the plot command, but the same operations are also available for set. For instance, the following command places a text label at three positions on the graph (I haven't formally introduced the set label option yet; we'll remedy that in chapter 7):

```
set for [j=1:3] label "Here!" at j,0
```

5.4.2 *Loops over strings*

In the previous examples, the loop extended over a range of numbers. This is convenient if you want to select columns from a file. But what if you want to loop over a bunch of files? To do that, you need to loop over strings. As it turns out, you can do this, too.

To make this concrete, let's say that you want to plot the daily temperatures as measured in three cities. The observations are contained in files named according to the locations. The following two commands are equivalent:

```
plot "newyork" u 1:2, "chicago" u 1:2, "sanfranciso" u 1:2
plot for [name in "newyork chicago sanfranciso"] name u 1:2
```

The loop condition in this case is

```
for [ variable in tokenstring ]
```

where tokenstring is a string of whitespace-separated tokens.

String operations are available when using loops. For instance, you can use string concatenation with the . (dot) operator to build up more complicated filenames:

```
plot for [name in "newyork chicago sanfranciso"] name . ".txt" u 1:2
```

The `word()` and `words()` functions (see table 5.1) may come in handy. Remember: `words(s)` returns the *number* of tokens found in the string s; `word(s,k)` returns the *k*th token found. For example, the following command iterates over all files. For each file, it plots the second against the first column, but it also adds a vertical offset:

```
files = "newyork chicago sanfranciso"
plot for [j=1:words(files)] word(files,j) using 1:($2+5*j)
```

5.4.3   *Summary of inline loops*

Inline loops are a new addition and have the potential to change the way you interact with gnuplot. For this reason, I want to summarize their most relevant features.

> **TIP**   Become familiar with inline loops. They can help a lot, in particular if you work with many files or with files containing a large number of columns.

Inline loops are available for the `plot` and `set` commands. In both cases, they follow directly after the command keyword. The loop's body extends to the next comma (for `plot`) or to the end of the command line. Loops can extend over numeric ranges or over the tokens found in a string. (See the following listing.)

Listing 5.3   Inline loop syntax for the `plot` and `set` commands

```
plot for [ variable = start : end : increment ] ...
plot for [ variable in tokenstring ] ...

set for [ variable = start : end : increment ] ...
set for [ variable in tokenstring ] ...
```

For numeric ranges, the loop includes both start *and* end points. The increment should be an integer; it's truncated to an integer otherwise. For iteration over strings, the loop condition consists of a string of whitespace-separated tokens.

Finally, I'd like to point out that inline loops nest. The following is legal and works as expected:

```
set for [j=1:3] for [k=1:3] label sprintf( "%d-%d", j, k) at j,k
```

Inline loop expressions are evaluated statically when the command is first parsed. This means you can't use the value of the first loop variable to control the range of the second loop dynamically: `set for [j=1:3] for [k=1:j] ...` does *not* work as expected. You must use one of the general looping constructs for such applications (see chapter 11).

5.5 *Gnuplot's internal variables*

The frequent use of internal variables is a relatively recent trend in gnuplot development. I use the term *internal variable* to refer to variables that are set by gnuplot itself and that can be read out by the user to investigate various aspects of gnuplot's internal state. The purpose of these internal variables is twofold:

- To make information about gnuplot's internal state available to user-defined functions and expressions
- To report the result of an internal calculation back to the user

If you know the name of a specific variable, then you can inspect its value using `print`, and you can use it in expressions just like a user-defined variable (as introduced in section 3.2.3). If you don't know the variable name, you can list *all* currently defined variables using

```
show variables all
```

To avoid conflicts with user-defined variables, the names of internal variables are prefixed by strings in capital letters (see table 5.2). You can restrict the `show` command to variables beginning with a certain prefix. For example,

```
show variables GPVAL
```

only shows internal variables with names beginning with `GPVAL`. (Note that there are no quotes around the prefix in the argument to the `show` command.)

Table 5.2   **Prefixes used for internal variables**

Prefix	Description
GPVAL_	For general-purpose variables, most of which refer to the most recent plot and graph (see section 9.5.4).
GPFUN_	For every user-defined function (see section 3.2.3), the function definition—as a string—is stored in an internal variable prefixed with GPFUN_, followed by the name of the user-defined function.
MOUSE_	For variables containing information about mouse events (see section 12.4.3).
STATS_	For variables set by the `stats` command (see section 5.6).
FIT_	For variables used by the `fit` command (see section F.7).

An example will help to clarify. In section 3.5.2, you learned that the `smooth kdens` algorithm can calculate a default bandwidth. Upon completion of the `plot ...` `smooth kdens` command, the internal variable `GPVAL_KDENSITY_BANDWIDTH` contains the numerical value of the default bandwidth. I also mentioned that this default bandwidth often appears too broad to reveal all the details in the data. You may therefore want to redraw the graph with a bandwidth that is (for instance) half the

default bandwidth. You can achieve this effect through the sequence of the following two commands:

```
plot "data" u 1:(1) s kdens
plot "data" u 1:(1) s kdens bandw 0.5*GPVAL_KDENSITY_BANDWIDTH
```

You first need to run the `plot` command without specifying a bandwidth, because this operation creates and populates the internal variable. This leads us to two important properties of internal variables:

- *Internal variables are only created when needed.* Before the first `plot` command (in a newly started gnuplot session), most internal variables aren't just empty—they aren't even defined (as `show variables all` will tell).
- *Internal variables are read-only.* Internal variables inform the user of gnuplot's internal state, but not the other way around: assigning a new value to such a variable does *not* change gnuplot's behavior (even if gnuplot allows the assignment, the new value is silently ignored). Assigning to internal variables isn't a way to modify options: you must use the `set` command to do that.

Not all option values can be inspected through internal variables at the moment, but it's relatively easy to add a new internal variable to the gnuplot implementation (if you're a gnuplot developer, that is). You should therefore expect that the information available through internal variables will grow over time. Use `show variables all` to see what's been included by the time you read this.

5.6   *Inspecting file contents with the stats command*

With gnuplot's `stats` command, you can examine the contents of a file and obtain information about the number of valid and invalid records, as well as find summary statistics (such as mean and standard deviation) for each column. The results are made available through internal variables (see section 5.5) and can therefore be used in subsequent calculations.

The syntax of the `stats` command is similar to that of `plot`. To examine the men's results from listing 3.1, you can use

```
stats "records" u 1:2
```

The complete output is shown in listing 5.4. The first part gives some information about the data file itself, such as the number of records, the number of blank lines, and the number of data blocks (that is, blocks of consecutive records separated by double blank lines). The entry labelled `Out of range` indicates the number of records disregarded because they're eliminated by the application of ranges (using ranges with `stats` is discussed in a moment). The subsequent parts of the output give the value for several typical *summary statistics*, such as mean and standard deviation.[3]

---

[3] In general, the definitions for the various quantities are taken from their respective entries in Wolfram Math-World (http://mathworld.wolfram.com). The abbreviation `Avg Dev` stands for the mean *absolute* deviation from the mean. The standard gnuplot reference documentation provides some additional details.

The second-to-last block in the output shows extremal values and percentile statistics. For the minimum and maximum, the number of the record where the extremal value is reached is displayed in square brackets. To find the median and quartiles, the data is sorted first. If the number of records N is odd, then the value of the record at position (N+1)/2 is reported as the median; if the number of records is even, then the mean of the two points at positions N/2 and 1+N/2 is reported. Equivalent treatment is used for the quartile values. The final section shows the result of a linear regression calculation, when treating the first column as a predictor for the second.

> **Listing 5.4 Typical output from the `stats` command**

```
* FILE:
  Records:            11
  Out of range:        0
  Invalid:             0
  Blank:               0
  Data Blocks:         1

* COLUMNS:
  Mean:            7753.9091           1287.8518
  Std Dev:        12439.9995           2169.5345
  Sample StdDev:  13047.1815           2275.4270
  Skewness:           1.9336              2.0063
  Kurtosis:           5.5228              5.7949
  Avg Dev:         9095.9091           1562.0000
  Sum:            85293.0000          14166.3700
  Sum Sq.:        2.36364e+09         7.00199e+07

  Mean Err.:       3750.8010            654.1393
  Std Dev Err.:    2652.2168            462.5463
  Skewness Err.:      0.7385              0.7385
  Kurtosis Err.:      1.4771              1.4771

  Minimum:          100.0000 [ 0]        9.5800 [ 0]
  Maximum:        42195.0000 [10]     7377.0000 [10]
  Quartile:         400.0000             43.1800
  Median:          1500.0000            206.0000
  Quartile:       10000.0000           1577.5300

  Linear Model:      y = 0.1743 x - 63.9
  Slope:             0.1743 +- 0.001633
  Intercept:         -63.9 +- 23.94
  Correlation:       r = 0.9996
  Sum xy:            4.066e+08
```

5.6.1 *The stats command and internal variables*

The `stats` command makes all of its results available through internal variables. To prevent clobbering existing variables, the prefix STATS_ is prepended to all variable names. Furthermore, the suffix _x or _y, respectively, is appended to the variable name to indicate the first or second column; if only a single column is identified in the `using` specification, no suffix is appended. (Use the `show variables` command to list all the variables, their names, and their values.)

Let's look at an example. The following code snippet prints the mean of the first and second columns to screen:

```
stats "records" u 1:2
print STATS_mean_x, STATS_mean_y
```

You can change the prefix with the name keyword. (You can't leave the prefix blank or empty.) For example, to use the string "REC" as a prefix, you use this command:

```
stats "records" u 1:2 name "REC"
```

Custom prefixes can be useful if you're working with multiple files. By prefixing each set of variables with its own identifier, you can compare summary statistics across several data sets.

A WORKED EXAMPLE

Because stats assigns all of its results to internal variables, they can be used seamlessly in subsequent operations. Here's an example. Back in section 3.5.2, you learned about gnuplot's smooth kdensity feature. I also pointed out that to obtain a *normalized* estimate, it is necessary to give each data point a weight equal to $1/n$, where n is the number of points in the data set. Back in chapter 3, I supplied the number of points manually and explicitly. Now, using stats, I can do better:

```
stats "measurements" u 1
plot "measurements" u 1:(1.0/STATS_records) smooth kdensity
```

Running the stats command first populates all the variables, including STATS_records: the number of records in the file. You now don't need to know the *value* of this quantity—instead, you refer to it by its symbolic name in the plot command. This same technique is widely applicable; you'll see examples in later chapters (see, for example, sections 11.1.2, 11.6, and F.6).

5.6.2 *Further options for the stats command*

To apply the stats command to a single column, you include only one column identifier in the using specifier. Of course, it doesn't make sense to perform the calculation for the linear model in this case.

You can limit the application of stats by using ranges. Ranges can be included in the command line (as for plot): stats [0:2000] "records" u 1:2. The stats command also honors ranges set using the set _range family of functions (see section 8.2).

To send the output to a file, specify a filename using set print (see section 5.3.1). You can suppress all output by appending the nooutput keyword to the command line.

5.7 *Command history*

Gnuplot has a history feature, making it possible to recall the most recent commands. The history feature is bound to the up- and down-arrow keys, as well as to the Ctrl-p and Ctrl-n key combinations. The easiest way to modify an existing plot is to recall the

command (using the up-arrow) and make whatever change is desired to the command line.[4]

    The most common application is to regenerate the current graph. In the next section, we'll discuss some special commands to do that. In section 5.7.2, we'll examine gnuplot's `history` command, which gives you greater control over the command history.

5.7.1   Redrawing a graph

You've already (in listing 2.2) encountered the `replot` command. It runs the most recent `plot` command again. You usually use it after making changes to options using `set`.

    You can add new data or functions to `replot`:

```
plot sin(x)              ⟵——  Plots a sine wave
replot cos(x)            ⟵——  Plots both a sine wave and a cosine wave
```

The `refresh` command is like `replot`, except that it *does not read the data again.* This is convenient if data was read from standard input (using the special filename `"-"`) or when the file may have changed on disk. It is *not* possible to add further data or functions using `refresh`.

> **TIP**   The `refresh` command reruns the most recent `plot` command (just like `replot` does), but without having to reread the data from disk.

It's fairly common to load a script using `load` and then want to edit the `plot` command. Unfortunately, `load` doesn't enter the contents of the file into the command history, so the `plot` command from the file isn't available using the up-arrow key. A new option to the `show` command can help:

```
show plot add2history
```

This not only prints the command line containing the `plot` command to screen, but also enters it as the most recent entry in the command history.

> **TIP**   You can enter `plot` commands read from a file into the command history using the (optional) `add2history` argument to `show plot`.

5.7.2   The general history feature

If you want more control over the command history, you can use `history`. The `history` command can be used three ways: to print all or parts of the command history, to search the command history, or to reexecute a command:

---

[4] Gnuplot supports various key bindings for editing the command line. The exact selection depends on the specific version of the readline library that was used to compile the local gnuplot executable. See appendix A and the gnuplot standard reference documentation for details.

```
history [quiet] [{int:max}] ["{str:filename}" [append]]     ◁── Shows the command history
history ?"{str:cmd}"                                          ◁── Searches the history
history ![ {int:pos} | "{str:cmd}" ]                         ◁── Executes a command
```

The first line shows how to print the command history to the screen or (if a filename is given) to a file. Optionally, `history` clobbers any existing file of the same name, unless you use the `append` keyword. By default, the command history is shown together with line numbers, but they can be suppressed with the `quiet` option. Finally, you can restrict the output to the last `max` entries by providing a numerical argument. For example, the command `history quiet 5` plots the last five commands preceding the current one to screen, suppressing line numbers. On my system, this command currently produces the following output:

```
plot cos(x)
plot sin(x)
plot [-2:2] sin(x)
plot [-2:2] sin(x), x
plot [-2:2][-1:1] sin(x), x
history quiet 5
```

Alternatively, you can give a command, or just its beginning, as an argument to `history`, prefixed by either a question mark (`?`) or an exclamation point (`!`). If the question mark is used, gnuplot searches the command history and shows only entries that start with the string provided. If the exclamation point is used, the most recent matching entry in the command history is executed again. For example, `history ?"set term"` shows all commands that changed the `terminal` setting. Finally, the exclamation mark, together with a number, executes the command at the specified history position.

There is also a `history` option that can modify the behavior of the `history` command:

```
set history [ size {int:max} ] [quiet|numbers] [full|trim] [default]
```

The `size` keyword takes an integer argument that specifies how many entries in the history should be saved to file when a gnuplot session is ended. A value of -1 allows an unlimited number of entries to be written to file.

Specifying `set history quiet` suppresses numbers generally when using the `history` command. The `trim` option removes duplicate entries in the command history, retaining only the most recent one. The `default` keyword restores the default settings, which are `set history size 500 numbers trim`.

5.7.3   *Restoring session defaults*

At times you may want to remove all user-defined variables and options and restore your gnuplot session to a defined state. You can do this using `reset`. The reset command does *not* affect options having to do with terminal settings (`set terminal` and `set output`), with locale settings (`set locale`, `set encoding`, and `set decimalsign`), and with search paths (`set loadpath`, `set fontpath`, and `set psdir`).

The `reset` command restores gnuplot to its default settings, period. If you also want to reload your personal initialization file (see chapter 12), then you must use `reset session`.

> **TIP** Use `reset session` to restore gnuplot's default state *and* to load your personal initialization file.

Finally, there are two special forms of this command: `reset errors` clears only the two internal variables `GPVAL_ERRNO` and `GPVAL_ERRMSG`. The command `reset bind` restores all hotkey bindings (see chapter 12) to their default values.

5.8 *Summary*

This chapter was something of a grab bag of useful features that make life easier. Such material often appears dry, so let's highlight what was important and why.

Strings and string operations are important whenever you want to include textual information on a graph. That's usually not a concern for initial and exploratory work, but it's crucial to provide semantics to finished graphs intended for public presentation. Feel free to come back to this topic whenever the need arises!

String macros and inline loops can simplify repetitive tasks. Inline loops (which are a relatively new gnuplot feature) can be a major convenience; you should at least be aware of this feature.

Internal variables, compile-time options, and the command history are all methods to inspect gnuplot's internal state. The need for these features usually arises only when you're trying to tackle more complex tasks—for now, it should suffice to know that these possibilities exist.

This concludes the introduction to gnuplot fundamentals. You have all the features and commands at your disposal to work efficiently with gnuplot. Now we'll take the basics for granted and talk in detail about the different ways you can affect the appearance of a graph: the "looks." Stay tuned.

A catalog of styles 6

This chapter covers

- Directives for choosing styles
- Gallery of plotting styles
- Pointers to additional styles

The next few chapters describe the different ways you can control the appearance of a plot: how to make it look just right. In this chapter we'll discuss the various ways to display *data*, and in the next chapter we'll talk about all the *other* stuff that goes onto a plot, such as labels, borders, arrows, and similar decorations. Because axes and their labels can provide so much relevant information about a plot, they're given their own chapter (chapter 8). And finally, in chapter 9, we'll discuss how you can customize the appearance of all plot elements, in case you don't like the defaults.

Most of this chapter is taken up by a comprehensive gallery of available plotting styles. Feel free to treat it as a *catalog*: look at the pictures to see whether there's anything you like and might want to use right away. As your needs change, you can always come back and check out what other styles are available.

In any case, be sure to read section 6.2, because it explains the mechanics of how to choose plot styles in gnuplot. The remainder of the chapter is optional (at this point).

6.1   *Why use different plot styles?*

Different types of data call for different display styles. For instance, it makes sense to plot a smooth function with one continuous line but use separate symbols for a sparse data set in which each individual point counts. Experimental data often requires error bars together with the data, whereas counts of discrete events call for histograms. Choosing an appropriate style for the data leads to graphs that are both informative and aesthetically pleasing.

But the choice of plot style isn't just an aesthetic issue: different graphical representations give a data set context and may even imply specific *semantics*:

- Continuous lines indicate a function or a dense data set with little noise.
- Explicit point symbols emphasize the discrete nature of a sparse data set and make individual data points stand out.
- Boxes are mostly used for histograms and so imply counts of discrete events.
- Error bars make the uncertainty in a data set explicit.

TIP   Remember: the style you choose determines not only what people are seeing, but also how they will interpret it!

6.2   *Styles and aspects*

Gnuplot can create graphs using different overall styles: you've already encountered `with lines`, `with points`, and `with linespoints`. In this chapter, you'll learn about additional styles for drawing boxes, error bars, circles and arrows, and so on. The style determines the overall appearance of the plot. Styles are usually specified using the familiar `with` keyword.

All graphs are constructed from lines and point symbols. In addition to the overall choice of style, you can also customize many aspects of lines and points: things like the width, color, and dash pattern of lines; and the shape, size, and color of points. These aspects can be set globally or locally for each data set individually.

Unless you specify aspects explicitly, gnuplot will choose a different set of aspects for each new data set you include in a graph. For example, if you create a graph from four data sets, by default each data set will be drawn using a different color (or dash pattern) and point shape. You can override this default sequence locally (as part of the `plot` command) or redefine the overall sequence globally.

In this section, we'll first review how to use inline customizations to modify the appearance of graph elements as part of the `plot` command. Then I'll give you an overview of the aspects that can be changed this way.

6.2.1   *Choosing styles inline through with*

Generally, plot styles are chosen *inline*, as part of the `plot` command. (There are also ways to set global style preferences—we'll come back to this in chapter 9.) You already saw inline styles in chapter 2; by giving the `with` keyword as part of the `plot` command, you can specify which style to use:

```
plot "data" u 1:2 with lines, "" u 1:3 with linespoints,
➡️    "" u 1:4 with points
```

As usual, keywords can be abbreviated to the shortest unambiguous form, so it's more convenient to write w l, w linesp (or even: w lp), and so on.

6.2.2 *The default sequence*

In graphical analysis, you often want to plot several similar data sets, possibly together with functions, all on the same plot—usually so that they can be compared directly to one another:

```
plot [0:4][-0.25:1.5] "sequence" u 1:2 w lp, "" u 1:3 w lp, "" u 1:4 w lp,
➡️                     -log(x)+(x-1) w l
```

All data sets are plotted using the same overall style (with linespoints), but gnuplot makes sure to change the color (or dash pattern) of the lines used. Gnuplot also uses a different point symbol for each data set (see figure 6.1).
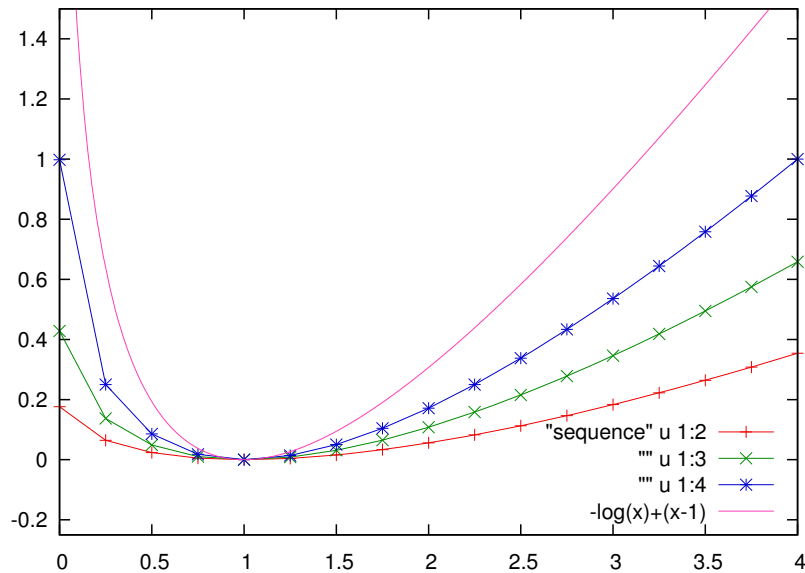
Figure 6.1   Gnuplot automatically chooses a different line style for each curve.

By default, lines and points are chosen automatically from a sequence of available patterns. If there are more data sets in the plot than there are patterns in the sequence, the selection starts again at the beginning.

You can override the automatic selection by explicitly specifying the index of one particular element from the default sequence. For example, if in the previous example you wanted to draw the second data set in the same style as the first, you could say

```
plot [0:4][-0.25:1.5] "sequence" u 1:2 w lp, "" u 1:3 w lp lt 1,
➥                     "" u 1:4 w lp, -log(x)+(x-1) w l
```

Here, lt is short for linetype (which refers to elements of the default sequence), and lt 1 asks for the *first* element from the sequence. Fixing a specific line like this doesn't affect the way the internal counter is incremented, so the ways the fourth column of the data and the function are displayed don't change.

So far, I've only shown you how to select a specific entry from the default sequence. Instead of making such a local change, you can also customize the default sequence globally—or even use *different* default sequences for different purposes. We'll discuss how to do that in chapters 9 and 12, but it's pretty advanced stuff. Most of the time, you'll find that gnuplot cycles through the default sequence in a way that just works.

6.2.3 *Customizing graph elements*

You can exercise detailed control over various aspects of graph elements. Table 6.1 lists all commonly used properties. Many are reasonably self-explanatory, but some may not be. In particular, color specifications and customized dash patterns require more detail—we'll come back to them in chapter 9.

There are several contexts in which the properties from table 6.1 can be specified:

- Some properties can be set *globally*. For example, you can use the set pointsize option to adjust the size of point symbols globally. Other properties can be configured globally using set style (see chapter 9).
- Several *terminals* allow you to modify the line widths and point size used for the selected terminal (see chapter 10).
- By far the most common way to specify appearance aspects is *inline* as part of the plot command. It's straightforward to do this, as I'll show next.

INLINE CUSTOMIZATIONS

To change any appearance property, you tack the desired specifiers onto the function or data set in the plot command:

```
plot sin(x) w l lw 3          ⟵── Uses line width 3
plot "data" u 1:2 w lp lt 2   ⟵── Inherits all properties from line type 2
```

If there are multiple curves in a plot, attach the specifiers to the desired curve:

```
plot "data" u 1:2 w lp lt 3, sin(x) w l lt 1
```

This plots the data set using the properties of line type 3 and the sine function with the properties of line type 1.

You can include more than one modifier (but gnuplot will prevent you from including the same modifier multiple times):

```
plot "data" u 1:2 w lp pt 6 ps 2        ⟵── Point type 6, at point size 2
plot cos(x) w l lw 3 lc rgb "red"       ⟵── Triple line width, in red
```

Table 6.1   Modifiers for the appearance of lines, points, and other graph elements

Keyword	Abbreviation	Description
`linetype`	`lt`	A *line type* is a combination of *all* aspects of both lines and points: line width, color, and dash type, as well as point type, point size, and point interval. If not instructed otherwise, gnuplot will cycle through a predefined sequence of line types for each new curve in a plot.
`linestyle`	`ls`	User-defined combination of appearance aspects. Similar to `linetype` but not eligible for automatic assignment to curves. (See section 9.4.)
`linewidth`	`lw`	Numerical multiplier, multiplying the default line width.
`linecolor`	`lc`	Color used to draw graphical elements (lines and points). (See section 9.1 for the syntax of color specifications.)
`dashtype`	`dt`	Pattern of dots and dashes used to draw the line. (See section 9.2.2 for the syntax of dash-pattern customizations.)
`pointtype`	`pt`	Shape or symbol used to display a data point. Chosen from a selection of available symbols. (See section 9.2.1.)
`pointsize`	`ps`	Numerical multiplier, multiplying the default point size.
`pointinterval`	`pi`	Controls whether only a subset of records is displayed with an explicit symbol when using the `linespoints` style. (See section 6.3.1.)
`textcolor`	`tc`	Color used for textual labels. (See section 9.1 for the syntax of color specifications.)
`fillstyle`	`fs`	Closed shapes (`boxes`, `circles`, `ellipses`, `candlesticks`) can be filled with a color or a pattern. The keyword `solid`, followed by a density between 0.0 and 1.0, fills the shape with the current color at the specified density; `pattern` uses the current pattern from the range of available patterns; and `empty` leaves the shape unfilled. (See section 9.4.4 for more information on fill styles.)
`fillcolor`	`fc`	Color used to fill closed shapes. (See section 9.1 for the syntax of color specifications.)
`border`	`bo`	Describes the border used for filled areas. Expects any combination of line options as argument. (See the tip at the end of this section for the `lineoptions` shorthand.)
`arrowstyle`	`as`	Selects a separately defined arrow style when using `with vectors`. (See section 7.3.2 for more information on arrows and section 9.4.3 for information on arrow styles.)

Line types and line styles are special: they're containers for *all* available properties. Each line type (or line style) indicates a line width, color, and dash pattern, as well as a point shape and size. By specifying either a line type or a line style, you choose values for all properties at once. Gnuplot allows you to override individual aspects, and doing so is common:

```
plot sin(x) w l lt 3 lw 2
plot "data" u 1:2 w p lt 4 pt 3
```

◁ ——— **Uses line type 3, but doubles the width**

◁ ——— **Uses color and point size of line type 4, but overrides the shape of the points**

As you can see, if a line type (or line style) is given, the curve inherits the applicable properties, unless they're explicitly overridden. In the previous example, the data set was plotted `with points` (that is, without connecting lines), so the graph inherited point shape, size, and color from the indicated `lt 4`. But because the point type was selected explicitly, the resulting graph used color and point size from `lt 4` but the point shape corresponding to `pt 3`.

SHORTHAND: LINE OPTIONS

Most styles and decorations introduced in this and the next few chapters are constructed from *lines*, and the appearance of these lines is usually subject to the appropriate modifiers from table 6.1. In particular, `linetype`, `linestyle`, `linewidth`, `linecolor`, and `dashtype` are generally recognized.

To condense the presentation, I won't list these modifiers individually going forward but will instead refer to them as *line options*. In command summaries, the string `lineoptions` is henceforth used as a *placeholder* for any combination of the aforementioned explicit appearance modifiers. Of course, `lineoptions` is *not* a legal gnuplot keyword—it's just a notational shorthand.[1]

> **TIP** In command summaries, the placeholder `lineoptions` stands for any combination of the explicit line-appearance modifiers listed earlier. This is just a convenient shorthand; `lineoptions` is *not* a legal gnuplot keyword!

6.3 A catalog of plotting styles

Well over two dozen styles are available in gnuplot. Here we look at those most useful for ordinary, two-dimensional data. I'll provide pointers to additional styles (for more complicated data sets) in section 6.5, toward the end of this chapter.

---

[1] That being said, as a symbolic placeholder for *all* available line modifiers, the `lineoptions` shorthand is more robust and often more accurate than an explicit list that becomes incomplete whenever gnuplot's selection of line modifiers grows. The standard gnuplot reference documentation contains many instances where explicit lists of line modifiers haven't kept up with gnuplot's increased capabilities and are now incomplete.

6.3.1   *Core styles: lines and points*

There are four styles I consider *core* styles, because they're so generally useful: `with points`, `with lines`, `with linespoints`, and `with dots`. These styles represent data with simple symbols or lines on the plot (see figure 6.2).
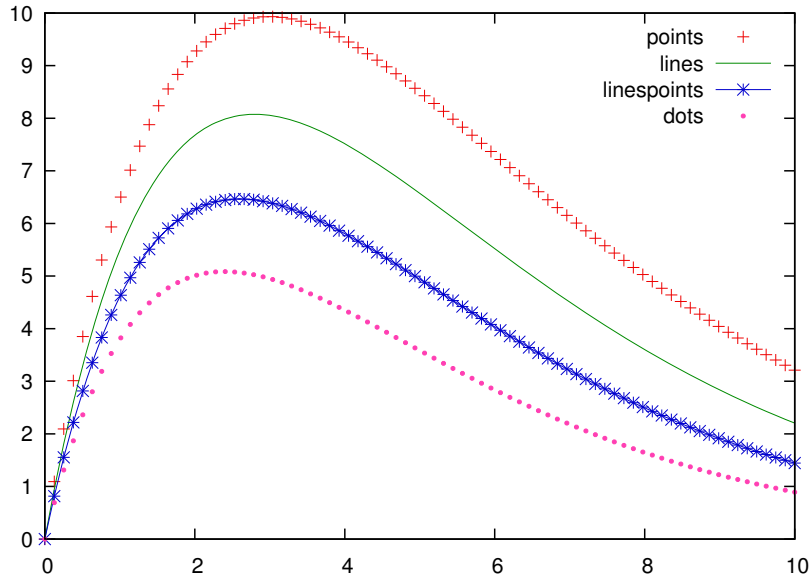
Figure 6.2   The four core styles: `with points`, `with lines`, `with linespoints`, and `with dots`

POINTS

The `points` style plots a small symbol for each data point. The symbols aren't connected to each other. This is the default style for data (see figure 6.2).

The size of the symbol can be changed globally using the `set pointsize` command. The parameter is a multiplier, defaulting to 1.0:

```
set pointsize {flt:mult}
```

You can also change `pointsize` (abbreviated `ps`) inline:

```
plot "data" u 1:2 w points pointsize 3
```

The point type (or point shape) can be selected using `pointtype` (or `pt`). Remember that `ps` stands for `pointsize`, *not* for *point style* (there is no such thing)! Figure 9.7 shows all commonly available point shapes.

In addition to the selection of available point types, you can also specify a *single letter* as argument to `pointtype`. The selection of symbols available depends, of course, on

the fonts installed on your system. Modern fonts that provide glyphs for a large number of Unicode code points may offer some interesting possibilities (see figure 6.3)! Note that when you use a character as a symbol, you must change its color using `text-color` instead of using `linecolor`. (See the sidebar in section 5.1 for information on Unicode and how to access characters from extended sets.)
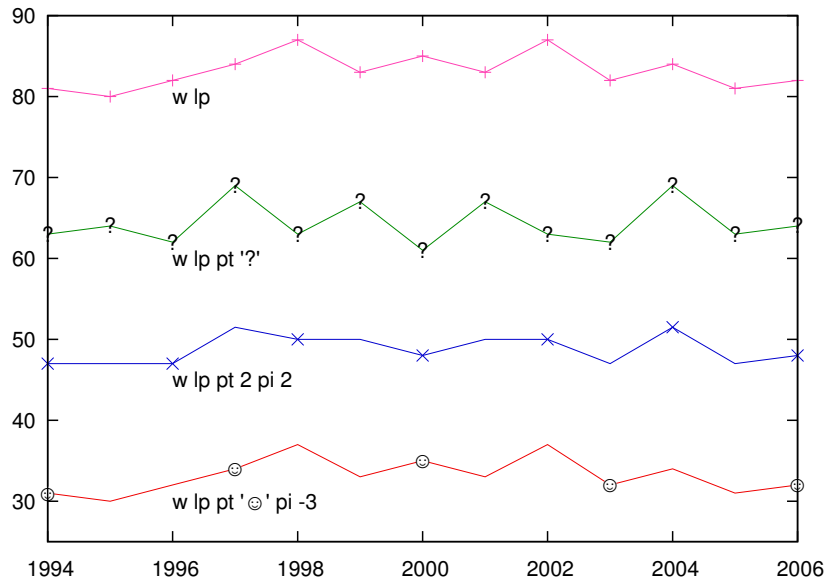
Figure 6.3 Variants of the `with linespoints` style. Top to bottom: default behavior, using a (regular) character as symbol, omitting every second symbol, and using a character from an extended range of glyphs while omitting two out of every three points. Note that in the last case, the lines have been erased to make room for the characters by giving a negative argument to the `pi` keyword.

LINES

The `lines` style doesn't plot individual data points, only straight lines connecting adjacent points. This is the default style for functions and the preferred style for dense data sets without too much noise.

Many aspects of lines, including their width, color, and dash pattern, can be customized. For instance, to double the width of a single line in a plot, you can use the following inline directive:

```
plot "data" u 1:2 w l linewidth 2
```

LINESPOINTS

The `linespoints` (abbreviated `lp`) style is a combination of the previous two: each data point is marked with a symbol, and adjacent points are connected with straight lines. This style is mostly useful for sparse data sets.

What has been said about the `with points` and `with lines` styles generally applies to `with linespoints`: the default size of the symbol is controlled by the `set pointsize` option but can be overruled inline, as can the line type and the point type.

A property that is *only* available for `linespoints` is `pointinterval` (short: `pi`).[2] If this property is set to a *positive* integer *n*, then gnuplot plots a symbol only for every *n*th data point; all other points are drawn `with lines` only (see figure 6.3). This can be extremely useful for dense data sets.

If `pointinterval` is *negative*, then it has the same effect as just explained, but in addition, a circular region centered at the location of the point is cleared, using the special `bgnd` color before the symbol is drawn (see figure 6.3). (The background color `bgnd` is a property of the terminal; use `set terminal ... background ...` to set it. See chapter 10.) The size of the cleared region is controlled using the `set pointintervalbox` option:

```
set pointintervalbox {flt:mult}
```

The value of `pointintervalbox` multiplies the point size to find the radius of the region to be cleared. All this is demonstrated in figure 6.3.

DOTS

The `dots` style prints a minimal dot (a single pixel for bitmap terminals) for each data point. This style is occasionally useful for very large, unsorted data sets (such as large scatter plots). Figure 1.2 was drawn using dots.

6.3.2   *Indicating uncertainty: styles with error bars or ranges*

Sometimes you don't just want to show a single data point; you also want to indicate a range with it. This may be the experimental uncertainty (the *error bar*), or it may be the range over which some quantity has changed during the observation interval (this is typical of financial charts). Gnuplot offers several styles that place an indicator for such a range onto the plot. First we'll look at styles that draw regular error bars (both in vertical and in horizontal directions). Then we'll go on to discuss styles that let you indicate several ranges at once (but only in the vertical direction).

STYLES WITH ERROR BARS

There are two basic styles to show data with error bars in gnuplot: `errorbars` and `errorlines` (see figure 6.4). The `errorlines` style is similar to the `linespoints` style (a symbol for each data point, adjacent points connected by straight lines), whereas the `errorbars` style is similar to the `points` style (unconnected symbols).

These styles draw error bars in addition to the data. Error bars can be drawn in either the x or y direction or both. To select a direction, prefix the style with x, y, or xy, respectively, as in `plot "data" with xyerrorlines`. Table 6.2 summarizes all available combinations.

---

[2]   This feature is a relatively recent addition to gnuplot.
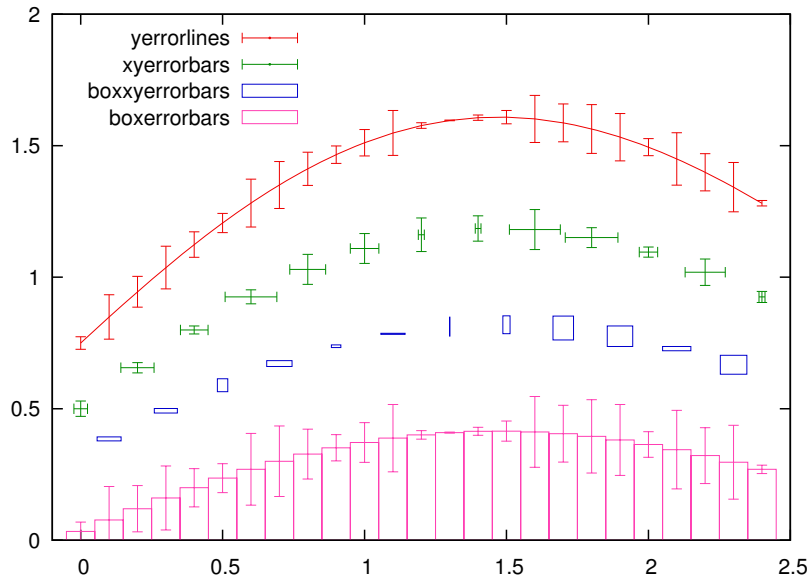
Figure 6.4   Different plot styles showing uncertainty in the data. From top to bottom: connected symbols using `errorlines`, unconnected symbols using `errorbars`, ranges indicated as boxes using `boxxyerrorbars`, and errors on top of a histogram using `boxerrorbars`.

Table 6.2   All possible combinations of `errorbars` and `errorlines` styles

	Error bars in x direction	Error bars in y direction	Error bars in both directions
Unconnected symbols	`xerrorbars`	`yerrorbars`	`xyerrorbars`
Connected symbols	`xerrorlines`	`yerrorlines`	`xyerrorlines`

The appearance of both the `errorlines` and `errorbars` styles is determined by the current line style. Both styles draw a symbol at the location of the data point—which is unnecessary, because the intersection of the error bars already indicates this position! You can suppress the symbol by specifying `pointtype 0`, like so: `plot "data" u 1:2:3 with yerrorlines pt 0`.

The error bars themselves are drawn in the current line style. A tic mark is placed at the ends of each error bar (see figure 6.4). You can control the size of the tic mark using the `set bars` option:

```
set bars [ small | large | fullwidth | {flt:mult} ]
```

The parameter is a multiplier, defaulting to 1.0. The symbolic names `small` and `large` stand for the values 0.0 and 1.0, respectively. The value `fullwidth` is only relevant to

histogram styles (more on that in a minute). Finally, you can turn off the tic marks using `unset bars`.

   Error bars require additional information. Just the x and y coordinates aren't enough: you must also provide data about the size of the uncertainties. Usually, this data comes from the data file, in the form of one or two additional columns. If one additional column is given, it's interpreted as a range *dy* to be added to and subtracted from the corresponding data value, so the error bar is drawn from (x, $y−dy$) to (x, $y+dy$). If *two* additional columns are supplied, they're interpreted as the absolute coordinates of the lower and upper ends of the error bar (*not* the ranges), so error bars are drawn from (x, *ylow*) to (x, *yhigh*). Corresponding logic applies to error bars drawn in the x direction.

   As usual, the columns to use are indicated with the `using` directive to `plot`:

```
plot "data" using 1:2:3 w yerrorbars              ⟵  ( x, y, dy )
plot "data" using 1:2:3:4:5:6 w xyerrorbars       ⟵  ( x, y, xlow, xhigh, ylow, yhigh )
```

Data transformations (see section 3.3) are often useful in this context. Here are some examples:

- If the input file contains only the variance (instead of the standard deviation, which is usually plotted as error) together with the data, you can apply the necessary square root inline: `plot "data" u 1:2:(sqrt($3)) w yerrorb`.
- If you know that the uncertainty in the data is a fixed number (such as 0.1), you can supply it directly: `plot "data" u 1:2:(0.1) w yerrorl`.
- If the data supplied in the file is of the unsupported form ($x, y, ylow, yhigh, dx$), you can build up the required `plot` command manually: `plot "data" u 1:2:($2-$5):($2+$5):3:4 w xyerrorl`.

As a final style to visualize data with uncertainty in both directions, there's `boxxyerrorbars`. It's similar to the `xyerrorbars` style, except that the range of uncertainty is shown as a rectangular box centered at the data point, rather than as a cross of error bars. (If you prefer to use circles or ellipses instead of boxes to indicate a confidence range, you may want to peek ahead at section 6.3.5.)

   The last style that uses error bars is `boxerrorbars` (not to be confused with `boxxyerrorbars`), which is a combination of the `boxes` and `yerrorbars` styles. It's displayed as a box with a vertical error bar centered at its top. It might be used, for instance, for histograms that have some uncertainty in their counting statistics. The additional values required for the error bar are supplied as the third (or third and fourth) arguments to the `using` directive. The box width is provided as the last argument to `using`.

   The styles discussed in this section are mostly used to plot data stemming from scientific experiments or calculations, where you want to show the uncertainty in the data clearly. But there are other situations where you may want to indicate a range (or even several ranges) together with the data. Those are the topic of the next section.

TIME-SERIES STYLES

Gnuplot offers two styles that are mostly useful for time-series data, although they can be used for other purposes as well: `candlesticks` (also known as a *bar-and-whiskers plot*) and `financebars` (see figure 6.5). Both can show *two* ranges in a single (vertical) direction: for instance, the typical band of variation and the highest and lowest values ever. Both are frequently used for financial data (such as stock prices), and I'll discuss them in those terms. The `candlesticks` style in particular is quite versatile and can be used to good effect in a variety of situations.
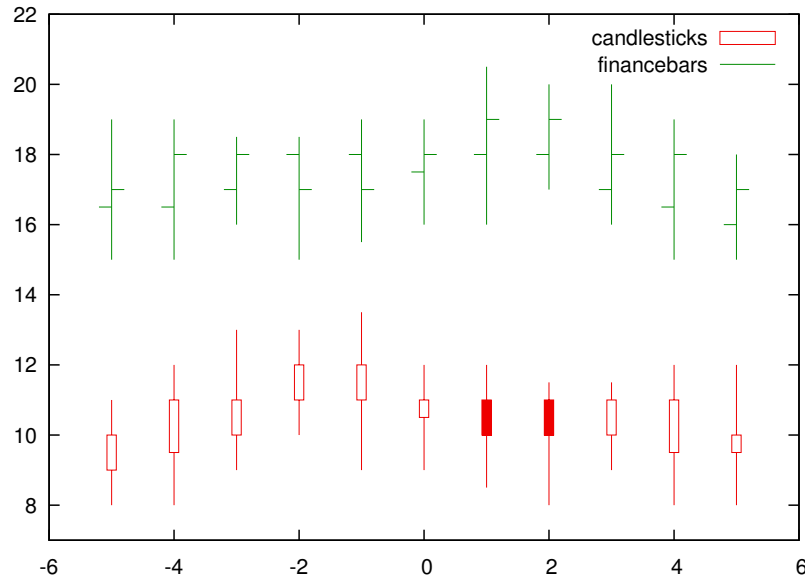
Figure 6.5   Styles for time series: `financebars` and `candlesticks`. Filled bars indicate that the closing value is less than the opening value for the current record.

Both styles require five columns of data: the x value followed (in order) by the opening, low, high, and closing prices. As usual, the appropriate columns are selected with the `using` directive to the `plot` command. Additional columns may optionally be specified to indicate the box width or the color.

Both styles represent the maximum range (low to high) with a vertical line. They differ in the way the secondary (opening to closing) range is displayed: in the `candlesticks` style, a box of finite width is overlaid on the vertical line; in the `financebars` style, tic marks indicate the opening and closing values. The size of the tic marks is controlled by the `set bars` option familiar from `errorbars` styles (see figure 6.5).

You can control the details of the `candlesticks` style using some additional options. If the current fill style is `empty` and the closing value is less than the opening one, the box is filled with the current color; otherwise, the box is left empty. (If you choose a particular fill style using `set style fill`, then this style is used to fill the box,

independent of the opening and closing values. See section 9.4.4 for more informa-
tion on `set style fill`.)

You can change the width of the box either by using the `set boxwidth` option or
by supplying a value in a sixth column. If the width is read from a column, it's assumed
to be given in the same units as the x coordinate. (If `boxwidth` is unset, the value of
`set bars` is used instead, but this usage is deprecated for `candlesticks`. For `finance-
bars`, the `set bars` option is the only way to change the size of the opening and clos-
ing markers.)

To place tic marks at the ends of the vertical line, append the keyword `whisker-
bars` (or `whisker`) to the `plot` command. You can control the size of these tic marks
independently from the box width by appending a numerical value to the `whisker-
bars` keyword. This value is interpreted as a multiplier giving the length of the tic
mark relative to the box width.

A few examples will make this clearer:

Plain ──▷
```
plot "data" u 1:2:3:4:5 w candlesticks
plot "data" u 1:2:3:4:5 w cand whiskerbars      ◁──── Tic marks the same
plot "data" u 1:2:3:4:5 w cand whisker 0.1      ◁──     length as boxwidth
                                                     Tic marks one tenth
                                                     the length of boxwidth
```

Neither the `financebar` style nor the `candlesticks` style connects consecutive
entries. If that's what you want, you'll have to do so explicitly. Keep in mind that it's
not clear what should be connected in these styles—they don't have a concept of a
"middle" value. You must supply this information separately.

6.3.3   *Styles with steps and boxes*

Box styles, which draw a box of finite width, are sometimes useful for counting statis-
tics or for other data sets where the x coordinates have only discrete values.

STEPS

Gnuplot offers three styles to generate steplike graphs consisting only of vertical and
horizontal lines (see figure 6.6). The only difference between the three styles is the
location of the vertical step:

- `histeps` centers each bin around the supplied x value.
- `steps` places the vertical step at the *end* of the bin.
- `fsteps` places the vertical step at the *front* of the bin.

If in doubt, the `histeps` style is probably the most useful.

BOXES AND IMPULSES

In contrast to the step styles from the previous section, the `boxes` style plots a box cen-
tered at the given x coordinate from the x axis (not from the graph border) to the y
coordinate (see figure 6.7). The width of the box can be set three ways:

- Supplied as the third parameter to `using`.
- Set globally through the `set boxwidth` option.
- Otherwise, boxes are sized automatically to touch adjacent boxes.
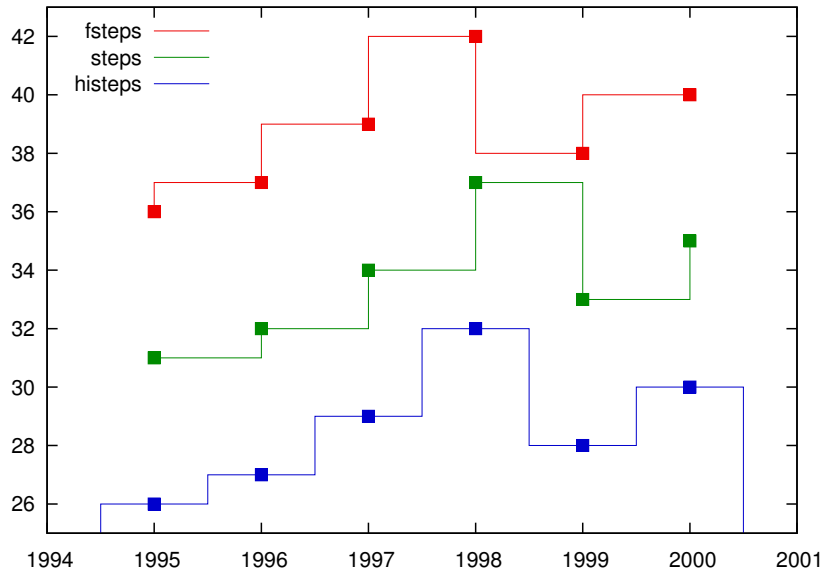
Figure 6.6 The three step styles. The same data set is shown three times (vertically shifted). Individual data points are represented by symbols; the three `steps` styles are shown with continuous lines. Note how different the *same* data set can appear, depending on the exact location of the vertical steps.
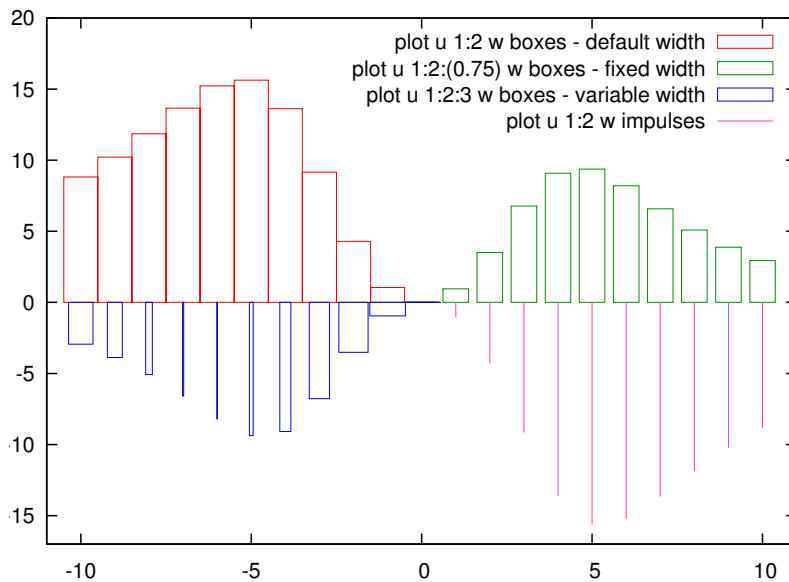
Figure 6.7 Box and impulse styles. The widths of boxes can be set globally or for each box individually. The second data set uses a fixed width (enclosed in parentheses in the `using` directive); the third one reads values for variable box widths from file.

If you supply a third column in the using directive, it's interpreted as the total width of the box in the same coordinates that are used for the x axis. The set boxwidth option has the following syntax:

```
set boxwidth [ {flt:size} ] [ absolute | relative ]
```

The *size* parameter can either be a measure of the absolute size of the box in x axis coordinates or denote a fraction of the default box size, which is the width of the box if it touches adjacent boxes. If absolute mode isn't stated explicitly, relative sizing is assumed. You can use a boxwidth of -2 to force automatic sizing of boxes (with adjacent boxes touching each other). The impulses style is similar to the boxes style with boxwidth set to zero. The examples in figure 6.7 clarify this.

Boxes can be filled or shaded, according to the global set style fill option (see section 9.4.4) or the inline fillstyle specification. The fillsteps style is a filled equivalent of the steps style.

6.3.4   *Filled styles*

You can fill the area between two curves in two-dimensional plots with color or patterns using the filledcurves style. The appearance of the filled regions is determined by the settings of the fill style, which is controlled by the set style fill option (see section 9.4.4) or given inline.

Filling the area between two curves requires a data set with at least *three* columns: for the x coordinate and the y coordinates of the two curves. The two curves constitute the boundaries of the filled area. If the two lines cross each other, you can distinguish the enclosed areas depending on whether the first or the second line is greater than (that is, above) the other. By default, all enclosed areas are shaded, but you can restrict shading to only one of the two kinds of areas by appending the keyword above or below.

This style is only available when plotting data from a file, but you can "fake" it using the "+" pseudofile (see section 4.5). That's what's done in the following code snippet (see figure 6.8):

```
plot [0:3*pi][-1.1:2.1] "+" u 1:(sin($1)):(0.5*sin($1)) w filledc,
➡                          "" u 1:(1+sin($1)):(1+0.5*sin($1)) w filledc above,
➡                          "" u 1:(2+sin($1)):(2+0.5*sin($1)) w filledc below
```

All three versions are shown: the default (with all enclosed areas filled) and—vertically shifted—the same curves with only the areas above and below filled.

In addition to filling the areas between two explicitly given curves, gnuplot can also handle other boundaries:

- Fill the area between one curve and one straight line (which may be one of the coordinate axes or a plot boundary).
- Treat a single curve as a closed polygon, and fill its interior.
- Specify an additional point that will be included in the construction of the polygon.
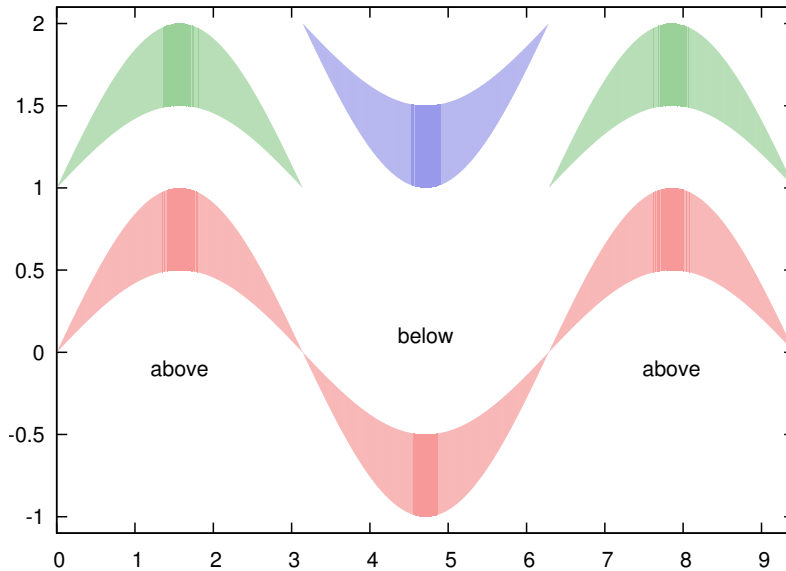
Figure 6.8 Shading the area between two curves in a single data set: `plot "data" u 1:2:3 w filledcurves`. You can select only those areas where one of the curves is above or below the other curve. (See the text for details.)

I suggest you consult the standard gnuplot reference documentation for the pertinent details.

6.3.5 *Beyond lines and points: multivariate visualization*

Several styles allow you to encode information by other means than the position on the graph: `with labels` reads a text string from the input file and places it on the plot. Using `pointsize variable`, you can change the symbol size according to the values in the data set. Finally, you can place arrows or entire geometric shapes on the plot using the `with arrows`, `with circles`, and `with ellipses` styles.

All these styles have in common that they consist of graphical elements that aren't simple but that instead have *internal degrees of freedom*—that's what enables them to encode additional information beyond their position on the graph. (We'll come back to multivariate visualization in section 14.2.) At the same time, because they're more than "lines and points," the distinction between these styles and some of the *decorations* discussed in the next chapter is a bit fuzzy—in fact, these styles share many implementation details with the corresponding decorations. Frequently, what applies to one applies to the other, even if it doesn't say so explicitly in the text!

> **TIP** The `with labels`, `with arrows`, `with  circles`, and `with ellipses` styles share many properties with the `set label`, `set arrow`, and `set object` decorations.

LABELS AND POINT SIZE

Let's look at an example that demonstrates both the `with labels` style and the `point-size variable` facility. Listing 6.1 shows a short data file containing the additional information required for labels and symbol sizes as additional columns. Given this file, you can generate the plot in figure 6.9 using the following commands:

```
plot [0:6][0:3.5] "labels" u 1:2:3 w p pt 6 ps var,
➥                 "" u ($1+0.25):($2-.25):4 w labels
```

Both styles require a third column as part of the `using` declaration, the contents of which are interpreted as labels or desired symbol sizes, respectively. Variable symbol sizes are most easily recognized if the symbols are circles, which are chosen by `point-type 6` (`pt 6` for short). Then follows the `pointsize variable` (abbreviated `ps var`) specification. Labels are chosen using `with labels`, and all the labels are offset a little down and to the right so they don't overlap with any of the circles.

Listing 6.1   Data for figures 6.9 and 6.10 (file: labels)

```
# x      y       size    label
1       2.6     3       ABC
2       2.1     6       EFG
3       1.0     2       PQR
4       1.2     1       UVW
5       1.6     4       XYZ
```
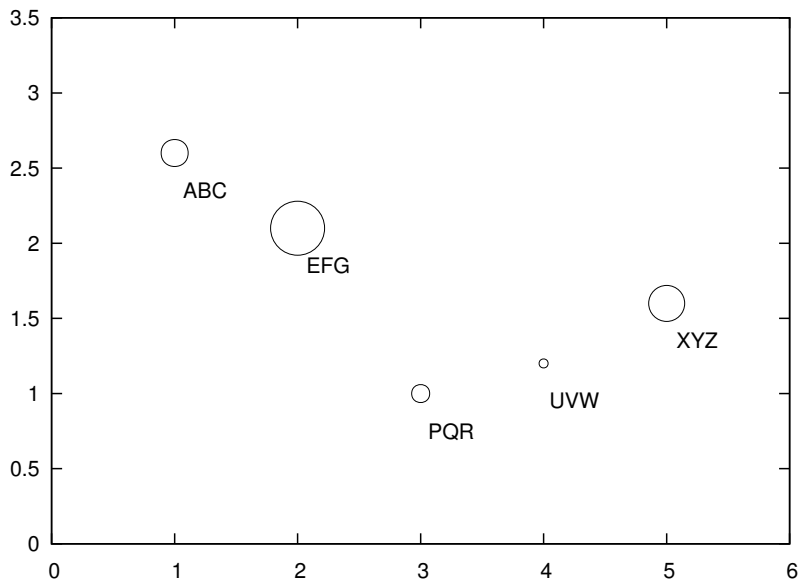

Figure 6.9   Encoding additional information through symbol size or textual labels: `pointsize variable` and `with labels`. The corresponding data file is shown in listing 6.1.

You should exercise some caution when using `pointsize variable`: it's often difficult for the observer to judge the size of symbols accurately. Moreover, it's not necessarily clear to the observer whether the radius or the area of the symbol is proportional to the encoded quantity. I'll have more to say about visual perception in chapter 14.

In contrast, the `with labels` style is frequently useful and very versatile. As mentioned earlier, many of the options of `set label` carry over to `with labels`; see section 7.3.3 for details. Here are only a few remarks:

- You can rotate the label using `rotate by`.
- You can change the text font and its size using `font "..."`.
- You can select the text color with `textcolor` (`tc`). Data-dependent text coloring is available through `tc var` (see section 9.1.5 for more on data-dependent color selection).
- You can draw a point symbol using `point` and shift the text label relative to it using `offset`.
- If the `hypertext` directive is included, the text label is usually hidden and is displayed only when you *hover* with the mouse near the point's location. (Obviously, this is relevant only for interactive terminals.)

Figure 6.10 shows some of these options in action—the data is again that shown in listing 6.1. The `plot` command is as follows:

```
plot [0:6][0:3.5] "labels"
         u 1:2:(sprintf("{/=%d %s}",5*column(3), strcol(4))):1
         w labels rotate by 30 point pt 5 offset 1,-1 tc var
```
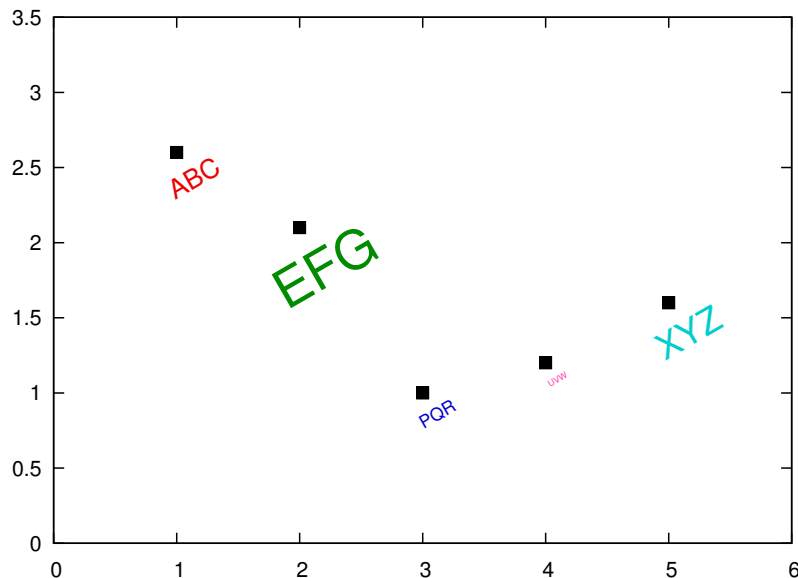

Figure 6.10 Further capabilities of the `with labels` style: plotting a point symbol together with the text label, and changing the color, size, and orientation of the label

Mind you, I'm not saying this is an example of a successful visualization! But it does demonstrate many of the concepts just introduced. (The variable text size uses *enhanced text mode*—I'll formally introduce this concept in section 10.2.3.)

A final word of warning: in contrast to what's customary elsewhere, the `with labels` style directive can't be abbreviated. Neither `with lab` nor `with label` will work!

SHAPES AND ARROWS

Gnuplot defines three shapes that can be used as plotting symbols: arrows, circles, and ellipses. What's different about these plot elements is that they have a nontrivial *internal structure* that can be used to encode information: the arrow has not only a position (as does any other point symbol) but also a direction and a length. An ellipse has a variable shape and orientation; and circles are interesting, because gnuplot lets you draw arc segments in addition to full circles. (As I pointed out earlier, these styles share many properties with `set arrow` and `set object`—see sections 7.3.2 and 7.3.4 for additional details.)

As an example, consider the short data file shown in listing 6.2. In listing 6.3, I use the first two columns as x and y coordinates and display the data in the third and fourth columns as size and angle, respectively. (Compare figure 6.11.)

Listing 6.2   Data for figure 6.11 and listing 6.3 (file: shapes)

```
# x     y       size    angle
1       1       0.1     0
2       1.2     0.2     15
3       1.8     0.3     45
4       1.7     0.2     90
5       1.6     0.15    160
6       1.5     0.2     250
7       1.4     0.3     300
```

Arrows are drawn using the `with vectors` style. They require four columns in the `using` directive: x and y position, followed by the offset of the arrow's tip from its starting position. You can control the way the arrow is drawn either by giving explicit style details inline (see section 7.3.2 to learn how to control an arrow's appearance) or by referencing a separately defined arrow style (see section 9.4.3 for globally defined styles). With `arrowstyle variable`, you can also read the index of a predefined arrow style from an additional column:

```
plot ... using 1:2:3:4 with vectors nohead              ⟵  Inline
plot ... using 1:2:3:4 with vectors arrowstyle 3           ⟵  Global style
plot ... using 1:2:3:4:5 with vectors arrowstyle variable    ⟵  From file
```

Circles require only two columns, for the x and y positions. If you specify a third column, it's used for the radius; otherwise the radius is taken from the global `set style circle` setting. (The radius is always measured in x coordinates.) The fourth and fifth columns, if present, measure the beginning and ending angle of the drawn arc segment (the "pie slice"). Both are measured in degrees, counterclockwise, against
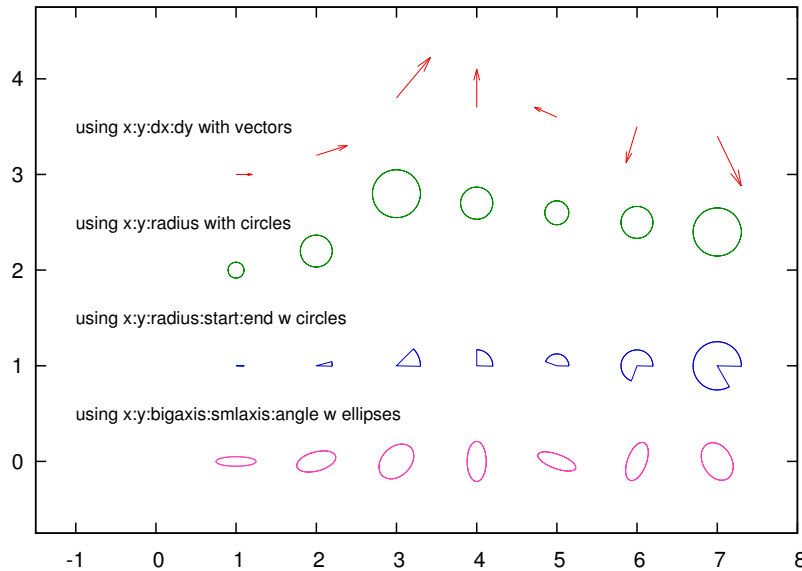
Figure 6.11 The data from listing 6.2, plotted using `with vectors`, `with circles`, and `with ellipses`. See listing 6.3 for details of the `plot` command.

the positive x axis. The circle (or circle segment) can be filled using either an inline `fillstyle` specification or the global `set style fill` setting. Finally, the color of the entire shape may be read from an additional column, using `linecolor variable`. Keep in mind that the resulting shape is *always* a circle, independent of the selected y range. If you want to control the ratio of the horizontal to the vertical axis, you must use `with ellipses`.

Ellipses require four columns, for the x and y coordinates of the center and for the length of the major and minor diameters. You can use a fifth column to give the angle of the ellipse with the positive x axis. Fill and color properties are the same as for circles. (There are some fineries about the units in which the diameters of an ellipse are measured. Check the standard gnuplot reference manual for details.)

Let's now discuss the commands used to create figure 6.11, as shown in listing 6.3. This listing is worth studying in detail, because it demonstrates how you can adapt plot styles to a given purpose through judicious application of inline transformations.

Because the various shape styles expect angles in degrees (not radians), you first need to make sure trigonometric functions will interpret their arguments accordingly by using the `set angle` option (for more information, see appendix F). The data set is then plotted several times, shifted vertically. Basic trigonometric relations are used to calculate the *offsets* of the arrow's tip from the angle and the radius. In the first row of complete circles, the command ignores the angle argument in the fourth column. For both the incomplete circles and the ellipses, the y coordinate is ignored instead, and all shapes are plotted at the same vertical elevation.

> **Listing 6.3   Commands for figure 6.11 using data from listing 6.2 (file: shapes.gp)**

```
set angle degrees

plot "shapes" u 1:($2+2):(2*$3*cos($4)):(2*$3*sin($4)) w vectors,
➥      "" u 1:($2+1):($3) w circles,
➥      "" u 1:(1):($3):(0):4 w circles,
➥      "" u 1:(0):(0.5):($3):4 w ellipses
```

6.4   *Putting it together*

This section combines everything you've learned in this chapter to create figure 6.12. Imagine that you've been given a set of daily weather observations (the beginning of the file is shown in listing 6.4). It includes the day of the month followed by the temperature (in Celsius), the dew point,[3] the cloud cover (in percent), and visibility (in miles, where 10 miles means unrestricted visibility). The last four columns give the wind speed (in knots), wind direction, air pressure (in millibars), and the pressure's short-term trend (up or down). That's a lot of data, and we'd like to present it both compactly and intuitively.
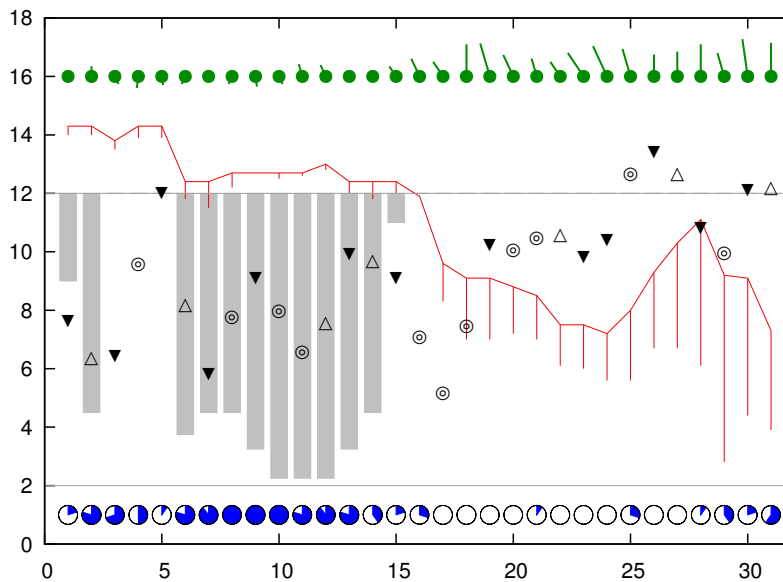
Figure 6.12   All-in-one visualization of the data set in listing 6.4: temperature and dew point as solid lines, atmospheric pressure and its trend with symbols, visibility with grey bars; wind speed and direction across the top, cloud cover along the bottom. See listing 6.5 for the commands.

---

[3]   The *dew point* is the temperature at which water vapor in the air begins to condense. The dew point is always lower than or equal to the actual temperature, and the closeness of those two temperatures is a measure of the humidity in the air.

Listing 6.4   Incomplete data for figure 6.12 with commands in listing 6.5 (file: weather)

```
# Day    Temp    Dew     Cloud   Visib   Wind    Direct    Press    Trend
1        14.3    14      20      7       0       0         1007.6   dn
2        14.3    14      80      2.5     7       0         1006.4   up
3        13.8    13.5    70      10      6       150       1006.4   dn
4        14.3    13.9    50      10      8       190       1009.6   --
5        14.3    13.9    10      10      6       170       1012     dn
6        12.4    11.8    80      1.75    6       210       1008.2   up
7        12.4    11.5    90      2.5     3       190       1005.8   dn
...
```

In figure 6.12, temperature and dew point are shown using continuous lines, pressure is shown using unconnected symbols (in millibars above 1,000), and the symbols indicate the most recently observed tendency: light triangles pointing upward for rising tendency; and dark, downward triangles for falling tendency. Visibility (or, rather, restrictions to visibility) are shown as grey bars in the background—the shorter the remaining white area at the bottom of the graph, the shorter the visible range. Finally, wind is shown with vectors along the top, and cloud cover is shown using circles along the bottom. The complete commands are given in the following listing.

Listing 6.5   Commands for figure 6.12 using data from listing 6.4 (file: weather.gp)

```
set angle degrees
unset key                           Disables whiskers
                                    on the error bars
set bar 0                                              Fixes the width of
set boxwidth .7                                        the candlesticks

sym(s) = s eq 'up' ? "△" : (s eq 'dn' ? "▼" : "◎")
                                                              Visibility ①
plot [0:32][0:18]
"weather" u 1:($5+2):($5+2):(10+2):(10+2) w cand lc rgb "grey" fs solid,
➡  "" u 1:2:3:2 w yerrorlines pt 0 lt 1,                                ②
➡  "" u 1:(1):(0.4):(90-0.01*$4*359):(90) w circ lc rgb "blue" fs solid,
➡  "" u 1:(1):(0.4) w circ lc rgb "black",                      Temperature
➡  2 lc rgb 'dark-grey', 10+2 lc rgb 'dark-grey',     Horizontal   and dew
➡  "" u 1:(16) w p lt 2 pt 7 ps .75,              ④  dividers     point
➡  "" u 1:(16):(.05*$6*cos(90-$7)):(.05*$6*sin(90-$7))
➡     w vectors lw 2 lt 2 nohead,
➡  "" u 1:($8-1000):(sym(stringcolumn(9))) w labels
```

Cloud cover ③

Wind speed and direction ⑤   Symbols for trend ⑥

This set of commands uses many of the concepts introduced previously. The `plot` command in this example is unusually complicated (including nine rather different quantities!). Here are a few pointers as you step through it.

   This example uses the `candlestick` style to draw the grey bars that indicate the visible range ①. This style is convenient because it draws a box, all dimensions of which you can control explicitly. The whiskers have zero length.

Gnuplot draws items onto the graph in the order in which they appear in the plot command. For this reason, it's important that the visibility comes *first*: the grey bars should be in the background and not obscure any of the other plot elements.

The yerrorlines style comes in handy to draw the temperature and the dew point ❷.

Next come the circle segments for the cloud cover ❸. There is a little math to account for the fact that gnuplot measures angles counterclockwise from the horizontal axis, but angles should increase clockwise from the 12 o'clock position in this graph. The filled pie slices are drawn first; the empty black circles that provide a rim for each symbol are drawn afterward, so they're visually in front.

Two horizontal lines are included to divide the graph visually ❹. This part of the plot command could also have come earlier or later.

Now the wind speed and direction indicators are added ❺. First a circular symbol (pointtype 7) is drawn to indicate the origin of each vector—this is important if the wind speed is so low that the vector has zero length. Next comes the vector, again adjusting the angles to be measured clockwise from the top rather than counterclockwise from the right. The arrow head is suppressed using nohead: the origin of each arrow is already clearly marked, so there's no need to clutter the graph by drawing explicit heads.

Finally, the symbols are drawn for the pressure tendency ❻. This part of the command exploits gnuplot's ability to use a single character as plotting symbol. The symbols can be found in the "Geometric Shapes" section of the Unicode standard: the light upward triangle has code point U+25b3, the filled downward triangle corresponds to U+25bc, and the bullseye character has code point U+25cd. The latter was chosen to be as neutral as possible compared to the other two symbols (no angles and of intermediate visual weight, due to the inner circle on an unfilled background).

The sym(s) function consists of two nested conditional operators and converts the strings used in the data file to the characters used in the plot. You need to use the stringcolumn(i) function here—the $9 shortcut doesn't work in this evaluation context. Also remember that stringcolumn(i) takes a numeric argument: the position of the column from which a value should be taken. (See section 4.4 for more information about stringcolumn(i) and the sidebar in section 5.1 for more information about Unicode.)

Figure 6.12 is certainly impressive, and so are the details of listing 6.5—but remember that most graphs (and commands) are much, much simpler! More important is the question of whether figure 6.12 *succeeds* in its visualization task. Here are some things to think about:

- The graph packs a lot of information into a relatively compact space. The middle section is shared by three totally different quantities, but because they're represented using different visual cues (lines, symbols, and filled regions), their interference is reduced. The amount of information encoded in the different

symbols used to represent pressure readings seems too high: I find it difficult to get an intuitive sense of the behavior of the air pressure from this graph.

- The dew point is represented particularly well, I think. The absolute value of the dew point is less relevant than its distance from the temperature, and it's precisely this distance that is emphasized by the vertical error lines. The continuous lines used for the ambient temperature highlight the actual values.

- For the visibility data, I wanted to stress the *reduction* in visibility, represented by the grey bars intruding from above, possibly leaving only a small, unobscured visible region at the bottom of the graph. Although good in concept, I don't think this idea works well in practice: drawing the visual range `with lines` would have been clearer.

- Both the wind and the cloud cover visualizations represent their underlying data intuitively: you can, literally, see which way the wind blows. It would have been weird to use `with lines` to indicate the wind direction.

  Nevertheless, the visualization of the wind speed and direction has a couple of problems: although you have a relative sense of the wind speed, there's no indication of its numerical range—if you're trying to land a plane, that matters a lot! And if the wind speed is high and the direction is variable, the arrows will have a tendency to obscure each other. (That this isn't an issue here is solely a consequence of this particular data set.)

- One last problem: although the example uses good numerical tic marks for the temperature and dew-point values, things are less clear for the visibility and pressure readings. A single axis isn't enough to represent quantities that have such totally different numerical ranges.

6.5   *Other styles*

The styles discussed in this chapter are all very general: they can be used to represent all sorts of data in a variety of situations. In addition to these general-purpose styles, gnuplot offers further possibilities for special applications:

- Appendix E explains boxplots (to represent point distributions), histograms (for counting statistics), and parallel coordinates (a particular kind of multivariate technique).

- Appendix C and appendix D introduce surface and contour plots, as well as false-color plots and heatmaps. All of these methods apply to cases where you'd like to represent how one dependent quantity depends on *two* independent variables.

- Appendix F introduces some special plot types for non-Cartesian coordinates.

6.6   *Summary*

This chapter is like an illustrated catalog: it demonstrates all of gnuplot's major plot styles by way of example. As such, I'm not disappointed if you didn't read the chapter

from start to finish: it's fine to look at the figures until you find one that does what you need and then read the accompanying section.

The only part of this chapter that you *should* have read, even on a first pass, is section 6.2, because it explains how styles are chosen and how you can customize them to meet your needs. This is fundamental material that you'll need basically every time you use gnuplot.

Everything in this chapter exists to visualize data. In contrast, chapter 7 looks at things you may want to put on a plot *in addition to* the data: labels, arrows, and other decorations.

Decorations: labels, arrows, and explanations

7

This chapter covers

- Understanding layers and locations on a graph
- Adding arrows, labels, and other decorations
- Providing explanations using a key
- Changing a graph's overall appearance

Data alone doesn't tell a story. To be useful, the data needs to be placed in context: you must tell the observer what the data is (such as position versus time, particle count versus scattering angle, stock price versus date, and so on) and what units the data is plotted in (centimeters or inches, seconds or minutes, dollars or Euros). No plot is complete without this information.

But you can do much more to make a graph useful and informative: you can add arrows and annotations on the graph to point out and explain interesting features. You may also want to provide special tic marks and labels to make quantitative information stand out more. Or you can change the size and shape of the entire graph to accommodate a specific data set.

In this chapter, we'll discuss all the means that gnuplot offers to put ancillary information on a plot (in addition to the data). Because much of this material is quite dry, I've gathered the most important commands and options together in the next section. Unless you have specific requirements, this may be all you need to know right now—gnuplot is good at automatically doing the right thing in most situations.

7.1   *Quick start: minimal context for data*

For the sake of concreteness, let's go back to the diffusion limited aggregation (DLA) example of section 1.1.2 and in particular figure 1.4. We'll use this plot as our example throughout this section.

The absolute quickest way to add the most important contextual information to the plot is to give it a title, such as

```
set title
➥ "Run Time (in seconds) vs. Cluster Size (in thousands of particles)"
```

This tells the observer both what is plotted (computation time versus system size) as well as the relevant units (time is given in seconds, and particles are counted by thousands), conveniently packaged in a single command. This is the bare minimum.

The title is centered at the top of the graph (as in figure 7.1—you may want to go back to section 1.1.2 to learn about the context in which this example arose). With a little additional effort, you can put labels on the axes using the `set xlabel` and `set ylabel` commands. This frees up the title for contextual information about the data in the plot, like so:

```
set xlabel "Cluster size [thousands]"
set ylabel "Running time [sec]"
set title
➥ "DLA Growth Program: Run Time vs Cluster Size (Double Logarithmic)"
```

Finally, there's the key (or legend), which relates line types to data sets. By default, it's placed in the top-right corner of the graph, but this may not be suitable if it will interfere with data. You can change the position of the key using the keywords

```
left, right, top, bottom, center
```

in the `set key` command: for example, `set key right center`, to place the key vertically centered on the right side of the graph. The order of the keywords doesn't matter, and if only one is given, all the other options retain their values. Finally, you can suppress the key using `unset key`.

You saw in section 2.1.2 how you can change the string describing the data set in the key using the `title` option to the `plot` command. To suppress an entry in the key, you can either use the `notitle` option or provide an empty title string. In the following command, the data curve has an entry in the key, but the fitted curve doesn't:

```
plot [1:100][0.1:] "progress" u 1:2 title "Data" w p, (x/2.5)**3 notitle
```
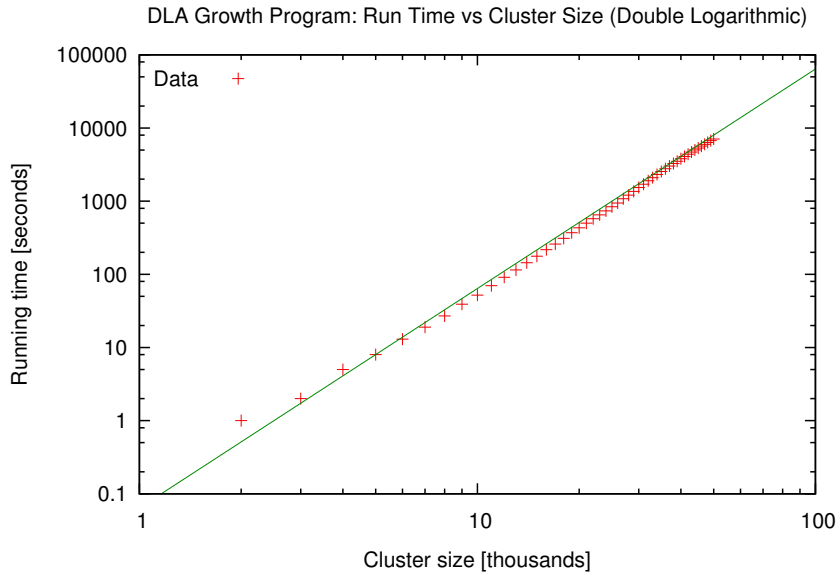
Figure 7.1 Providing minimal context for a plot using a title and axes labels. See listing 7.1.

Now let's put all this together. The following listing summarizes the commands used to build figure 7.1.

Listing 7.1   Commands for figure 7.1 (file: progress.gp)

```
set xlabel "Cluster size [thousands]"
set ylabel "Running time [seconds]"
set title
➡ "DLA Growth Program: Run Time vs Cluster Size (Double Logarithmic)"

set key top left
set logscale

plot [1:100][0.1:] "progress" u 1:2 title "Data" w p, (x/2.5)**3 notitle
```

This concludes our quick start, and it may be all you need for now. The rest of this chapter discusses the miscellaneous options that are available if you aren't happy with the defaults. But first I need to describe the different ways you can refer to a location on a plot.

> **TIP**   You can use `set xlabel` and `set ylabel` to tell the viewer what quantities are plotted along the horizontal and vertical axes. The `set title` command is an easy way to add general context to a plot. And don't forget the `plot ...` `title "..."` facility to add a meaningful string to the plot's legend.

7.2   *Understanding layers and locations*

To add decorations to a plot, you must be able to tell gnuplot *where* on the graph the new elements should be placed. Two different concepts are involved. In the next section, I'll explain how to specify positions laterally *within* the plane of the graph. Then, you'll learn about the *stacking order* that enables you to place elements visually in the foreground or background relative to each other.

7.2.1   *Locations*

First, let's establish some terminology. The entire area of the plot is referred to as the *screen* or the *canvas*. On the canvas is the *graph*, which is surrounded by a *border* (unless you explicitly turn off the border). The region outside the border is called the *margin*. All these are indicated in figure 7.2.
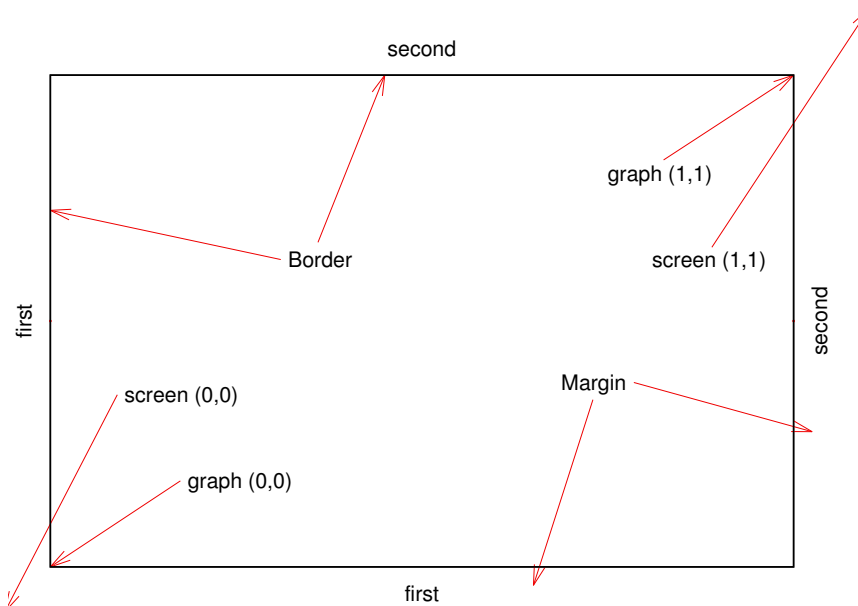
Figure 7.2   The parts of a gnuplot graph: canvas, borders, and margins

You can provide up to two different sets of axes for a plot. This is occasionally useful when comparing different data sets side by side: each data set can be presented with its own coordinate system in a single graph. The primary coordinate system (named `first`) is plotted along the bottom and left borders. The secondary coordinate system (`second`) is plotted along the top and right borders. By default, the secondary system isn't shown; instead, the primary system is displayed on all four borders. (You'll find more information on axes and coordinate systems in chapter 8.)

Now that you know what all parts of a graph are called, we can talk about the different ways to specify locations. Gnuplot uses *five* coordinate systems:

```
first, second, graph, screen, character
```

The first two refer to the coordinates of the plot itself. The third and fourth (`graph` and `screen`) refer to the graph area and the entire canvas, respectively, placing the origin (0,0) at the bottom-left corner and the point (1,1) in the top-right corner (see figure 7.2). Finally, the `character` system gives positions in character widths and heights from the origin (0,0) of the screen area. Obviously, positions in this last coordinate system depend on the font size of the default font for the current terminal.

Coordinates are given as pairs of numbers separated by a comma. As necessary, each number in this pair can be preceded by one of the five coordinate specifiers. The default is `first`, and if no coordinate system is given explicitly for y, the one for x is used for both values.[1]

You'll see many examples of coordinate specifications in the rest of this chapter.

7.2.2 *Layers*

Gnuplot draws graph elements onto the canvas in a well-defined sequence. You can use this stacking order to place some elements visually "in front of" or "behind" other elements.

Plot elements are organized into *layers* (see figure 7.3). Graph elements in a layer that's closer to the front obscure elements in layers further back.
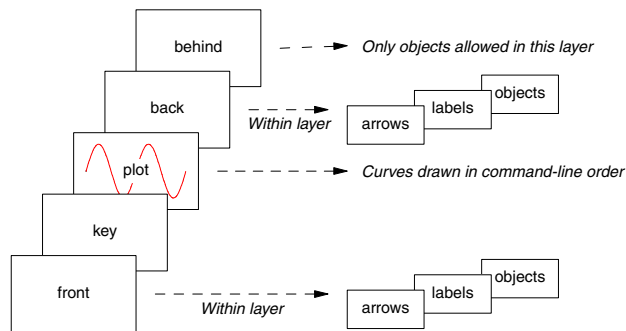
Figure 7.3   Stacking order of layers, front to back. Within each layer, objects are drawn first, followed by labels and arrows. Curves are drawn in the order in which they appear in the `plot` command.

Labels, arrows, and geometric shapes (*objects*) are organized in two layers, one of which is (visually) behind the plotted data; the other one is on the top of the visual stack. When creating a decoration, you can assign it to either of these two layers using the keywords `front` and `back`; the default is `back`. Within each layer, arrows are visually in front of labels, and geometric shapes are even further behind. As you'll learn in the next section, each arrow (or label or object) has a numeric tag, and this tag is used to sort elements of the same type: arrows with lower tag numbers are drawn first, and all arrows are drawn before any text labels are drawn.

A special layer, identified using the keyword `behind`, resides at the bottom of the visual stack. Only objects (but no arrows or labels) can be added to this layer. You can

---

[1] Although it may seem weird to use different coordinate systems in the same coordinate pair, this feature is occasionally useful. For example, you can use the following command to place a vertical line at a fixed x location and let it extend over the full range of the graph, independent of the y range: `set arrow from 1.25, graph 0 to 1.25, graph 1`.

use this layer to provide a visual background (such as a solid color or a color gradient) to the entire graph.

Data and functions are plotted in the order in which they appear in the `plot` command, left to right. This means elements specified *later* in the `plot` command appear *in front* of those given earlier.

Whereas the locations within the plane of the graph are essential when working with decorations, the stacking order becomes relevant only for complicated graphs consisting of many plot elements. For most day-to-day operations, you probably won't need to concern yourself with it. But you have control over the visual stack for those special situations when you need it!

7.3 *Additional graph elements: decorations*

I use the term *decorations* for all additional graphical elements that can be placed on the graph but that don't (primarily) represent data. The most important decorations are text labels, closely followed by arrows, but gnuplot can also draw rectangles, circles, ellipses, and polygons.

7.3.1 *Common conventions*

Labels, arrows, and objects share some conventions that apply to all of them equally. I'll summarize these conventions here so I don't need to repeat this information for each of the individual styles.

CREATING DECORATIONS

All decorations are created using the `set ...` command (see chapter 5). It's very important to remember that this command does *not* generate a replot event: the decorations won't appear on the plot until the next `plot`, `splot`, `replot`, or `refresh` command has been issued!

Keep in mind that decorations aren't taken into account by the autoscale feature, which automatically attempts to adjust the plot ranges in such a way as to display the relevant parts of the data. In other words, if a decoration (or part of it) falls outside the plot range (either explicitly given or automatically selected), it will *not* appear on the plot. (More on autoscaling in section 8.2.)

> **TIP** Decorations created with `set ...` won't appear on the graph until the entire plot is redrawn through a `plot`, `replot`, or equivalent command.

IDENTIFYING DECORATIONS WITH TAGS

So that you can later refer to a specific object (such as an arrow or a text label), you can give the object a numeric tag: for instance, `set arrow 3 ....` Now you can make changes that apply only to this arrow by providing the tag in the next call to `set arrow`; or you can eliminate the arrow using `unset arrow 3`. Arrows, labels, and objects have separate counters.

If you omit the label, gnuplot will assign the next unused integer automatically (when using `set ...`) or apply the command to *all* instances of the object (when using

unset ...). The command unset arrow, without an explicit tag, removes all arrows from the graph!

You can see all defined arrows, together with their location and appearance options, by using show arrow and including a tag (for example, show arrow 3 limits the command to the corresponding element). Labels and objects can be listed using equivalent commands. Because decorations are drawn onto the plot in the order of their tag number, you can control visual stacking through appropriately assigned tag numbers.

> **TIP** When you're using gnuplot interactively, it's usually a good idea to assign numeric tags explicitly and not rely on the automatically generated tags. Almost invariably, you'll want to modify a graph element created earlier, and always using an explicit tag number ensures that you only edit the one element you intend to.

7.3.2 *Arrows*

Arrows are generated using the set arrow command, which has the following options:

```
set arrow [{idx:tag}] from {pos:from} to {pos:to}
set arrow [{idx:tag}] from {pos:from} rto {pos:offset}
set arrow [{idx:tag}] from {pos:from} length {pos:len} angle {flt:degrees}

set arrow [{idx:tag}] [ nohead | head | heads | backhead ]
                      [ size {flt:length} [,{flt:angle}] [,{flt:backangle}]
                        [fixed] ]
                      [ nofilled | empty | filled | noborder ]
                      [ front | back ]
                      [ lineoptions ]

set arrow [{idx:tag}] [ arrowstyle | as ] {idx:style}
```

Each arrow has a beginning and an end point. The starting point is always given explicitly, as a coordinate pair in any of gnuplot's coordinate systems, but the end point can be given in three different ways (see the first three lines in the previous snippet):

- As an explicit end point coordinate using the to keyword. Example: set arrow from 0,0 to 2,1.
- As a relative offset from the starting point, using rto. The horizontal and vertical length of the offset are measured in units of the x and y axis, respectively.[2] Example: set arrow from 0,0 rto 2,0 draws a horizontal arrow that is two x-axis units long.
- As a length and an angle. The length is measured in units of the horizontal axis but can be preceded by any of the permissible coordinate system specifiers. The angle is given in degrees and measured counterclockwise from the positive x axis. Example: set arrow from 0,0 length 2 angle 45.

---

[2] For logarithmic coordinates, the relative offset is multiplicative, not additive.

Many aspects of the arrow's appearance can be changed using the `set arrow` command. These specifications can be given either explicitly or by referencing a previously defined arrow style (using the `arrowstyle` keyword). In the next section, I'll explain the different appearance options; we'll come back to predefined styles in chapter 9.

Appearance options can be either specified separately or combined with the coordinates in a single call. The following two code snippets show two equivalent ways to define the same arrow:

```
set arrow 1 from 0,0 to 2,1
set arrow 1 nohead front

set arrow 1 from 0,0 to 2,1 nohead front
```

CUSTOMIZING ARROW APPEARANCE

You can customize the appearance of an arrow, in particular the size and shape of its head (or heads) and the look of its "backbone":

- *Heads*—By default, a head is drawn at the destination endpoint. Heads can be suppressed entirely using `nohead`. The keyword `heads` draws heads at both ends; `backheads` draws only a single head at the start point.

- *Head size and shape*—You can change the size and shape of the head using the keyword `size` followed by the length and the angle at the arrow tip, and—optionally—the angle at the back of the head (see figure 7.4). Unless otherwise specified, the length is measured in units of the primary horizontal axis; this can be changed using any of the coordinate system specifiers (see figure 7.2). All angles are measured in degrees; the back angle is ignored if the fill style is `nofilled` (which is the default).[3]

  By default, the size of the arrow head is reduced for very short arrows; this can be disabled using `fixed` immediately following the size.

- *Head color and fill style*—By default, arrow heads aren't filled and the back angle is ignored. The keyword `empty` draws only the outline, `filled` draws the outline and fills it, and `noborder` draws the filled area but not the outline. This matters, because the outline *extends*, by one line width, beyond the end position of the arrow; hence you may need to suppress the outline. It also matters if you're using a dashed line for the arrow and want to suppress a dashed outline for the head.

  You can't choose the fill color separately from the outline color. If you want a red-filled arrow head with a black outline, you need to construct it from *two* arrows, one on top of the other.

- *Layer assignment*—Arrows can be assigned to the `front` or `back` layer.

- *Line options*—All available line options can be used. They apply to the arrow's backbone and its head or outline.

---

[3]   The default values for the head size are `size screen 0.0225,15,90`. Using the `screen` coordinate system guarantees that the head size doesn't change if the plot range does.
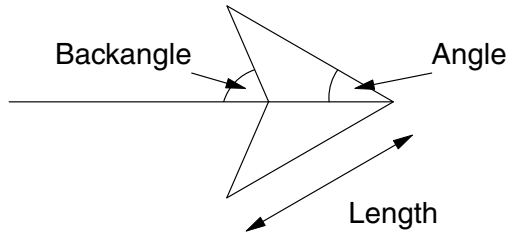
Figure 7.4 The parameters that control the shape of an arrowhead

Some examples of custom arrow heads are shown in figure 7.5.
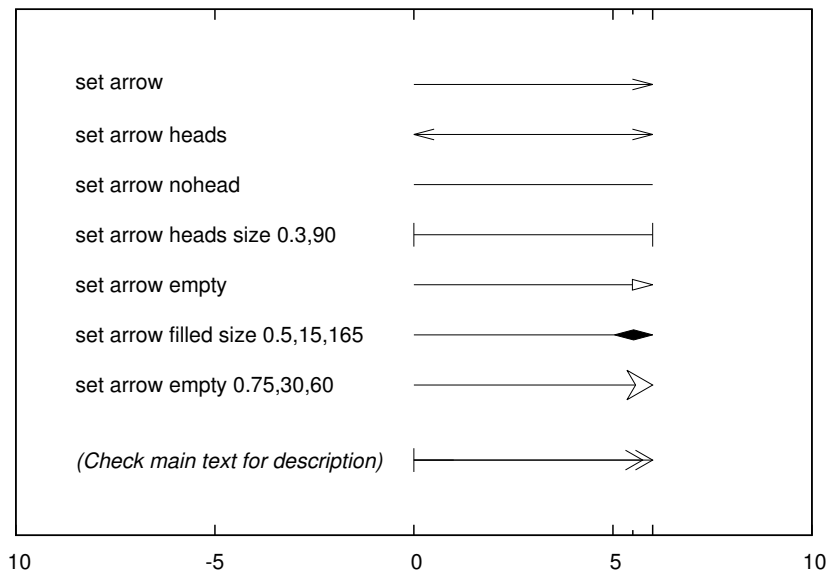
Figure 7.5 Different arrow forms and the commands used to generate them. See listing 7.2 for the last one.

MORE TRICKS WITH ARROWS

The arrow facility is versatile and can be used for purposes other than placing explicit arrows on a plot. Here are some ideas:

- Because an arrow without a head is just a straight line, you can use `set arrow` . . . to draw arbitrary straight lines on a plot. In particular, arrows can be used to draw *vertical* lines. (You don't need arrows to draw *horizontal* lines, of course; it's generally easier to use `plot 1, -1` to draw horizontal lines at y = -1 and y = +1.)
- The ability to mix different coordinate systems in the same coordinate specification can come in handy in this context. For example, if you want to draw a vertical line at x = 0.5 from the lower to the upper boundary, you can say `set arrow`

from 0.5, graph 0 to 0.5, graph 1 without having to worry about the exact values of the vertical plot range.

- You can use short arrows without heads to draw custom tic marks, if gnuplot's built-in tic marks are insufficient (see chapter 8 for more information on tic marks).

- Arrows with customized heads can be used to indicate a scale or range on a graph—figure 7.8, later in the chapter, contains an arrow used for this purpose.

- You can generate an arrow with two *different* heads by generating two single-headed arrows back to back. You can also overlay arrows to generate more sophisticated effects. The following listing shows the commands used to generate the bottom arrow in figure 7.5.

Listing 7.2   Commands to generate the double-feathered arrow in figure 7.5

```
set arrow 1 from 0,-9 to 6,    -9 size 0.5,30
set arrow 2 from 0,-9 to 5.75, -9 size 0.5,30
set arrow 3 from 0,-9 to 1,    -9 backhead size 0.3,90
```

As you've seen, arrows are a versatile graph element. But more often than not, you'll want to combine them with some textual explanation. That's what labels are for.

7.3.3   *Text labels*

Text labels are a natural companion to arrows. The arrow shows the observer where to look, and the label explains what's happening:

```
set label [{idx:tag}] [ "{str:text}" ] [ at {pos:location} ]
                       [ left | center | right ]
                       [ rotate [ by {int:degrees} ] | norotate ]
                       [ [no]boxed ]
                       [ front | back ]

                       [ point lt|pt {idx:pointtype}
                               | ps {flt:pointsize} | nopoint ]
                       [ offset {pos:off} ]
                       [ hypertext ]

                       [ noenhanced ]
                       [ font "{str:name}[,{int:size}]" ]
                       [ textcolor | tc [ {clr:colorspec}
                                        | lt {idx:type} | ls {idx:style} ] ]
```

The label text is typically a constant, but it can also be a string variable or any string-valued expression. To obtain a formatted string, the sprintf() and gprintf() functions are often useful. For example, to create a label (at the origin) showing the value of π, truncated to two digits, you can use set label at 0,0 sprintf("%.2f", pi). (See section 5.1 for more information about string operations, in particular table 5.1.)

By default, the text is placed flush left at the position specified by the `at ...` location. This can be changed using the `left`, `center`, and `right` keywords.

Gnuplot allows text to be rotated, but not all terminals support arbitrary angles. Use the `test` command to see what's possible (see section 10.1.3). If the text hasn't been rotated, you can draw a frame around it using `boxed`. The stacking order can be controlled with `front` and `back`, as for arrows.

Using `point`, you can place a symbol at the position specified with `at`; the text label is shifted relative to it. The point style and size of the symbol can be fixed using `lt` (line type), `pt` (point type), and `ps` (point size), and the offset can be customized using `offset`. The `nopoint` option suppresses the point.

The `hypertext` option creates a label that is visible *only when the mouse hovers* near the location of the label—specifically, near the symbol placed at the location of the label. (The `hypertext` option can only be used together with the `point` option.) The `hypertext` option isn't available for all terminals. Currently, only the (interactive) `wxt`, `qt`, and `windows` terminals support it, as well as the (browser-based) `svg` and `canvas` terminals.

You can select the color of the text using `textcolor` (abbreviated `tc`), followed by either a color specification (see chapter 9) or the keyword `linetype` (`lt`) and the index of a predefined line type. In this case, the color of the line type is applied to the text.

Finally, you can choose a specific font using the `font` option. The keyword `noenhanced` suppresses the interpretation of enhanced-mode control characters in the text string, even if enhanced mode is active for the current terminal. (Read chapter 10 for an in-depth discussion of font selection and enhanced text mode.)

In addition to the fully general `set label` facility that I just described, gnuplot offers two specialized labeling commands: to add a label to any axis of the plot, or to add a title to the entire graph. Both provide convenient shortcuts for common situations.

TITLE

The `title` option is a `label` with some special defaults. For instance, it's automatically placed centered at the top of the graph, and the top margin of the graph is adjusted accordingly.

The `set title` option takes the following arguments, which should be familiar from the discussion of `set label`:

```
set title [ "{str:text}" ] [ offset {pos:off} ]
          [ font "{str:name}[,{int:size}]" ][ noenhanced ]
          [ textcolor | tc [ {clr:colorspec}
                           | lt {idx:type} | ls {idx:style} ] ]
```

Only the behavior of the `offset` directive requires an explanation. It can be used to shift the title from its default position by a specified amount. What's unusual is that by default, it interprets its argument as given in the `character` coordinate system. With this in mind, the following commands are entirely equivalent—both shift the title down by the height of a single character:

```
set title "..." offset 0,-1
set title "..." offset character 0,-1
```

AXIS LABELS

The commands that place labels on the axes of a plot are also variants of the standard `set label` command. Because labels can be placed on any axis, there's an entire family of related commands, all differentiated by prefixes (such as x and y), that indicate which axis a label belongs to. (We haven't discussed these prefixes yet but will do so in chapter 8.) In the following syntax summary, the underscore (_) stands for any of the permissible prefixes:

```
set _label ["{str:text}"] [ offset {pos:offset} ]
                          [ rotate by {int:degrees} | parallel | norotate ]
                          [ font "{str:name}[,{int:size}]" ] [ noenhanced ]
                          [ textcolor | tc [ {clr:color}
                                          | lt {idx:type}
                                          | lt {idx:style} ] ]
```

The options are an almost strict subset of those available for the plain `set label` command. Check section 7.3.3 for details. The only new feature is the `rotate parallel` directive. It applies only to three-dimensional plots (see appendix C) and rotates each label such that it's parallel to its axis (by default, the labels are printed horizontally in three-dimensional plots).

   Finally, keep in mind that both `set title` and the axis label commands are merely standard text labels with some convenient defaults. If they don't give you sufficient control to achieve the effect you're looking for, you can always use explicit `set label` commands instead.

   In addition to arrows and labels, which primarily are intended to add explanations to a graph, gnuplot also lets you place arbitrary graph *objects* on a plot. You do so using the `set object` command.

7.3.4 *Shapes or objects*

The `set object ...` facility can place a variety of geometrical objects on a graph, such as rectangles, circles, ellipses, and even arbitrary polygons. Shapes are different from other graphical elements described so far: they're less suited to represent data, because their coordinates can't be read from a file and instead must be set manually. Shapes also don't add semantic information to a graph in the way arrows and labels do. They introduce functionality to gnuplot that you'd expect in a general-purpose drawing program rather than in a data-visualization tool—but here they are, in case that (special) need arises.

   Because gnuplot's shapes and objects are a bit outside this book's primary goal (which is to help you *understand data with graphs*), I won't discuss them much further, except for some cautionary notes. (See the standard reference documentation for details.)

All objects share a single tag counter, which is separate from the tag counters for arrows and labels; but keep in mind that there is no separate tag counter for (say) circles as opposed to rectangles. All shapes are of type `object`; the particular shape (whether circle, rectangle, ellipse, or polygon) is specified as a property of each individual object. For rectangles, you can define a global style using `set style rectangle`; the other shapes inherit properties from the global fill style, which is defined through `set style fill`. (See section 9.4 for information on global styles.) Finally, let me remind you that elements of type `object` are the only items that can be placed in the layer at the bottom of the visual stack (using the `behind` keyword). This can be useful to give a colored background to the entire graph. (Most contemporary terminals let you specify a background color, which may be a more convenient alternative, provided you want to give a single color to the entire canvas area.)

Arrows, labels, and objects are graphical elements that you can use to make graphs more interesting and more informative. But they aren't the only elements that can go on a plot besides data, and they aren't even the most useful ones. The single most useful element is probably the graph's *legend* or *key*, which explains what all the lines and symbols on the plot stand for. Gnuplot's `set key` facility is very powerful and therefore deserves a section by itself.

7.4   *The graph's legend or key*

We all know the little boxes on maps that explain what the different symbols mean: thick red lines indicate highways, white is for country roads, and thin dashed lines mean unpaved gravel. (And the odd-looking symbol with a roof? Right: that's an outhouse.) In gnuplot, this box is called the *key*.

The key (or legend) explains the meaning of each type of line or symbol placed on the plot. Because gnuplot generates a key automatically, it's the most convenient way to provide this sort of explanation. On the other hand, relying on the key separates the information from the actual data. I therefore often find it preferable to place arrows and labels directly on the graph instead, to explain the meaning of each curve or data set. A separate key excels when there are so many curves that individual arrows and labels would clutter the graph.

As usual, almost anything about the key can be configured. All available options are summarized next:

```
set key [ on|off ] [ default ]

        [ [ at {pos:position} ]
        | [ inside | outside ] [ lmargin | rmargin | tmargin | bmargin ] ]
        [ left | right | center ] [ top | bottom | center ]

        [ vertical | horizontal ] [ Left | Right ]
        [ [no]reverse ] [ [no]invert ]
        [ maxrows [ auto | {int:rows} ] ] [ maxcols [ auto | {int:cols} ] ]

        [ [no]opaque ]
        [ samplen {flt:len} ] [ spacing {flt:factor} ]
        [ width {int:chars} ] [ height {int:chars} ]
```

```
[ [no]title "{str:text}" ]
[ [no]box [ lineoptions ] ]

[ [no]enhanced ]
[ font "str:font,int:size" ] [ textcolor | tc {clr:color} ]

[ [no]autotitle [columnheader] ]
```

As you can see, there are a lot of sub-options! To make information easier to find, I've broken the following explanation into separate sections under their own headings.

7.4.1   *Turning the key on and off*

You can suppress the key using either of the following two commands:

```
set key off
unset key
```

The command `set key on` enables it again.

7.4.2   *Placement*

The key can be placed at any position on the canvas. The most straightforward (but not necessarily the most convenient) method is to fix a specific location using the `at` option. The location can be prefixed by any of the standard coordinate system prefixes. The keywords `left`, `right`, `top`, `bottom`, and `center` align the key relative to the specified position. For instance,

```
set key at 0,0 top left
```

places the top-left corner of the key at the origin of the coordinate system.

As usual, with great power comes great responsibility. When using the explicit `at` option, gnuplot doesn't rearrange the plot in any way to make room for the key: you must do this explicitly. It's usually more convenient to use the predefined keywords that place the key relative to the graph; when you use these keywords, the borders of the graph are automatically adjusted to make room for the key.

The keywords `left`, `right`, `top`, `bottom`, and `center` push the key into the desired position. You can place the key inside the plot using `inside` (this is the default) or on any one of the margins using `lmargin`, `rmargin`, `tmargin`, and `bmargin` (meaning the left, right, top, and bottom margin, respectively). You can also use combinations of position specifiers, such as `set key top left`. The effect of these options is cumulative, so the following two examples have the same effect:

```
set key bottom left
```

or

```
set key bottom
set key left
```

One surprising exception is that the keyword `center` by itself is interpreted as the *center of the graph* (both horizontally and vertically).

7.4.3   *Layout*

The samples in the key can be either stacked vertically or aligned horizontally using the `vertical` and `horizontal` keywords. The alignment of the labels within the key is controlled using the `Left` and `Right` options (note the capitals). The default is `vertical Left`.

Usually, the textual description of the line sample is to the left, with the line sample on the right. The keyword `reverse` interchanges this arrangement.

Entries in the key are made in the order in which the corresponding data sets occur in the `plot` command. If you want to sort entries in a specific way, you need to list the data sets in the appropriate order in the `plot` command. But you can invert the stacking order of the samples in the key using `invert`. This is mainly useful to force the ordering of labels in the key to match the order of box types in a stacked histogram (see section E.4).

In general, the key is built up in its primary direction as far as possible. Assuming that key entries are vertically stacked (which is the default), gnuplot will continue to add lines to the key and only begin a new column when there is no more vertical room on the graph. To change this behavior, use the `maxrows` keyword together with a numeric argument: gnuplot will create a new column in the key whenever the number of rows exceeds the specified value. The `maxcols` keyword has the equivalent effect with a key that is built up horizontally.

7.4.4   *Appearance*

By default, each entry in the key is created at the same time the corresponding line is drawn on the graph. Later plot elements may therefore obscure parts of the key. When the `opaque` keyword is given, the key is drawn *after* the graph is complete; moreover, the key area is cleared before the key is created.

The length of the line sample in the key can be controlled using `samplen`. The sample length is the sum of the tic length (see chapter 8) and the argument given to `samplen` times the character width. A negative argument to `samplen` suppresses the line sample.

The vertical spacing between lines in the key is controlled using `spacing`. The spacing is based on the product of the point size and the argument to `spacing`. This parameter does *not* influence the horizontal spacing between entries in `horizontal` mode.

You can give the entire key a title (which is placed at the top of the key) using the `title` option, and draw a frame around the key using `box`. In this case, you may need to adjust the size of the box using the `width` and `height` parameters. Both take the number of characters to be added to or subtracted from the calculated size of the key before the box is drawn. In particular, if any text label in the key contains control characters for enhanced text mode, the size of the autogenerated box may be incorrect.

Finally, you can adjust the appearance of the lines used to draw the box using the usual line options.

Enhanced text mode (see chapter 10) can be switched on or off for the key, independent from the terminal's behavior. You can set the default font for the key by using the `font` keyword, but keep in mind that these settings can be overruled in the explanations (see the next section).

7.4.5   *Explanations*

The *explanation* is a bit of text that assigns a meaning to each line sample or symbol type included in the key. The string for the explanation can come from one of two places: the `plot` command or the data file. Let's look at both methods.

TAKING EXPLANATIONS FROM THE PLOT COMMAND

Usually, the explanatory text is taken from the plot command, using the `title` keyword:

```
plot "data" u 1:2 title "Experiment" w l, sin(x) title "Theory" w l
```

If no explicit title is set in the `plot` command, by default gnuplot will generate a standard description based on the filename and the selection of columns plotted. If you don't like this behavior, you can turn it off using `set key noautotitle`: now only curves with an explicit `title` keyword in the `plot` command will have an entry in the key. (The command `set key autotitle` restores the default behavior.)

There are two further ways to suppress parts or all of a key entry:

- If an *empty* string is given as argument to the `title` keyword, gnuplot will generate no key entry for the corresponding data set: neither a line sample nor an explanation will be shown.
- To generate a key entry consisting only of the line sample, without a visible explanation, use a string consisting only of *whitespace* as argument to the `title` keyword in the `plot` command.

TAKING EXPLANATIONS FROM THE DATA FILE

Instead of taking explanations for the key from the `plot` command, gnuplot can take them from the first *non-comment* line in the data file. This is the same information used in section 4.4.1 to identify a column in the `using` specification—now it's used for the graph's key, as well.

Although the strings used in the key always come from the first non-comment line in the data file, there are three ways to instruct gnuplot what to do:

- Globally, using `set key autotitle columnhead`
- As part of the `plot` command, via `title columnhead`
- To have maximum control, using the function `columnheader()` as part of an inline transformation in the `using` specification

Let's look at an example. The next listing shows a data file that you've seen before (in listing 4.7). Note how the first line, containing the column headings (which will be used in the key of the plot), isn't a comment line.

Listing 7.3 Data for figures 7.6 and 7.7 (file: grains)

```
Year    Wheat    Barley  Rye              ⊲──── Not a comment line!
1990    8        6       14
1991    10       5       12
1992    10       7       15
1993    11       5       13
1994    9        6       12
```

You already know how to plot a file like this *and* generate proper key entries using explicit explanations in the `plot` command (technically, you need to use `skip 1`, because the first line isn't commented out):

```
plot "grains" skip 1 u 1:2 t "Wheat" w lp, "" skip 1 u 1:3 t "Barley" w lp,
➡    "" skip 1 u 1:4 t "Rye" w lp
```

If you use global style settings instead, the `plot` command can be particularly simple (see figure 7.6):

```
set style data linesp                    ⊲──── A global style setting …
set key autotitle columnhead             ⊲──── … and global key configuration …
plot "grains" u 1:2, "" u 1:3, "" u 1:4  ⊲──── … result in a simple plot command.
```
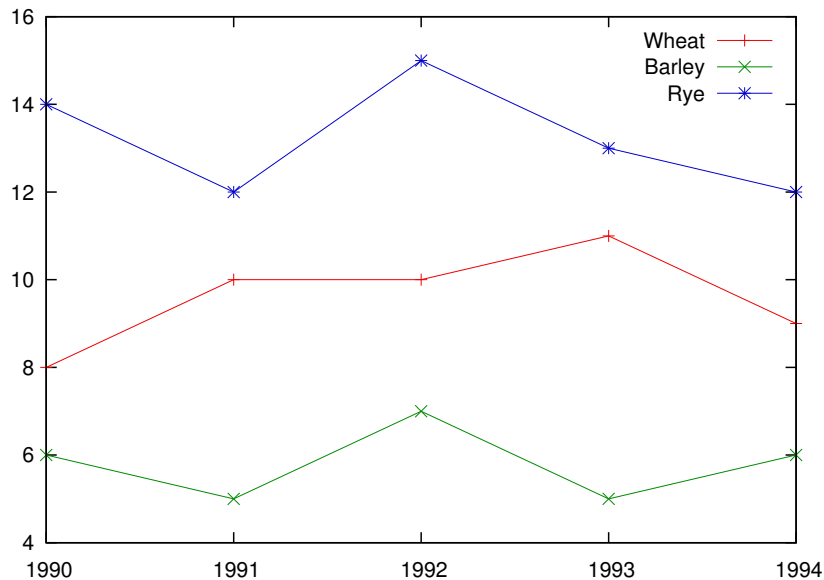

Figure 7.6 Using `set key autotitle columnhead`, key entries are taken from the first non-comment line in the data file. See listing 7.3 for the data file.

Rather than use a global directive, you can tell gnuplot to take the explanation from the data file within the `plot` command. Doing so lets you modify the behavior on a case-by-case basis:

```
plot "grains" u 1:2 t columnhead w lp,
➥    "" skip 1 u 1:3 t "Brown Barley" w lp
➥    "" skip 1 u 1:4 t "" w lp
```

Takes the explanation from the file

Explicitly provided explanation

Suppresses the key entry

Using `title columnhead` as part of the `plot` command instructs gnuplot to take the explanation for that column from the header entry for the *same* column. If you want more control (for instance, because you want to specify a different column, or you're applying an inline transformation that combines multiple columns), you can use the `columnheader()` function.[4] The argument to `columnheader` should evaluate to an integer, which is interpreted as a column number; the function returns the contents of the first non-comment line in this column as a string. This string can either be supplied to the `title` keyword in the `plot` directly or be part of a string expression. The next code snippet shows all these variants:

```
plot "grains" u 1:2 t columnhead w lp,         ⟵  Same column
➥    "" u 1:3 t columnhead(3) w lp,            ⟵  Arbitrary column
➥    "" u 1:4 t "Black " . columnhead(4) w lp  ⟵  String expression
```

Note that the title can be taken from a different column than the data, which can be useful in combination with data transformations: commands such as `plot "data" using 1:($2/$3) title columnhead(2)` are legal.

The ability to take explanations directly from the data file is an interesting feature, adding significant convenience in particular when you're including many data sets in a single plot. Nevertheless, I'm a bit uncomfortable with the way it mixes data and comments in the same file without an easy way to distinguish the two; files in this format are harder to use with data-processing filters or as input to other programs. (I'd probably have preferred to make the line containing the column heads a comment line and indicate its use as title information through a special marker, such as a doubled hashmark or other comment character at the beginning of the line.) Judge for yourself.

PLACING EXPLANATIONS ON THE GRAPH

I quickly want to mention another new feature. Traditionally, explanations are visually grouped together to form the key. You can now also put some (or all) explanations onto the graph itself, either at the beginning or the end of the line they're describing (see figure 7.7). To do so, include `at beginning` or `at end` after the `title` keyword in the `plot` command:

---

[4]   The `columnheader()` function is new. In previous versions of gnuplot, this effect was achieved by supplying the column number directly as an argument to the `title` keyword.

```
unset key
plot [1989:1995] "grains" u 1:2 t "Wheat" at beginning w lp,
➡                "" u 1:3 t columnhead at end w lp,
➡                "" u 1:4 t at beginning w lp
```

**Explicit description
at beginning**

**Default description
at beginning**

**Description from
column head at end**
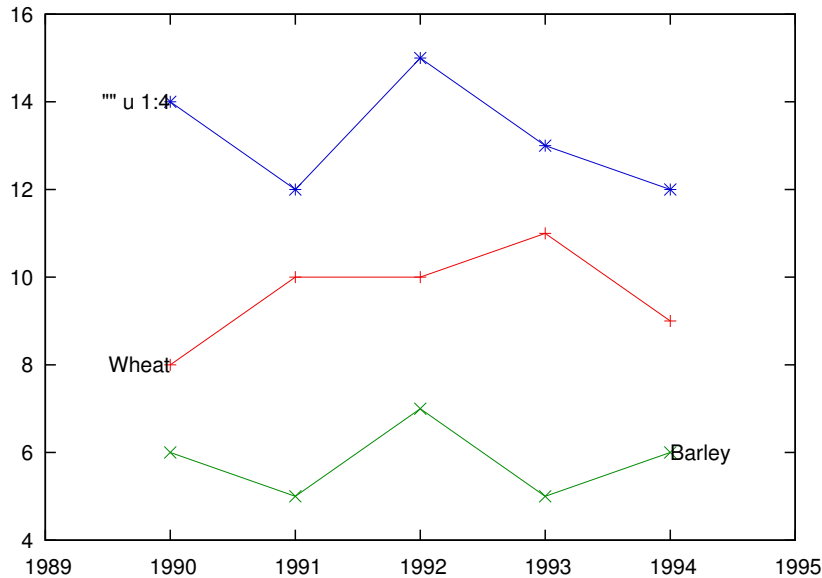
Figure 7.7   Explanations can be placed at the beginning or end of the curves rather
than grouped together in a global key.

Notice that the explanation string may be missing, but the `title` keyword (or its
abbreviated form `t`) must be present. Also keep in mind that you have no fine control
over the placement of the explanation. Furthermore, gnuplot won't adjust plot ranges
automatically to make room for these labels: you have to do so yourself. If you want
more control, use the `set label` facility (see section 7.3.3).

7.4.6   *Default settings*

Just to provide a reference, the default settings for the key are

```
set key on right top vertical Right noreverse noinvert autotitle
➡   samplen 4 spacing 1.25 title '' nobox
```

The previous command is equivalent to the much shorter

```
set key default
```

which can occasionally be helpful to restore sanity when you're experimenting with
key placement and layout.

Now that you've seen all the different graph elements you can use in a gnuplot figure, it's time to put everything together and look at a worked example. That's what we'll do next.

7.5   *Worked example: features of a spectrum*

To demonstrate how to use all the features together, let's study a more extensive example (see figure 7.8). The plot shows some experimental data, together with two theoretical curves that might explain the data from the experiment.
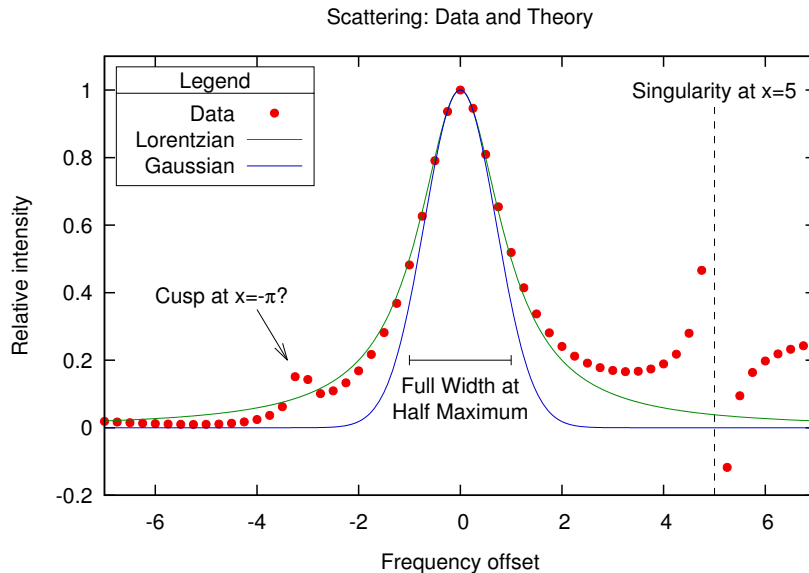
Figure 7.8   A complicated plot. See listing 7.4 for the commands used to generate it.

The folllowing listing gives the commands used to generate this graph, as they would be entered at the gnuplot prompt. Let's step through them in detail—there is much to learn.

Listing 7.4   Commands for figure 7.8 (file: spectrum.gp)

```
set title "Scattering: Data and Theory"                          ❶ Title

set xlabel "Frequency offset"                                    ❷ Axis labels
set ylabel "Relative intensity"

set key top left box title "Legend" height .5                    ❸ Key

set arrow 1 from -4,.35 to -3.4,0.2                              ❹ Arrow and label
set label 1 "Cusp at x=-π?" at -6,0.4

set arrow 2 from 5,-.2 to 5,.95 nohead dashtype '-' lc black     ❺ Vertical line
set label 2 "Singularity at x=5" at 5,1 center

set arrow 3 from -1,0.2 to 1,0.2 heads size 0.1,90              ❼ Label with line break
set label 3 "Full Width at\nHalf Maximum" at 0,0.125 center
```

Scale indicator ❻

```
plot [-7:7][-.2:1.1] "spectrum" u 1:2 t "Data" w p pt 7,
➡                      1/(1+x**2) t "Lorentzian", exp(-x**2) t "Gaussian"
```

The graph receives an overall title ❶, and both axes are labeled ❷. The key is moved to the top-left corner, so it doesn't interfere with the data, and surrounded with a box ❸. The key gets its own title, which makes it necessary to increase its overall height a bit to give it a less crowded appearance.

An arrow and a label are used to point out an oddity in the data ❹. The label contains the Unicode character π (see the sidebar in section 5.1 for more information on Unicode and how to enter Unicode characters in gnuplot).

An arrow without heads is used to create a dashed, vertical line ❺. The dashed line shows the vertical asymptote as the data approaches the singularity at x=5. (Dash types will be introduced in section 9.2.2.)

The scale indicator is yet another creative use for an arrow, this time with two customized heads ❻. This arrow doesn't point out a feature in the graph, but gives an indication of scale. ("Full Width at Half Maximum" is commonly used in spectroscopy to measure the width of a spectral line.) The label for this arrow contains an explicit line break so that it spreads over two lines ❼. Remember from chapter 5 that escaped characters (such as \n here) in double-quoted strings are interpreted as control characters. Finally, finally, the data is plotted ❽!

I hope this example gives you an idea how you can use decorations to improve your graphs. Arrows and labels can be used to explain data and point out specific features, but nothing prevents you from getting more creative.

7.6   *Summary*

This chapter explained techniques for putting additional, decorative graphical elements on a plot: things like arrows and text labels. Because it's such an important tool to help a viewer understand a complicated graph, I spent a lot of time explaining how to customize and tweak the graph's key or legend.

You also learned how to specify locations on a graph; I introduced the coordinate systems gnuplot uses for that purpose. I also discussed the layers that make up the graphical stacking order.

Decorative graphical elements (such as those in this chapter) are important, even though they don't represent data. Through these additional decorations and text labels, a graph receives its *semantics*: only if there is text can an onlooker understand what the data shown in the graph *means*.

In the following chapter, we'll look at the last important elements that give a plot context: the axes and their subdivisions and labels. Stay tuned.

All about axes

This chapter covers

- Using multiple axes
- Selecting plot ranges
- Customizing tic marks and tic labels
- Plotting time series with gnuplot

In this chapter, we finally come to coordinate axes and their labeling. Treating them last, after discussing plot styles and decorations, may seem surprising, given how critical well-labeled coordinate axes are to achieving an informative graph. On the other hand, gnuplot's default behavior for axes-related options is adequate in almost all situations, so explicit customization is rarely required.

One topic deserves special consideration: the use of multiple axes on the same plot. We'll discuss this first. Then we'll move on and describe all the ways axes and their labels can be customized. Finally, we'll treat the special case when one axis (usually the x axis) represents *time*: in other words, when the plot shows a time series. Time-series plots pose special challenges, because the labels aren't numeric. Instead, you need to worry about things such as the names of months and week-days—potentially in different languages! This has long been a problem to gnuplot users, so I'll devote significant space to this application. But first, let's talk about multiple axes on the same plot.

8.1   *Multiple axes*

Gnuplot gives you the ability to plot graphs using two different coordinate systems within the same plot. Typically, these coordinate systems share one axis (otherwise there's no good reason to have them on the same plot), but they may also be entirely independent.

Plots involving two different y axes usually make the most sense when you want to compare side by side two data sets that have very different units. As a typical example, let's study figure 8.1, which compares the average ice cream consumption (in some community) over consecutive four-week periods with the mean temperature during the same period.[1]

Figure 8.1 is a good example of why you might want to use multiple axes on a plot: the two quantities (ice cream consumption and temperature) are of a different nature and are also numerically quite different. Yet once you put them next to each other, the correlation becomes clear (not too surprisingly, in this example).
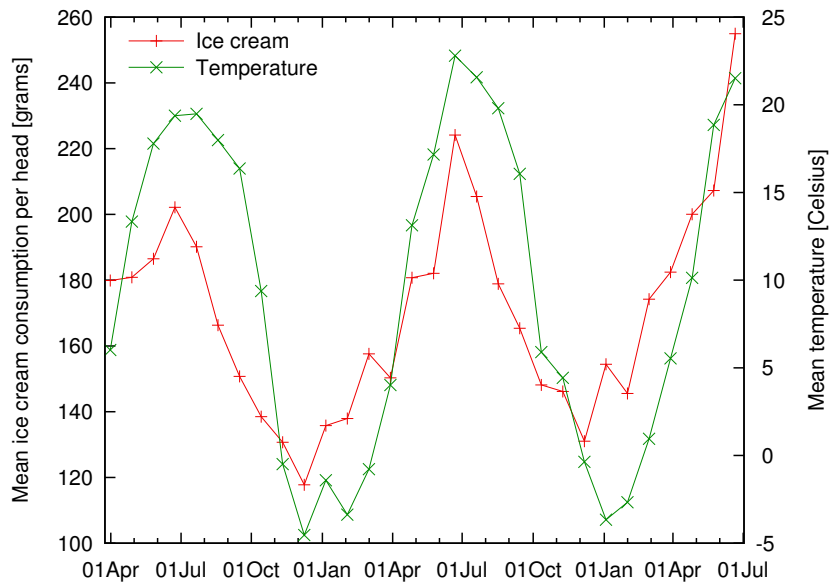
Figure 8.1   Using multiple axes on a plot to compare two different quantities side by side. (See listing 8.2 to find out how this plot was made.)

8.1.1   *Terminology*

As you've just seen, gnuplot can handle two sets of axes on a single plot. The consequence is that all commands and options to manipulate axes-related properties come

---

[1]   This example was inspired by the "Ice Cream Consumption" story, found in the StatLib's Data and Story Library (DASL) at http://lib.stat.cmu.edu/DASL/Datafiles/IceCream.html.

in two versions—one for each set of axes. This section summarizes the naming conventions associated with these commands.

The primary coordinate system is usually plotted along the bottom and left borders of the graph. (This corresponds to the `first` coordinate system introduced in section 7.2.) If the secondary system (`second`) is used, it's plotted along the top and right borders.

By default, the secondary system isn't used. Instead, the tic marks (but not the labels) of the primary system are mirrored on opposite sides of the plot (that is, on the top and right borders, respectively).

All options that modify aspects of the coordinate systems can be applied to any of the axes: either the x or the y axis, in either the primary or the secondary coordinate system. The commands and options are prefixed to indicate which specific axis a command should be applied to (see table 8.1). Omitting the prefix applies the option to *all* axes.

	Primary	Secondary
x axis	x	x2
y axis	y	y2

Table 8.1   **Prefixes used to indicate the selected coordinate system**

In the rest of this chapter, I'll frequently discuss only one variant of any option—typically the one for the x axis of the primary coordinate system. You should keep in mind that everything applies to all other axes as well, just by selecting the appropriate prefix per table 8.1.

8.1.2   *Plotting with two coordinate systems*

The best way to understand how multiple coordinate systems are used in the same plot is through an example. Listing 8.1 shows the beginning of the data file from figure 8.1; the complete set of commands used to generate the plot from the data file is in listing 8.2.

Listing 8.1   Incomplete data for figure 8.1 (file: icecream)

```
# Date        Consumption[g]    Temperature[Celsius]
1951-04-01    179.8              6.01
1951-04-29    180.8             13.34
1951-05-27    186.5             17.78
1951-06-24    202.1             19.38
1951-07-22    190.1             19.48
...
```

Listing 8.2   Commands for figure 8.1 using data from listing 8.1 (file: icecream.gp)

```
set xdata time
set timefmt "%Y-%m-%d"
set xtics format "%d %b"
```

```
set ytics nomirror          ⟵——— Switches mirroring of the principal system off
set y2tics                  ⟵——— Switches the secondary system on

set ylabel  "Mean ice cream consumption per head [grams]"
set y2label "Mean temperature [Celsius]"

set key top left reverse Left

plot ["1951-03-25":]
➡    "icecream" u 1:2 t "Ice cream" axes x1y1 w lp,
➡    "" u 1:3 axes x1y2 t "Temperature" w lp
```

The first three lines (from `set timefmt` to `set xdata`) tell gnuplot how to parse and format the calendar date used along the x axis. We'll discuss them in section 8.4.

Next, the mirroring of the primary axis' tic marks is switched off on the opposite (right) side of the plot (`set ytics nomirror`), and instead tic marks for the secondary y axis (`set y2tics`) are switched on.

Explicit labels are placed on both vertical axes—this step is crucial when using multiple axes, because otherwise the viewer has no chance of figuring out which data set goes with which axis. The key is modified from its default location and appearance (see section 7.4 if you need a refresher on any of the options).

Finally, the only new thing in the actual `plot` command is the use of the `axes` keyword with its arguments. These directives tell the `plot` command which combination of axes to use for each data set. For example, `axes x1y2` means that the data should be plotted according to the primary x axis, but the secondary y axis. There are four possible combinations of axes that can be used, and they can be selected using `x1y1`, `x1y2`, `x2y1`, and `x2y2`.

Should you do it?

Multiple axes on a single plot are occasionally frowned upon, because they can easily be abused to manipulate the message of a graph. Look at the figure at the end of this sidebar. The top-left panel shows both data sets drawn to the same scale. Both curves grow, but one grows more strongly than the other.

The other panels show exactly the *same* data, but look how different the appearance is in each one! The curve that seems to be growing most strongly in the top-right panel is the one that is lagging behind in the original graph. And in the bottom two panels, both curves seem to grow at an equal rate. These dramatically different appearances were achieved solely by manipulating the plot ranges for each curve individually.

Dual axes have their uses. They can be helpful (even necessary) to combine two data sets with very different scales in a single graph. But you must always take care, because the ability to select different plot ranges in a single graph makes dual-axes plots open to the kind of abuse you see in the figure.

(continued)

If you choose to use multiple axes, it's *mandatory* that you indicate which scale applies to which curve. In the figure shown here, this information is intentionally (and maliciously) omitted, to emphasize the problems that can occur in this kind of graph. When using dual axes, *always* include an explicit label stating which axis applies to which curve!
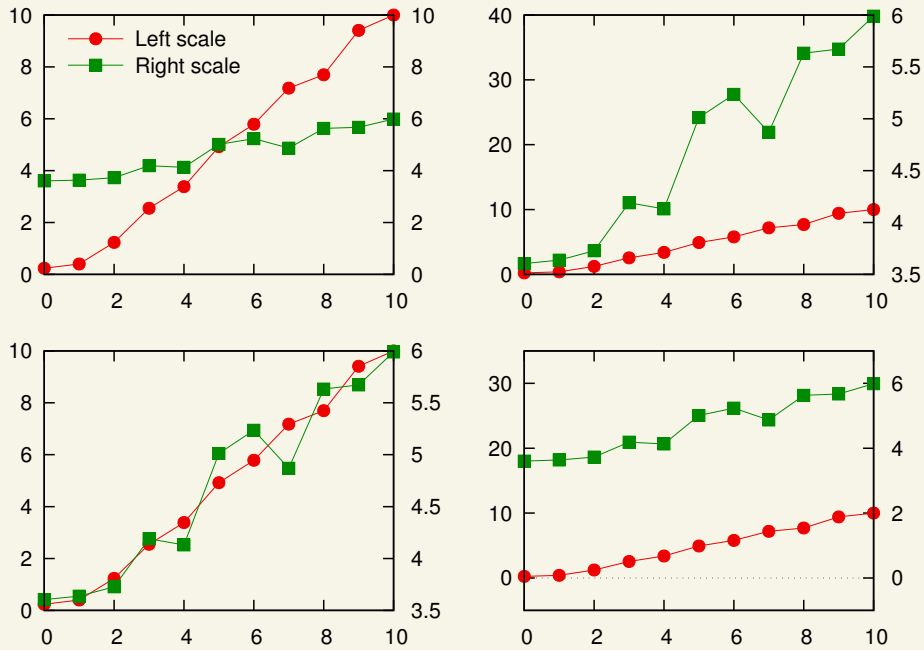
The malicious effect of not-to-scale graphs. The data in all four panels is the same, but the scales have been changed for both curves independently.

8.1.3   *Linking axes*

In the previous section, you learned how to plot two curves representing different and independent quantities (ice cream consumption and temperature) on the same plot. In this section, we'll consider a related problem: how to plot two curves representing different but *related* quantities.

Let's look at an example. Figure 8.2 shows two data files, side by side. Both contain temperature readings for the same date range, but for two different locations (New York City, USA and Frankfurt a.M., Germany) and in two different units (Fahrenheit and Celsius).[2]

---

[2] Weather data is available from the respective government services: http://mng.bz/V299 and www.dwd.de.

```
# Frankfurt/M  Celsius          # New York      Fahrenheit
2014-11-01      9.3             2014-11-01      45
2014-11-02     10.2             2014-11-02      45
2014-11-03     12.9             2014-11-03      50
2014-11-04     12.3             2014-11-04      61
2014-11-05      9.3             2014-11-05      60
2014-11-06      6.6             2014-11-06      53
2014-11-07      5.8             2014-11-07      47
```

Figure 8.2   Two data sets showing daily temperatures. Note that the data sets use different units.

Because both data sets show the same *kind* of data, it makes sense to plot them in the same graph. Nevertheless, because the *units* are different, the two data sets can't just share a single y axis. Of course, you could use inline transformations (see section 3.3) to transform (say) the Fahrenheit values into Celsius and plot them—but in order to easily go back and forth between both temperature systems, it would be nice if you could retain *both* units in the same plot.

This is where gnuplot's new set link facility comes in. Using set link, you can make the values on one axis dependent on the values on the opposite axis. All you need to do is provide a link function that maps values from one system to the other, together with its inverse.

In this example, Celsius must be mapped to Fahrenheit and vice versa. The two temperature systems are related to each other by the following formulas:

```
Fahrenheit = (9/5)*Celsius + 32
Celsius    = (5/9)*(Fahrenheit - 32)
```

Using these formulas, you can set up the link between the two vertical axes:

```
set link y2 via (9/5.)*y+32 inverse (5/9.)*(y-32)
```

This establishes a link between the primary vertical axis (the y axis, on the left), which shows temperatures in Celsius, and the secondary vertical axis (the y2 axis, on the right), which shows values in Fahrenheit. You can now plot both files using the commands from listing 8.3; the resulting graph is shown in figure 8.3. (See listing 8.2 for an explanation of the commands at the beginning of this example.)

Listing 8.3   Commands for figure 8.3 using data from figure 8.2 (file: link.gp)

```
set xdata time
set timefmt "%Y-%m-%d"

set xtics format "%d%b%y"
set ytics nomirror
set y2tics

set ylabel "Celsius"
set y2label "Fahrenheit"

set link y2 via (9/5.)*y+32 inverse (5/9.)*(y-32)

plot "frankfurt" u 1:2 axis x1y1 w lp, "newyork" u 1:2 axis x1y2 w lp
```
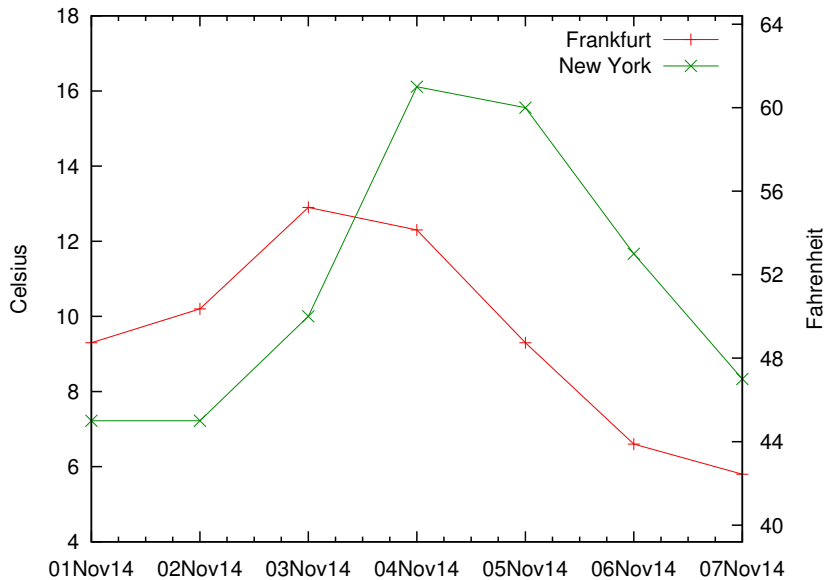

Figure 8.3   Linking two axes using `link`. See the text for details and listing 8.3 for the commands.

This example should give you a sense how the `set link` feature can be used. Here are a few additional details:

- Links can be established between the two horizontal axes and between the two vertical axes using `set link x2` and `set link y2`, respectively.
- When linking the horizontal axes using `set link x2`, you must call the dummy variable in the `link` functions `x`; when linking the vertical axes using `set link y2`, the dummy variable in the `link` functions must be called `y`.
- You can remove the link between either set of axes using `unset link x2` or `unset link y2`.

- Keep in mind that the link function must be invertible! This isn't a problem when the link function is linear (as in the example), but non-linear functions are, in general, not invertible (or are invertible only over a limited domain—the function x^2, for instance, is invertible for all non-negative values of x).

The `set link` feature is relatively new and strikes me (at the time of this writing) as still somewhat immature. Feel free to experiment further!

Do-it-yourself log plots

The `link` feature can be used to create log plots. Doing so may shed more light on what `link` can do.

Let's consider the data set displayed in the sidebar in section 3.4. Back then, I used `set logscale y` to generate a semi-logarithmic plot. Now, I'll use `link` instead.

In a semi-log plot, you plot the logarithm of the y value. You can do this using a simple inline transformation:

```
plot "data" u 1:(log10($2)) w lp
```

This gives the appearance of a semi-log plot, but the tic marks along the vertical axis show not the actual y values from the data sets, but rather their logarithms (to base 10)—so, a value of 10 corresponds to a tic mark labeled 1, a value of 100 to a tic mark labeled 2, and so on. (See the scale on the right in the figure in the sidebar in section 3.4.)

The `link` feature makes it possible to have an axis showing the *original* y values:

```
set link y2 via 10**y inverse log10(y)
```

One problem is that gnuplot isn't very clever at figuring out where to place tic marks for the dependent (linked) axis, which means you have to specify them explicitly using a command like this (the `set _tics` family of options is explained later in this chapter):

```
set y2tics ( 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 )
```

Although you must specify the tic marks explicitly, `link` lets gnuplot figure out where along the vertical axis these tic marks should go. The resulting plot is equivalent to the figure in section 3.4 (with the tic marks on the left and right interchanged).

8.2   *Selecting plot ranges*

I introduced plot ranges in chapter 2, but only in a limited form as an inline specification to the `plot` command, something like this:

```
plot [-10:10][-2:2] sin(x)
```

The first pair of numbers in brackets sets the desired x range, and the second pair of numbers in brackets fixes the y range. Both terms are optional—if you leave them out, gnuplot will select ranges automatically.

This is enough—most of the time. But using this syntax, only the plot ranges of the primary coordinate system can be fixed, which is insufficient if you want to use multiple axes on the same plot. Also, the inline syntax doesn't work well when you're attempting to change the plot range with the mouse. Let's look at gnuplot's facilities for controlling plot ranges in more detail.

> **TIP**   A distinctive property of good graphs is that they show what is *relevant*. Plot ranges are the primary means of achieving this goal. Well-selected plot ranges are essential!

8.2.1   *What you need to know for interactive work*

The inline syntax is a shorthand for the family of `_range` commands. (In this discussion, the underscore is a placeholder for any one of the prefixes from table 8.1.) To adjust plot ranges for the primary and secondary systems independently, you need to issue separate `set _range` commands using different prefixes.

The explicit `set _range` commands expect a pair of numbers enclosed in square brackets, similar to the syntax for inline range specifications. The following listing shows some examples of `_range` commands in action.

Listing 8.4   Permissible syntax for setting plot ranges

```
set xrange [-1:5]          ⟵——— Explicit min and max
set xrange [:10]           ⟵——— Leaves the min unaffected; sets the max explicitly
set yrange [3:*]           ⟵——— Sets the min explicitly; uses autoscaling for the max
set yrange [:sqrt(2)]      ⟵——— Numeric expressions are legal.
set yrange [1:0]           ⟵——— Inverted axes are possible.
```

In addition to providing explicit lower and upper boundaries, you can leave one or both of the numbers *blank*, in which case the corresponding value won't be changed. Alternatively, you can supply a star (*) to turn on *autoscaling* for that particular value. Autoscaling means gnuplot selects a plot range that is at least large enough to include all data points, but gnuplot may *extend* the range in order to include the next full tic mark position. This behavior can be suppressed using the `noextend` specifier. Using the following setting, the vertical range is bounded precisely by the minimal and maximal values in the plotted data set:

```
set yrange [*:*] noextend
```

The inline shorthand (like `plot [0:1] ...`) only affects the command line it's part of—it doesn't reset the overall plot ranges, the way `set _range` does. This is why you don't need to use the * indicator to turn on vertical autoscaling when using only the shorthand: by default, the plot ranges are `[-10:10]` horizontally and `[*:*]` vertically. Issuing a command like `plot [][0:1] ...` doesn't change the global autoscale setting for the y axis, so a subsequent command like `plot [][0:] ...` will autoscale the positive y axis. But if the vertical range has been fixed globally—using `set yrange [0:5]`,

for example—then `plot [][0:] ...` will honor the explicit plot range. To reestablish autoscaling behavior, you need to turn it on explicitly, either inline (`plot [][0:*] ...`) or using `set yrange [0:*]`.

8.2.2   What you might want to know for batch processing

Special considerations apply when creating large numbers of graphs automatically. Imagine, for instance, that you want to monitor traffic across many websites—in hosted or enterprise environments, this may mean following hundreds of separate traffic streams. Furthermore, the traffic to these sites is likely to vary by several orders of magnitude: a busy site may see tens of thousands of clicks a day (or more), but other sites may see few if any clicks.

Obviously, you don't want to select plot ranges manually (for all the sites—every day!). But relying entirely on gnuplot to adapt the plot range to the available data often isn't satisfactory either. For some of the very slow sites (those that show at most a single click a day), gnuplot will select a vertical plot range of `[0:1]`, which isn't a visually pleasing solution. It would be better if you could make sure the plot range is always *at least* `[0:10]` but larger if need be. Fortunately, gnuplot let's you do just that: you can define *bounds* for autoscaled ranges. The next listing shows you the syntax.

> **Listing 8.5   Restricting plot ranges to bounded intervals**

```
set yrange [*:*]           ⟵———  Unbounded autoscaling
set yrange [0:5<*]         ⟵———  Lower bound is fixed; upper bound is 5 or more
set yrange [0:5<*<10]      ⟵———  Lower bound is fixed; upper bound is between 5 and 10
set yrange [-10<*:*<10]    ⟵———  Double-sided bound, not exceeding [-10:10]
```

Other times, it may make sense to let autoscaling apply to only *some* of the lines in a plot. For example, imagine that you want to plot two data sets together, one of which is corrupted by occasional crazy outliers. In that case, you may want to let gnuplot use only the smoother curve to determine the plot range. There are two ways to do this. Using the `noautoscale` (`noauto` for short) keyword, you can flag those data sets in a `plot` command that should be disregarded when determining the plot range:[3]

```
plot "smooth" u 1:2, "noisy" u 1:2 noautoscale
```

Another approach is to evaluate the data sets first and then use the information obtained to calculate the plot range. The `set _range` family of options provides a way of doing this (using `writeback` and `restore`), but the `stats` command (see section 5.6) and the internal variables it defines are probably more straightforward to use.[4]

---

[3]   The `noautoscale` directive is available only when plotting data files, not when plotting functions. If you want to use it with functions, you need to "fake" it using the `"+"` pseudofile facility (see section 4.5).

[4]   Another related option is `set autoscale`, possibly in conjunction with `set offset`, but it seems overly complicated and largely redundant. For more information about this, as well as about `writeback` and `restore`, see the standard reference documentation.

8.3   Tic marks

Subdivision markers or *tic marks* are placed on the axes of a plot to indicate the scale of the graph. Only if tic marks are present can a viewer infer *quantitative* information from a graph. Suitably chosen tic marks are therefore of *critical* importance to any well-constructed graph. Fortunately, gnuplot handles tic marks well on its own, and you rarely need to customize tic-mark generation and labeling. But for the few cases when you *do* have special requests, here's how to do it.

8.3.1   Overview and common conventions

Gnuplot distinguishes between *major* and *minor* tic marks. The difference is that major tic marks also carry a textual label (normally a number), whereas minor tic marks don't. By default, only major tic marks are used, except for logarithmic axes, where both major and minor tic marks are drawn by default.

Major tic marks are by far the more important of the two types. The appearance of major tic marks and their labels is controlled through the family of set _tics options. The available sub-options are as follows:

```
set _tics [ axis | border ]
          [ in | out ] [ front | back ]
          [ scale [ default | {flt:major} [,{flt:minor}] ] ]
          [ [no]mirror ]

          [ rotate [by {flt:ang}] | norotate ]
          [ left | right | center | autojustify ]
          [ offset {pos:offset} | nooffset ]
          [ font "{str:name} [,{int:size}]" ]
          [ textcolor | tc {clr:color} ]
          [ format {"str:format"} ]
```

As before, the underscore _ is a placeholder for any one of the available axes identifiers (see table 8.1), indicating which axis the settings should be applied to. If you omit the identifier (set tics ...), then the chosen style is applied to all axes.

The location and frequency of tic marks is controlled through another set of sub-options to set _tics. These sub-options only make sense when applied to a specific axis (meaning you can't leave the _ placeholder blank in this case):

```
set _tics ...
          [ add ]
          [ autofreq
            | {flt:incr}
            | {flt:start}, {flt:incr} [,{flt:end}]
            | ( ["{str:label}"] {flt:pos} [ 0 | 1 ]
                [, ["{str:label}"] ... ] ) ]
```

We'll next step through these options: first we'll discuss how to modify the appearance and placement of tic marks, and then we'll talk about labels, before discussing ways to control the location and frequency of tic marks. We'll round off this section with some related features that are handled using commands other than set _tics.

8.3.2 *Tic mark appearance and placement*

By default, gnuplot draws tic marks on the border of the plot, extending inward into the plot region, but they can be drawn toward the outside using the `out` option. This is useful in particular when the tic marks would interfere with the data. Tic marks can be drawn in either the `front` or the `back` layer of the visual stack (see section 7.2).

Instead of being placed along the borders of the plot, tic marks can also be drawn along the zero axis (for instance, `set xtics axis` draws tic marks along the line of the plot where y equals 0). If the zero axis isn't within the plot range, the tic marks are drawn along the border.

The `scale` parameter controls the size of both major and minor tic marks. If no size for the minor tic marks is given explicitly, it's set to half the size of the major marks. The size is given relative to the default size of 1.0 for major tic marks.

By default, the tic marks (but not their labels) of the primary coordinate system are mirrored on the opposite side. You can turn off this behavior using `nomirror`. You want to do this when you're using two coordinate systems in one plot (such as both the `y` and the `y2` systems), but there are also other situations when mirroring is undesirable.

Why gnuplot's defaults make sense

Graphs drawn using gnuplot can often be recognized by their overall appearance (unless the defaults have been modified significantly, of course): the plot is surrounded by a frame (or border), with tic marks and labels on the frame. This format may at times strike you as a bit tired, but I'd like to emphasize some advantages that should make you hesitate at least a moment before rejecting it:

- The frame on all four sides provides a clear visual boundary for the data. If only two axes were drawn along the bottom and left side, data points in the upper-right corner might get (visually) lost.
- The other two sides of the frame (top and right side) can be used to mirror the primary tic marks or to carry a set of secondary tic marks, thus making qualitative evaluation of the data easier.
- Because tic marks and their labels are drawn along the border of the graph (as opposed to along the zero axis) they're less likely to interfere with the data.

If you nevertheless want to give your graphs a different appearance, we'll discuss some options in section 12.6.

A WORKED EXAMPLE

Figure 8.4 shows a graph that doesn't use the gnuplot default format: it has no border, and instead uses explicitly drawn coordinate axes (zero axes). Because there is no border, the tic marks are drawn along the coordinate axes.
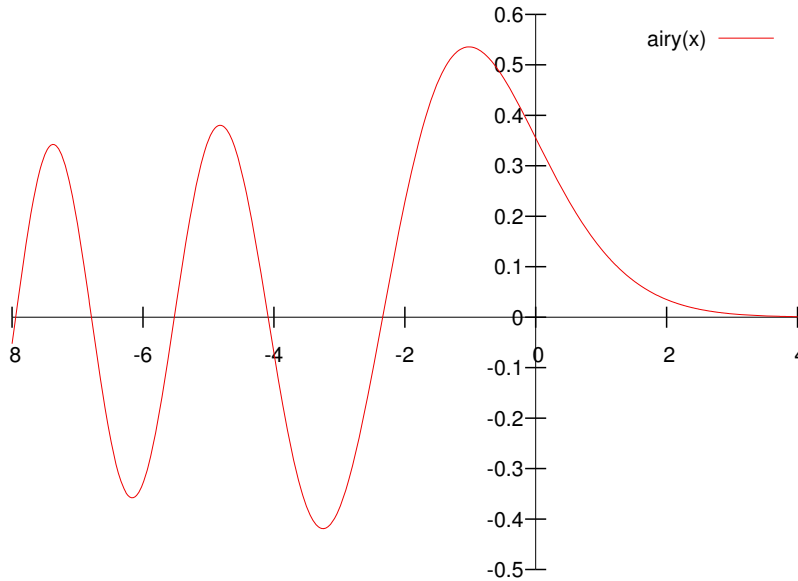
Figure 8.4   Nonstandard placement of coordinate axes and tic marks. See listing 8.6.

You'll find the commands for this figure in listing 8.6. First the border is switched off, but the coordinate axes are drawn with solid black lines. (See section 8.3.6 later in this chapter for the `set zeroaxis` command. The line type `-1` is guaranteed to draw a solid black line; see table 9.3.) Then gnuplot is instructed to place tic marks along the coordinate axis (instead of the border) with `set xtics axis` (and equivalent for the vertical axis). Finally, a function is plotted.[5]

<div style="background:green;color:white;">

Listing 8.6   Commands for figure 8.4 (file: var-axes.gp)

</div>

```
unset border
set zeroaxis lt -1

set xtics axis
set ytics axis

plot [-8:4] airy(x)
```

---

[5]   The *Airy function* is the solution to the differential equation $y'' - xy = 0$ that is bounded everywhere. It's important in the approximation theory of differential equations.

The resulting graph is closer to what is common in mathematical contexts, but the disadvantages are clear: the curve interferes with the tic marks whenever it crosses the axes. (The partial obscuring of the zeros near the origin could be avoided through smarter placement of the labels and isn't a fundamental problem.)

8.3.3 *Tic labels*

The text labels associated with major tic marks can be rotated and shifted using `rotate` and `offset`. If `rotate` is used without an explicit angle (in degrees), the labels are turned by 90 degrees to the left. You can adjust the position of the labels using `offset`. The relative shift can be specified in one of the five usual coordinate systems (see section 7.2); if none is specified, the `character` coordinate system is assumed.

By default, tic labels are justified depending on the axis and the rotation angle (`autojustify`). Alternatively, you can control the alignment explicitly with `left`, `right`, and `center`. Text font and color can be selected in the usual fashion.

FORMATTING TIC LABELS

You can change the formatting used for the labels placed at the major tic marks using `set _tics format`, followed by a string that describes the desired format.[6] The format string is similar to the format string used by the `printf()` family of functions from the standard C library. In addition, gnuplot uses some extra format (or *conversion*) specifiers, which are listed in table 8.2. These conversion specifiers only apply to *numeric* arguments; for date/time values, check section 8.4.

> **TIP** Don't attempt to build up date and time strings manually; use dedicated conversion specifiers for time series (see section 8.4).

If the `%` character is encountered in the format string, it's interpreted as the beginning of a conversion specifier. It must be followed by one of the characters from table 8.2. Several modifiers can be placed between the `%` character and the conversion specifier; see table 8.3. In general, gnuplot's functionality is built on top of the system's C library—check its documentation for additional details.

The format string can also contain arbitrary characters, which are placed verbatim onto the plot. This makes it possible, for instance, to print the units (such as `kg` or `cm`) together with the numerical values.

Finally, providing an empty string as the format specifier (as in `set xtics format ""`) is a way to suppress the generation of tic *labels*, although the tic *marks* are drawn.

---

[6] Traditionally, formatting of tic labels was accomplished using the `set format` option. This option is still available, but it's now redundant and should probably be avoided in favor of the more consistent method described in the text.

Table 8.2   Conversion specifiers understood by the `gprintf(...)` function. See table 8.4 and table 8.5 for conversion specifiers for date and time values, and table 5.1 for information on `gprintf(...)`.

Conversion specifier	Description
`%f`	Floating-point notation
`%e` or `%E`	Exponential notation, using `e` or `E` (respectively) to indicate an exponent
`%g` or `%G`	Uses the shorter of `%f` and `%e` (or `%E`)
`%h` or `%H`	Like `%g` or `%G`, but uses enhanced text mode to format the power of 10 as a proper superscript. When the encoding is UTF-8, labels are formatted as `a × 10^{b}`; when Unicode isn't available, the multiplication sign is replaced by the letter `x` (for `%h`) and by an asterisk `*` (for `%H`).[a]
`%x` or `%X`	Hexadecimal representation
`%o` or `%O`	Octal representation
`%P`	Multiple of π
`%c`	Character replacement for scientific powers, such as `k` (kilo) for 1,000, and so on
`%%`	Inserts a `%` character into the output (no conversion)
`%t` and `%T`	Coefficient and exponent to base 10
`%l` and `%L`	Coefficient and exponent to the base of the current logscale
`%s` and `%S`	Coefficient and exponent to the base of the current logscale; scientific power (restricts the power to a multiple of 3)
`%b` and `%B`	Coefficient and exponent using ISO/IEC 80000 notation

[a] This conversion specifier is new with gnuplot 5.

Table 8.3   Accuracy specifiers to be used with the `gprintf()` and `sprintf()` functions

Accuracy specifier	Description
`"-"`	Left-justifies the number
`"+"`	Explicitly signs all numbers
`" "`	Uses a space instead of a positive sign
`"#"`	Forces a decimal point, even if all following digits are zero
`"a.b"`	Field width *a* and number of decimal places *b*. (Both *a* and *b* must be positive integers.)
`"0a.b"`	Same as previous, but pads the number with zeros instead of spaces on the left

SCIENTIFIC NOTATION AND ENHANCED TEXT MODE

Several of the conversion specifiers in table 8.2 give you individual access to the coefficient and exponent of a number in *scientific notation*. They're intended for situations where you want to build up the combination of coefficient and exponent yourself.

In scientific notation, a number x is represented as

$$x = a \times b^c$$

where a is called the *coefficient* (also *mantissa* or *significand*), b is the *base*, and c is called the *exponent* or *power*. For example, the number 1,024 can be written as 1.024×10^3 (base 10) or 1×2^{10} (base 2).

When in *enhanced text mode*, gnuplot interprets certain regular characters embedded in string labels as control or formatting characters. This way, it's possible to change font face, size, and weight locally and on the fly.[7] A full discussion of enhanced text mode will have to wait until chapter 10; what is of interest in the context of tic marks and especially tic labels is the ability to include subscripts and superscripts in all text elements that are part of a graph. Subscripts are indicated with an underscore (_) and superscripts with a caret (^). A single character following either of these (or a sequence of characters enclosed in curly braces) is formatted as subscript or superscript, respectively.

In particular for powers of 10, the ability to use real superscripts is welcome. Imagine a number such as 5,000. In scientific notation, this becomes

$$5 \times 10^3$$

Using the customary E notation (see section 4.3), this is written as `5.0e+03`. But using enhanced text mode, you can format this string as

```
"5 × 10^{3}"
```

(or even as `"5 × 10^3"`, because the exponent consists only of a single character), and gnuplot will display the exponent as a proper superscript.

The new conversion specifiers `%h` and `%H` use enhanced text mode to format numbers this way *if* the proper encoding (such as UTF-8; see the sidebar in section 5.1) has been chosen. Otherwise, the lowercase version (`%h`) uses the letter x to signify the multiplication operator following the coefficient, whereas the uppercase version (`%H`) uses the asterisk `*`.

A WORKED EXAMPLE

Let's study an example. Figure 8.5 shows the weight of several mammals as a function of their heart rate in a double-logarithmic plot. Remarkably, the points seem to fall roughly onto a straight line: there is a distinct relationship between the size (or

---

[7] Enhanced text mode is now the *default* in gnuplot 5; in general, it's no longer necessary to enable it explicitly. In fact, at times you may need to *disable* it using the `noenhanced` keyword in order to include control characters in text labels.

weight) of an animal and the rate of its metabolism. This is known as *allometric scaling* and extends to other properties, such as the life span, as well: bigger animals live longer. Allometric scaling is a fascinating topic, but what interests us right now are the commands used to draw this graph. They're shown in the following listing.

Listing 8.7   Commands for figure 8.5 (file: mammals.gp)

```
set logscale
unset key

set ylabel "Weight"
set xlabel "Heart rate [beats per minute]"                1 Tic labels for vertical axis

set ytics format "%.0s%cg"
set xtics format "%.0t·10^{%T}"                           2 Tic labels for horizontal axis

plot [10:1000][:0.2e9]
     "mammals" u 3:2:1 w labels point pt 7 lc rgb 'red' offset 0,-1

                                          Custom point type and color  3
```
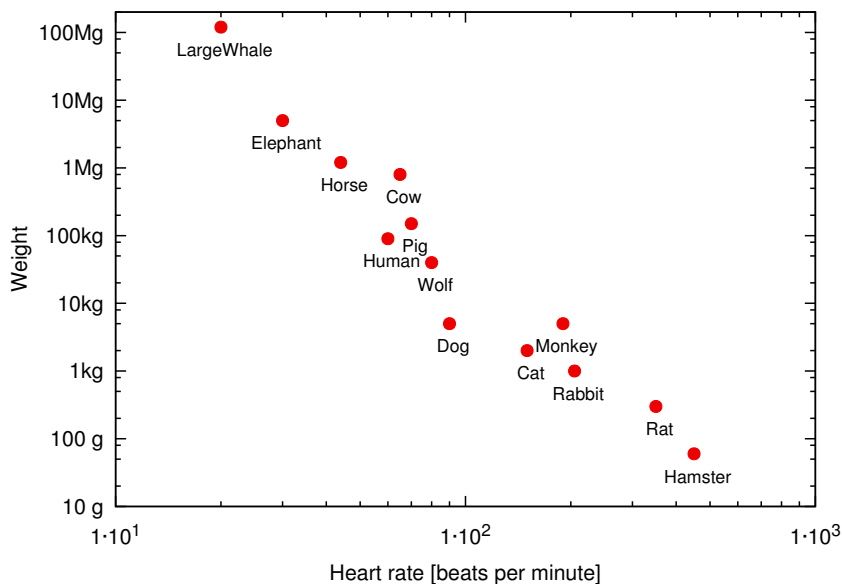

Figure 8.5   Allometric scaling: larger animals have a slower heart rate. Notice the customized tic labels along both axes. See the text for details and listing 8.7 for the commands.

Choosing logarithmic scales and setting axes labels should be familiar by now, but pay close attention to the way the tic labels are formatted. For the vertical axis, the format string is `"%.0s%cg"` 1. According to table 8.2, `%s` is the coefficient (in scientific notation) when the power is restricted to being a multiple of 3, and `%c` is a character replacement for that power (*k* for *kilo*, *M* for *Mega*, and so on). Hence a number like 2,000 is

split into the coefficient 2 (all decimals are suppressed because of `%.0s`), followed by *k* for the factor of 1,000. The example tacks on an additional *g*, which carries through unchanged, to represent the weight unit *gram*. The result is perfect for grams and kilograms, but megagrams are more commonly known as (metric) tons. The horizontal axes uses a custom format with a multiplication dot in place of the cross sign ❷.[8] Finally, the `plot` command is an interesting example of the use of the `with labels` style (see section 6.3.5), with an explicitly selected point type and color ❸.

8.3.4 *Tic mark location and frequency*

Finally, you can control where tic marks are drawn. Using `autofreq`, gnuplot will automatically generate tic marks covering the plot range.[9] Alternatively, you can provide an increment alone; then tic marks are drawn at integer multiples of the increment. Or you can specify a start point, an increment, and (optionally) an endpoint. (For logarithmic axes, the increment is interpreted multiplicatively instead of additively.)

Some examples will clarify (remember that the variable `pi` is predefined in gnuplot):

```
set xtics pi        ⟵──── Increment alone: tic marks at pi, 1*pi, 2*pi, …
set xtics 1, pi     ⟵──── Start and increment: tic marks at 1, 1+pi, 1+2pi, …
set xtics 0,0.1,1      ⟵─┐ Start, increment, and end point: tic
                         │ marks at 0, 0.1, 0.2, … 0.9, 1 only
```

You can also provide a list of explicit labels and locations at which to draw tic marks. The list must be enclosed in regular parentheses, with list entries separated by commas. Each entry in the list consists of the text label for the tic mark, its location, and a third, optional parameter that indicates whether the tic mark should be drawn as a major or minor tic mark: 0 for major and 1 for minor.[10] The text label can be an empty string (so that no label is drawn for the tic mark), or it can be missing (in this case, a textual representation of the location parameter is used as a label). If a text label is provided for minor tic marks, it's ignored. Here are some examples:

```
                                              ┌─ 5 major tic marks with
                                              │  autogenerated (numeric) labels
     set xtics ( 0, 1, 2, 3, 4 )          ⟵──┘
     set xtics ( "First" 1, "Second" 2, "Third" 3 )   ⟵──┐
 ⊳   set xtics ( "A" 0 0, 1 1, 2 1, "B" 3 0 )             │ 3 major tic marks using
 │                                                        │ provided text labels
 │
 └─ 2 major tic marks at 0 and 3 labeled "A" and
    "B", with 2 minor tic marks between them
```

---

8   The multiplication cross can be found at Unicode code point U+00d7 and the multiplication dot at U+22c5. The interpunct at code point U+00b7 provides an alternative for the latter that may be more readily available.

9   To generate tic marks covering only the actual range of the data, rather than the plot range, you can use the `rangelimited` keyword. See the standard gnuplot reference documentation for details.

10 You can add further levels of tic marks beyond major and minor. See the standard gnuplot reference documentation for `set tics` and `set xtics` for details.

ADDING AND OVERRIDING INDIVIDUAL TIC MARKS

Usually, any change to the location and frequency of tic marks wipes out any previous settings. To make changes to individual tic marks while retaining existing ones, use the `add` keyword. This can be useful for adding tic marks for special values to otherwise autogenerated tics, like so:

```
set xtics autofreq
set xtics add ( "π" pi )
```

These commands draw a tic mark at π in addition to the automatically generated ones. If you omitted the `add` keyword here, the second line would clobber the first, and the *only* tic mark would be the one explicitly set at π.

You can also use the `add` keyword to *override* the label of an autogenerated tic mark.[11] Consider the following:

```
set xtics -5,1,5                        ← Creates major tic marks at all integer
                                          values between -5 and +5
set xtics add ( "" 0, "max" 5 )         ← Wipes out the label (not the tic mark) at 0, and
                                          replaces the label for the right-most tic mark with
                                          the expression "max"
```

> **TIP** You can use `set _tic add` to add individual tic marks or to replace the label for autogenerated ones.

MINOR TIC MARKS

Minor tic marks aren't labeled and are typically drawn smaller than major tic marks. By default, minor tic marks are disabled for linear axes and enabled for logarithmic axes.

Minor tic marks can be switched on using the `m_tics` family of options, where the underscore is again used as a placeholder for any of the usual prefixes:

```
set m_tics [ {int:intervals} ]
```

The optional parameter counts the *number of subintervals* between major tics marks; the number of minor tic marks generated is one less than this number. For example, let's say there are major tic marks at x=0 and x=5, and you'd like to create minor tic marks for all the integers in between. The command `set mxtics 5` will do the trick, because there are five integer-sized intervals between neighboring major tics, but the number of minor tic marks generated is only four.

Minor tic marks are drawn only when there are regularly spaced major tic marks. If all major tics are individually placed, `m_tics` has no effect. You can still create minor tic marks manually, using `set _tics`.

---

[11] I'd like to thank Christoph Bersch for making me aware of this possibility.

A WORKED EXAMPLE

You may have wondered how I generated the plot in figure 4.2 using two different y axis scales, each covering only part of the plot. Now you have all the information at hand to discover the secret.

The plot shows the same data twice, but vertically shifted. You can achieve this—without using inline transformations!—by plotting the data once against the primary and once against the secondary vertical coordinate axis, with properly chosen plot ranges for both axes. Note how the visible range (from min to max) is the same for both systems, but the two ranges are offset from each other.

Next, the tic marks are added. The following listing specifies both a start *and* an end value for tic-mark generation—this way, you partially label each axis, but only for the part of the plot that's relevant to each curve.

> **Listing 8.8   Commands for figure 4.2 (file: missing.gp)**

```
set yrange [9:16]
set y2range [6:13]

set ytics 9,1,12 nomirror
set y2tics 10,1,13

set datafile missing "NaN"

plot [0:10] "missing" u 1:2 ax x1y2 w linesp lt 1 pt 7,
➥          "" u 1:($2) ax x1y1 w linesp lt 1 pt 7
```

8.3.5   *Reading tic labels from file*

Finally, gnuplot can read the tic labels from the input file using the `_ticlabels()` family of functions as part of the `using` directive to the `plot` command. (The underscore `_` is again a placeholder for any of the axis identifiers from table 8.1.)

Let's look at the data file in listing 8.9. The x values are present both in numeric form (column 1) and as strings. Of course, it would be nice to use the strings for the tic labels. Here's how to do that:

```
plot "months" u 1:2:xtic(3) w linesp
```

The `xtic()` function takes as argument the number of a column to be used for the tic labels. Equivalent functions exist for the other coordinate axes; labels for the y axis are specified after labels for the x axis.

> **Listing 8.9   Time series data for figure 8.7 (file: months)**

```
# Month   Data      Month Name
1         3         Jan
2         4         Feb
3         2         Mar
4         5         Apr
5         8         May
6         7         Jun
```

```
7        4        Jul
8        5        Aug
9        3        Sep
10       2        Oct
11       4        Nov
12       2        Dec
```

When you use any of the `_tic()` functions in a `plot` command, tic marks and labels are only drawn at the locations *explicitly* read from the data file—in other words, auto-generation of tic marks is turned off.

The `_tic()` functions allow a variety of further tricks beyond reading an explicit string from the data file. The following snippet calculates the quarter of the year for each month (such as Q1 or Q2) and uses it as the tic label:

```
plot "months" u 1:2:xtic( 'Q' . (1+int(($1-1)/3) )) w lp
```

In sections 8.4.1 and 8.4.2, I'll show you two more ways to produce figure 8.7 (shown later in the chapter) from the file in listing 8.9.

8.3.6   *Grid and zero axis*

In addition (or as an alternative) to tic marks along the border of the graph, you can overlay a scale grid on the data. Grid lines are drawn at the position of major and, optionally, minor tic marks:

```
set grid [ [no]_tics ] [ [no]m_tics ]
         [ layerdefault | front | back ]
         [ polar [ {flt:angle} ] ]
         [ majorlineoptions [, minorlineoptions] ]
```

You can switch the grid on to be drawn at major or minor tic marks, for the primary or secondary coordinate system. (The underscore again must be replaced by any one of the prefixes from table 8.1.) Tic marks must be enabled—instructions to draw a grid at nonexistent tic locations are ignored. The grid is drawn either in front of (`set grid front`) or behind the data (`set grid back`).[12] The lines to use for the grid can be set separately for grid lines drawn at major and minor tic marks. If no style or type is given, gnuplot uses the style `ls 0`, which draws the least-visible lines possible (often using a dotted line). The `polar` option is relevant only for plots using polar coordinates, which we'll discuss in section F.2.1.

Similar to grid lines, but less obtrusive, are zero axes. These are lines drawn across the graph for all the points where one of the coordinates is equal to zero:

```
set _zeroaxis [ lineoptions ]
```

---

[12] The option `layerdefault` is only relevant for three-dimensional graphs—see the standard reference documentation for details.

For example, `set xzeroaxis` switches on a horizontal line at *y*=0 (representing the x axis). The default line type is `ls 0`, the same as for the grid.

A WORKED EXAMPLE

A quick example will help to clarify. (See the following listing and figure 8.6.) Turning on the grid is simple; more interesting are the tic marks and labels along the horizontal axis.

Listing 8.10   The commands for figure 8.6 (file: ticmarks.gp)

```
set grid
set xtics pi format "%.0Pπ"

plot [-4*pi:4*pi] sin(x)/x
```

The command `set xtics pi` instructs gnuplot to place major tic marks at multiples of π. The format specification `format "%.0Pπ"` tells gnuplot to use the integer multiple of π as a label. Tacking on the character π makes it clear to the viewer what the tic marks represent.

It's important to understand that the format specifier `%P` interprets a value as a multiple of π but by itself does *not* ensure that tic marks will only be drawn at integer multiples of π. Instead, you must explicitly choose the locations where tic marks will be drawn using `set xtics pi` and then use `format "%P"` to format the labels at those positions accordingly.
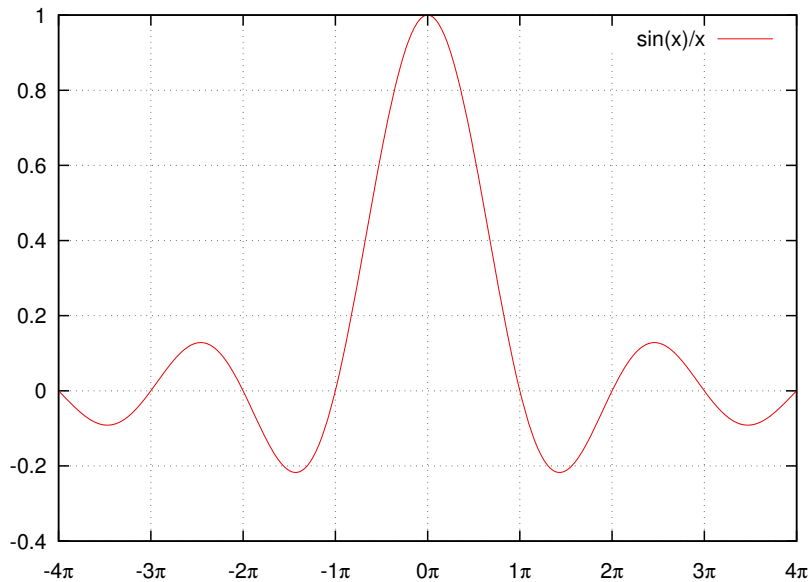
Figure 8.6   Tic marks at multiples of π and Greek letters used for the tic labels. See listing 8.10.

8.4   *Special case: time series*

Whenever you want to study how a quantity changes over time, you're dealing with a time series.[13] Time series are incredibly common—from stock charts and opinion polls to fever curves. Unfortunately, they pose special challenges, because the tic labels you'd like to use for the x axis (such as the names of months or weekdays) aren't strictly numeric. Worse, they aren't even universal, but locale-dependent. If you want to plot time-series data, you therefore need to be able to parse arbitrary date/time formats from a file, and you must be able to format timestamps in a suitable, locale-dependent format.

Gnuplot offers three different ways to deal with date/time information as part of axes labels:

- The classic approach uses `set _data`, `set timefmt`, and `set _tics format`, which let you parse and reformat arbitrary date/time information in the input file; I'll describe these in detail in section 8.4.2. This is gnuplot's most flexible and powerful method of handling timestamps.
- Alternatively, gnuplot can read fully formatted tic labels directly from the input file using the `_ticlabels()` functions introduced in section 8.3.5. This method is easy if the labels read from file are in the format you want, but it only works if the data set also contains numeric time information (to fix the x coordinate of the data points). It's also only suitable for relatively sparse data sets, because `_ticlabels()` creates a tic mark and label for every data point. And it doesn't allow locale-dependent formatting.
- For special cases when you don't require arbitrary date/time labels but merely want to use the names of months or weekdays in a plot, gnuplot provides the simplified `set _mtics` and `set _dtics` facilities (see section 8.4.1).

In the next section, we'll first discuss the simpler case of using month or weekday names as tic labels. Afterward, we'll tackle the harder problem of dealing with arbitrary date/time information, both for input and for output. (Reading tic labels in files was discussed previously in section 8.3.5.)

Locales

The names of months and days of the week differ from country to country and language to language. The set of parameters defining such details of the user interface is called the *locale*. By examining the current locale, gnuplot determines the strings to be used in tic labels involving the names of months or weekdays.

The default is taken from the `LANG` environment variable but can be changed using the following command:

```
set locale ["{str:locale}"]
```

---

[13] If the time intervals between successive measurements are equal, the data points form a *discrete time series*; otherwise, the time series is *continuous*. I have more to say about time-series analysis in chapter 14.

> The choice of available locales is system-dependent. On Unix systems, you can use the shell command `locale -a` for a list of available locales or check the directory /usr/share/locale/. Note that some locales have country-specific variations (such as `en_AU`, `en_CA`, `en_GB`, and `en_US`). In this case, it may not be sufficient to set the general locale (such as `en`), and a more specific locale must be chosen.

8.4.1 *Turning numbers into names: months and weekdays*

Gnuplot provides two simple commands to turn numbers into the names of months or days of the week. They offer much less flexibility than the general time-series commands discussed in the next section, but they're easy to use.

Let's look back at the data file in listing 8.9. Suppose you want to label the x axis with the names of the months, but without using the explicit names in the third column. You can do this using the `set  xmtics` command, which maps numbers to names of months (with 1="January", ..., 12="December"). Don't confuse this command with the `set mxtics` command introduced in section 8.3.4, which switches on minor tic marks!

The following sequence of commands produced the plot in figure 8.7:

```
set xmtics                          ⟵──── Sets format of major tic marks to months
plot [][0:10] "months" u 1:2 w linesp
```
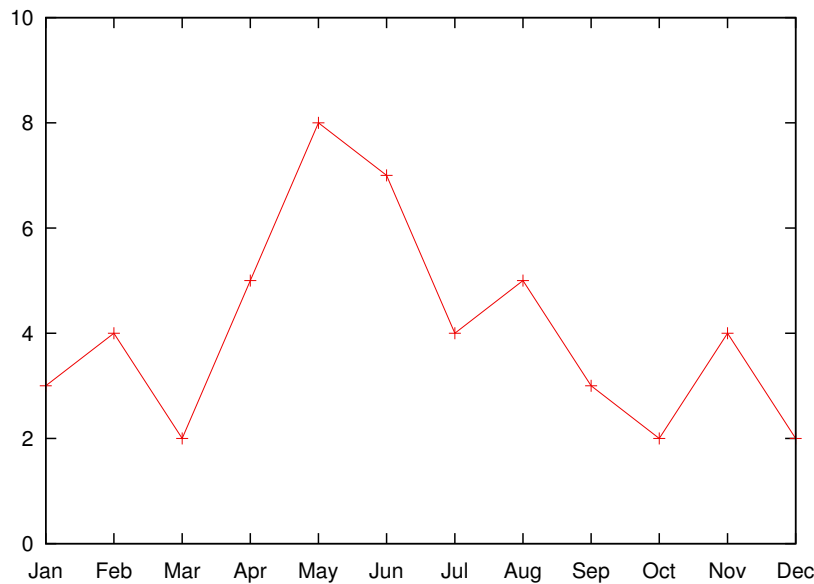

Figure 8.7   A time-series plot, using `set xmtics` to use months for the labels along the horizontal axis. Also see listings 8.11 and 8.12. The data is shown in listing 8.9.

In addition to the names of months, you can use days of the week (such as "Mon", "Tue", and so on) as tic labels. You enable them using `set xdtics` (with 0="Sunday", ..., 6="Saturday"), similar to what you've seen for `set xmtics`. Both `set xmtics` and `set xdtics` map overflows back into the legal range through a modulo operation (modulo 12 and modulo 7, respectively), as you'd expect. To restore normal (numerical) axes labeling, use `unset xmtics` or `unset xdtics`. Equivalent commands exist for all other axes, using the usual prefixes in table 8.1.

The `set xmtics` and `set xdtics` facilities are very limited. The strings follow the locale but can't be formatted any further. But as a quick-and-easy alternative to the general methods described next, the techniques introduced in this section are occasionally useful.

8.4.2   *General time series: the gory details*

For more general time series, you face two problems: you must be able to *read* arbitrary timestamps from the input file and then *format* them again for output. But before you can do anything, you must first put gnuplot into time-series mode (for the primary x axis) using

```
set xdata time
```

Issuing `set xdata` (without an argument) restores normal operation. Equivalent commands exist for all other axes, distinguished through the usual prefixes.

> **TIP**   To handle general date and time information in data files, you must first switch to time-series mode using `set xdata time`. In time-series mode, timestamps are parsed (for input) according to the value of `set timefmt` and are formatted (for output) according to `set _tics format`.

In time-series mode, input (parsing of timestamps from files) and output (formatting timestamps for inclusion in the plot) are controlled by the commands `set timefmt` for input and `set xtics format` for output.[14] Both accept a format string using a syntax similar to that found in the POSIX `strftime()` routine. (I introduced `set _tics format` in section 8.3.3 already, but there we only talked about the formatting of plain *numbers*. Now I explain the ability to format complex date/time values.)

Gnuplot assumes that all data is in universal time coordinates (UTC)—it has no facilities to perform time zone changes, adjust for daylight savings, or apply similar transformations. If they're required, they must be applied externally, before you attempt to plot the data.[15]

---

[14] The command `set format x` is an older alternative for controlling the output format. It's now redundant, but you'll still find references to it in the standard reference documentation.

[15] Gnuplot 5 stores timestamps internally as seconds since the beginning of the Unix epoch. This is a change from previous versions of gnuplot.

INPUT

Time/date information is parsed in a way reminiscent of the scanf() family of functions and shares its familiar challenges. The expected input format is indicated through a format string to set timefmt. The format string may contain several conversion specifiers, all of which begin with the % character, followed by a letter that indicates how an input value should be interpreted. Check tables 8.4 and 8.5 for a list of all possible conversion specifiers and their meanings.

The input format string may contain other characters besides format specifiers, but input strings must match the format *exactly* (with some exceptions regarding whitespace that we'll discuss shortly):

```
set timefmt "%Y-%m-%d"          ⟵——— Matches 2000-01-01, but also 2000-1-1
set timefmt "%d%b%y"            ⟵——— Matches 1JAN05, 01Jan05, 1jan05
```

If there are no characters separating different fields from one another, gnuplot consumes a fixed number of characters per field (left to right), and the fields must be left-zero-padded as necessary:

```
set timefmt "%Y%m%d"            ⟵——— Parses 20020101 as Jan 1, but parses 2002101
                                      as Oct 1, and will fail to parse 200211
```

Special rules apply when the date format to be parsed contains whitespace. First, gnuplot interprets whitespace-separated data as occupying several columns. A timefmt format containing whitespace in turn consumes several columns. A blank space (*not* an escaped tab: \t) embedded in a formatting string matches one or more blanks (not tabs) in the input file. So, "%H %M" matches 12␣20 and 12␣␣␣20 but not 1220. (The ␣ symbol indicates a whitespace character.)

An example will help. The following input file

```
2005-01-01     8:41      3
2005-01-01     9:17      4
2005-01-01    22:46      2
2005-01-02    03:05      5
```

will be correctly parsed and plotted by the following commands:

```
set xdata time
set timefmt "%Y-%m-%d %H:%M"
set xtics format "%F\n%T" rotate
plot "data" u 1:3 w lp
```

Note that the column used for the y values is the *third*, because the time format consumes two columns. Also, the format string contains a single whitespace, but in the data file several blanks separate the date from the time. The file won't parse correctly if the spaces between date and time are replaced by tabs.

Table 8.4   Alphabetically sorted conversion specifiers for date/time information for the `set _tics format` and `set timefmt` commands. See table 8.2 to format numeric values. See table 8.5 for a list sorted by topic.

Conversion specifier	Available for …		Values	Description
	input: `set timefmt`	output: `set _tics format`		
%a		✓	Sun, Mon, …	Abbreviated day of the week
%A		✓	Sunday, Monday, …	Full day of the week
%b	✓	✓ (also %h)	Jan, Feb, …	Abbreviated name of the month (three characters)
%B	✓	✓	January, February, …	Full name of the month
%d	✓	✓	01–31	Day of the month (always two digits on output)
%D		✓	Example: "03/25/08"	Shorthand for "%m/%d/%y" (US date format)
%F		✓	Example: "2008-03-25"	Shorthand for "%Y-%m-%d" (extended ISO date format)
%H	✓	✓	00–24	Hour—24-hour clock (always two digits on output)
%I		✓	00–12	Hour—12-hour clock (always two digits)
%j	✓	✓	001–366	Day of the year (always three digits on output)
%k		✓	0–24	Hour—24-hour clock (one or two digits on output)
%l		✓	0–12	Hour—12-hour clock (one or two digits)
%m	✓	✓	01–12	Month (always two digits on output)
%M	✓	✓	00–60	Minute (always two digits on output)
%p		✓	"am", "pm"	a.m./p.m. indicator
%r		✓	Example: "10:55:48 pm"	Shorthand for "%I:%M:%S %p" (US time format)
%R		✓	Example: "22:12"	Shorthand for "%H:%M" (24-hour clock time format without seconds)
%s	✓		0–…	Unix epoch seconds (input only!)
%S	✓	✓	00–60	Seconds (always two digits on output)
%T		✓	Example: "22:12:48"	Shorthand for "%H:%M:%S" (extended ISO time format: 24-hour clock with seconds)
%U		✓	00–53	Week of the year (weeks starting on Sunday; always two digits)
%w		✓	00–06	Day of the week (0=Sunday; always two digits)
%W		✓	00–53	Week of the year (weeks starting on Monday; always two digits)
%y	✓	✓	00–99	Year (two-digit; always two digits on output)
%Y	✓	✓	0000–9999	Year (four-digit; always four digits on output)

Table 8.5 Conversion specifiers for date/time information for the `set _tics format` and `set timefmt` commands, sorted by topic. See table 8.2 to format numeric values. See table 8.4 for a list sorted alphabetically by conversion specifier.

Conversion specifier	Available for		Values	Description
	input: `set timefmt`	output: `set _tics format`		
`%s`	✓		0–…	Unix epoch seconds (input only)
`%S`	✓	✓	00–60	Seconds (always two digits on output)
`%M`	✓	✓	00–60	Minute (always two digits on output)
`%k`		✓	0–24	Hour—24-hour clock (one or two digits on output)
`%H`	✓	✓	00–24	Hour—24-hour clock (always two digits on output)
`%l`		✓	0–12	Hour—12-hour clock (one or two digits)
`%I`		✓	00–12	Hour—12-hour clock (always two digits)
`%p`		✓	"am", "pm"	a.m./p.m. indicator
`%j`	✓	✓	001–366	Day of the year (always three digits on output)
`%d`	✓	✓	01–31	Day of the month (always two digits on output)
`%m`	✓	✓	01–12	Month (always two digits on output)
`%b`	✓	✓ (also `%h`)	Jan, Feb, …	Abbreviated name of the month (three characters)
`%B`	✓	✓	January, February, …	Full name of the month
`%y`	✓	✓	00–99	Year (two-digit; always two digits on output)
`%Y`	✓	✓	0000–9999	Year (four-digit; always four digits on output)
`%w`		✓	00–06	Day of the week (0=Sunday; always two digits)
`%a`		✓	Sun, Mon, …	Abbreviated day of the week
`%A`		✓	Sunday, Monday, …	Full day of the week
`%W`		✓	00–53	Week of the year (weeks starting on Monday; always two digits)
`%U`		✓	00–53	Week of the year (weeks starting on Sunday; always two digits)
`%R`		✓	Example: "22:12"	Shorthand for "%H:%M" (24-hour clock time format without seconds)
`%T`		✓	Example: "22:12:48"	Shorthand for "%H:%M:%S" (extended ISO time format: 24-hour clock with seconds)
`%r`		✓	Example: "10:55:48 pm"	Shorthand for "%I:%M:%S %p" (US time format)
`%F`		✓	Example: "2008-03-25"	Shorthand for "%Y-%m-%d" (extended ISO date format)
`%D`		✓	Example: "03/25/08"	Shorthand for "%m/%d/%y" (US date format)

Finally, gnuplot won't parse strings enclosed in quotes (see section 4.3.3). Therefore, you can't parse a file that contains date/time information as strings with embedded whitespace:

```
"2005-01-01 8:41"    3
"2005-01-01 9:17"    4
```
Will *not* parse

Also keep in mind that gnuplot's capabilities when it comes to parsing date/time information depend on the capabilities (and sensibilities) of the underlying system libraries. Don't expect edge cases to be absolutely portable!

OUTPUT

Compared to parsing time/date information, it's much easier to format the information into human-readable tic labels. To do so, specify the desired output format using `set _tics format "..."` (where the underscore again is a placeholder for any of the possible prefixes from table 8.1), and you're done.[16]

The format string can contain arbitrary text in addition to the formatting characters. Here's a useful snippet to stack the time on top of the date (note the embedded newline):

```
set xtics format "%T\n%D"
```
⟵——— **Time stacked on top of date**

But you can also use other text, such as formatting characters

```
set xtics format "%Y-%m-%D %H:%M"
```
⟵——— **Date followed by time**

and even plain text:

```
set xtics format "It happened on %A"
```
⟵——— **Full day of the week**

In particular when used together with string functions, there is almost no limit to the appearance of tic labels for plots displaying time series.

WORKING IN TIME-SERIES MODE

Keep in mind that when working in time-series mode (after issuing the `set xdata time` command), all x coordinates are interpreted according to the current setting of the `set timefmt` format option (or equivalent for all other axes). In particular, this means plotting ranges must be specified as quoted strings, in the format defined by `timefmt` (the *input* time format): `plot ["01Jan00":"15Jan00"] "data" u 1:2`. Similar concerns hold for the coordinates supplied to `set arrow` and `set label`.[17]

Also, don't forget that the currently selected locale (as inherited from the environment when gnuplot was started, or set using `set locale`) affects the tic labels (names of months and of days in the week).

---

[16] I also recommend that you avoid the former way of formatting tic labels based on the `set format` command. The new style, based on `set xtics format`, is both clearer and more robust.

[17] Alternatively, you can specify ranges and positions as integers, which are interpreted as Unix epoch seconds.

And finally, always remember that you need to switch on time-series mode explicitly using `set xdata time`. This necessary step is easy to forget, and doing so will lead to mysterious errors.

> **TIP** Before parsing date/time information from files and using it to plot data, you must enable time-series mode using `set xdata time`.

A WORKED EXAMPLE

Remember the file with monthly data from listing 8.9? In section 8.3.5, gnuplot read the tic labels for the horizontal axis directly from the data file. In section 8.4.1, the plot was created using the simple `set xmtics` formatting. Now you'll see a third way of plotting the file, this time using the full machinery of gnuplot's general time-series mode.

The next listing shows one way to do it. In this version, the strings in the third column aren't used. (See tables 8.4 and 8.5 for the conversion specifiers.) The resulting graph is exactly the same as figure 8.7.)

Listing 8.11   Commands for figure 8.7 using time-series mode (file: months1.gp)

```
set xdata time                 ◁──── Turn on time-series mode!

set timefmt "%m"               ◁──── Input format: 1, 2, …, 12
set xtics format "%b"              ◁──── Output format: Jan, Feb, …

plot [][0:10] "months" u 1:2 w lp          ◁──── Plot using the first column.
```

Amusingly, there's another way to achieve the same effect. Remember that the data in listing 8.9 is redundant: the first and third columns *both* contain information that can be interpreted as months. You can therefore ignore the first (numeric) column and base the graph on the third column. The following listing shows this approach.

Listing 8.12   Alternative commands for figure 8.7 (file: months2.gp)

```
set xdata time                 ◁──── Turn on time-series mode!

set timefmt "%b"               ◁──── Input format: Jan, Feb, …
set xtics format "%b"              ◁──── Output format: Jan, Feb, …

plot [][0:10] "months" u 3:2 w lp          ◁──── Plot using the third column.
```

Notice how the strings in the third column are used to give the horizontal position for each point. (You may want to change the output format to full, rather than abbreviated, month names using `set xtics format "%B"` to convince yourself that the strings read from the file are actually parsed and interpreted and aren't just passed through to the graph.)

8.4.3   Beyond tic labels: processing date/time information

The functionality I just described for use with time-series plots can be extended to more general applications. Most date/time manipulations involve the first three functions in table 8.6.

Table 8.6   Functions to parse and format arbitrary time strings. See tables 8.4 and 8.5 for conversion specifiers.

Function	Description
strftime(*format, seconds*)	Formats its second argument, which is interpreted as Unix epoch seconds, according to the format string in the first argument. Returns a formatted string.
strptime(*format, string*)	Parses its second argument, which must be a string containing a timestamp, according to the format string in the first argument. Returns the corresponding number of Unix epoch seconds as a floating-point number.
timecolumn(*column, format*)	Takes an integer column identifier and a format string. Parses the current entry in the identified column according to the format string. Returns the corresponding number of Unix epoch seconds. Equivalent to strptime(*format*, stringcolumn(*column*)).
time(*dummy*)	Returns the current system time in Unix epoch seconds. A numeric argument must be supplied, but its value is ignored.
tm_year(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the full year as a floating-point number.
tm_yday(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the day of the year as a floating-point number (0.0–365.0).
tm_mon(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the number of the month as a floating-point number (0.0–11.0).
tm_mday(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the day of the month as a floating-point number (1.0–31.0).
tm_wday(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the day of the week as a floating-point number (Sunday=0.0).
tm_hour(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the hour as a floating-point number.
tm_min(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the minute as a floating-point number.
tm_sec(*seconds*)	Takes a timestamp in Unix epoch seconds, and returns the second as a floating-point number.

As an example, consider again the list of athletic records in listing 3.1. This data set contains not only the finishing times, but also the calendar date when the record was set. You may want to include this information in a plot of the data (see figure 8.8).
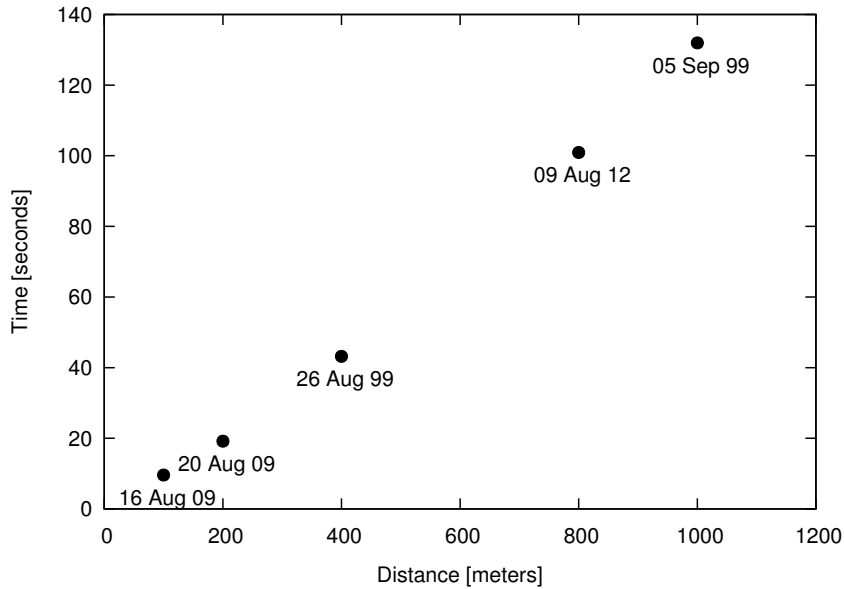
Figure 8.8   Men's world records, together with the date when the record was set. The dates are read from the data file, reformatted, and placed onto the graph. See the text for details and listing 3.1 for the data.

This `plot` command to create this figure is shown here:

```
plot [0:1200] "records"
➡              u 1:2:(strftime( "%d %b %y", timecolumn(4, "%Y-%m-%d")))
➡              w labels point pt 7 offset 0,-1
```

The primary plot style is `with labels` (see section 6.3.5), using the ability to plot a point symbol together with the label. The labels are first *parsed* (using `timecolumn()`) and then *formatted* (using `strftime()`). Notice that the date format in the graph is different than in the data file—the date strings are parsed and reformatted, not merely passed through to the plot.

Instead of using the `timecolumn()` function, you can use `strptime()` in conjunction with the basic `stringcolumn()` function:

```
plot [0:1200] "records"
➡              u 1:2:(strftime("%d %b %y", strptime("%Y-%m-%d", strcol(4))))
➡              w labels point pt 7 offset 0,-1
```

This example will suffice to demonstrate the possibilities. Keep in mind that `timecolumn()` and `strptime()` return a *numeric* representation of a timestamp, thus making all kinds of further date calculations and manipulations possible.

8.5   Summary

In this chapter, we talked about all the ways you can control the appearance of the axes on a plot. It's been a long chapter, but axes—or rather, the tic marks and labels placed on them—are important: they enable the viewer to gain *quantitative* insight from the data displayed in the plot.

This chapter concludes our overview of what I would call "basic" gnuplot. In the following chapters, we'll look at some exciting but distinctly more advanced topics, such as color in graphs, customizations, and scripting capabilities. You'll also learn just about everything there is to know about exporting graphs to standard file formats.

Part 3

Mastering technicalities

The topics treated in this part are of a more technical nature than in the preceding chapters. You'll need the material in these chapters when you're no longer satisfied with gnuplot's defaults and want to establish greater control over the appearance of your figures, or if you want to automate and configure the way you work with gnuplot. Chapter 9 explains how to customize many aspects of graph elements, such as their color, the shape of point symbols, and the dash patterns used to draw lines. You'll also learn how to change the overall appearance of a graph, such as its size and aspect ratio. Chapter 10 describes in detail how you can export graphs to file, using gnuplot's terminal capabilities. Chapter 11 discusses ways to automate repetitive tasks, either using gnuplot's own scripting facilities or by using gnuplot in conjunction with scripts or programs written in a separate programming language. Chapter 12 presents options to customize your gnuplot work environment.

Color, style, and appearance

This chapter covers

- Global appearance options
- Color
- Lines and points
- Size and aspect ratio

Whereas the previous few chapters dealt with very local aspects of a graph (such as an individual curve, a single text label, or the details of tic mark formatting), the present chapter addresses global issues. We begin with a long and important section on *color*. Color is an exciting topic, and it opens a range of options for data visualization; I'll explain gnuplot's support for color handling in detail.

Next, we discuss a few details that matter if you want to modify the default appearance of point types, dash patterns, and the automatic selection of visual styles. The overall *size* and *aspect ratio* of the graph are the subjects of the closing section of this chapter.

9.1 Color

Color is an important element in visualization. At the base level, colors are often easier to distinguish than dash patterns or different symbol shapes. But colors open

181

many more options for visualization—in fact, colors by themselves are sufficient to represent semantic information (we'll come back to this in appendix D). And finally, color is just plain fun!

At the same time, color adds complexity: what colors look good, and which are suitable for representing information? And then there's the problem of representing color in computer systems. In appendix D, we'll have more to say about choosing colors for visualization; here we're only interested in the technicalities of specifying color in gnuplot.[1]

There are three fundamentally different ways to select a color in gnuplot:

- Through *explicit specification*, either by using a known color name (such as `"red"`) or by giving its red-green-blue (RGB) components (like `"#FF0000"`). An explicit color specification is indicated by the keyword `rgb`.
- By *indexed lookup* from a selection menu of discrete colors. (For example, `lc 1` picks the *first* color from an existing list of color styles.) Indexed lookup is the default and doesn't use a special keyword.
- By *mapping a value into a continuous color gradient*. Color gradients are called *palettes* in gnuplot terminology (with keyword `palette`) and are the principal topic of appendix D. In this chapter, we'll give only a brief preview.

Each one of these methods can be used in two ways:

- *Fixed*—A specific color is selected for a given data set, to distinguish this data set from any other.
- *Variable or data-dependent*—The color is determined by the value of the data plotted so that the curve changes its color depending on location. Data-dependent coloring is switched on with the keyword `variable` in place of an explicit color name or index, and with `palette z` for gradient-mapped colors.

Keeping all these possibilities straight is one of our primary goals in this section!

No matter which way you use color, a color specification always requires the `line-color` keyword for graph elements (not just lines, but all graph elements, like points or boxes) and the `textcolor` keyword for all forms of text labels.

> **TIP** Remember: all color specifications are preceded either by the keyword `linecolor` (or `lc`) for graph elements or by `textcolor` (or `tc`) for text labels.

9.1.1 *Explicit colors*

The most fundamental way to indicate a color is to specify it explicitly. You can do so by giving its name or by spelling out its components.

---

[1] To find the pertinent information in the standard gnuplot reference documentation, use the command `help colorspec`.

COLOR NAMES

Several colors can be referred to by their *name*. Gnuplot knows 111 predefined color names (corresponding to only 95 distinct colors; the duplications are mostly due to both *grey* and *gray* being valid spellings). You can see all the color names and their corresponding RGB values using the `show colornames` command. Note that gnuplot's list of color names isn't identical to the list of X11 colors.[2]

RGB COMPONENTS

The most common way in computer graphics to indicate an arbitrary color is through its RGB (red-green-blue) values: a triple of numbers giving the relative intensities of the red, green, and blue components that make up the desired color. By convention, each component is represented through a single byte, so that each RGB component lies between 0 and 255. Conventionally, the components are written as two-digit hexadecimal numbers (from `00` to `FF`), and an entire triple can be represented through a string of the following format:

```
rgb "#RRGGBB"
```

This is the *X11 format.* You can also write the string explicitly as a hexadecimal constant in the format `"0xRRGGBB"`. Here are three equivalent ways to specify the same color:

```
rgb "red"
rgb "#FF0000"
rgb "0xFF0000"
```

Instead of representing the RGB triple as a string, you can also use the integer value corresponding to the value of the three bytes making up the triple. It may be convenient to use the hexadecimal representation of the integer, instead of the representation to base 10. The following are equivalent:

```
rgb 0xff0000
rgb 16711680
```

Here's a function that takes the three components (as integers between 0 and 255) and packs them into an equivalent integer value:

```
pack( r, g, b ) = 2**16*r + 2**8*g + b
```

This gives you an additional way to obtain the color red:

```
rgb pack( 255, 0, 0 )
```

---

[2] The Wikipedia pages for "X11 color names" and "Web colors" contain more information about commonly used color names. The amusing history of color names used in graphical user interfaces can be found in the article "'Tomato' versus '#FF6347'—the Tragicomic History of CSS Color Names" by Julianne Tveten (*Ars Technica*, October 11, 2015); http://mng.bz/R56B.

HSV COLOR MODEL

The RGB color model has the advantage of corresponding directly to the way a computer display operates, but it's not particularly intuitive for a *human* to use. (Quickly, what does `#2e8b57` look like?)

The HSV (hue-saturation-value) scheme is an alternative to the RGB color description and the most popular intuitive color model. The *hue* describes the shade of the color (by convention, a hue value of 0 corresponds to red followed by yellow, green, cyan, blue, and magenta at equal intervals). The *saturation* measures the richness of the color, from pale pastel shades to full saturation. Finally, the *value* element of the HSV triple describes the lightness of the color, from very dark to very bright. All HSV values vary from 0 to 1.[3]

Gnuplot's built-in function `hsv2rgb( h, s, v )` takes a triple of HSV values, performs the conversion to RGB, and returns an integer-packed representation of the corresponding RGB triple. This gives you yet another way to specify the color red (remember that red corresponds to a hue value of 0):

```
rgb hsv2rgb( 0, 1, 1 )
```

Table 9.1 shows all these forms together, this time for the color magenta, which is a mixture of full red and full blue components. (Its hue value in the HSV model is 5/6.)

Table 9.1   Equivalent ways to specify a color explicitly (here, magenta)

Description	Example
Color name	`rgb "magenta"`
X11 RGB string	`rgb "#FF00FF"`
String with hex constant	`rgb "0xFF00FF"`
Integer in hex format	`rgb 0xff00ff`
Packed integer	`rgb 2**16*255 + 2**8*0 + 255` or `rgb 16711935`
Conversion from HSV	`rgb hsv2rgb( 5/6., 1, 1 )`

TIP   The input values to the `hsv2rgb()` function *must* all lie in the interval [0:1]. The function silently replaces values outside this interval with the nearest legal value (so, 1.1 becomes 1.0, and -0.1 becomes 0). In particular, the hue parameter does *not* wrap around for values greater than 1.0—you must map it into the unit interval yourself!

You can use the expression `x – floor(x)` to reduce an arbitrary hue value to the unit interval. You may want to define a more general version of the `hsv2rgb()` function that removes the restriction on the hue value:

```
hsv( h, s, v ) = hsv2rgb( h – floor(h), s, v )
```

---

[3]   The Wikipedia page for the HSL and HSV color models provides more detail.
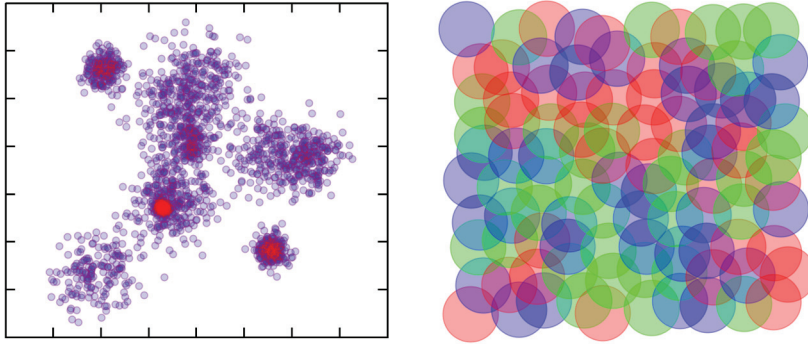
Color figure 1 Alpha shading and transparency. When a partially transparent color is added to an existing background, the added (foreground) color tends to prevail; adding it to a background of the same hue just increases the intensity. In the left panel, this effect is used to visualize point density in a dense data set. All data points are drawn with both blue disks and red rims. Because contributions from overlapping points add visually, regions of high point density show up as areas of high color intensity. Regions where the density is high enough for the rims to contribute significantly appear red. The right panel demonstrates what kinds of color mixtures you can expect when several colors are added together. (See chapter 9 for details.)
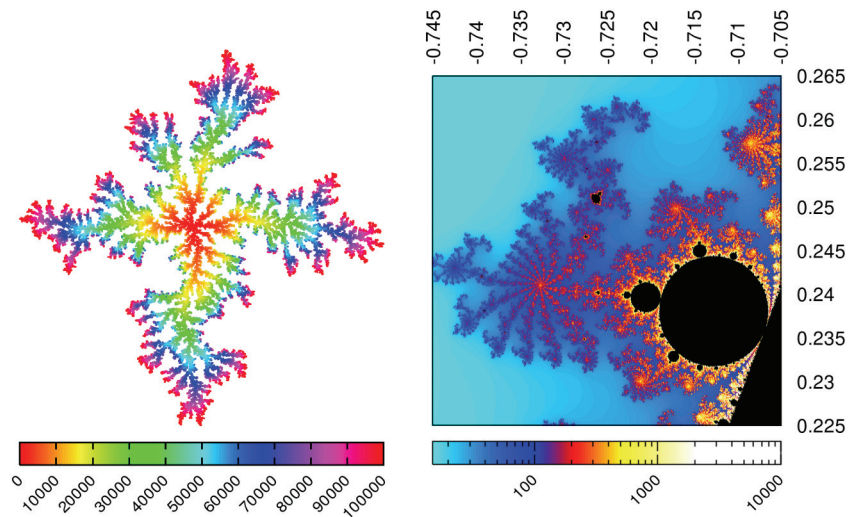
Color figure 2 Using color gradients to visualize data values. In the left panel, color is used to indicate the order in which particles were added to the cluster. The right panel shows a section of the complex plane, including part of the Mandelbrot set (black) and its fractal boundary. Color is used to visualize the number of iteration steps before the Mandelbrot iteration diverged. (See chapter 9 and appendix D.)
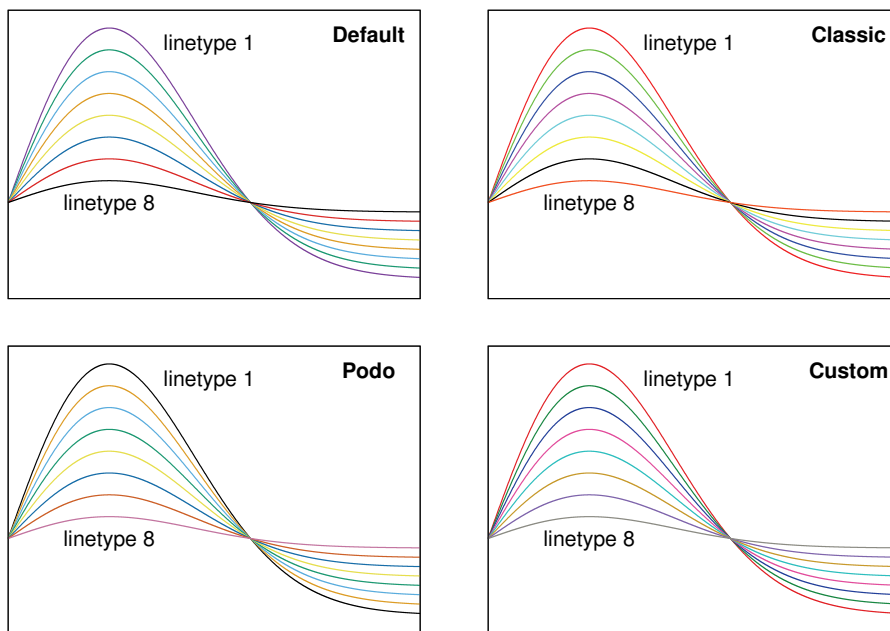
Color figure 3 The three built-in color sequences that can be selected using `set colorsequence`, and the custom sequence that was used for the color illustrations in this book. (See chapters 9 and 12 for details.)
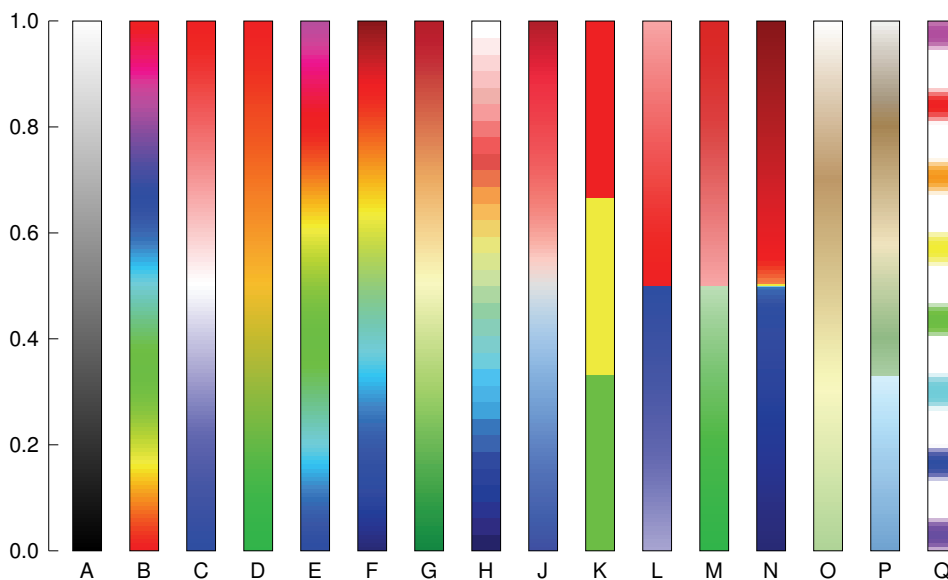
Color figure 4 Several color gradients (or palettes) for visualization purposes, as discussed in appendix D. Each panel in the figure displays a different palette.
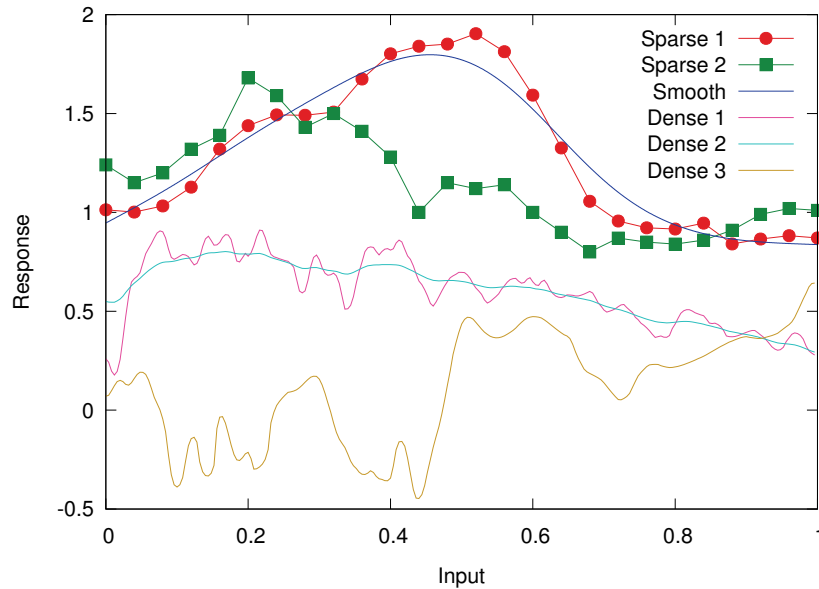
Color figure 5   Using stylesheets to change the appearance of a plot. The plot in this figure has been prepared with a stylesheet that uses relatively bright colors, but thin lines. (Compare color figure 6. Also see listing 12.7 in chapter 12 for details.)
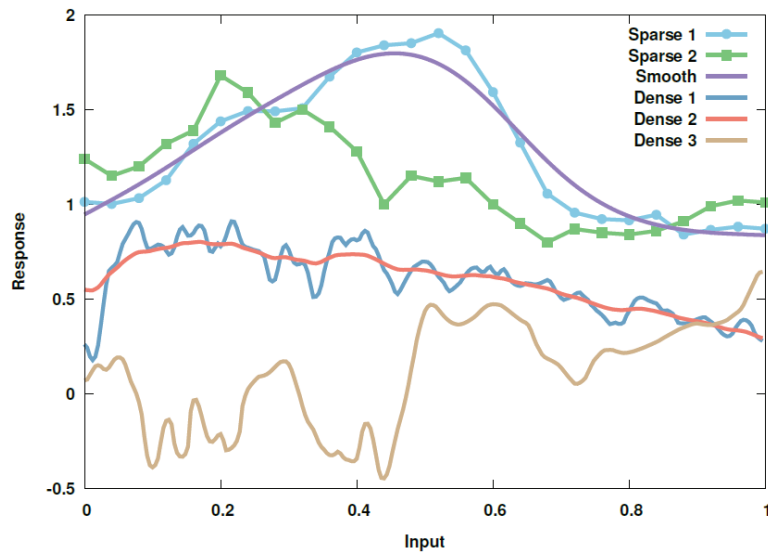
Color figure 6   Using stylesheets to change the appearance of a plot. The plot in this figure has been prepared with a stylesheet that uses soft pastel colors, but relatively thick lines. (Compare color figure 5. Also see listing 12.8 in chapter 12 for details.)
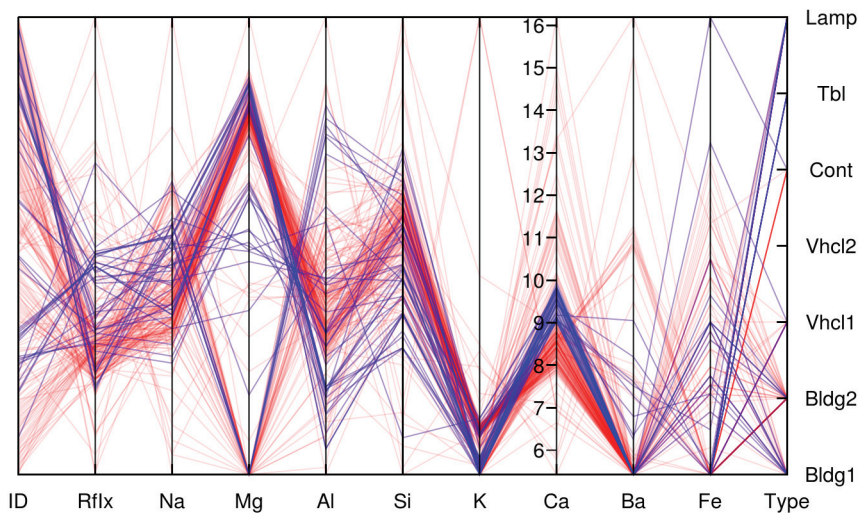
Color figure 7 Using a combination of several graphical techniques to represent a complicated, multivariate data set. The figure shows a parallel-coordinates plot of the entire glass data set. Because of the relatively large number of records, all lines are drawn partially transparent so they don't obscure each other. A subset of records has been highlighted in a different color: records with a Calcium (Ca) content between 9 and 10 have been selected for highlighting. (See chapter 14 for details.)
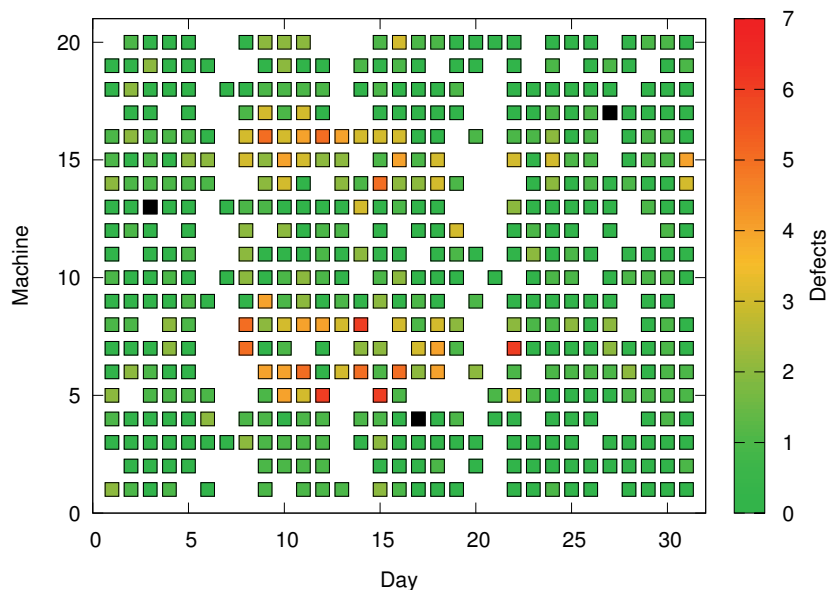
Color figure 8 A false-color plot prepared using a color gradient. The color of each square represents the number of defects produced by each machine on each day of the month. Missing squares indicate that the corresponding machine wasn't used on that day. Many different aspects of the operation are visible in this graph. You should be able to recognize weekends and two clusters of machines that seem more error-prone than the others. This data set contains a handful of outliers (that is, isolated instances with an excessive number of defects), which are drawn in black. (See appendix D for details.)

9.1.2 *Alpha shading and transparency*

So far, we've assumed that colors are fully *opaque*, so that an object drawn on top of another completely hides its background. But gnuplot can do more: several terminals allow for *transparent* colors.[4]

MAKING COLORS TRANSPARENT

Indicating that a pixel should be transparent requires additional information. In practice, you add a fourth byte to the RGB components; this is often called the *alpha channel*. In gnuplot's string representation, it precedes the RGB value:

```
rgb "#ααRRGGBB"
rgb "0xααRRGGBB"
```

Transparency information is encoded in the highest-order bit, as follows:

- $\alpha = 0$: fully opaque
- $\alpha = 255$: fully transparent

This convention has the advantage that a fully opaque color `rgb "#00RRGGBB"` has the same numerical value as `rgb "#RRGGBB"`, so opaque colors don't require special treatment.

ALPHA BLENDING

So, what happens when two transparent colors are overlaid? Obviously, the color "underneath" is still visible—but how? It turns out that there is no obvious answer to this question—the cairo graphics library defines 29 (!) different operators to combine two pixels containing alpha-channel information![5]

Even in gnuplot, the answer isn't unique: lines and similar graph elements are combined using cairo's OVER operator (the library's default), but filled objects use the SATURATE operator. (You can't specify an operator explicitly.)

To make the behavior of these two operators concrete, let's assume that you want to add a new object of color $\alpha_1 X_1$ to an existing color $\alpha_0 X_0$, resulting in color $\alpha_2 X_2$. (Here X stands for any one of the three RGB components.) The new color is then calculated separately for each RGB component according to the following formulas:

$$
\begin{aligned}
\text{OVER} \quad \alpha_2 &= \alpha_1 + \alpha_0(1 - \alpha_1) \\
X_2 &= (\alpha_1 X_1 + \alpha_0 X_0(1 - \alpha_1))/\alpha_2
\end{aligned}
$$

$$
\begin{aligned}
\text{SATURATE} \quad \alpha_2 &= \min(1, \alpha_1 + \alpha_0) \\
X_2 &= (\min(\alpha_1, 1 - \alpha_0)X_1 + \alpha_0 X_0)/\alpha_2
\end{aligned}
$$

---

[4] All cairo-based terminals support transparency, but several of the older terminals don't. In particular, neither the PostScript terminal nor the GD-based bitmap terminals can handle transparency.

[5] See http://cairographics.org/operators.

These formulas aren't so helpful to predict the outcome, but you can use them retrospectively to understand what happened. Two general observations hold (see figure 9.1):

- The two source colors don't commute: it matters which color is in the background and which is newly added. The color that is added on top of an existing background tends to prevail.
- When all pixels are of the same hue, things are simpler: the hue basically stays, and only the intensity increases (in a roughly additive way).
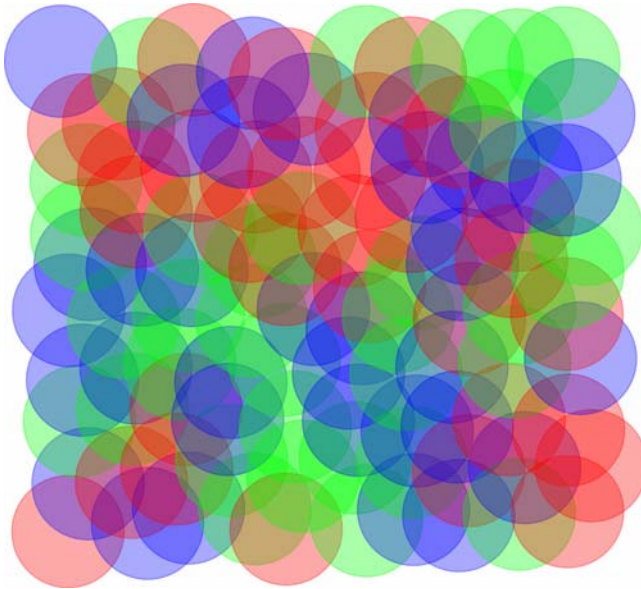
Figure 9.1   Alpha blending in action. The color that is added to an existing background tends to prevail; adding it to a background of the same hue just increases the intensity.

USES FOR TRANSPARENT COLORS AND ALPHA BLENDING

Transparent colors can be useful when you're visualizing *large* data sets consisting of many points. If such data is plotted in a scatter plot, points frequently overlap. With fully opaque colors, overlapping points completely obscure each other, making the local density of points indistinguishable. But with partially transparent colors, overlapping points contribute *additively*, so that a higher color intensity reveals a higher point density.

Figure 9.2 shows such a scatter plot. The plot command is as follows:

Blue filled circles ❶

```
plot [-0.4:1.2][-0.2:1.2] "alpha" u 2:1 w p pt 7 ps 1 lc rgb "#cc0000ff",
➥                         "" u 2:1 w p pt 6 ps 1 lc rgb "#ddff0000"
```

Red unfilled circles ❷
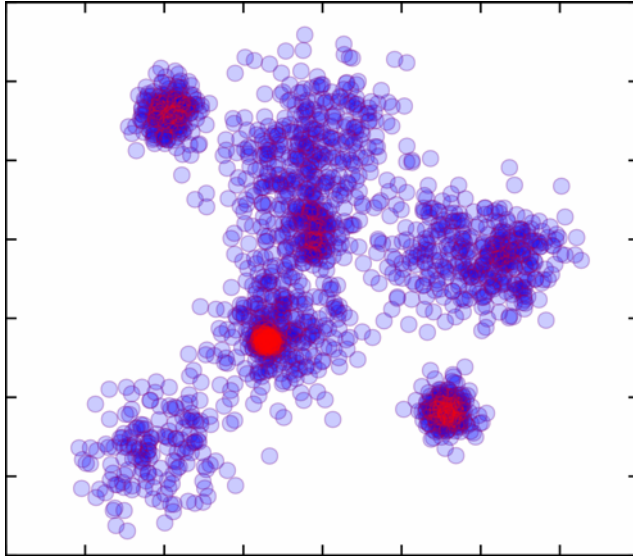
Figure 9.2   Using alpha blending to visualize point density in a dense data set. All data points are drawn both with blue disks and red rims. Because contributions from overlapping points add visually, regions of high point density show up as areas of high color intensity. Regions where the density is high enough for the rims to contribute significantly appear red.

All points are first drawn as filled circles (`pt 7`) in blue ❶ and then again as unfilled circles (`pt 6`) in red ❷. Adding an outline to the circles significantly improves the appearance of the graph; giving the outline a different color sets off areas of truly high point density even more strikingly.

You may want to experiment further with this—for example, I've obtained good results when using the same color (say, blue) for both disk and outline, but making the outline slightly *larger* than the disk itself (for example, `ps 1.5` for the outline, but `ps 1` for the disk).

9.1.3   *Selecting a color through indexed lookup*

The methods explained so far in this section let you select individual colors. When that's what you want, these techniques are convenient: `plot sin(x) lc rgb "red"` does exactly that—it plots the sine in red.

It gets trickier if you want to plot several curves at once. To distinguish them, each should be plotted in a different color, and it would be convenient if gnuplot would do so automatically. In other words, now you want colors to be chosen from a discrete selection (rather than specified individually).

This is exactly what gnuplot does: as I explained in section 6.2, gnuplot maintains a set of distinct colors and (by default) uses the next one for each new function or data

set. (Later in this chapter, you'll learn how to customize the overall selection of colors available in this way.)

Colors (and line types and styles more generally) are identified by a positive integer. If you want to override gnuplot's automatic choice of color, or if you want to pick a particular color from the predefined set, you do so by supplying the appropriate integer value:

```
plot "data" u 1:2 w l lc 2, "" u 1:3 w l lc 2
```

This plots both the second *and* third columns against the first using the second color from the predefined set of colors for both data sets.

All of this is straightforward. There's only one possible glitch: don't confuse `lc 1` with `lc rgb 1`. The former specifies the first color from the selection of predefined colors (whatever color this may be!). The latter selects a *specific* color by giving its RGB components as an integer-packed value. (The value 1 corresponds to `"#000001"`, which is almost black.)

SPECIAL COLOR INDICES

Gnuplot knows about two special color indices: `black` and `bgnd`. The first is a shorthand to obtain a black line through indexed lookup (that is, without having to identify a color explicitly). It's mostly convenient for drawing graph elements like arrows, borders, and labels: `set arrow 1 from 0,0 to 1,1 lc black`. (Note the absence of either quotes or the `rgb` keyword.)

All contemporary terminals allow an explicit background color to be set using the `background` keyword: `set terminal pdfcairo background rgb "grey"` (see chapter 10). This color is then available through the `bgnd` index for all graphs sent to this terminal. This is mostly useful to rub out or erase parts of the graph:

```
set object 1 circle at 0,0 radius 1 fill solid fc bgnd front
```

This clears a circular area of radius 1, centered at the origin, by filling it with the background color.

9.1.4   *Mapping a value into a continuous gradient*

Continuous color gradients (called *palettes* in gnuplot) are the topic of appendix D. But a discussion of color in gnuplot would be incomplete without mentioning them, so I'll give a brief preview here.

First, you need to define a gradient, for example like this:

```
set palette defined ( 0 'blue', 0.5 'white', 1 'red' )
```

This defines a color gradient from blue over white to red; white is centrally located in the mapped interval. You can now use any of the colors from this gradient in a plot like this:

```
plot sin(x) lc palette frac 0.25
```

This plots the curve in the color that is halfway between blue and white. (The keyword `palette` can be abbreviated to `pal`.)

When you select a color from a palette using the `frac` keyword, the numeric argument must lie in the interval [0:1]. You can also map the gradient to a different interval using `set cbrange`. (The identifier `cb` stands for *color box*.) For example, `set cbrange [-5:5]` maps the color gradient into that interval. You can now specify a color by providing a value from this interval together with the `cb` keyword. The following `plot` command produces the same curve as the previous listing, because -2.5 lies at the same relative position within the interval [-5:5] as 0.25 lies within the interval [0:1]:

```
set cbrange [-5:5]
plot sin(x) lc pal cb -2.5
```

Remapping a gradient to a specific numeric range makes sense when you want to indicate a specific numeric *value* with a color. That's the topic of the next section and—in much more detail—of appendix D.

9.1.5 *Using data-dependent colors*

Gnuplot can change the color of a line or point based on the values of the *data*. As usual, there are three ways to do this: explicitly (using the `rgb` keyword), by indexed lookup (using an integer index), and using a gradient (with the `palette` keyword). In any case, you must supply an additional column with the `using` specification of the `plot` command; the color information is always taken from the *last* (rightmost) column in the `using` specification. (Value-dependent color is only available for data—if you want to use it with functions, you have to fake it using the `"+"` pseudofile—see section 4.4.)

EXPLICIT AND INDEXED DATA-DEPENDENT COLORS

Let's consider an example. Here's a very short data file:

```
# x     y      color
1       2.0    1
2       2.1    2
3       1.9    3
4       2.0    4
5       2.0    5
```

Either of the following two commands will work:

```
plot "data" u 1:2:3 w lp lc rgb variable      ⟵  Explicit
plot "data" u 1:2:3 w lp lc variable          ⟵  Indexed
```

The first thing to notice is that although the plot style is `linespoints`, *three* columns are specified: the last (rightmost) is used for color information. Next, notice the keyword `variable` in the color specification: that's what turns on data-dependent coloring. If `variable` follows the `rgb` keyword, the value in the third (color) column is taken as an integer-packed RGB (or αRGB) tuple; if `variable` is used by itself, the entry in the color column is taken as the index of the color to use. (It's not possible to use color names or RGB strings together with `variable`.)

A WORKED EXAMPLE USING EXPLICIT DATA-DEPENDENT COLORS

In practice, you'll almost never read either the color value or the color index from a file. Instead, think of the third column in the using specification as a place to perform some creative inline transformation (see section 3.3) to calculate a data-dependent color. Let's consider a few examples.

Remember DLA clusters, like the one in figure 1.2? For each particle, the data set contains not only the particle's coordinates but also the sequence order in which it was added to the cluster. If you map the latter information onto a color spectrum, you can generate "tree rings" that indicate how the cluster grew (see figure 9.3).

An easy way to turn sequence numbers into colors is to map them onto the hue value in the HSV model. Using the hsv2rgb() built-in function, you can do this on the fly (on a 100,000 particle cluster, in this case):

```
plot "dla" u 2:3:(hsv2rgb($1/100000.,1,1)) w d lc rgb var
```

This graph shows clearly that a DLA cluster tends to grow mostly at the rim. Despite the tenuous structure, it's rare for a particle to penetrate deeply into an existing cluster; it's much more likely to get caught by one of the branches extending outward.

Although it worked reasonably well in this example, in general you should use caution when mapping a quantity to the rainbow spectrum of the HSV color model. One problem is that this particular color spectrum doesn't have any inherent sense of ordering: is yellow larger than blue, or vice versa? Another problem is that the spectrum closes in on itself: red represents both very small and very large values. In this
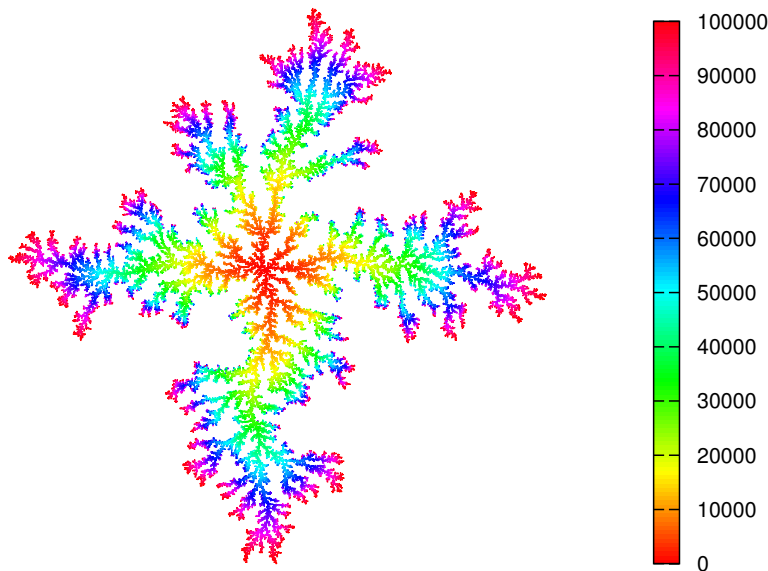
Figure 9.3   A DLA cluster (see section 1.1.2) using the HSV color model to indicate the order in which particles were added to the cluster

example, it doesn't matter, because the geometric structure of the cluster provides the required sense of ordering; but in more general cases, a rainbow plot like this can be highly confusing. Carefully designed *color gradients* are almost always a better idea. Fortunately, gnuplot makes this easy—a topic we'll come back to in appendix D.

WORKED EXAMPLES USING INDEXED DATA-DEPENDENT COLORS

Indexed data-dependent colors make sense when the data contains an attribute that assumes only a discrete set of values (or *levels*). In listing 9.2, some attributes (such as the physical dimensions, the year of presentation, and the available horsepower) are numeric and can have essentially any value. But others, such as the number of cylinders and the country of manufacture, assume only a finite number of discrete values. Those can be represented through distinct, indexed colors. (See listing 9.1 for the commands and figure 9.4 for the resulting plot.)

Listing 9.1 Command for figure 9.4 using data from listing 9.2 (file: supercars.gp)

```
unset key

set xlabel "Year"; set ylabel "Horsepower"

set for[k=1:3] label k sprintf( "% d Cylinders", 4*(k+1) )
➥              at 2003.75,1200+k*50 point pt 7 lc k offset 11,-0.175 right

plot [2003:2014][500:1400] "supercars" u 1:2:($3/4-1) pt 7 ps 1.0 lc var,
➥                         "" u 1:2:8 w labels offset 1,-1
```

Listing 9.2 Data for figure 9.4 (file: supercars)

# Year	PS	Cyl	Len	Height	Weight	Country	Name
2005	1014	16	4462	1159	1888	France	"Bugatti Veyron"
2006	1061	8	4475	1092	1270	USA	"SSC Aero"
2012	1261	8	4655	1079.5	1244	UK	"Hennessey Venom GT"
2011	940	8	4293	1120	1435	Sweden	"Koenigsegg Agera"
2005	806	8	4293	1120	1456	Sweden	"Koenigsegg CCX"
2009	1104	8	4665	1198	1688	Denmark	"Zenvo ST1"
2005	650	8	4460	1114	1100	Germany	"Gumpert Apollo"
2009	750	12	4601	1222	1630	UK	"Aston Martin One-77"
2011	700	12	4780	1136	1575	Italy	"Lamborghini Aventador"

But data-dependent indexed colors can also be used to represent *metadata*. Here's a clever trick:[6] consider a data file containing several data sets, separated by double blank lines. The index of each data set is available in the pseudocolumn -2 (see section 4.4). You can use this information to let gnuplot select a different color for each data set, by using the value of the pseudocolumn as an index into the set of predefined colors:

```
plot "data" u 1:2:(1+column(-2)) w lp lc var
```

---

[6] The idea for this application was taken from the standard gnuplot reference documentation.
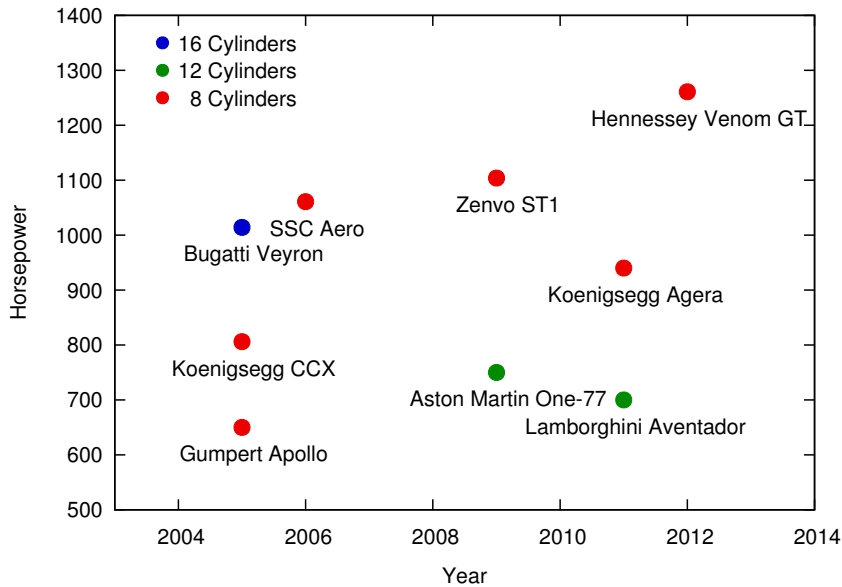
Figure 9.4   Using indexed color to visualize discrete values or levels in a data set. For a set of current "supercars," the date of presentation and the available horsepower are visualized through the *location* on the graph, and the number of cylinders is represented via *color*. Compare listing 9.2 for the data and listing 9.1 for the commands.

Because this example uses indexed lookup, the keyword `rgb` is absent. You need to use the `column()` function, because the dollar-sign shorthand doesn't work for a negative column number. And finally, you need to add 1 to the return value, because data sets in a file are counted starting at 0, whereas line colors are counted starting at 1.

COLOR GRADIENTS

Indexed and explicit colors work well for discrete colors, but if you want a curve that changes color continuously, then color *gradients* are the way to go. Using data-dependent color with a continuous gradient is similar to what you've seen so far, except the keyword is `z` instead of `variable`. Assuming an appropriate color gradient has been defined, the following command will color the curve according to the gradient. Again, you need to specify a third column for the color information:

```
plot "data" u 1:2:3 w l lc palette z
```

Figure 9.5 shows a way that a color gradient can be used with a regular curve. The idea is that this curve monitors an important quantity that must not exceed some threshold. The graph uses a color gradient to change the appearance of the curve depending on the value of the monitored quantity. It's easy to set up the gradient and create the plot (more details in appendix D):

```
set palette defined ( 0 'green', 3 'green', 4 'red', 5 'red' )
plot [][0:5] "monitoring" u 1:2:2 w l palette z lw 2
```
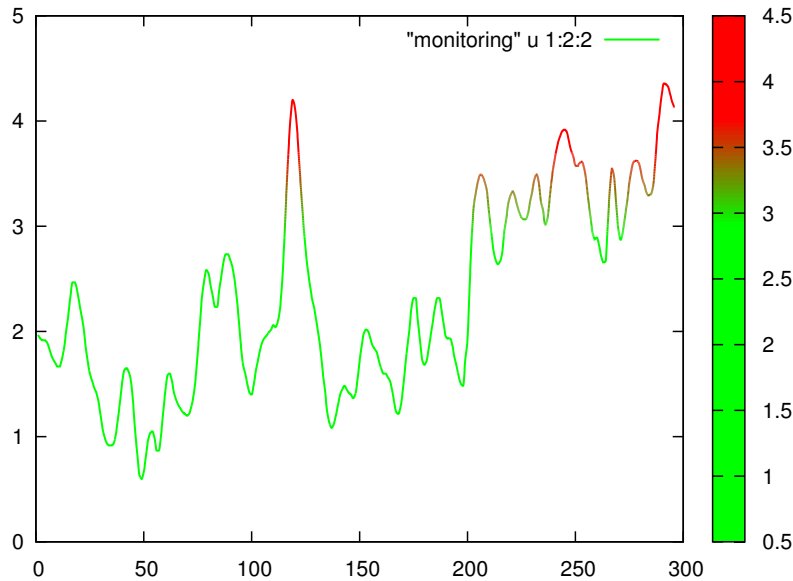
Figure 9.5   Gradient-mapped color. The control curve changes its color if the monitored quantity exceeds its safe value.

9.1.6   *The built-in color sequences*

For many years, the default color sequence for the most common terminals consisted of full-intensity red, green, and blue, followed by magenta, cyan, and yellow, because these are the three basic colors of computer graphics and their binary combinations. It's a simple sequence, but not a very good one: the colors are bright and garish, and yellow is exceptionally difficult to see against a white background. Moreover, this sequence of colors (with its emphasis on red and green) is particularly challenging for people who are color-blind! There are also more subtle problems, having to do with color perception: the blue line, for example, always seems "heavier" than any of the other colors and tends to dominate a graph.

In gnuplot 5, you can choose from three different color sequences, using the set colorsequence command (see figure 9.6):

```
set colorsequence [ default | classic | podo ]
```

The three color schemes are identified using keywords:

- default—A terminal-independent sequence of eight colors
- classic—The original behavior, which differs from terminal to terminal but usually produces the red/green/blue sequence mentioned earlier
- podo—A terminal-independent sequence of eight colors, intended to be easy for color-blind people to distinguish[7]

---

[7] Bang Wong, "Points of View: Color Blindness," *Nature Methods* 8 (2001): 441.
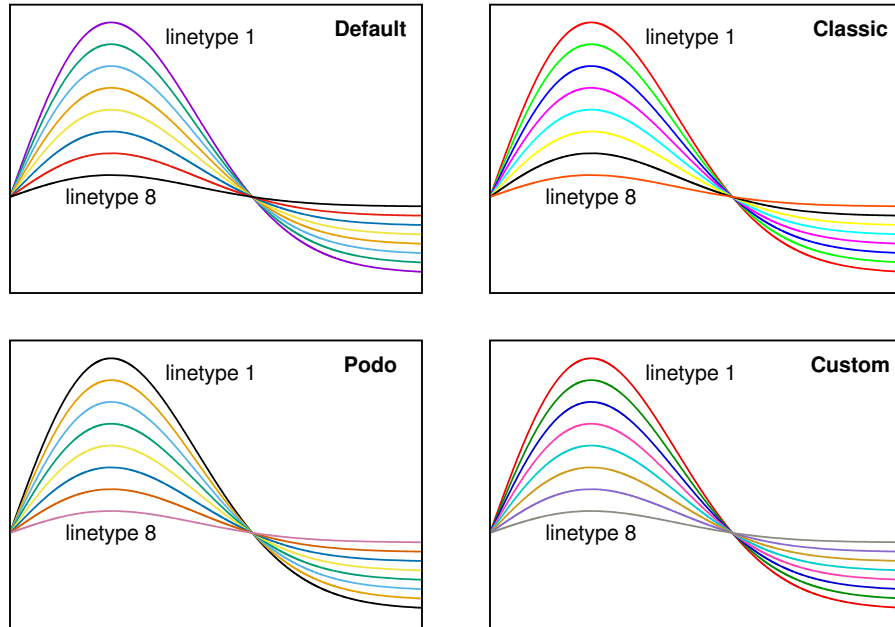
Figure 9.6 The three built-in color sequences that can be selected using `set colorsequence`, and the custom sequence used for the color illustrations in this book. (See section 12.6 for details.)

Unfortunately, none of these three options constitutes a particularly good choice, in my opinion: the new default scheme, for instance, may be much less garish than the classic scheme, but its colors are so muted that I now find it difficult to distinguish different lines! The `set colorsequence` facility is also strangely limited (for example, you can't extend or modify the provided color choices), so you'll probably want to construct your own sequence of colors (and other line properties) from scratch. That's what I do, and that's the topic of section 9.3.

> **TIP** Because the three built-in color sequences aren't very good, you should consider defining and using a custom sequence.

9.1.7 *Tips and tricks*

Color specifications tend to be clumsy: they're both lengthy and obscure. It may therefore be a good idea to introduce shorthands. Here are some ideas to get you started.

Instead of using a hex string, you can assign its value to a semantically named variable:

```
gold = "#FFD700"                      ⟵── Hex string
gold = pack( 0xff, 0xd7, 0 )          ⟵── Equivalent: integer value

plot "data" u 1:2 w l lc rgb gold
```

This isn't ideal, because you still have to type the keyword `rgb` every time. You can absorb it into the variable if you use gnuplot's macro facility (see section 5.2):

```
blue = "rgb '#0000ff'"
plot "data" u 1:2 w l lc @blue
```

You can even include the `lc` keyword in the macro:

```
green = "lc rgb 'green'"
plot "data" u 1:2 w l @green
```

Rather than use color names, you may also want to consider semantic names for colors:

```
alarm  = "red"
normal = "#00cd00"
```

The options are endless—adapt them to your needs.

9.2   *Lines and points*

Gnuplot graphs are constructed from *lines* and *points*: these are the fundamental building blocks. (In keeping with the usage in the gnuplot documentation, the term *point* here and in the following refers to the symbol used to represent a single data point.) All aspects of lines and points can be customized: the width, color, and dash pattern of lines, and the shape, size, and color of points.

This isn't news—I introduced all these concepts at the beginning of chapter 6 and have been using them ever since. (Table 6.1 provides a handy summary.) Most of these properties are straightforward and won't need explanation beyond what you can find in the table. But there are a few exceptions: the selection of available *point types* deserves a few words, and so do user-defined *dash patterns*.

9.2.1   *Point types and shapes*

Each terminal provides a set of shapes or *points* that can be used to represent a single data point. Because these shapes are relatively low-level graph elements, they're hard-coded into the implementation of each terminal driver. In general, a gnuplot user can't change the existing shapes or add new ones.[8] All commonly used terminals provide at least a dozen (or so) different shapes, but some terminals define many more (the old PostScript terminal defined 75 distinct point types!). In practice, fewer than about 10 point types are sufficiently different that they can be distinguished visually.

The *sequence* of shapes has been unified across all (contemporary) terminals, at least for the first few point types (see figure 9.7). Although you can't change the shape of the symbol, you can modify its size, line width, and color using the `pointsize` (or `ps`), `linewidth` (or `lw`), and `linecolor` (or `lc`) properties.

---

[8]   You may be able to use symbols beyond those provided by gnuplot by using a single character from a suitable font as a point symbol—see section 6.3.1. Some terminals, such as `tikz`, permit user-defined point shapes.

·	0 : no symbol	△	8 : Up Triangle, empty
+	1 : Cross	▲	9 : Up Triangle, filled
✕	2 : Saltire	▽	10 : Down Triangle, empty
✳	3 : Cross and Saltire	▼	11 : Down Triangle, filled
☐	4 : Square, empty	◇	12 : Diamond, empty
■	5 : Square, filled	◆	13 : Diamond, filled
○	6 : Circle, empty	⬠	14 : Pentagon, empty
●	7 : Circle, filled	⬟	15 : Pentagon, filled

Figure 9.7   Available point symbols

Point shapes are usually indicated using their index: `plot ... w p pt 7` represents each data point with a filled circle. You may want to define symbolic names for point types:

```
frame = 4                           ⟵ Square, empty
box   = 5                           ⟵ Square, filled
circ  = 6                           ⟵ Circle, empty
disk  = 7                           ⟵ Circle, filled

plot "data" u 1:2 w p pt frame                      ⟵ Uses empty squares
```

9.2.2   *Dash pattern*

The ability to customize dash patterns is a new feature in gnuplot 5; support in older terminals may be spotty. A dash pattern can be specified inline or defined separately using `set dashtype`. Once a dash pattern has been defined this way, you can reference it by its (integer) index. There are two ways to specify a dash pattern: using a symbolic string or using a tuple that gives the relative lengths of solid and empty line segments.

SYMBOLIC STRINGS

A dash pattern can be defined using a symbolic string consisting of the symbols shown in table 9.2. The dash pattern can consist of up to four non-whitespace characters; leading whitespace and any further characters are ignored. Figure 9.8 shows some examples.

Symbolic strings can conveniently be used inline in a `plot` command. The following two commands are equivalent:

```
plot sin(x) dt '-'            ⟵ Inline specification

set dashtype 1 '-'           ⟵ Separate definition …
plot sin(x) dt 1             ⟵ … and indexed lookup
```

Table 9.2 Symbols for dash patterns

Symbol	Numeric	Description
. (dot)	`(2,5)`	A dot followed by a short space
– (dash)	`(10,10)`	A dash followed by a space
_ (underscore)	`(20,10)`	A long dash followed by a space
␣ (whitespace)	`(...,10)`	A space

EXPLICIT TUPLES

The most general way to specify a dash pattern is to supply a tuple of numeric values. These values represent the length of subsequent solid and empty line segments. There can be at most eight elements in the tuple (so that a dash pattern can have at most four solid parts); the first entry in the tuple is interpreted as solid. Here are two examples (compare figure 9.8):

```
set dashtype 1 (2,2)              ⟵  Dotted line
set dashtype 2 (20,10,2,4,2,10)   ⟵  Long dash followed by a double dot
```

Lengths of line segments are measured in units of the line width. A tuple entry of 2 (not 1!) results in a line segment that is as long as it is wide.

Or so it may appear. If you're using very short line segments, you're likely to run into an obscure subtlety: line caps. The cairo library defines three types of line ends: butt,

(20,10,2,4,2,10)

square   round   butt
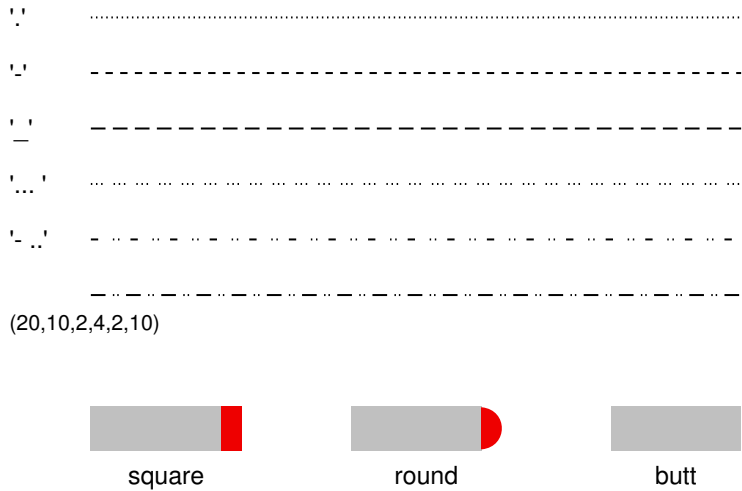
Figure 9.8 Top: Dashed lines resulting from various combinations of the standard dash symbols or given by an explicit numeric tuple (as shown on the left—the pattern for the bottom line is given by an explicit numeric tuple). Bottom: Effect of the different line-cap styles. The line end and the added cap are shown in different colors.

round, and `square` (see figure 9.8, bottom),[9] which can be selected when choosing a cairo-based gnuplot terminal (see section 10.2.3). Unless the chosen style is `butt`, the line extends by half a line width past its actual end point. But the default style for the cairo-based terminals in gnuplot is `square`, meaning all solid line segments in a dash pattern appear longer by half the line width at *both* ends—and the empty segments appear that much shorter. You now have two options: select the `butt` option when setting the terminal (for example, `set terminal wxt butt`) or take the extent of the line caps into account when setting up the dash pattern. (You may find it amusing to work out why the dash pattern `(2,2)` using `butt` line caps is equivalent to the dash pattern `(0,4)` when using `square`.)

9.3   *Customizing color, dash, and point sequences*

Let's first establish when a predefined *sequence* of colors (and possibly of other line properties, such as dash patterns) is relevant: it comes into play only if you're plotting many functions (or data sets) *and* you want to let gnuplot automatically choose a different line color (or pattern) for each new curve. If you don't use gnuplot's automatic style selection, because you specify the plotting style explicitly for each curve instead, the existence of a predefined color sequence is immaterial. The topic of this section therefore applies only to a very limited purpose.

That being said, letting gnuplot choose line styles automatically is such a huge convenience (in particular when you're doing casual and exploratory work) that it can't be ignored. (It matters less when you're preparing presentation graphics, because in that case you naturally want to have full control over all aspects of the plot.)

Gnuplot's automatic style selector chooses from a sequence of *line types*. Line types are gnuplot's original method for specifying appearance options, and nowadays they sit rather awkwardly next to the more modern paradigm (based on the `set style` facility; see section 9.4). Line types are still essential for gnuplot's automatic assignment of visual styles to different curves but otherwise have lost much of their previous importance; in particular, line types are no longer used as containers for general appearance options.

A line type is a combination of all graphic line attributes together with a point type and the other point attributes. By construction, every terminal provides an (in principle) *infinite* sequence of different line types. This guarantees that gnuplot never runs out of different styles, no matter how many curves you plot. In practice, not all of these line types *look* different, of course—gnuplot uses only about a dozen different colors and dash patterns before it repeats itself. But every new line gets its own line type, with a unique numerical index.

---

[9]   See http://cairographics.org/samples/set_line_cap.

9.3.1 *Customizing line types*

You can customize the default sequence of colors and other properties using the `set linetype` command:

```
set linetype {int:idx} [ lineoptions ]
```

Here, the first argument is the index of the line type that you wish to modify; it's followed by specifications of graphical attributes (line color and width, point type and size, and so on—see the tip at the end of section 6.2.3 for an explanation of `lineoptions`). The terminal-specific default values prevail for attributes that aren't changed explicitly.

> **TIP** Line types are persistent: they prevail until changed explicitly. In particular, line types aren't affected by calls to `reset`. (They're reinitialized by `reset session`.)

In practice, you'll only want to redefine a few (the first few) line types and have gnuplot cycle through them repeatedly when plotting many curves. But because the number of available line types is always infinite, gnuplot has no way of knowing when to begin cycling: you have to tell it explicitly. You do so with the command `set linetype cycle`:

```
set linetype cycle {int:period}
```

Let's consider an example. You want curves to be drawn with alternating thin red and wide black lines (only two styles, in other words); the red line should use an empty square as point symbol and the black line a filled circle (`pt 4` and `pt 7`, respectively— see figure 9.7). The following commands achieve this behavior. Because there are odd precedence rules between explicit customizations and `set linetype cycle`, it's necessary to wipe out previous settings first:

```
unset for [i=1:32] linetype i              ⟵── Erases any existing customizations
set linetype 1 lw 1 lc rgb "red" pt 4             ⟵── Defines the first line style
set linetype 2 lw 2 lc rgb "black" pt 7           ⟵── Defines the second line style
set linetype cycle 2                  ⟵── Tells gnuplot to alternate between two styles
```

Unfortunately, the `set linetype cycle` facility applies only to *line* properties, but not to point types! It also obeys non-obvious precedence rules, which make its behavior surprising and hard to predict.[10] My recommendation, therefore, is to avoid it and instead set all line types explicitly:

```
set for [i=1:64:2] linetype i lw 1 lc rgb "red" pt 4
set for [i=2:64:2] linetype i lw 2 lc rgb "black" pt 7
```

This will give you the desired behavior for the first 64 curves you include in a graph (and, of course, you can always increase the upper limit in the loop as far as necessary).

---

[10] At the time of this writing, this entire area appeared to be somewhat in flux. Check the gnuplot standard reference documentation for up-to-date details.

The same method can be used, even if you want more than two distinct line types—just increase the step size in the loop accordingly!

9.3.2   *Special line types*

Every terminal defines two special line types that can't be modified by the user; see table 9.3. Line type `0` is frequently useful to add reference or grid lines to a plot. Line type `-1` can be used to add arrows or other decorations, with the guarantee that they appear solid black.

Table 9.3   Special line types defined by all terminals

Line type	Description
`linetype 0`	A minimally visible line; often a thin dotted line in black or gray
`linetype -1`	A solid black line

9.4   *Global styles*

You can control all graphical aspects of plot elements *inline* by spelling out the desired appearance options as part of the `plot` (or `set arrow` or `set object`) command. But you can also define *global* styles. There are three primary reasons to do so:

- *Convenience*—Instead of listing all the desired options explicitly (possibly more than once, if you plot more than one data set or have more than one arrow or other decoration), you can refer to a style with just its identifier.
- *Semantics*—You can set up and refer to styles based on their function (such as "important" in bright red or "reference" as a dotted hairline).
- *Display targeting*—Imagine that you need graphs that look good in different contexts: clear and crisp on the screen, bold and bright in a presentation, and black-and-white in a print publication! Using collections of styles (*stylesheets*) for each intended use, this becomes much easier than having to redo all the graphs by hand.

You create styles using the `set style` command and can define them for lines and points, arrows, objects, and so on. In addition to defining global styles, `set style` is also used to fix the options on some special-purpose plots—we won't discuss these applications here but in their proper context (appendix E).

9.4.1   *Data and function styles*

By default, functions are plotted with lines (`with lines`), and data from files is plotted using both lines and points (`with linespoints`). These defaults can be changed using these commands:

```
set style data ...
set style function ...
```

For example, to plot data using `with lines`, you say `set style data lines`.

9.4.2 *Line styles*

You define line styles using `set style line`. This command takes an integer index of the style you want to create or change, followed by a combination of line options that define the desired appearance of the style (see the tip at the end of section 6.2.3 for the `lineoptions` shorthand):

```
set style line {int:index} [ lineoptions ]
set style line {int:index} default
```

Unless explicitly overridden, all aspects of a line style default to the corresponding value of the *line type* of the same index. The keyword `default` restores a line style to the equivalent line type.

This begs the question: what's the difference between line styles and line types? In brief:

- *Line types* are persistent: all of their properties prevail until explicitly changed. Every terminal provides an infinite number of line types and their properties. When assigning lines to curves automatically, gnuplot chooses from the available line types only.
- *Line styles* aren't persistent: they're deleted by calls to `reset`. The properties of line styles default to those of the line type with the same index, unless explicitly redefined. Line styles aren't available for automatic assignment to data sets and functions.

I use the following little mnemonic to remember which is which:

Line-T-ypes are sys-T-em settings.
Line-S-tyles are u-S-er preferences.

My recommendation is to prefer line styles over line types whenever possible and to restrict the use of line types to the automatic selection feature (where line types are required).

Here are some examples that give you an idea how line styles can be used:

```
solid     = 100
dotted    = 101
important = 102
reference = 103
strut     = 104

set style line solid lw 1 lc rgb "green" pt 5
set style line dotted lw 1 lc rgb "blue" dt '.'

set style line important lw 2 lc rgb "red" pt 0
set style line reference lw 0.5 lc rgb "grey" dt (2,4)

set style line strut lw 2 lc black
```

9.4.3   *Arrow styles*

You can predefine arrow styles using `set style arrow`. The available appearance options are summarized here—they should all be familiar from section 7.3.2:

```
set style arrow {idx:index} default

set style arrow [{idx:tag}] [ nohead | head | heads | backhead ]
               [ size {flt:length} [,{flt:angle}]
                 [,{flt:backangle}]
                 [fixed] ]
               [ nofilled | empty | filled | noborder ]
               [ front | back ]
               [ lineoptions ]
```

Given all the different appearance options for arrows (in particular with regard to arrow heads), it may make sense to define common arrow types, for example like so:

```
nohead  = 100
filled  = 101
outline = 102
bold    = 103

set style arrow nohead nohead
set style arrow filled size screen 0.025, 30, 45 filled
set style arrow outline size screen 0.025, 30, 45 empty
set style arrow bold lw 2 lc rgb "red"
```

Of course, using symbolic names for the arrow styles is entirely optional. You can now create an arrow by referencing one of these styles:

```
set arrow 1 from 0,0 to 1,1 as filled          ⟵── Uses the symbolic name
set arrow 1 from 0,0 to -1,0 as 103            ⟵── Uses the explicit style index
```

Remember that it's a good idea to use the `screen` coordinate system to define the arrow head: this way, the head size won't change if the plot range changes!

9.4.4   *Fill styles*

Several plot styles and decorations can be *filled* with a color or pattern. This section introduces the syntax of fill specifications. The three pertinent keywords are `fill-color` (or `fc`), `fillstyle` (or `fs`), and `border` (or `bo`).

Two of these don't present surprises: `fillcolor` must be followed by a color specification (see section 9.1), and `border` must be followed by any combination of line options (see the tip at the end of section 6.2.3). The keyword `noborder` suppresses the border. That leaves `fillstyle`, which does warrant some discussion.[11]

---

[11] At the time of this writing, the `border` and `noborder` keywords are permissible only if they follow immediately after the `fillstyle` or `fs` directive on the command line.

There are three forms of fill specifiers:

```
fillstyle empty
fillstyle [transparent] solid [{flt:density}]
fillstyle [transparent] pattern [{idx:style}]
```

The first one leaves the area empty. The second one fills the area with color; an optional numerical argument between 0 and 1 is interpreted as density, with `1.0` standing for full intensity and `0.0` for an invisible fill color. If the optional `transparent` keyword is present, then the density value is interpreted as the value of the α channel for the fill color.

It's also possible to fill closed areas with cross-hatched patterns: that's what the `pattern` keyword does. All contemporary terminals define a set of eight different patterns (use the `test` command to see them—this command will be formally introduced in section 10.1.3); you select one of them through an integer argument after `pattern`. If the `transparent` keyword is present, then the background remains visible through the grid pattern.

> **TIP** Areas may appear unfilled even if you set an explicit fill style using `fill-style`, unless a fill color has also been selected using `fillcolor`.

Using `set style fill`, you can set a global default for many filled styles, such as `with boxes`, `with candlesticks`, `with filledcurves` (all in chapter 6), and `with histograms` (appendix E). It also becomes the default fill style for circles, ellipses, and polygons— but *not* for rectangles: for those, you have to use their own `set style rectangle` command. See the standard reference documentation for details.

9.4.5 *Other global styles*

In addition to the applications just discussed, you can use the `set style` command to express other global appearance preferences. In particular, you can create predefined styles for objects (or shapes) in a way similar to that discussed for arrows. Check the gnuplot standard reference documentation for `set style circle`, `set style ellipse`, and `set style rectangle`.

The other available applications of `set style` are more specific. The command `set style textbox` applies only to boxed labels (see section 7.3.3). Again, the standard reference has the details.

Finally, `set style boxplot` and `set style histogram` apply only to special-purpose plotting styles that I haven't yet introduced. I'll do so in appendix E.

9.5 *Overall appearance: aspect ratio and borders*

In addition to the graph elements that go *onto* a plot, there are some features that determine its general appearance: its size and aspect ratio, and the presence or absence of borders and margins.

9.5.1   *Size and aspect ratio*

We have to distinguish between the size of the *plot* and the size of the *canvas*. The latter determines the size of the output file (or screen), whereas the former affects the size of the graph on the canvas. The canvas size is specified using the following command

```
set terminal {enum:terminaltype} size {flt:x},{flt:y}
```

which we'll discuss in chapter 10 on files and output devices. Here, we're interested in the size of the plot on the screen or canvas, which is controlled by the `set size` command:

```
set size [ [no]square | ratio {flt:r} | noratio ]
         [ {flt:x} [,{flt:y}] ]
```

The numeric values x, y scale the plot with respect to the screen. For values less than `1.0`, the graph doesn't fill the entire screen; for values greater than `1.0`, only part of the graph is shown. If y is omitted, the plot is scaled in both directions by x. For example, `set size 0.5` reduces the extent of the plot by half along both axes, while keeping the overall canvas size fixed.

If the size of the graph is smaller than the size of the canvas, it's not clear where on the canvas the graph should be located. You can fix the graph's position on the screen using the `set origin` option:

```
set origin {flt:x},{flt:y}
```

The arguments give the origin of the graph in `screen` coordinates (see figure 7.2 in chapter 7 for an overview of gnuplot's coordinate systems).

The `ratio` option and its relatives (`square` and `noratio`) provide a shorthand to control the aspect ratio of the plot: that is, the ratio of the y-axis length to the x-axis length. Setting `set size ratio 2` therefore results in a plot that's twice as tall as wide, and `set size square` is a synonym for `set size ratio 1`.

Negative values for the ratio have a special meaning: `set size ratio -1` scales the graph so that the length of the x-axis *unit* is equal to the length of the y-axis unit. For `ratio -2`, the y-axis unit has twice the length of the x-axis unit; for `ratio -0.5`, the x-axis unit is twice the y-axis unit's length. Figure 9.9 demonstrates all these settings.

With `nosquare` and `noratio`, you reset the aspect ratio of the graph to the default value of the terminal. Note that neither command resets the scale given by x, y.

It's possible to give contradictory hints to gnuplot when prescribing both scale values in addition to the aspect ratio. In such cases, gnuplot attempts to maintain the desired aspect ratio without exceeding the minimum area given by the scale values. You can always avoid these situations by fixing only two of the values. My recommendation is to control *the size of the plot on the canvas* through either the x or y scale value and to control *the aspect ratio* using the `ratio` option. The effect of both of these options is intuitively clear, and no ambiguity is possible.
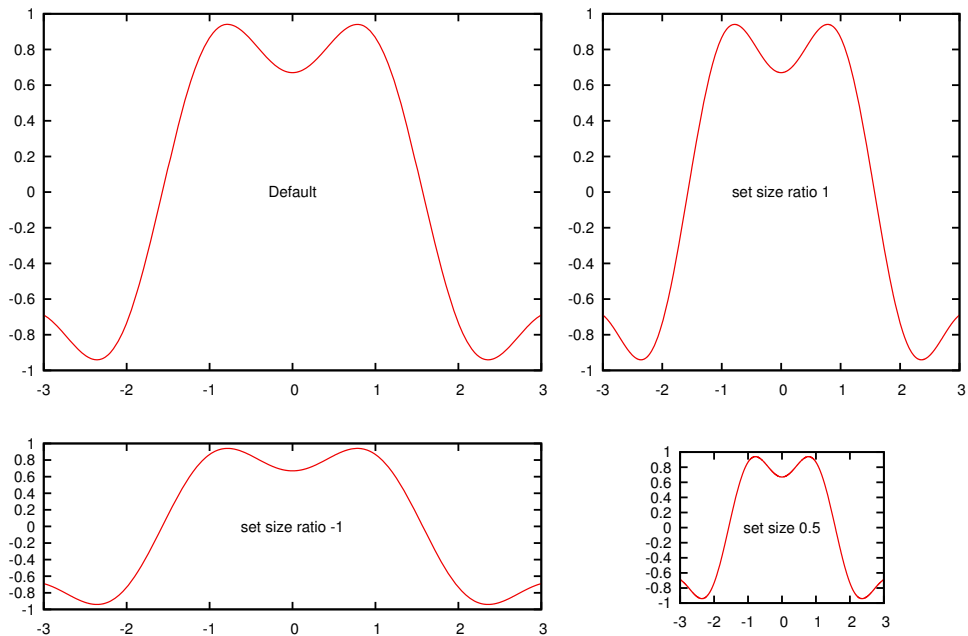
Figure 9.9 Controlling overall image size and aspect ratio with `set size`. Clockwise from top left: default settings, aspect ratio 10/7; `set size ratio 1`, full plot size, aspect ratio 1:1; `set size 0.5`, reduced plot on a full-size canvas, default aspect ratio; `set size ratio –1`, full plot size, x unit with the same apparent length as the y unit.

9.5.2 Borders

By default, gnuplot draws borders on all four sides of the plot. This is suitable if the graph will be used in a larger text document (because it gives the plot a "frame" separating it from surrounding text) but can feel cluttered for a standalone diagram; but it's simply unsuitable for special diagrams that don't fit neatly in a rectangular box. For these reasons, gnuplot gives you the ability to control all parts of the border using the `set border` option:

```
set border [ {int:mask} ] [ front | back | behind ] [ lineoptions ]
```

The only argument that isn't familiar by now is `mask` (see figure 7.3 for layers, and the tip at the end of section 6.2.3 for the meaning of `lineoptions`). This argument is a bit mask in the form of an integer, each bit referring to one side of the border (see figure 9.10). The bit positions are counted *clockwise from the bottom*, so that a mask value of `1` refers to the bottom, `2` to the left, `4` to the top, and `8` to the right: `15` (=1+2+4+8) paints borders on all four sides. (Additional bits are used for three-dimensional plots using `splot`—we'll discuss them in appendix C.) To draw borders on some but not all sides, you switch on only those you want to see:

```
set border 1+2          ⟵——  Switches on the bottom and left borders
set border 3            ⟵——  Same thing
```

Figure 9.10   The set border command takes as its first argument an integer that encodes the choice of borders to switch on in the form of a bit mask. The graph indicates which borders correspond to which bit (in decimal representation) and also gives the overall mask value for two common cases: borders on all sides and borders on the bottom and left.

If tic marks are drawn on the border (as is the default), unsetting the border does *not* unset the tic marks. They must be controlled separately using the options discussed in chapter 8.

CREATING AN EMPTY SPACE BETWEEN AXES AND DATA

With the `set offsets` command, you can introduce an empty space between the data and the borders. This can be desirable to avoid interference between the data and tic marks and other axes decorations. The `set offsets` command controls the size of the empty space on all four sides:

```
set offsets {flt:left}, {flt:right}, {flt:top}, {flt:bottom}
```

By default, the arguments are interpreted as given in the `first` coordinate system; using the keyword `graph`, you can also specify the arguments as fractions of the size of the overall graph. Gnuplot extends the plot ranges to make room for the empty space as required by `set offsets`—keep this in mind when using this command.

CLIPPING DATA POINTS CLOSE TO THE BORDERS

The `set clip` option controls how gnuplot plots points that are too close to or outside the plot area:

```
set clip points
set clip [ one | two ]
```

The first version, `set clip points`, is relevant only when you're using a style that shows discrete plotting symbols (`with points`, `with linespoints`, and so on). If it's active, symbols that would touch or overlap the borders of the plot are suppressed. Exactly how many points are clipped depends on the symbol size: for larger symbols, more points need to be clipped. (See chapter 6 for more detail on styles and ways to influence the symbol size.)

The second version controls how gnuplot plots line segments connecting points if at least one of the points falls outside the currently visible plot range. If `set clip one` is active, line segments are drawn if at least one of the end points falls into the visible plot range. If `set clip two` is active, line segments are drawn even if both end points are outside the current plot range but a straight line connecting them crosses the visible range. In no case are parts of the line segment drawn outside the visible range. By default, `set clip one` is on, but `set clip two` is off.

9.5.3 Margins

The empty areas outside a plot's borders make up the *margins* (see figure 7.2). By default, gnuplot calculates the size of the margins automatically, based on the presence or absence of tics, axis labels, a plot title, and other decorations placed outside the borders. If necessary, you can fix margins manually using the `set margin` command.

Traditionally, there have been four different commands, for each of the margins separately. Gnuplot 5 also contains a version that combines all four into one:

```
set bmargin [ {flt:chars} | screen {flt:frac} ]      ⟵  Bottom margin
set lmargin [ {flt:chars} | screen {flt:frac} ]      ⟵  Left margin
set tmargin [ {flt:chars} | screen {flt:frac} ]      ⟵  Top margin
set rmargin [ {flt:chars} | screen {flt:frac} ]      ⟵  Right margin

set margin [ {left}, {right}, {bottom}, {top} ]

show margin                                   ⟵  Shows the current settings
```

The argument is interpreted differently depending on whether the keyword `screen` is present:[12]

- Without `screen`, the argument is interpreted as the *width* of the margin around the graph, measured in character widths or heights. Supplying a negative value restores automatic calculation of an appropriate margin width.
- With `screen` present, the argument is interpreted as the *absolute position* of the graph's border, measured in `screen` coordinates. Using `screen` ignores and overrides the current values of `set size` and `set origin`.

It's important to clearly distinguish between these two modes of operation! (See figure 7.2 for the meaning of `screen`; see section 9.5.1 for `set size` and `set origin`.)

The `set margin` command combines all four separate commands into a single invocation. Each argument can be either a desired width (in character units) or an absolute position (if preceded by the `screen` keyword). You can combine both modes in a single invocation: a command like `set margin 10, screen 0.9, -1, -1` is valid.

9.5.4 Internal variables

You can inspect gnuplot's internal size and margin information by examining the values of several internal variables. (See section 5.5 for general information on internal variables.) The variables that are of interest in the current context all have the prefix `GPVAL_TERM_`. You can list them and their values as follows:

```
show variables GPVAL_TERM_
```

There are seven such variables; table 9.4 explains what they represent.

---

[12] The gnuplot standard reference documentation states the syntax as `at screen`, but the `at` keyword is optional and will be ignored henceforth.

Frequently, the numerical value of these variables is less useful; what really matters are various ratios. For example, the position of the left border in screen coordinates (see figure 7.2 for the different coordinate systems) is

```
GPVAL_TERM_XMIN/(1.0*GPVAL_TERM_XSIZE/GPVAL_TERM_SCALE)
```

In this expression, the multiplication with 1.0 is necessary to force floating-point evaluation of the fraction. In general, the units of the variables representing the canvas size are *different* from the units representing the location of the graph borders. The variable GPVAL_TERM_SCALE holds the conversion factor: you must always divide the variables holding the size by this variable, as in the expression given earlier.

Table 9.4   Internal variables that contain information about the size of the canvas and the position of the borders

Variable name	Description
GPVAL_TERM_XSIZE, GPVAL_TERM_YSIZE	Size of the entire canvas (or screen) in the horizontal and vertical directions.
GPVAL_TERM_XMIN, GPVAL_TERM_YMIN	Positions of the bottom and left borders with respect to the canvas. These are the lower limits of the graph in both directions.
GPVAL_TERM_XMAX, GPVAL_TERM_YMAX	Positions of the top and right borders with respect to the canvas. These are the upper limits of the graph in both directions.
GPVAL_TERM_SCALE	Conversion factor between the units of the canvas size and the location of the borders. Variables representing the size must be divided by this variable to be comparable to the other variables in this table.

9.6   *Summary*

In this chapter, we focused less on individual graph elements and their properties and instead discussed several topics that influence the overall appearance of a graph. Color, and the way it's made available in gnuplot, occupied the first part of this chapter. The discussion was pretty technical; appendix D will return to the color topic, but then we'll emphasize how to employ color for visualization tasks.

Next, we described how the appearance of lines and points can be controlled in minute aspects. Gnuplot 5 gives you the ability to define custom dash patterns for lines, and we explained how to do that.

The last part of this chapter explained how to control the overall appearance of a graph, such as the presence or absence of graphical borders around a plot, and how to modify the size and aspect ratio of the entire graph. Gnuplot's defaults for all these properties are probably sufficient for the vast majority of your work, but there are situations where being able to go beyond the basics can give your graphs a palpable lift.

Terminals and output formats

Gnuplot can generate graphs in all common graphics formats (and some uncommon ones, too). The components within gnuplot that are ultimately responsible for creating all graphical output are called *terminals*. Different output formats require different terminals.

Because terminals are the final step in the creation of a graph, the ultimate appearance of a graph depends directly on the capabilities of the terminal used to generate it. Some of gnuplot's terminals produce output of very high quality and are among gnuplot's primary assets. Unfortunately, terminal handling in gnuplot is a bit different than comparable functionality in other programs. We'll therefore

begin our discussion of terminals with a brief historical digression that explains the motivation behind gnuplot's *terminal* abstraction.

An important capability that is provided by each terminal is the handling of *fonts* or typefaces: the selection of fonts and the available effects (italics, boldface, sub- and superscript) are determined by the choice of terminal. So, we'll discuss fonts and related topics in this chapter as well.

Next, we'll examine the most important terminals, beginning with terminals that generate output in standard graphics file formats (specifically, PDF and PNG). Then we'll talk about ways to include mathematical notation and formulas on graphs, with and without the use of LaTeX. We'll introduce gnuplot's `svg` and `canvas` terminals, which generate graphics suitable for viewing in contemporary web browsers. And finally, we'll quickly survey gnuplot's interactive terminals.

10.1   *The terminal abstraction*

If you look in the standard gnuplot reference documentation, you'll find that gnuplot supports some 70 different terminals! This may seem daunting, but the reality is that most of them are obsolete today—the number of terminals that are relevant is (thankfully) much smaller. But this leaves unanswered the question of where these 70 different terminals originally came from. For this, we need a bit of history.

10.1.1   *Historical digression*

Gnuplot was originally conceived and implemented at a time (the late 1980s!) when graphics support was much less standardized than it is today. It wasn't safe to assume that a computer even *had* an interactive graphics terminal: producing graphics usually required special hardware (such as an attached pen-and-ink plotter or a dedicated graphics display).

The generic name for such a device that produced visible output was *terminal*. To keep gnuplot portable, all code specific to a single terminal was encapsulated into the terminal facility—today, we'd call it the terminal *driver*. As graphic terminals developed and proliferated, gnuplot acquired a selection of such drivers. Nowadays, many of these legacy devices are obsolete, but the drivers are still part of the gnuplot distribution. Most of them can safely be ignored.

Nevertheless, the *abstraction* has proven itself capable of adapting to changing circumstances. Today, the established architecture supports the two forms of graphical output that are still relevant: it drives interactive, screen-based displays (*interactive terminals* such as the `wxt` and `qt` terminals), and it creates output *files*, using any of a variety of standard graphics formats and command languages.

10.1.2   *The terminal workflow*

The primary reason terminal handling is often seen as obscure and confusing is that it requires *multiple steps*. Moreover, if the sequence of steps is interrupted, the gnuplot session may end up in a different *state* than at the beginning. This is confusing,

because the near-unequivocal behavior of most contemporary programs is that an action "export to file" is both atomic and stateless (or, at least, it appears to be). In other words, you usually issue a single command to create a file, and then you're right back to where you started. Not so in gnuplot.

THE STANDARD WORKFLOW

The complete workflow to create a graphics file from an interactive gnuplot session and to restore the interactive session again consists of the following six steps (also compare listing 2.2):

1 Save the current state of the `terminal` option with `set terminal push`.
2 Select a new terminal using `set terminal`.
3 Select the desired output file or channel name with `set output`.
4 Generate a plot using `plot`, `splot`, `replot`, or `refresh`.
5 Restore the original terminal setting using `set terminal pop`.
6 Restore the original file or channel name with `set output`.

One reason for this multistep process is that the same terminal abstraction is used for both the interactive session display as well as the printable export, requiring you to switch between different modes in a way that seems cumbersome today.

Two issues in particular tend to trip up the unwary. The first is the need to choose the file format (via the terminal) and the desired filename through two *separate* commands (`set terminal` and `set output`). The second is the requirement to *restore* the interactive session explicitly after the output file has been generated. (If this doesn't happen, subsequent `plot` commands continue to be sent to the file, clobbering its contents.)

Some day, gnuplot will acquire an `export` command that will perform all the required steps automatically. Until then, you have to either make sure to complete all the necessary steps manually or use a macro of some form. (You can find implementations of macros for this purpose in sections 5.2.2, 11.2.1, 11.2.2, and 12.1.2.)

> **TIP** To export a plot to a file from an interactive session requires multiple steps. Make sure you complete the process manually or use a macro.

FLUSH YOUR BUFFERS!

As it turns out, there is a particularly nasty third problem, which is that output buffers aren't necessarily *flushed* after a `plot` (or equivalent) command. This means when the `plot` (or equivalent) command returns, the output file may still be incomplete and not yet ready to be used as input to other programs.

There's a simple remedy, though: the next `set terminal` or `set output` command is guaranteed to complete the output file and flush any open buffers. Hence, even if you don't want to resume your interactive session, you still need to issue either a `set output` or `set terminal` command. Also be aware that quitting gnuplot does *not* necessarily flush open files!

TIP   Output files may not be complete even after the `plot` command returns. To guarantee that all buffers are flushed and the file is completed and closed properly, you must issue either a `set output` or `set terminal` command after the `plot` command.

THE PUSH AND POP PSEUDOTERMINALS

The pseudoterminals `terminal push` and `terminal pop` can be very helpful in this context. The command `set terminal push` saves not only the current terminal type, but also all selected options. The command `set terminal pop` restores it. It's therefore possible to restore the previous terminal state even without having explicit information about it.

CHANGING DEFAULT TERMINAL PROPERTIES

A special command exists to change certain terminal options without having to reset the entire terminal. This can be useful when you want to modify some terminal aspect, regardless of the current choice of terminal. Five terminal properties can be changed using this command:

```
set termoption font "{str:name} [,{flt:size}]"
set termoption fontscale {flt:scale}
set termoption [no]enhanced

set termoption [ linewidth | lw ] {flt:factor}
set termoption [ dashlength | dl ] {flt:factor}
```

Be aware that *only one aspect can be changed per invocation* of `set termoption`; the different aspects can't be combined into a single call.

SCALING A COMPLETE PLOT UP OR DOWN

It's occasionally necessary to scale an entire, finished graph up or down—for a presentation, say, or for inclusion in a publication. Increasing the size of the resulting graph using `set terminal ... size` isn't enough, because the line widths and font sizes won't be scaled correspondingly. Instead, you want to adjust the `fontscale` and `linewidth` sub-options to the `set terminal` command as well.

The `set termoption` command may be helpful in this context, because it can adjust these two settings separately from the `set terminal` command. There is no equivalent terminal option for the size of symbols; you must use the `set pointsize` command for this purpose.

10.1.3 *Terminal capabilities and the test command*

Output devices (terminals) obviously can differ widely in their capabilities—you can't get color plots from a black-and-white printer. Gnuplot has a built-in command called `test` that generates a standard test image. The test image shows all available line styles and fill patterns and also attempts to demonstrate more advanced terminal capabilities, such as the ability to rotate text through an arbitrary angle. To use the `test` command, you first need to select the terminal you're interested in and set the name of the output file (if it's not an interactive terminal) like so:

```
set terminal pdfcairo
set output "test.pdf"
test
```

Note that the command is `test`—not `plot test`!

   In figure 10.1, you see the available line and symbol styles along the right side, fill patterns along the bottom, and line widths on the left. You can also see what kinds of arrows the terminal supports and whether it has the ability to rotate text. For the terminal as shown in the figure, all these features are supported.
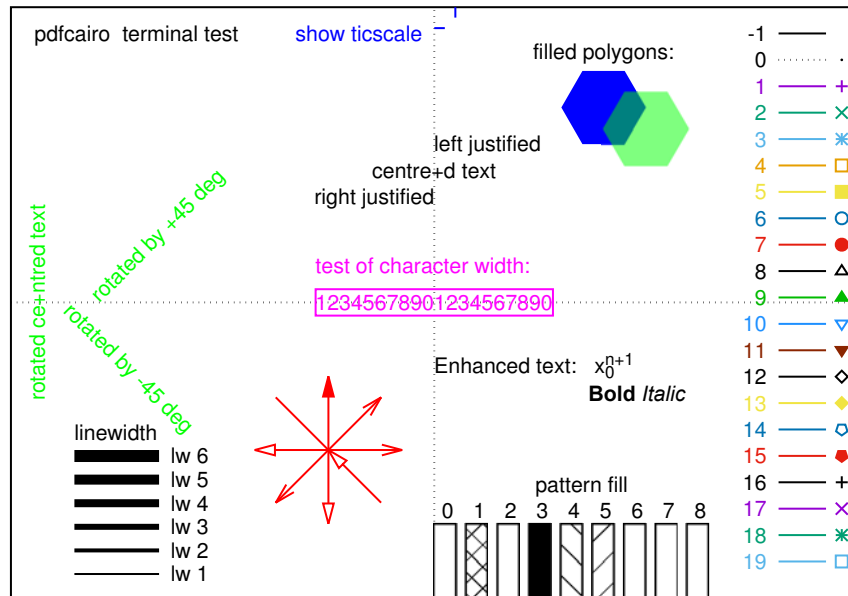
Figure 10.1   The standard test image to demonstrate terminal capabilities, shown here for the `pdfcairo` terminal and the default color sequence `set colorsequence default`

Gnuplot terminals today

In recent versions of gnuplot, terminal handling has become easier and more consistent than ever. The improvements in terminals and terminal handling *alone* justify upgrading to gnuplot 5 at the earliest possible moment!

A huge advance is the development of a family of terminals that are all based on a common set of high-quality libraries for graphics and font handling. The `pngcairo`, `pdfcairo`, `epscairo`, and `wxt` terminals all use the cairo and pango libraries for graphics and font rendering, and the `fontconfig` facility for font selection. These libraries provide advanced capabilities, enabling gnuplot terminals to produce high-quality graphics output. Moreover, because an entire family of terminals shares a common back

end, it's now possible to generate bitmap (PNG), vector (PDF and EPS), and interactive (wxWidgets) versions of the same graph, with only minor differences between them.[a]

Even outside this family of terminals sharing a common back end, terminals in gnuplot 5 exhibit more consistent behavior than ever before. A common set of options is available for basically all contemporary terminals. Terminals are guaranteed to offer at least eight line types that have the same appearance (with regard to color, dash pattern, and point type) across all terminals. A background color can be selected as part of the `set terminal` command.

Font handling, in particular, has become much easier, because all contemporary terminals obtain their font support from the underlying font subsystem in the same way. Enhanced text mode (see section 10.2.3) is supported and is enabled by default. At the same time, some traditional uses of enhanced text mode are no longer as relevant: thanks to Unicode and UTF-8 encoding, special symbols can now be requested without an explicit font change, and the font subsystem will do its best to provide a suitable glyph.

By default, terminals produce colored graphs using solid lines. The option `mono` is still available for all terminals to enforce graphs in black and white, but it does *not* automatically turn on dashed lines, essentially rendering this option useless. (The options `solid` and `dashed` are still accepted by most terminals but are ignored.) You therefore have to select dash patterns explicitly (either globally using `set linetype` or inline with `dashtype`); thus you're overall better off using a stylesheet (see section 12.6) to define a set of line types that have both the desired dash type *and* the required color when you want to create graphs in black and white.

---

[a] In particular, many of the configuration tricks and tweaks required by the bitmap terminals based on the GD library and the PostScript terminal are no longer necessary.

10.2   *Font selection and enhanced text mode*

Fonts aren't handled by gnuplot itself—instead, gnuplot relies on capabilities of the installed system libraries (and, of course, on the selection of installed fonts). In this section, we investigate how to tell gnuplot to use a specific font from among the available ones, and we introduce *enhanced text mode*, which allows additional formatting options for text.

10.2.1 *Font selection*

There are three ways to select a font, in increasing order of granularity:

- Terminals have a *default font* that's used for all text labels (unless overridden). You can select a different default font using the `font` keyword in the `set terminal` command.

- Most commands that generate textual output (such as `set label` and `set key`) let you *select a font explicitly*, again using the `font` keyword. The font is used only for the specific label in question.
- Using enhanced text mode, you can change the font *inside a string*. This makes it possible to render individual characters using a specific font if necessary or desired.

Another reason the default font is important is that the average size of its characters is used to define the `character` coordinate system (see section 7.2) and to set the size of the graph's margins (see section 9.5.3). For variable-width fonts, gnuplot makes a best-effort attempt to determine an average character size.

10.2.2 Font resolution

In general, all terminals rely on some underlying system library for font resolution. Most contemporary terminals, and in particular the ones based on the cairo library, use the `fontconfig` utility for this purpose. One important service this library provides is *font substitution*: given a request for a font, it selects that font from among the available ones that *most closely matches* the request, taking into account not just the name of the requested font but also font characteristics such as style, slant, and weight. This "ensures that a font will always be returned, but doesn't ensure that it is anything like the requested pattern," as the `fontconfig` user manual says.[1] The details of the matching process, and in particular the relative priority of the various attributes, are complicated and not necessarily transparent to the user. The substitution occurs glyph by glyph: I've encountered instances where each character in a string of special characters was rendered using a different font, depending on which font in the list of candidates first provided a glyph for the requested character.

One particular problem concerns the rendering of mathematical symbols and Greek characters. In general, textual labels on a graph should support the graph but not distract from it. That calls for a clear and neutral font, like Helvetica or another linear sans-serif font with little line-width variation. In contrast, mathematical symbols and Greek letters are traditionally drawn with varying stroke weight. If a sans-serif font is selected as the default font for a terminal, then the font subsystem will attempt to find sans-serif glyphs for all requested characters, even for special symbols. To avoid this, I've found it necessary to switch fonts inside a string, particularly when I wanted to use letters from the Greek alphabet.

Figure 10.2 shows the results produced on my system when various free fonts are requested using the test script in listing 10.1. (Depending on what fonts you have installed, your results may differ.) The test string consists of a set of regular characters chosen to exhibit the particulars of the font and a selection of mathematical symbols. The first group of fonts consists of non-serifs, the second group of serif fonts, and the last group of three special math fonts. (OpenSymbol is distributed with Open-Office/LibreOffice, AsanaMath is intended as complement to Palatino, and the STIX project intends to provide a "comprehensive set of fonts" for scientific publishing.)

---

[1] See www.freedesktop.org/software/fontconfig/fontconfig-user.html.

Helvetica	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Nimbus Sans L	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
FreeSans	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
TexGyreHeros	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Arial	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Verdana	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
DejaVu Sans	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Liberation Sans	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Droid Sans	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Times	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Palatino	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Bitstream Charter	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Utopia	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Century Schoolbook L	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
OpenSymbol	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\phi \to \infty$
STIXGeneral	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\varphi \to \infty$
Asana Math	Quarges Gyt R17	$\sum \alpha \lambda \propto \int \Gamma(\theta \cdot \pi) \, d\phi \to \infty$

Figure 10.2 Output produced when various free fonts are requested on *my* computer—your version may look different. The first part of the test string contains a selection of characteristic regular characters, and the second part consists of common mathematical symbols and Greek letters.

Listing 10.1 Commands for figure 10.2 (file: font-demo.gp)

```
unset border; unset tics; unset key

set t pdfcairo size 14cm,17cm
set output "fonttest.pdf"
```

```
s = "Quarges Gyt R17"
t = "∑αλ ∝ ∫ Γ(θ · π) dφ → ∞"

f = "Helvetica 'Nimbus Sans L' FreeSans TexGyreHeros Arial "
f = f . "Verdana 'DejaVu Sans' 'Liberation Sans' 'Droid Sans' ? "
f = f . "Times Palatino 'Bitstream Charter' Utopia 'Century Schoolbook L' ? "
f = f . "OpenSymbol STIXGeneral 'Asana Math'"

a = 0; b=2.75; c=5.75
n = words(f); m = n

do for[k=1:n] {
  m = m - 1
  w = word(f,k)
  if( w ne '?' ) {
    set label w at a,m font "Helvetica,12"
    set label s at b,m font sprintf("%s,16", w)
    set label t at c,m font sprintf("%s,16", w)
  }
}
plot [-0.25:9.25][-1:n] -2
```

10.2.3 *Enhanced text mode*

All contemporary terminals support *enhanced text mode*, which allows additional formatting options. In enhanced text mode, some characters become control characters and indicate, for example, sub- and superscripts. As of gnuplot 5, *enhanced text mode is now the default* for all terminals. Enhanced text mode can be enabled or disabled for individual elements (such as text labels or the key) that contain strings. This may be useful if you want to use the special characters in regular text. Table 10.1 lists the special characters and their effects.

Here are some additional remarks on enhanced text mode:

- Curly braces {...} are used both to group several characters together and to limit the effect of a font declaration to the contents of the braces only. They can be left off for single-character sub- and superscripts.
- Phrases in curly braces nest, as in e^{-x^2} or a^{x_i}.
- Font specifiers must be preceded by a slash / that immediately follows the opening curly brace.
- Font specifiers can be combined into a single expression: for example, {/Times:Italic*2 Hello}.
- Font specifiers apply to everything within the enclosing braces: {/:Italic a_k} creates a label where both characters are italic. To prevent the subscript from being italic, you must either take it outside the braces, as in {/:Italic a}_k, or use an explicit :Normal, like so: {/:Italic a_{/:Normal k}}.
- A character within the single-byte ASCII range can be selected by supplying its ASCII code as a three-digit octal number preceded by a backslash.

Table 10.1   Control characters for enhanced text mode

Control character	Example	Result	Description
`{ }`			Establishes groups and scopes
`^`	`x^2`	x^2	Superscript
`_`	`A_{ij}`	A_{ij}	Subscript
`@`	`x@^s_i`	x_i^s	Aligns sub- and superscripts vertically
	`x@_i^{-3/2}y`	$x_i^{-3/2}y$	You need to put the shorter one *first*…
	`x@^{-3/2}_iy`	$x_i^{-3/2}y$	… rather than last.
`{/ }`	`{/Times Hello}`	Hello	Changes the font
`{/= }`	`{/=24 A}`	A	Selects an absolute font size (size in printer points)
`{/* }`	`{/*0.5 A}`	^A	Changes the font size relative to the enclosing font
`{/:Normal }`	`{/:Normal Hello}`	Hello	Selects regular font characteristics
`{/:Italic }`	`{/:Italic Hello}`	*Hello*	Selects italics
`{/:Bold }`	`{/:Bold Hello}`	**Hello**	Selects boldface
`&`	`[&{abc}]`	[ ]	Space, corresponding to the length of the enclosed argument
`~`	`~B/`	$\not{B}$	Overlays the two following characters or groups
	`~x{.5-}`	$\bar{x}$	Overlays a – character on the previous character, raised by 0.5 times the current character size
`\`	`\{ x \~ y \}`	{ x ~ y }	Escapes a special character by preceding it with a backslash in *single*-quoted strings
`\\`	`\\{ x \\~ y \\}`	{ x ~ y }	The backslash must be doubled to escape special characters in *double*-quoted strings.
`\NNN`	`\101`	A	Selects a symbol using its octal code (single-byte characters only)
`\NNN\NNN`	`\316\261`	α	Selects a multibyte character using a sequence of octal numbers, one for each byte in the binary encoding of the character. (The Unicode character α at code point U+03B1 has the binary representation 0xCE 0xB1 in UTF-8, which translates to \316\261 in octal numbers.)

- Similarly, a multibyte Unicode character can be selected by encoding each byte of its binary representation as a three-digit octal number. Remember that the Unicode codepoint U+xxxx does *not* equal the binary representation in UTF-8. Instead, you have to look up the binary representation, convert each byte separately to a three-digit octal number, and then supply the sequence of numbers thus obtained.

- Special characters can be used in enhanced text mode by preceding them with a backslash \ for single-quoted strings and by preceding them with a double backslash \\ for double-quoted strings.

- Overlays and the escaping of special characters aren't available for the original postscript terminal. Both work normally for the pdfcairo and epscairo terminals.

Enhanced text mode vs. Unicode

Enhanced text mode used to be very important for two applications that today are better solved through Unicode. Overlays (using the ~ operator) were a way to create accented characters, and special characters (such as Greek letters and certain mathematical symbols) could be accessed by switching to a different font, like the PostScript Symbol font.

Today, both of these effects are better achieved through Unicode. Dedicated glyphs for accented characters almost always look better than an ad hoc solution using overlays; and system-wide font subsystems do their best to return a suitable glyph when a special symbol is requested, without you having to declare a specific font.

This leaves enhanced text mode as a solution for basic formatting tasks: sub- and superscripts, and local changes in font size and appearance (italics and boldface). Nevertheless, font changes are still available, and you're free to use them if you have reason to override the selection made by the font subsystem.

10.2.4 Worked example

Let's look at some examples of enhanced text mode (see listing 10.2 and figure 10.3). Most of the commands should be clear; I only want to point out a few details. Enhanced text mode can be used anywhere that gnuplot expects a string: in the example, it's used in the labels, for the x tics, and for the entries in the graph's key. The scope of a font selection can be limited using curly braces. This is done several times: for example, in expressions such as {/OpenSymbol φ}(x), which prints the first character using the OpenSymbol font but the last three using the regular font.

A word of warning: enhanced text mode works best for simple applications, such as sub- and superscripts, boldface and italics, and simple changes to the font size or font face. But complicated, nested expressions may not always work out perfectly (in particular when using the PostScript terminal—see the gnuplot standard reference documentation for details).

Listing 10.2   Commands for figure 10.3 (file: enhanced-mode.gp)

```
set label 1 "{/OpenSymbol φ}(x) = e^{-x^2/2} / √2{/OpenSymbol π}" at 1.2,.25
set label 2
➥    "{/OpenSymbol Φ}(x) = ∫@_{-∞}^x &{i} {/OpenSymbol φ}(t) dt" at 1.2,0.8

set label 3 "{/:Bold*2 Big and Bold}" at -2.5,0.6

set key top left title "{/:Bold Legend}"

set xtics ( "{/OpenSymbol π}/2" pi/2, "0" 0, "-{/OpenSymbol π}/2" -pi/2 )

plot [-3:3] exp(-0.5*x**2)/sqrt(2*pi) t "{/OpenSymbol φ}(x)",
➥            norm(x) t "{/OpenSymbol Φ}(x)"
```
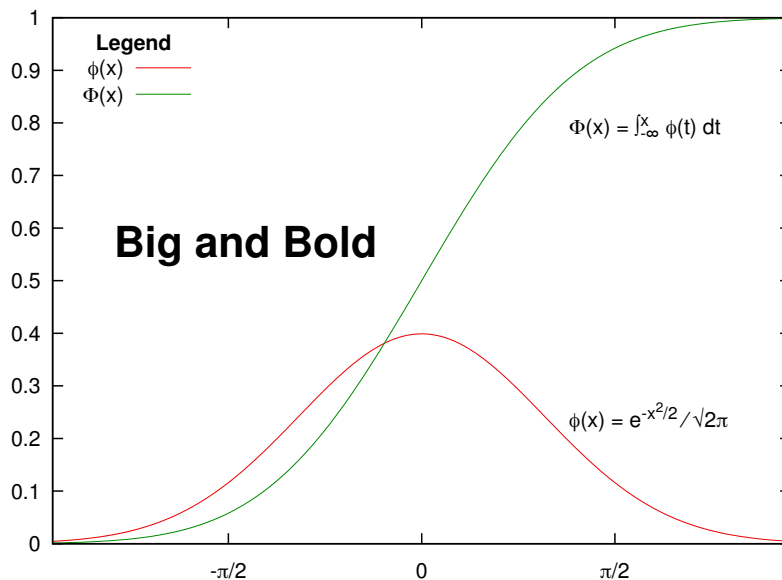

Figure 10.3   Enhanced text mode. See listing 10.2.

10.3   *Generating PNG and PDF with cairo-based terminals*

The pngcairo and pdfcairo terminals are the first choice when it comes to generating graphs in a standard bitmap (PNG) or vector (PDF) file format. (There is also the epscairo format, which is equivalent to the pdfcairo format but produces encapsulated PostScript.) These terminals share a common set of options:

```
set terminal [ pngcairo | pdfcairo | epscairo ]
             [ size {flt:x}[ in|cm ],{flt,y}[ in|cm ] ]
             [ font "{str:face} [, {int:size}]" ] [ fontscale {flt:scale} ]

             [ background rgb {colorspec} ] [ [no]transparent ]
```

```
[ [linewidth|lw] {flt:factor} ][ [dashlength|dl] {flt:factor} ]
[ butt | square | rounded ]
[ [no]enhanced ] [ [no]crop ] [ mono | color ]
```

The only option that requires some care is `size`, because it treats units differently depending on the output format:

- For PNG, the size defaults to 640×480 pixels (aspect ratio 4:3). If no units are specified, pixels are assumed; if either `cm` or `in` is given, the size is converted to pixels, assuming 72 pixels per inch.
- For PDF and EPS, the size defaults to 5×3 inches (aspect ratio 5:3). Other sizes can be specified using either `cm` or `in` as units; if no units are given explicitly, inches are assumed.

Most of the other options are straightforward:

- You can choose a typeface and size for the terminal's default font. The default is `font "sans,12"` (but see `fontscale`). You can give only the size, leaving the font blank: `font ",16"`.
- Using `fontscale`, you can adjust the size of the font separately and relative to the current font size. Be aware that the default value of this option is 0.5!
- You can specify a background color using any one of the *explicit* color specifications (those using the `rgb` keyword—see section 9.1.1). Using `transparent` makes the background color transparent.
- Line widths (for vector formats) default to 0.25 points (with 72 points per inch). Both the line width and the dash length can be changed by a multiplicative factor. Note that it's *not* possible to change the point size in the same manner; you have to use the global `set pointsize` option instead.
- The keywords `butt`, `square`, and `rounded` control how the cairo library renders line ends and joins (see figure 9.8). The default is `butt`.
- Enhanced text mode is the default. You can disable it using `noenhanced`—for instance, because you want to use its control characters as regular characters.
- When you use `crop`, the terminal attempts to trim empty space around the graph before saving it. The plot isn't rescaled, so the resulting graph is in general *smaller* than the size specified in the `size` directive. (The `crop` option is only available for `pngcairo`.)
- By default, terminals produce colored output. You can force monochrome output using `mono`, but be aware that this does *not* automatically turn on dashed lines—you have to do so explicitly for each line in your graph.

10.4 Using gnuplot with LaTeX

One very common use of gnuplot graphs is to include them as illustrations in a LaTeX document. This section presents a couple of cookbook-style recipes. First, I describe how to include a regular graphics file as an image in a LaTeX document.

Then we discuss gnuplot's special `cairolatex` terminal, which allows you to combine gnuplot graphics with LaTeX text in the same illustration so that you can use the full power of LaTeX for mathematical typesetting in gnuplot graphs.

10.4.1 *Including a graph in a LaTeX document*

Including an external figure in LaTeX is fairly easy. One possibility is to use the `graphicx` package. This LaTeX package can handle both bitmap and vector formats, but for best results you want to use a vector format when available.[2]

As an example, let's assume you want to include the graph from figure 10.3 in a LaTeX document. You'd have to export the graph to PDF, using the following commands:

```
...                      ⟵——— Commands from listing 10.2 go here.
set terminal pdfcairo
set output 'enhanced.pdf'
replot
```

The resulting file can then be included into a LaTeX document. The complete LaTeX document is shown in the following listing.

> **Listing 10.3   Importing a PDF into LaTeX: see figure 10.4 (file: doc1.tex)**

```
\documentclass{article}
\usepackage{graphicx}

\begin{document}

\section{The First Section}

Here is a very short paragraph. The plot will be included
after this paragraph.

\begin{figure}[h]
  \begin{center}
    \includegraphics[width=10cm]{enhanced}
  \end{center}
  \caption{A gnuplot graph, included in \LaTeX}
\end{figure}

And here is a second paragraph. The graph should have
been included before.

\section{The Second Section}

The second section really contains only a very short text.
\end{document}
```

The `graphicx` package provides the `\includegraphics` command, which takes the name of the graphics file to include as a mandatory parameter. (The filename extension isn't required, and it's recommended that you omit it.) The `\include-graphics` command takes a number of optional parameters as key/value pairs, which

---

[2] Also keep in mind that you may have to use `pdflatex` when using graphics in PDF format, but plain `latex` for EPS graphics. Some newer TeX installations perform the necessary format conversion automatically.

let you perform useful operations on the image as it's included: you can trim, scale, and rotate it. Here, its size is adjusted ever so slightly (from 5 inches to 10 cm).[3] The document's final appearance after processing with LaTeX is shown in figure 10.4.
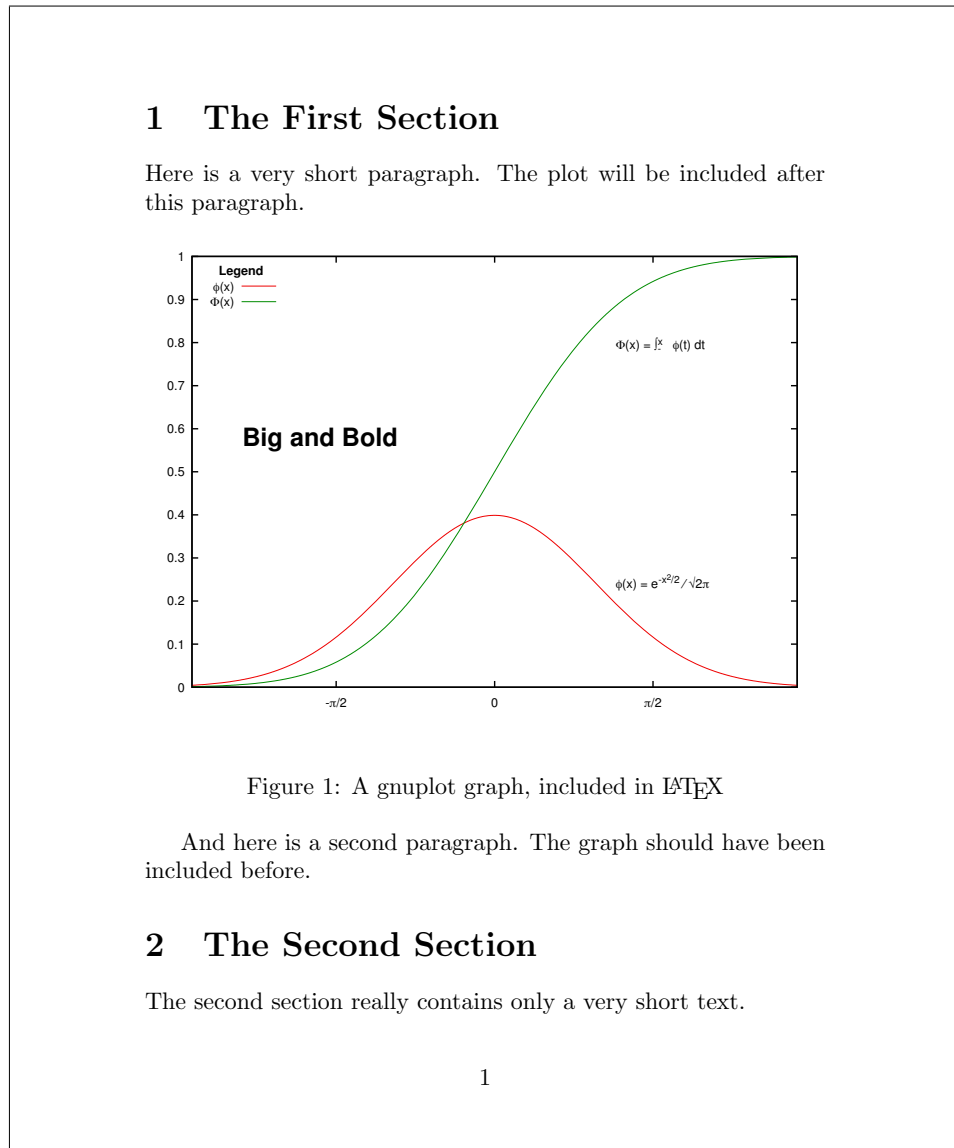
Figure 10.4 The final appearance of the LaTeX document shown in listing 10.3. Note the labels using enhanced text mode in the included gnuplot graph.

---

[3] There are many more options—check your favorite LaTeX reference for details. A good place to start is H. Kopka and P. W. Daly, *Guide to LaTeX*, 4th ed. (Addison-Wesley, 2004).

10.4.2   *Using the cairolatex terminal*

In the previous example, a graphics file containing enhanced-mode text was included in a LaTeX document. This seems inconvenient, to say the least: because LaTeX has such powerful capabilities to format text (and mathematical expressions specifically), there should be ways to use them to lay out the text, rather than dealing with the much more limited possibilities available through enhanced text mode.

The `cairolatex` terminal does exactly that: it splits a gnuplot graph into its graphical and textual components. The graph is stored as a PDF (or EPS) file, and the text is saved to a LaTeX file. You then include this LaTeX document, which in turn imports the graphics file, into your LaTeX master file.[4]

> **TIP**   When gnuplot generates a file that has to be postprocessed using LaTeX, it's particularly important to close the file properly by invoking `set output` after the plot is finished: otherwise, the generated file won't be complete, and LaTeX will fail. (This applies equally to the `cairolatex` terminal, as well as to the `tikz` terminal later in this section, and other comparable terminals.)

A WORKED EXAMPLE

An example will make this clearer. Let's re-create the graph from figure 10.4, this time using LaTeX formatting commands instead of enhanced text mode. See the next listing (compare to listing 10.2 for a version of this graph using enhanced text mode).

Listing 10.4   Combining gnuplot and LaTeX using `cairolatex` (file:cairolatex.gp)

```
set label 1 '$\phi(x) = \frac{1}{\sqrt{2 \pi}} e^{-\frac{1}{2} x^2}$'
➥   at 1.2,0.25
set label 2 '$\Phi(x) = \int_{-\infty}^x \phi(t) dt$' at 1.2,0.8

set label 3 '\textbf{\Large Big and Bold}' at -2.5,0.6

set key top left Left title '\textbf{Legend}'

set xtics ( '$-\pi/2$' -pi/2, '$0$' 0, '$\pi/2$' pi/2 )

plot [-3:3] exp(-0.5*x**2) /sqrt(2*pi) t '$\phi(x)$',
➥          norm(x) t '$\Phi(x)$'

set terminal cairolatex pdf
set o "cairolatex.tex"          ⟵  **Supply the name of the output LaTeX
replot                              file with extension!**
set o          ⟵——  **Make sure to flush and close the output file.**
```

The text for labels and tic marks that are added to the graph now contains LaTeX directives. In particular, note the `$...$`, which indicate *math mode* in LaTeX and which are required for many of the special mathematical formatting commands.

---

[4]   The `cairolatex` terminal is a modern replacement for the `epslatex` terminal, but `epslatex` remains available as well.

All the strings in listing 10.4 use *single quotes*, and therefore the backslashes they contain don't need to be doubled. If you used double quotes instead, then all the contained backslashes would have to be escaped. The third label, for example, would have to be written like so:

```
set label 3 "\\textbf{\\Large Big and Bold}" at -2.5,0.6
```

The `cairolatex` terminal generates *two* files: the graph (as a PDF or an EPS file) and the text (as a LaTeX file). Gnuplot expects the full name (including extension) of the LaTeX file to be set using `set output` when you use the `cairolatex` terminal (see the third-to-last line in listing 10.4). The following listing shows the LaTeX master file.

> **Listing 10.5** **LaTeX document that imports cairolatex.tex (file: doc2.tex)**

```
\documentclass{article}
\usepackage{graphicx}

\begin{document}

\section{The First Section}

This is a different document. We will include a \LaTeX\ file
containing the graph below.

\begin{figure}[h]
  \begin{center}
    \input{cairolatex}
  \end{center}
  \caption{A graphics file included in \LaTeX, with \LaTeX\ labels}
\end{figure}

And here is a second paragraph. The graph should have
been included before.

\section{The Second Section}

The second section really contains only a very short text.

\end{document}
```

Note how similar this document is to the one in listing 10.3. The `graphicx` package must again be specified explicitly. The document then uses `\input` (instead of `\includegraphics`—that's the only difference) to include the LaTeX file written by gnuplot. Again, it's recommended that you omit the extension: LaTeX will look for a file with .tex extension. The final document (after LaTeX processing) is shown in figure 10.5.

1   The First Section

This is a different document. We will include a LaTeX file containing the graph below.

Figure 1: A graphics file included in LaTeX, with LaTeX labels

And here is a second paragraph. The graph should have been included before.

2   The Second Section

The second section really contains only a very short text.

1

Figure 10.5   The final appearance of the LaTeX document shown in listing 10.5. Note the true LaTeX labels and tic marks on the graph. Compare to figure 10.4.

OPTIONS FOR THE CAIROLATEX TERMINAL

Now that you've seen how the `cairolatex` terminal is used, we can take a closer look at its features. It shares most of its features with the `pngcairo` and `pdfcairo` terminals:

```
set terminal cairolatex [ eps | pdf ] [ standalone | input ]
            [ header "{str:header}" | noheader ]

            [ size {flt:x}[ in|cm ],{flt,y}[ in|cm ] ]
            [ font "{str:latexfont}[,{int:size}]" ][fontscale {flt:scale}]

            [ background rgb {colorspec} ] [ [no]transparent ]
            [ [linewidth|lw] {flt:factor} ][ [dashlength|dl] {flt:factor} ]

            [ butt | square | rounded ] [ [no]crop ]
            [ blacktext | colortext ] [ mono | color ]
```

Only three options are new: using the `eps` and `pdf` keywords, you choose the format of the graphics part of the output. Be aware that the default is `eps` (not `pdf`).

If you select `standalone`, the generated LaTeX document is complete and can be compiled by itself. In other words, it doesn't have to be included into another LaTeX document for further processing. This way, you can create standalone graphics with LaTeX-style annotations.

In `standalone` mode, the name of the generated graphics (PDF or EPS) file is modified by appending *-inc* to the base name: the two files written by `cairolatex` would be called, for example, graph.tex and graph-inc.pdf. This means that the resulting graph, which combines both plot and text, can still be named graph.pdf. The default value for this option is `input`, which generates an incomplete LaTeX document for inclusion in a master file, as shown earlier.

The last new option is `header`. With it, you can add an arbitrary set of LaTeX commands into the LaTeX file containing the text component of the plot. For instance, you could use this

```
set terminal cairolatex header "\\bfseries"
```

to print all text in the plot in boldface.

If LaTeX finds the file gnuplot.cfg when processing a standalone file, it's included in the preamble of the document. It's included *before* the argument to the `header` option, so the latter can be used to override global settings from the gnuplot.cfg file.

Finally, the `font` option changes its meaning for the `cairolatex` terminal, compared to the `pdfcairo` and `pngcairo` terminals. Instead of the names of system fonts, `cairolatex` obviously expects names for LaTeX fonts. The font description consists of three parts: the font name, the font series, and the font shape. All three parts are optional, but the commas are mandatory. Here are two quick examples:

```
set terminal cairolatex font "cmr,b,it"    ⟵——— ComputerModern, bold, italic
set terminal cairolatex font ",,sc"        ⟵——— Surrounding font, with small caps
```

You can give the size of the font after the font description: `set terminal cairolatex "cmr" 14`.

The `cairolatex` terminal provides additional ways you can access LaTeX features from within a gnuplot command script. Check the gnuplot standard reference documentation and your favorite LaTeX reference for more details.

10.4.3 *Letting LaTeX generate the graph*

When you use the `cairolatex` terminal, gnuplot draws the graph, and LaTeX provides the text labels. You can go a step further and let LaTeX draw both the text and the graph. Gnuplot merely creates a set of LaTeX commands that accomplish both tasks. Several terminals take this approach; here I'll briefly introduce the `tikz` terminal as an example for the overall concept. The `context` terminal follows a similar idea.

The `tikz` terminal uses the Lua language to create the required LaTeX commands. The next listing re-creates the graph from listing 10.4 but uses the `tikz` terminal.

Listing 10.6   Commands for figure 10.6 (file: tikz-demo.gp)

```
set label 1 '$\phi(x) = \frac{1}{\sqrt{2 \pi}} e^{-\frac{1}{2} x^2}$'
➥  at 1.2,0.25
set label 2 '$\Phi(x) = \int_{-\infty}^x \phi(t) dt$' at 1.2,0.8

set label 3 '\textbf{\Large Big and Bold}' at -2.5,0.6

set key top left Left title '\textbf{Legend}'

set xtics ( '$-\pi/2$' -pi/2, '$0$' 0, '$\pi/2$' pi/2 )

plot [-3:3] exp(-0.5*x**2) /sqrt(2*pi) t '$\phi(x)$', norm(x) t '$\Phi(x)$'

set terminal tikz standalone
set o "tikz.tex"              ⟵——— Supply the name of the output LaTeX file with extension!
replot
set o                ⟵——— Make sure to flush and close the output file.
```

Notice that the `tikz` terminal is invoked with the `standalone` option: in this case, you're creating just the graph. Gnuplot produces a LaTeX file called tikz.tex that can be processed using `pdflatex`. Some auxiliary files that are required by LaTeX are part of the gnuplot distribution, but you can also generate local copies of them in the current directory by including the keyword `createstyle` in the call to `set terminal tikz` .... The resulting graph is shown in figure 10.6. Be aware that you need to have the `pgf` (TeX Portable Graphics Format) package installed, which is part of the LaTeX distribution but may not be installed by default.

How does this entire approach compare to the alternatives? Historically, one advantage of letting LaTeX generate the graph was that LaTeX allowed greater control over the appearance of lines, dash patterns, and the like. This aspect has become less important as gnuplot has acquired some of these capabilities.

Terminals such as `pdfcairo` produce a separate (standalone) graph, which potentially has to be included into a document. Using the `tikz` or `context` terminal, you can embed the commands that generate the graph into the document, so only a

$$\Phi(x) = \int_{-\infty}^{x} \phi(t)dt$$

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$
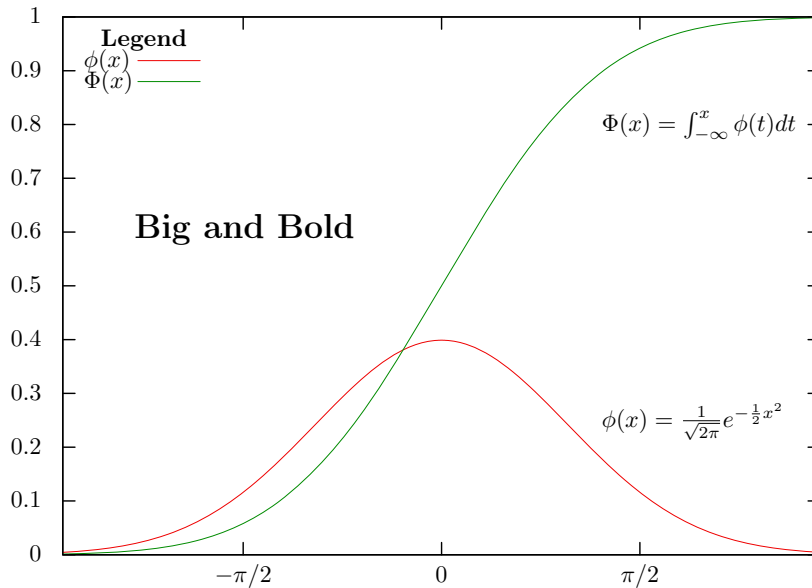
Figure 10.6   A graph drawn using the `tikz` terminal. See listing 10.6 for the commands.

single file has to be processed. Whether this is a good idea is another question—personally, I prefer the *separation of concerns* provided by having standalone graphs and a document that references them. Also be aware that LaTeX may encounter resource (specifically, memory) limitations when processing large graphs.

10.5   Scalable graphics for the Web with SVG and HTML5

Traditionally, graphics to be displayed in web pages were bitmaps—and you can still do this, using the pngcairo terminal, for instance. But gnuplot can also generate graphs using *scalable* formats that can be rendered in a web browser. The svg terminal creates a graph using the Scalable Vector Graphics (SVG) format; the canvas terminal prepares JavaScript commands for use with the HTML5 canvas element.

10.5.1   The svg terminal

The svg terminal behaves similarly to the pngcairo and pdfcairo terminals, but it creates SVG code, which can be captured in a file using set output:

```
set terminal svg [ size {int:x},{int:y} [ fixed | dynamic ] ]
                 [ font "{str:name} [, {int:size} ]" ]
                 [ fname "{str:fontname}" ] [ fsize {int:fontsize} ]
                 [ fontfile {str:filename} ] [ [no]enhanced ]

                 [ rounded | butt | square ] [ background rgb {colorspec} ]
                 [ [linewidth|lw] {flt:fac} ] [ [dashlength|dl] {flt:fac} ]

                 [[ mouse ] [ standalone | jsdir "str:dirname" ]]
                 [ name "{str:title}" ]
```

You can specify a size in pixels; the default size is 600×480. By default, or if the keyword fixed is given, the resulting graph has an absolute size; if dynamic is given, then the resulting graph adapts to the size of the viewer (typically filling the entire available area).

A font can be specified using the usual format, but you can also specify only a font name or font size using the keywords fname and fsize. Using fontfile, a file with font information can be embedded into the SVG document; gnuplot searches for the requested file using the information in the GNUPLOT_FONTPATH environment variable. The font must be an SVG font, not a PostScript or TrueType font.

The keyword mouse enables mousing: mouse coordinates are continuously displayed when the mouse is over the image; and individual curves can be toggled on or off by clicking their entry in the graph's key. Mousing requires an external JavaScript script, which is usually found in a standard location of the local gnuplot installation; you can change the location by providing a path name to the jsdir option. The standalone option embeds the mousing code in the SVG file, rather than referring to an external document. Finally, you can use name to give the plot an identifier by which it can be referred to by JavaScript or a web page that includes the graph.

An SVG file can be used by itself or included in a web page. It's recommended that you use the <object> tag for this purpose, like so:

```
<object type="image/svg+xml" data="graph.svg">
  Your browser does not support SVG.
</object>
```

This also provides a fallback message to the user, in case the user's browser doesn't support SVG.

10.5.2 *The canvas terminal*

The canvas terminal creates JavaScript commands that draw onto an HTML5 canvas element. It can be called in two different modes:

- Using standalone, the terminal produces a web page (that is, HTML code) containing a <canvas> element and the JavaScript to draw on it. The set output option should therefore specify a file with extension .html (or equivalent).
- The name option identifies the name of a JavaScript function that will be generated, and the terminal produces just the set of JavaScript commands that implement that function. The set output option should specify a file with extension .js.

The canvas terminal supports various options:

```
set terminal canvas [ size {int:x},{int:y} ]
                    [ font "{str:name} [,{int:size}] ] [ fsize {int:size} ]

                    [ [no]enhanced ]
                    [ title "{str:title}" ]
                    [ rounded | butt | square ] [background rgb {colorspec}]
                    [ [linewidth | lw] {flt:fac} ]
```

```
[ [dashlength | dl] {flt:fac} ]
[ standalone [mousing] | name "funcname" ]
[ jsdir "{str:url}" ]
```

You can specify a font using `font`. This setting is currently ignored, but a font *size* can be selected using `fsize`.

In `standalone` mode, the terminal creates a complete web page, which includes the JavaScript code for the graph. In addition, the web page refers to various other JavaScript files that are part of the gnuplot installation and that it expects to find in a standard location. You can specify a different directory using the `jsdir` option; this may be appropriate when the web page isn't viewed locally, but is served on the Web. If the `mousing` keyword is used in conjunction with `standalone`, then some additional elements are included in the page that show the mouse coordinates and provide further ways to interact with the graph (such as zoom and unzoom functionality).

When the `name` option is used, gnuplot generates only the JavaScript. The resulting code contains the implementation of a JavaScript function with the desired name. For example, assume that you've created a JavaScript file called plotter.js, containing a function called `drawplot()`, using the following commands:

```
...                        ⟵——— plot commands go here.
set terminal canvas name "drawplot"
set output "plotter.js"
replot
```

These files can be used in a web page like the one in the following listing.

Listing 10.7   Using the `canvas name` terminal (file: canvas-demo.html)

```
<html>
  <head>
    <script src="/usr/share/gnuplot/5.0/js/canvastext.js"></script>
    <script src="/usr/share/gnuplot/5.0/js/gnuplot_common.js"></script>
  </head>

  <body onload="drawplot();">
    <script src="plotter.js"></script>
    <canvas id="drawplot" width=600 height=400>
      <div id="err_msg">Your browser does not support HTML5 canvas.</div>
    </canvas>
  </body>
</html>
```

Several details are important:

- The header section references two external scripts, which are part of the gnuplot distribution. Make sure the path to these files is correct in the header and that the files are visible.
- The name of the function appears twice in the page: as the value of the `onload` attribute of the <body> tag, and as the value of the `id` attribute of the <canvas> tag.

- The name of the generated JavaScript file is used in the `<script>` tag within the page's body.

10.6 *Interactive terminals*

In contrast to the large range of options available for file-based terminals, interactive terminals are rather straightforward.

10.6.1 *Common options*

Gnuplot can have multiple plot windows open simultaneously, although only one of the windows will receive `plot` commands at any time. You can create a new plot window by specifying the terminal ID (which is just an integer expression) when choosing the terminal, like so:

```
set terminal wxt 1
```

To switch between open windows, use `set terminal` together with the terminal type followed by the window ID to select the active terminal. The keyword `close` in a `set terminal` command closes the identified window. (Depending on the window manager, the window may persist until it has received mouse or keyboard focus for the first time after the `set terminal ... close` command has been issued.)

To distinguish between multiple plot windows, you can give each one a title, which will (depending on the window manager) be displayed in the window's header bar. (This is the title of the *window* only and has nothing to do with the title that can be placed on the plot using `set title`.)

Plot windows are automatically raised and receive input focus when a new plot is drawn. You can disable this behavior using the `noraise` option, so that keyboard focus stays with the command window even when a `plot` command is issued.

By default, pressing the space bar while the plot window has focus raises the command window, and pressing the Q key closes the plot window. Using the `ctrl` option, these bindings are replaced by combinations of the Ctrl key together with either the space bar or the Q key (thus freeing up those keys to be rebound otherwise: see section 12.4.3).[5]

When the `persist` option has been enabled, plot windows stay up even when gnuplot is exited (or prevent gnuplot from terminating completely until all plot windows have been closed explicitly).

You can use three commands to manipulate interactive terminals directly: `raise`, `lower`, and `clear`. The first two take an optional window identifier as optional argument. The `clear` command (which always acts on the most currently active terminal) is useful even for non-interactive terminals: it clears the plot area (and forces a form-feed or page break for non-interactive terminals). It's required, for example, in some applications of multiplot mode (see section E.1).

---

[5] Everything having to do with keyboard focus and the raising and lowering of windows depends on the interplay of the window manager and the widget set implementing the terminal. As of this writing, not all of this functionality works with all window managers.

10.6.2 *The wxt and qt terminals*

The wxt and qt terminals are very similar in their functionality but are built using different widget sets. Because both widget sets are cross-platform, both terminals are available on all current operating systems: Linux, Windows, and Mac OS X.

Both terminals share the most important options:

```
set term [ wxt | qt ] [ {int:id} ] [ title "{str:title}" ]
        [ size {int:width},{int:height} ] [ position {int:x},{int:y} ]
        [ font "str:face[,{int:size}]" ] [ [no]enhanced ]
        [ [no]raise ] [ [no]persist ] [ [no]ctrl ]
        [ close ]
```

In addition, both terminals support the dashlength (or dl) option. The wxt terminal (but not the qt terminal) also offers the background (or backg) and linewidth (or lw) options.

10.6.3 *The aqua terminal*

The aqua terminal is a native Mac OS X terminal for gnuplot. It relies on AquaTerm.app:

```
set term aqua [ {int:id} ] [ title "{str:title}" ]
            [ size {int:width},{int:height} ]
            [ font "str:face[,{int:size}]" ] [ [no]enhanced ]
```

10.6.4 *The windows terminal*

The windows terminal is for use on Microsoft Windows platforms:

```
set term windows [ {int:id} ] [ title "{str:title}" ]
        [ size {int:width},{int:height} ] [ position {int:x},{int:y} ]
        [ font "str:face[,{int:size}]" ] [ [no]enhanced ]
        [ close ]
```

The windows terminal supports some additional GUI features; see the standard gnuplot reference documentation for details.

10.7 Other terminals

As mentioned in the introduction to this chapter, gnuplot supports a *lot* of terminals—many more than I discussed in this chapter. Although most of them are no longer relevant, some may be of use in specialized situations.

Traditionally, the postscript terminal was very important, and so were the gif, png, and jpeg bitmap terminals based on the GD library. These terminals have largely been replaced by the pdfcairo and pngcairo terminals discussed earlier in this chapter.[6]

---

[6] The default appearance of graphs produced using pdfcairo differs somewhat from those generated with postscript. Explicitly specifying the aspect ratio with set terminal pdfcairo size 5in,3.5in and adjusting the border thickness and point size with set border lw 1.5 and set pointsize 0.75 bring the results from the pdfcairo terminal much closer to the look familiar from the postscript terminal. If you want to create monochrome graphs, you'll need to define your own dash patterns—see the stylesheet in listing 12.9 for an example. You may also want to specify the Helvetica font, but the result will depend on the selection of fonts installed on your system (whereas PostScript graphs were guaranteed to use the actual Helvetica, because it's one of the 35 PostScript core fonts).

There are many more terminals that generate commands suitable for TeX or LaTeX processing than I described here. If you're interested, you may want to check out the standard gnuplot reference documentation for `eepic`, `latex`, `pslatex`, `pstricks`, `texdraw`, and `tpic`.

In addition to LaTeX, gnuplot supports some other graphical command languages. Check out the `fig` and `tgif` terminals.

You should also know about two terminals that are truly obsolete but are still supported and may provide a safe fallback solution if necessary: `x11` and `dumb`. The `x11` terminal is built on the X11 system without the benefit of a modern widget system. As an X11 application, it honors many of the standard X11 conventions, such as geometry, color resources, and app-defaults. The `x11` terminal has been superseded by terminals such as `wxt` and `qt` that are built using contemporary widget sets.

Finally, a true classic: the `dumb` terminal (see figure 10.7). This is a terminal for character-oriented output devices. It uses characters to draw an ASCII-art rendition of the plot: it's the fallback for console-oriented systems or over *really* slow network connections. For all of these terminals, see the standard gnuplot reference documentation.
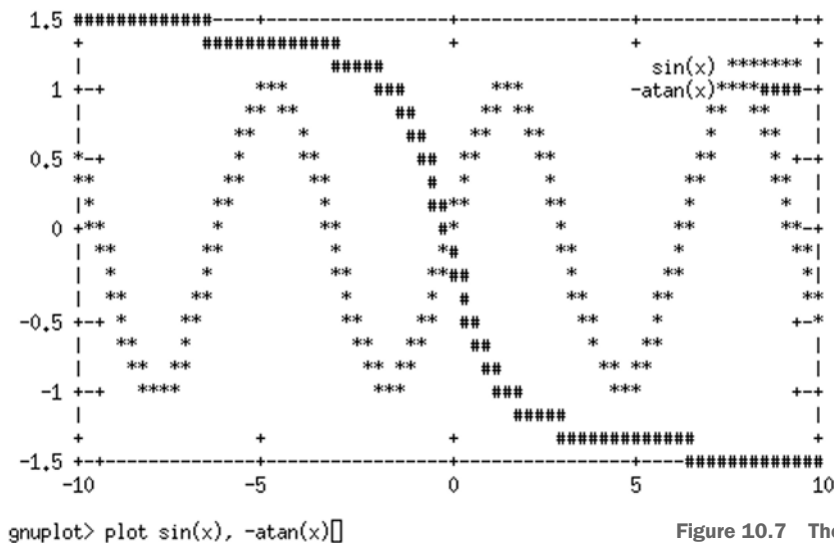
`gnuplot> plot sin(x), -atan(x)`   **Figure 10.7   The `dumb` terminal**

10.8   *Summary*

Gnuplot uses the terminal facility to create output for other applications and devices. The `pngcairo` and `pdfcairo` terminals create graphs in standard bitmap (PNG) and vector (PDF) formats. The `svg` and `canvas` terminals can generate vector graphics suitable for inclusion in dynamic web pages.

The appearance of strings and text labels depends on the available fonts and the capabilities of the chosen output format. Enhanced text mode lets you embed simple formatting commands (such as sub- and superscripts) in text labels.

For those who are familiar with LaTeX, the `cairolatex` terminal makes it possible to use LaTeX's text (and, in particular, math) formatting capabilities together with gnuplot's plotting facilities to achieve particularly high-quality plots for publications or other printable documents. Finally, I provided a brief overview of interactive terminals for various platforms.

Automation, scripting, and animation

This chapter covers
- Loops and conditionals
- Command files and batch processing
- Calling gnuplot from other programs
- Animations

In the preceding chapters, we've studied most commands and options that can be used from within gnuplot. But up to this point, our discussions assumed that you'd be running gnuplot interactively: entering commands, viewing plots, and then entering more commands.

But sometimes that's not what you want. For instance, you may have a large number of data sets and just want to convert all of them to plots: it would be great to automate this task through *scripting*. Or you may have a `cron` job running nightly to scan web server logs and want it to also generate some plots showing the number of hits per hour, without requiring human intervention: that's *batch processing*. Or you might want to use gnuplot as a *graphing back end*, while tendering control to some other program.

236

All these situations involve tasks that have been *automated* to some degree. In this chapter, we'll study ways to do this, either using gnuplot's own facilities or using gnuplot in conjunction with an external programming language.

11.1 Loops and conditionals

Gnuplot has support for loops and conditionals: this makes it possible to automate repeated tasks. In contrast to the *inline loops* that we encountered earlier (see section 5.4), the control structures introduced in this section are more general. They can be applied to any command, or any combination of commands—not just a single `plot` or `set` command, as was the case for inline loops. In addition to two different loops and a conditional construct, there's also a special operator that calculates the sum of a set of numbers. Let's look at all of them in detail; table 11.1 provides a summary.

The `do for` loop is a generalization of the inline loop concept. Loops can extend over a numeric range (start and end inclusive; the increment is optional and defaults to 1). Alternatively, the `do for` loop can also iterate over whitespace-separated tokens in a string.

The other loop is a `while` loop. It executes the statements in the loop body as long as the supplied condition doesn't evaluate to 0.

The `if` statement is used to construct conditionals. The `else` clause is optional; at present, there is no provision for repeated conditions (no `elsif` or similar).

Finally, the `sum` construct is an *expression* (meaning it returns a value). It iterates over a set of integers, evaluates a given expression, and adds the expression's value to a running total. Upon completion, the value of the running total is returned. It isn't possible to give a step size, and the evaluated term need not include the iteration variable:

```
sum[var=start:end] expression
```

The following examples will make this clearer:

```
print sum[k=1:10] 1                    ⟵——— Prints 10.0
print sum[k=1:10] k                     ⟵——— Prints 55.0
plot "data" u 1:(sum[k=2:5] column(k)/4.)   ⟵——— Plots the average of cols 2–5
```

Limitations of gnuplot's programming model

At the present time, gnuplot offers only limited support for scripting. It has control structures (loops and conditionals) but neither data structures[a] nor subroutines—and of course no such thing as local variables! This shouldn't be a surprise: gnuplot was never intended as a programming environment, and the available scripting support is little more than an ad hoc collection of convenience features, intended to take the tedium out of repeated tasks.

---

[a] But see the footnote on page 15.

(continued)

The problem is that control structures like loops require arrays and subroutines to work properly: they don't make much sense otherwise. But because gnuplot doesn't offer such facilities, existing features find themselves "press-ganged" into roles that they aren't intended for: whitespace-separated strings, together with the `word()` and `words()` functions, are used to mimic iterable data structures; array indexes are faked through numeric suffixes on variable names; and individual command files have to stand in for subroutines. Depending on your perspective, you may find this either hideous or ingenious.

You can't very well have "a little" programming support. Either you don't have any, or you'll find that each "small" feature before long leads to the addition of yet another "small" feature, until you end up with a complete programming environment anyway—simply by following the law of unintended consequences.

Because of this, I doubt the wisdom of adding programming support to gnuplot through opportunistic accretion and without clarity about the intended programming model. The evolution of programming languages over the last 30 years (and the last 15, in particular) has shown how difficult it is to create a programming environment that "wears well as one's experience with it grows."

Table 11.1 Gnuplot loop constructs and conditionals

Syntax	Example

`do for [var=start:end:increment] {`
`  statement`
`  statement`
`  ...`
`}`
	`do for[k=1:10] {`
	`    print k`
	`}`

`do for [var in tokenstring] {`
`  statement`
`  statement`
`  ...`
`}`
	`do for [s in "abc uvw xyz"] {`
	`    print s`
	`}`

`while( condition ) {`
`  statement`
`  statement`
`  ...`
`}`
	`x = 0`
	`while( x < 10 ) {`
	`    x = x + 1`
	`    print x`
	`}`

`if( condition ) {`
`  statement`
`  statement`
`  ...`
`} else {`
`  statement`
`  statement`
`  ...`
`}`
	`x = 0`
	`if( x < 0 ) {`
	`    print "Negative"`
	`} else {`
	`    print "Not negative"`
	`}`

11.1.1 *Worked example: making graph paper*

To kick off our list of examples for gnuplot's looping constructs, let's make some quad-ruled graph paper. This application demonstrates how *repetitive* tasks can be automated with loops. It also has an immediate graphical representation (see figure 11.1).
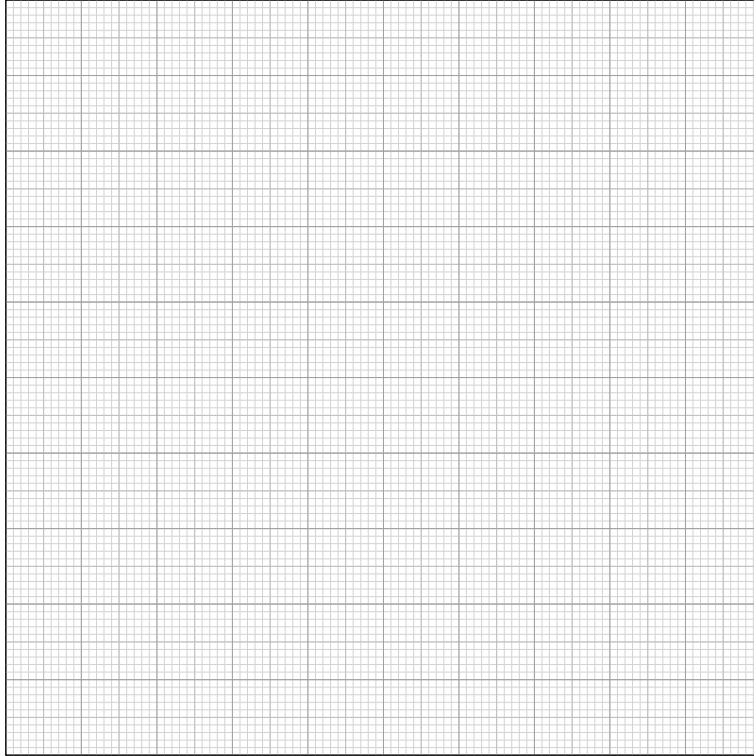
Figure 11.1   Graph paper: an easy application of gnuplot's loops. See listing 11.1.

On commercially produced graph paper, every fifth line is printed a little wider than the remaining lines to impose some larger-scale, visual guidance to the user. In the graph in figure 11.1, I instead made every fifth line a little *darker* than the other lines, using different shades of grey. For best effect, it's important to add the darker lines later to the plot than the other lines.

The commands for the graph paper in figure 11.1 are shown in listing 11.1. The size of the entire graph is set to 12.5cm✕12.5cm, but the actual plot (the *canvas*) is scaled to 80% of that, leading to a finished 10cm✕10cm square with millimeter spacing of individual lines.

Listing 11.1   Commands for figure 11.1 (file: graphpaper.gp)

```
set terminal pdfcairo size 12.5cm,12.5cm
set o 'graphpaper.pdf'

set size 0.8,0.8
set origin 0.1,0.1
set margins 0,0,0,0

unset key
unset tics

set style arrow 1 nohead lw 1

set for [i=1:99] arrow from i/10.,0 to i/10.,10 as 1 lc rgb 'light-grey'
set for [i=1:99] arrow from 0,i/10. to 10,i/10. as 1 lc rgb 'light-grey'

set for [i=5:95:10] arrow from i/10.,0 to i/10.,10 as 1 lc rgb 'grey'
set for [i=5:95:5] arrow from 0,i/10. to 10,i/10. as 1 lc rgb 'grey'

set for [i=10:90:10] arrow from i/10.,0 to i/10.,10 as 1 lc rgb 'dark-grey'
set for [i=10:90:10] arrow from 0,i/10. to 10,i/10. as 1 lc rgb 'dark-grey'

plot [0:10][0:10] 1/0
set o
```

11.1.2 *Worked examples: iterating over files*

It's no surprise that gnuplot's looping facilities come in handy whenever you're dealing with a large number of similar files. Let's consider two examples—one straightforward, the other one not.

PLOTTING SEVERAL FILES

Imagine that you have a directory containing many data sets. Each data set is in a separate file, and the files are all named in a predictable fashion: data01, data02, and so on. You want to generate a plot of every data set, each in its own file, with appropriately chosen filenames. Here's how to do it (the `sprintf()` function comes in handy to format the input and output filenames properly):

```
set terminal pdfcairo                              ⟵——— Chooses the output file format
do for[k=1:50] {
  set output sprintf( "graph%02d.pdf", k )         ⟵——— Creates the output file
  plot sprintf( "data%02d", k ) u 1:2              ⟵——— Plots the data
  set output                                       ⟵——— Flushes the output channel
}
```

MAKING DATA COMPARABLE

The next example is more complicated. In this case, you have only three files (data1, data2, and data3), but you don't want to plot each data set in a separate graph. Instead, you want to combine them onto a single graph so that they can be compared easily. To compare disparate data sets meaningfully, it's often necessary to *normalize* the data before plotting it. For now, and to keep the exposition simple, let's assume that you just want to subtract the respective average value from each data set before

plotting it. (You'll see another example of normalizing data for the sake of comparison in section 14.1.3.)

It's easy enough to loop over the files and use the stats command (see section 5.6) to calculate the respective average (or mean) values. The problem is that you need to *store* the separate values temporarily, so that you can use them later in the plot command. Because gnuplot doesn't have an array data structure, you need to bend over backward to accomplish this goal. I'm going to show you two different ways to get there.

In the first implementation, variables with numeric suffixes stand in as array entries. But because it's impossible to create variables dynamically, a *string* containing the variable definition is constructed instead, which is then executed using eval. The subsequent plot command, using an inline loop, does the same thing—it builds up the variable name as a string and obtains the value of the corresponding variable using the value() function (see section 5.1 for information about value()):

```
do for[file in "data1 data2 data3"] {
  stats file u 2 nooutput
  eval sprintf( "mean_%s=%f", file, STATS_mean )
}

plot for[file in "data1 data2 data3"]
➥    file u 1:($2-value(sprintf("mean_%s", file)))
```

The second implementation may be more straightforward, but personally I find it even more ominous. It fully embraces the idea that an array is nothing more than a sequence of values and hence can be conveniently represented as a string. String operations are then used to extract the desired entry.

In the first loop, a string is built up from the values calculated by stats (don't miss the whitespace in the sprintf() format specifier that indicates token boundaries). The plot command contains an inline loop and uses the word() function to extract each appropriate element in turn from the string of values (see table 5.1 for information about the word() function):

```
files = "data1 data2 data3"
means = ""
do for[file in files] {
  stats file u 2 nooutput
  means = means . sprintf( " %f", STATS_mean )
}

plot for[k=1:3] word(files,k) u 1:($2-word(means,k))
```

11.1.3 *Worked examples: Taylor series and Newton's method*

In this section, I demonstrate two other applications of looping constructs. Both require some familiarity with calculus—feel free to skip this section.[1]

---

[1] This isn't the place to explain either Taylor series or Newton's method; the purpose of this section is to demonstrate gnuplot's looping features. For this reason, my presentation of the mathematical concepts is intentionally brief. The Wikipedia pages for both topics are reasonably accessible.

TAYLOR SERIES

According to Taylor's theorem, every smooth function can be locally approximated by a polynomial: the higher order the polynomial, the better the approximation. We demonstrate this here for the function $\sin(x)$, which has the following Taylor approximation:

$$\sin(x) \approx \sum_{k=1}^{n} (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!}$$

Using gnuplot's `sum` feature, together with the `!` operator to indicate the factorial, this can be written very compactly in gnuplot (also see figure 11.2—the function `s(n)` has been introduced to fix the key entry when there is only a single term in the series):

```
f(x,n) = sum[k=1:n] (-1)**(k-1)*x**(2*k-1)/(2*k-1)!
s(n) = (n==1 ? "1 Term" : sprintf( "%d Terms", n ) )
plot [-8:8][-1.2:1.2] sin(x) lw 2, for[n=1:4] f(x,n) t s(n)
```
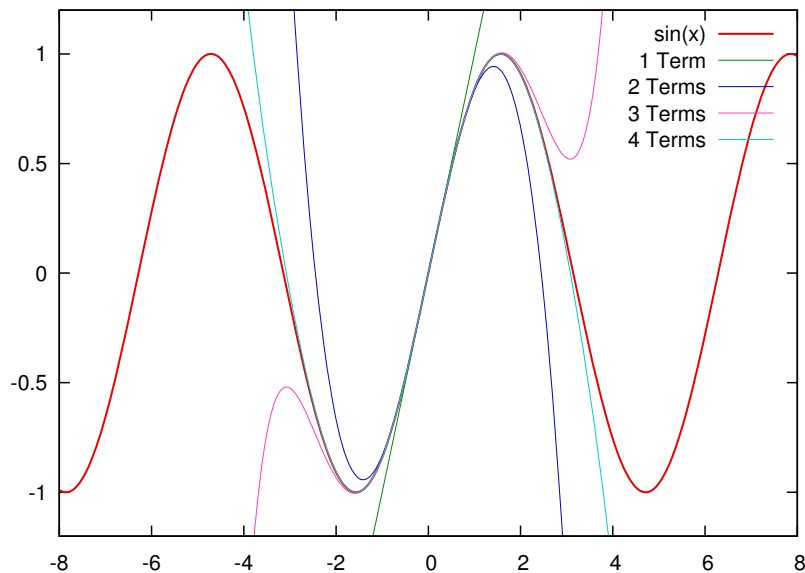

Figure 11.2   Successive approximations to sin(x) using the `sum` facility

NEWTON'S METHOD

The other example involves Newton's method for finding the roots of an analytic function. Newton's method consists of an iteration of the following form:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

This can be implemented using a `while` loop. The derivative $f'(x)$ is evaluated through a finite-difference approximation:

```
f(x) = cos(x)
steps=0; x=0.5; eps=1e-6

while( abs(f(x)) > 1e-10 && steps < 15 ) {
  df = (f(x+eps) - f(x) )/eps
  x = x - f(x)/df
  steps = steps + 1
  print steps, x, f(x)
}
```

With the given choice of parameters, Newton's method converges to the first positive root of the cosine in five steps. Here's the total output from running this snippet:

```
1 2.33048604607464 -0.688696509312027
2 1.38062453566562 0.189027591805359
3 1.57312250575387 -0.0023261768611035
4 1.57079632259625 4.19864216189357e-09
5 1.5707963267949 6.12323399573677e-17
```

11.2 Command files

Gnuplot commands can be not only entered at the command prompt but also captured in a script file. Such a file can be loaded into a gnuplot session, included in other command files, or processed by gnuplot in batch mode.

So far, we've encountered command files mostly as the product of the `save` command. The files generated by `save` capture the entire state of the gnuplot session at that moment: the values of all options and variables, all user-defined functions, as well as the most recent `plot` command. If you `load` such a file, it re-creates exactly the previous session state, and—in particular—it re-creates exactly the previous plot.

It's worth emphasizing that this isn't the only way to use command files; in particular, it isn't necessary for command files to contain a `plot` or `splot` command. Command files without either of these commands obviously don't generate visual output, but they can be very useful nevertheless:

- As *setup* or *configuration* scripts—Such scripts contain nothing but function definitions, customized line types and styles, or shorthands for commonly used color gradients. We'll explore some of these possibilities in chapter 12.
- As *subroutines*—I mentioned earlier that gnuplot doesn't have subroutines: it's not possible to group several statements together into a single, callable unit. Command files provide a workaround for this limitation.
- As *batch files*—The contents of a command file can be processed by gnuplot in the background, without an interactive user session.

11.2.1 Scripts as subroutines

Gnuplot doesn't have a way to group several commands or statements together into a single, callable unit. Gnuplot's *functions* are expressions (they return a value), and their definitions must consist entirely of expressions as well. It isn't possible to evaluate a gnuplot command within a gnuplot function, much less a sequence of commands.

Scripts provide a workaround. A script can contain any valid gnuplot code and is callable as a single unit. The generic export script is a good example (compare section 10.1.2):

```
set terminal push
set terminal pdfcairo
set output "graph.pdf"
replot
set output
set terminal pop
```

This sequence of commands can be run with a single invocation of load or call.

PASSING PARAMETERS TO SCRIPTS

The script in the previous example didn't take any parameters: the name of the output file was hardwired. That's not ideal; it would be better to supply a filename at runtime. There are two ways to do this:

- Prepopulate a gnuplot session variable, and use it in the script.
- Use call to invoke the script, and specify the parameters as part of the command.

To use the first method, the export script must be changed to use a variable:

```
set terminal push
set terminal pdfcairo
set output outfile          ◁──┐  The outfile variable holds
replot                          the desired filename.
set output
set terminal pop
```

Be sure to assign the proper value to this variable *before* invoking the script (assuming that the script is called export.gp):

```
outfile = "graph.pdf"
load "export.gp"
```

The alternative is to use the call command to invoke the script (instead of load). The call command accepts up to nine parameters on the command line, which are available in the script in variables called ARG1, ARG2, and so on. (The variable ARG0 is set to the name of the script file, and the variable ARGC holds the number of arguments supplied to call.) A version of the export script, suitable for call, looks like this:

```
set terminal push
set terminal pdfcairo
```

```
set output ARG1                    ⟵——— Uses the special variable
replot
set output
set terminal pop
```

You call it like this:

```
call "export.gp" "graph.pdf"
```

Personally, I prefer the `call` mechanism over prepopulating session variables, because it makes the parameter passing explicit. You need to be aware of an important limitation, though: parameters are passed by `call` as *strings* and therefore must be representable by strings (numbers are converted to strings on the fly). This makes passing functions inconvenient, whereas a function that exists in the current session can be used by the script. (You'll see an example in a moment, when we discuss a script that implements Newton's method.)

SIMULATING LOCAL VARIABLES

Gnuplot doesn't have local variables: all variables that are created live in the global session space. This means if you call a script that defines variables, these variables are still part of the session when the script finishes. This may be desirable (because it provides a way to retrieve values from the script), but more often it isn't. The problem also exists in reverse: if a variable existed in the session *before* the script was run, then it will have been clobbered afterward.

The solution is to give the names of all strictly internal variables a unique *prefix*. You can even use the `undefine` command to clean up the session by removing all variables with a given prefix upon completion of the script.

FURTHER POSSIBILITIES WITH SCRIPTS

Scripts may contain `load` and `call` commands. Their effect is to include the contents of the loaded file at the position of the `load` or `call` command. (Gnuplot safeguards against infinite recursive inclusion.)

When you use the command `quit` in a script, it doesn't terminate the gnuplot session; instead, it *terminates execution of the current script*. It therefore acts similarly to a `return` statement in most programming languages, initiating an immediate return to the calling code (but without a return value, of course).

A WORKED EXAMPLE: NEWTON'S METHOD (AGAIN)

Listing 11.2 is another implementation of Newton's method that demonstrates some of the ideas just introduced. The script defines three internal variables, all of which are prefixed with `NEWTON_` and are removed from the session on the last line of the script. Parameters are passed to the script implicitly: both the function `f(x)` and the starting value `x` must exist in the session before the script is called. Upon completion, the variable `x` holds the position of the found root. A sample use of this script might be as follows (the last line is the final output from the script):

```
f(x) = cos(x**2); x=1.0              ⊲──── Prepopulates function f(x) and starting value x
load "newton.gp"                     ⊲──── Invokes script
f(1.253314137316) = -0.000000000001 in 4 steps    ⊲──── Output
```

> **Listing 11.2   Newton's method in gnuplot (file: newton.gp)**

```
# Must prepopulate f(x) and x before call

NEWTON_steps = 0
NEWTON_eps = 1.e-6

while( abs(f(x)) > 1e-12 && NEWTON_steps < 15 ) {
  NEWTON_df = (f(x+NEWTON_eps) - f(x) )/NEWTON_eps
  x = x - f(x)/NEWTON_df
  NEWTON_steps = NEWTON_steps + 1
}
print sprintf("f(%.12f) = %.12f in %d steps", x, f(x), NEWTON_steps)

undefine NEWTON*
```

11.2.2 *Worked example: export script*

The script in listing 11.3 is intended to demonstrate how far you can push the "script-as-subroutine" concept in gnuplot. You decide whether this is a good idea.

The purpose of the script is my old standby: saving the commands of the last plot, and exporting the plot to a file in a standard graphics file format.[2] But this script does more:

- It infers the desired file format by examining the extension of the target file.
- The target file can include an absolute or relative path. The script places both the command file and the graphics file in the specified directory.
- You can specify additional options when invoking this script. These options are supplied to the chosen terminal when the graph is exported.

The script makes a number of (platform-specific) assumptions:

- The filename extension indicating the desired graphics file format consists of exactly three letters.
- The path separator is / (Unix convention—you need to change this if you're on a different platform).

The script itself is straightforward but messy, because much of the string handling has to be done explicitly. Moreover, gnuplot doesn't follow the convention of most contemporary programming languages of using zero-offset strings and asymmetric bounds, thus necessitating the explicit conditional when splitting the argument into path and basename. (To reduce clutter, I have refrained from prefixing all internal variables to make them "private.")

---

[2] If gnuplot should ever acquire a simple `export` command that bundles all the required steps into a single invocation, I'll lose one of my best sources for examples.

As I said, I regard this example as a "technology demonstrator" that shows how you can encapsulate complex tasks in scripts and use them like subroutines. You decide whether you think this is the right approach.

> **Listing 11.3 Full-featured export script (file: mega-export.gp)**

```
# Invoke as: call "mega-export.gp" "graph.png" "terminal options"
# File format MUST be a three-letter extension.

arg = ARG1
opts = ARG2          Saves arguments
len = strlen( arg )

k = 0
do for [i = 1:len] {
  if( arg[i:i] eq "/" ) {      Finds the last "/"
    k = i                      in the filename
  }
}

if( k == 0 ) {        ◁────── No "/" found: uses current directory
  path = ''
  file = arg[1:len-4]                                          Splits the
} else {              ◁────── At least one "/" found            filename
  path = arg[1:k]                                              into path,
  file = arg[k+1:len-4]                                        file, and
}                                                              extension
type = arg[len-2:]    ◁────── Assumes a 3-letter extension

term = ''
if( type eq 'eps' ) { term = 'epscairo' }
if( type eq 'pdf' ) { term = 'pdfcairo' }
if( type eq 'png' ) { term = 'pngcairo' }     Selects a terminal
if( type eq 'svg' ) { term = 'svg' }          based on the extension
if( term eq '' ) {
  print "Unknown file type - using pdf"
  term = 'pdfcairo'
}

save path . file . '.gp'

set t push
eval "set t " . term . " " . opts
set o path . file . "." . type
replot
set o
set t pop

print "Done!"
```

11.3   *Batch processing*

Gnuplot can be run by itself as a standalone command interpreter, executing scripts from files—basically like a Perl or Python interpreter interpreting a program in its respective language. The only difference is that in gnuplot's case, the final output is a *graph*. Running gnuplot non-interactively as a command interpreter like this is known as *batch mode*.

TIP   Batch mode isn't terribly common, because you usually want to *see* and interact with the graph that's created. Batch mode is useful whenever you want to create a large number of file-based graphs using a predetermined set of commands, or when you need to re-create the same graphics file again and again as the data set that's plotted changes.

Running gnuplot in batch mode is straightforward and equivalent to the way Perl and Python behave: any files listed after the `gnuplot` command are expected to contain gnuplot commands. They're executed in the order specified, as if they had been loaded using `load`. Alternatively, gnuplot will read commands from standard input. There are thus three ways to execute a set of command files:

- Using command-line arguments:
  ```
  shell> gnuplot plot1.gp plot2.gp plot3.gp
  ```
- Reading from standard input:
  ```
  shell> cat plot1.gp plot2.gp plot3.gp | gnuplot
  ```
- From within a gnuplot session:
  ```
  load "plot1.gp"
  load "plot2.gp"
  load "plot3.gp"
  ```

Gnuplot doesn't start an interactive session when invoked with command-line arguments: it just processes all commands (including any `plot` commands) and terminates. This implies that *plots sent to an interactive terminal usually aren't visible*, or rather, they're visible for a tiny moment as gnuplot opens the terminal window, draws the graph, and immediately closes the plot window again and exits.[3] It's a common mistake to forget to set appropriate (file-based) terminal and output options in gnuplot batch files!

TIP   By default, gnuplot uses one of the interactive terminals for all graphical output. That's great for interactive sessions but makes no sense when you're using gnuplot in batch mode. Don't forget to choose a meaningful output format and destination.

You can force an interactive session by using the special filename - (hyphen) on the command line. The following command runs a setup script before dropping you into an interactive session (you'll need to issue an end-of-file character before quitting):

```
shell> gnuplot setup.gp -
```

---

[3] You can also specify the command-line flag –persist when invoking gnuplot so that the plot remains visible on the screen.

11.3.1 *Using gnuplot in shell pipelines*

It's possible to use gnuplot as part of a command pipeline,[4] wherein gnuplot reads both commands *and* data from standard input. The special filename - (see section 4.5) tells gnuplot to read data from the current source: that is, from the same input device from which the most recent command was read. So if the command was read from standard input, then `plot "-"` will continue reading data from standard input as well. This makes it possible to build a pipeline like the following:

```
shell> cat script.gp data | gnuplot
```

Here, the command file script.gp may be as simple as

```
set t pngcairo
set o "out.png"
plot "-" u 1:2 w lp
```

and the data file may contain any data set, as long as it has at least the two columns referenced in the `plot` command.

The pipeline concept can be taken a step further: remember that `set output` defaults to standard output (if no filename is given as argument). Now the script is even simpler

```
set t pngcairo
plot "-" u 1:2 w lp
```

but in the pipeline you need to deal with the results that gnuplot sends to standard output. The easiest way is to capture them in a file

```
shell> cat script.gp data | gnuplot > graph.png
```

but it would of course also be possible to pipe gnuplot's output to some other post-processing step instead.

A word of warning: to use gnuplot in a pipeline like that, both commands and data must be sent to standard input explicitly. Just listing them on the command line doesn't work: gnuplot won't read a combination of command and data files from the command line.

> **TIP** Using gnuplot in a shell pipeline may seem unusual at first, but it can be very convenient. Any pre- or postprocessing can be done in one fell swoop. Moreover, sending the graph to standard output avoids all of gnuplot's awkward file-handling issues.

---

[4] Provided the platform supports pipes, of course.

11.4   Calling gnuplot from other programs

Sometimes it's necessary or desirable to let gnuplot be controlled by another computer program. Three situations come to mind:

- You need a level of automation and control that is difficult or impossible to achieve with gnuplot's limited scripting capabilities.
- You want to use gnuplot to visualize data that is generated by some other program without having to save the data to an intermediate file.
- You'd like to use gnuplot as a generic graphics back end for a system that has no graphics capabilities otherwise.

You can use gnuplot this way, and it isn't even particularly difficult—but it isn't especially *convenient*, either. Gnuplot doesn't offer an API or a robust method of interprocess communication. Gnuplot must be run as a subprocess, and all communication has to occur via pipes (where available) or otherwise through the file system. Bidirectional communication (that is, sending data back from gnuplot to the controlling process) is particularly challenging.

> **TIP**   Before investing effort in one of the solutions sketched in the following sections, consider whether batch operations are sufficient. In batch mode, your program creates a *file* with gnuplot commands (and a heredoc containing the data) on disk and then invokes gnuplot on that file (in a subshell). The detour through the file system effectively *decouples* gnuplot from your program, which makes operations more robust and easier to debug.

In the following two sections, I demonstrate how to run gnuplot from either Perl or Python. The master process sends some data to gnuplot (via a pipe), and gnuplot creates a graphics file on disk—this is probably the most common case. The main differences between these examples concern the way Perl and Python handle subprocesses and have little to do with gnuplot itself. (The Python example also makes use of a heredoc, whereas the Perl example uses the - special file.)

11.4.1   Worked example: calling gnuplot from Perl

Perl, the "duct tape of the internet," is (still) a common choice for gluing processes together. It lets you open a subprocess as a file handle—in the same way you'd open a file—provided the first character in the filename is a | (pipe symbol). You can then write anything to this file handle using `print`, and it will be sent to the standard input of the subprocess. In listing 11.4, this method is used to send both commands and data to gnuplot. The special file - is useful here, because it instructs gnuplot to read data from standard input (see section 4.5 for more information on pseudofiles). It's important to close the file handle describing the subprocess explicitly when you're done, because Perl won't terminate while gnuplot is still running!

Listing 11.4 Calling gnuplot from Perl (file: driver.pl)

**Opens gnuplot as a subprocess and
assigns it to the file handle PROC**

Writes gnuplot options to the file handle

```perl
open PROC, "| /usr/bin/gnuplot " || die "Could not start gnuplot: $!";

print PROC "set xlabel 'Time'; set ylabel 'Amplitude';"

print PROC "set o 'graph.png';";
print PROC "set t pngcairo;";

print PROC "plot '-' u 1:2 t 'data' w l\n";

for( $x = 0.0; $x  <= 10.0; $x += 0.1 ) {
  $y = exp(-$x/5)*sin( $x );
  print PROC "$x $y\n";
}
print PROC "e\n";

close PROC;
```

Writes the name of the output file …
… and the desired terminal to the file handle

**Instructs gnuplot to expect
data to follow inline**

Sends the data to the file handle

**The data stream is
terminated by the
single letter e.**

**Closes the file handle
and terminates the
gnuplot subprocess**

11.4.2 *Worked example: calling gnuplot from Python*

Listing 11.5 shows a simple Python program that uses gnuplot to create a graph. It
uses the `Popen` facility from the `subprocess` module in the standard library. The con-
structor returns an object that provides a handle to the subprocess. The script sends
commands to the subprocess via the `stdin` member of this handle. In contrast to the
Perl example in the previous section, this script doesn't use the special - filename.
Instead, it creates a heredoc named `$d` and references it in the `plot` command. The
heredoc is particularly useful here, because this example involves *two* columns of data.
The - filename would require looping through the data twice: once for the first col-
umn and once for the second. The heredoc submits all the data to gnuplot in one fell
swoop and then provides access to it as needed.

Listing 11.5 Calling gnuplot from Python (file: driver.py)

```python
import subprocess

data = [ [0, 12, 25],
         [1, 14, 22],
         [2, 15, 23],
         [3, 11, 25] ]

gnuplot = "/usr/bin/gnuplot"

gp = subprocess.Popen( gnuplot, stdin=subprocess.PIPE )

gp.stdin.write( "$d << EOD\n" )
for row in data:
    for col in row:
```

Columns: [x, y1, y2]

```
        gp.stdin.write( "%f\t" % (col,) )
    gp.stdin.write( "\n" )
gp.stdin.write( "EOD\n" )

gp.stdin.write( "set t pngcairo; set o 'graph.png';\n" )
gp.stdin.write( "plot $d u 1:2 w l, $d u 1:3 w l\n" )
```

11.4.3 *Helpful hints*

Using gnuplot in such a way from another program works quite well but can appear a bit fickle at first, because you need to re-create exactly those conditions that are usually fulfilled by input coming from the interactive command-line environment. Diagnosing glitches in this area isn't helped by error messages, which are intended for interactive use. Here's a checklist of trouble spots to look for when things don't work out at first:

- Commands must be separated from one another by *explicit* semicolons or newlines. A common mistake is to write code like this (also see listing 11.4):

```
print PROC "set key";            ⟵——— WRONG: missing newline or semicolon!
print PROC "set border";
```

  The two set commands *appear* to be broken down onto two separate lines, but gnuplot will see them as one consecutive string. This will work:

```
print PROC "set key\n";          ⟵——— RIGHT: explicit newline
print PROC "set border";
```

  Instead of the newline, a semicolon could have been used as well.

- The line containing the plot command *must* be terminated by an explicit newline. Gnuplot doesn't parse a command line until it has encountered a newline. Because the plot command is the last command, it must end with an explicit newline.

- You *must* use an explicit using directive when using the special filename - to tell gnuplot how to parse the incoming data stream.

- For each occurrence of - in the plot command, there *must* be a separate data stream. Gnuplot will continue to interpret incoming data as data until it has encountered a corresponding number of end-of-file characters. (This is another reason heredocs are probably more convenient in this situation.)

- Keep in mind that some characters in gnuplot code may have a special meaning in a programming environment. For example, the $ character, frequently used to indicate the value of a column in gnuplot, is also used by Perl and Unix shells. To prevent interpretation by the programming environment, the character needs to be protected (that is, *escaped*). Alternatively, you can use the column() function, which eliminates the need for the $ character in gnuplot code.

- Don't forget to separate data lines from one another using newlines as well.

One final comment: when generating many graphs from the same program, it's usually a good idea to start gnuplot only once and use a single process instance for all the graphs, rather than starting a separate gnuplot process for each of them. This doesn't

matter much when you're preparing two or three graphs, but when the number of graphs is large, the time savings are significant. Just be sure to reset all relevant options (and specifically the output filename) between invocations of the `plot` command.

Thoughts for the design of a gnuplot access layer

If you find that you need to drive gnuplot from some other program a lot, it may be worth investing in an *access layer*. A quick internet search will turn up several gnuplot *wrappers* for a variety of languages. Most of them attempt to replicate gnuplot's set of commands and options as an API in the host language. This approach is fundamentally mistaken in my opinion, because whenever new features are added to gnuplot or existing ones change, the access layer is no longer up to date.

Instead, I recommend building an access layer in the spirit of the thin database wrappers that exist in all common programming languages for interfacing to relational databases. They don't try to replicate SQL in the host language; they merely provide a way to submit well-formed SQL to the database and to move data back and forth.

I think such an access layer needs to have only three basic functions:

- `open()`—Starts the subprocess, and maintains a handle to it. This function hides all the messy details.
- `exec()`—Takes a string (or an array of strings), and sends them to gnuplot. It's the user's responsibility to make sure the strings contain valid gnuplot commands, but the access layer may apply safeguards against common oversights, such as ensuring that each command is terminated with a semicolon or a newline.
- `data()`—Takes the identifier for a heredoc (such as `"$d"`) and a two-dimensional data structure in the host language. It turns the data structure into a sequence of strings, each corresponding to one line in the data "file," and then sends commands to gnuplot that create a heredoc under the given name, containing the supplied data. Subsequent `plot` commands can then reference this heredoc by its identifier.

The primary idea is to play on the strengths of the host language as much as possible, rather than try to replicate gnuplot's command set. For example, the user should be able to build up a native data structure and manipulate it in the host language—only at the last moment does the access layer turn it into strings that can be passed to gnuplot as a "data file." The heredoc facility comes in handy, because it means data needs to be submitted to gnuplot only once.

These are just considerations that you might find useful if you're designing an access layer for gnuplot. Because every language is different, it doesn't make sense to provide a skeleton implementation here: in particular, what is considered the most suitable "native" data structure is likely to differ so much from one host language to the next that little can be said in general. Also keep in mind that getting the edge cases right will require a disproportionate amount of effort (this is especially true for the serialization of the data structure into strings). A one-off, specialized implementation that takes the specifics of the intended usage into account may be able to skirt some of these issues.

> *(continued)*
> Finally, be aware of the false sense of security that such an access layer instills. A
> proper API lets you detect and recover from error conditions in the encapsulated
> code (by throwing exceptions or returning error codes). This isn't true for gnuplot,
> because gnuplot has no way of notifying you when things go wrong: it just spews
> (human-readable) error messages to standard error. It's difficult to respond program-
> matically to that.

11.5   Animations

All the facilities you've encountered in this chapter support *automation* in some form:
events occurring without human intervention. You can take this a step further and
think about a series of graphs being created, by themselves, in sequence. In other
words, you have all the ingredients for *animation*: moving pictures.

11.5.1   Introducing a delay

To achieve satisfactory animation, you'll need one additional feature: a *controlled delay*
between successive figures. This is where the pause command comes in. The pause
command takes the number of seconds to wait (-1 waits until a carriage return is
encountered or—if mousing is enabled—a mouse click has occurred) and an optional
string argument, which is printed to the command window to prompt the user:

```
pause {int:seconds} [ "{str:message}" ]
```

The pause command is usually used in conjunction with one of the loop constructs.[5]
For a first example, let's go back to the Taylor series example from section 11.1.3 ear-
lier in this chapter. Using pause, you can create an *animated* version of this example:

```
f(x,n) = sum[k=1:n] (-1)**(k-1)*x**(2*k-1)/(2*k-1)!

do for[n=1:8] {
  plot [-8:8][-1.2:1.2] sin(x) lw 2, f(x,n) t sprintf("%d Terms",n)
  pause 2
}
```

This plots a new graph every two seconds, each one representing a better approxima-
tion than the one before. (Try it yourself—there's no point showing a static graph of
an animated effect.)

---

[5] In previous gnuplot versions, the same effect was achieved using the reread command. When included in a
command file, reread instructs gnuplot to begin executing the current file from the beginning, resulting in
an infinite loop.

11.5.2 *Waiting for a user event*

A second form of the `pause` command waits until a specific user event has occurred:

```
pause mouse [ {eventmask} ] [ "{str:message}" ]
```

The event mask can contain any combination of the following keywords, separated by commas: `keypress`, `button1`, `button2`, `button3`, `close` (meaning the plot window was closed), and `any`. If `pause` was terminated through a keyboard event, the ASCII value of the selected key is stored in the gnuplot variable `MOUSE_KEY` and the corresponding character in the variable `MOUSE_CHAR`. If a mouse event occurred, the mouse coordinates are stored in the variables `MOUSE_X`, `MOUSE_Y` or `MOUSE_X2`, `MOUSE_Y2`, respectively, and are available for further processing. The command `pause mouse` (without an event mask) is equivalent to `pause -1`.

11.5.3 *Further examples*

Some of the most striking applications of animations involve three-dimensional graphics. We'll discuss these in detail in appendix C and appendix F, but I want to give two quick demos here (see listings 11.6 and 11.7).[6]

The logic of both listings is straightforward: a parameter-dependent graph is continuously re-created in a `while` loop. Because the parameter is slowly changed at the same time, no two successive graphs are exactly equal, giving the appearance of a smooth change.

It's essential to include a delay in the loop (using `pause`) and to provide an explicit break condition for the loop—gnuplot doesn't recover gracefully if the loop is stopped using Ctrl-C. You can stop the animations in the example scripts by pressing the X key on your keyboard (we'll formally introduce the `bind` command in chapter 12).

> **TIP** Always include an explicit break condition. Gnuplot doesn't recover if the loop isn't stopped cleanly.

Listing 11.6 Animation demo (file: pebble.gp)

```
unset key; unset border; unset tics

bind "x" "end=1"

end = 0; t = 0
while( end==0 ) {
  t = t + 0.1
  splot [][][-1:1] exp(-0.2*sqrt(x**2+y**2))*cos(sqrt(x**2+y**2) - t)
  pause 0.001
}
```

---

[6] The file world_110m.txt required by the second listing can be found at www.gnuplotting.org/plotting-the-world-revisited/.

For another stunning example of what is possible using gnuplot animations, check out the demo/games directory of the gnuplot distribution. There you'll find, among other things, a rather addictive implementation of a Tetris-like game—entirely in gnuplot!

Listing 11.7   Animation demo (file: globe.gp)

```
unset key; unset border; unset tics
set hidden3d

set mapping spherical; set angles degrees
set parametric; set urange [-90:90]; set vrange [0:360]

bind "x" "end=1"

end = 0; t = 0
while( end==0 ) {
  t = (t + 1)%360

  set view 90-30*cos(t), 360-t, 1.5, 1.25

  splot cos(u)*cos(v),-cos(u)*sin(v),sin(u) w l lc rgb "grey", \
        "world_110m.txt" w l lt 1

  pause 0.01
}
```

11.6   *Case study: continuously monitoring a live data stream*

Monitoring a live stream[7] means *animated* figures that are continuously changing as they're being updated with newly arriving data. This isn't a situation that gnuplot was designed for, but—as it turns out—gnuplot can handle it quite gracefully.

In this section, I'll present several different approaches to this problem. We'll start with a very basic, ad hoc solution; the final solution is more involved and can be regarded as the beginnings of a production-level, real-time monitoring system that uses gnuplot as its graphics back end. Some of the initial ideas use Unix tools for simplicity, but the final solution is fully general and can, in principle, be adapted to any platform.

11.6.1   *Using gnuplot to monitor a file*

First, let's assume that you would like to monitor the tail end of a log file. Messages are appended to the log file asynchronously by some other process (or processes). For demonstration purposes, let's assume that each log entry consists of a timestamp followed by the actual value. The goal is to have a constantly updated plot of the last few entries in the log file.

The minimal gnuplot script to achieve this goal is just a few lines long:

```
while( 1 ) {
  plot "< tail -200 logfile" u 0:2 w lp
  pause 0.1
}
```

---

[7] This section was partially inspired by the blog post "Visualize real-time data streams with Gnuplot" by Thanassis Tsiodras. You can find it at http://mng.bz/N3kJ.

The script consists of an infinite loop, which is executed 10 times per second (`pause 0.1`). The `plot` command uses the Unix `tail` facility to extract the last 200 lines from the log file. The output from `tail` is piped directly to gnuplot, without accessing the file system. The pseudocolumn `0` (that is, the line number in the actual data set) is used for the x coordinate. (See chapter 4 for information on pseudofiles and pseudocolumns.)

This works quite well, although the result isn't very pretty (we'll remedy this next). But as a minimal-effort solution, this script can't be beat. It should be committed to memory for all situations where you have an ad hoc need for real-time monitoring of data.

A DATA GENERATOR FOR TESTING

If you want to try the solutions described in this section and you don't have access to a live data stream, you may find the script in listing 11.8 helpful. It generates data at random intervals; you must specify the average time between write events as a command-line parameter. So, to have the script generate about 10 events per second, you call it like this (depending on your needs, you may want to redirect the generator's output to a file):

```
shell> python -u datagen.py 0.1
```

The `-u` option forces Python to use unbuffered I/O. This is important in this application, because you're interested in seeing real-time data. Don't forget. The generated "data" here is a sine wave. The script writes out about 20 points per period of the sine, regardless of the specified wait time between write events.

Listing 11.8   A random data generator (file: datagen.py)

```
import sys
import time
import math
import random

tmax = float( sys.argv[1] )

t = 0
while 1:
    dt = random.uniform( 0, 2.0*tmax )
    time.sleep(dt)
    t += dt

    print t, math.sin(0.1*math.pi*t/tmax)
```

A BETTER MONITOR

Let's improve the monitoring script given earlier. The data does contain timestamps, but so far, they're ignored. The graph therefore gives the impression that log entries are equally spaced in time—which isn't true.

But if you try using the timestamp as the x coordinate, the visual effect is unsatisfactory. The reason has to do with the way gnuplot handles the plot range: unless instructed otherwise, it makes sure all data points are included, *and* it extends the plot

range to the next tic mark. But because data points occur asynchronously (at random times), the range of seconds spanned by the points in the sample changes as new points are added and old points are removed. This constant change in the plot range leads to the undesirable "flicker" effect. The following script uses a fixed plot range to avoid this problem.

Listing 11.9   A real-time data monitor (file: monitor.gp)

```
logname = "logfile"          ⊲────── Name of the log file to monitor
tmplen  = 200                ⊲────── Number of records to save to a temporary file
tmpname = "tmp"              ⊲────── Name of the temporary file

bind "x" "end=1"
end = 0
while( end==0 ) {
  system sprintf( "tail -%d %s > %s", tmplen, logname, tmpname )
  stats [*:*][*:*] "tmp" u 1:2 nooutput
  set xrange [STATS_max_x-11.0:STATS_max_x+1]
  plot "tmp" u 1:2 w lp
  pause .01
}
```

The first few lines of the script are used to configure the script to local conditions: you must supply the name of the log file to monitor, as well as the name of a temporary file and the number of records that will be saved to it. (We'll come back to the latter parameter later.) The script then invokes the Unix `tail` utility via gnuplot's `system` command to dump the configured number of data points from the end of the log file into a temporary file. The `stats` command is used to determine the value of the last (most recent) timestamp in this temporary file. The `set xrange` command then chooses a *fixed* plot range that is based on this value: the plot range extends 1 second into the future and 11 seconds into the past, measured from the last entry in the temporary file. (That the upper limit of the plot range isn't made to coincide with the largest data point is sheerly for visual effect. Try it both ways: personally, I find it helpful to see explicitly where the available data ends.) The range indicators on the `stats` command are important: without them, `stats` would discard data outside the plot range and hence fail to reflect changes to the input file.

The result works quite well in practice. (Try it!) Here are a few additional ideas you may want to explore:

- The script contains three adjustable parameters: the wait time in the `pause` command, the number of lines that `tail` extracts, and the size of the plot range in seconds. All three should depend on the frequency with which entries are added to the log file. Begin by deciding how many previous seconds you want to see in the graph. If you know at what rate new lines are appended to the log file, you can figure out how many records you need to extract. The wait time in `pause` should be smaller (by a factor of 2–10) than the time between log-file updates to ensure smooth animation, but keep in mind that reducing this number drives up resource consumption.

- Listing 11.9 uses the Unix `tail` command to extract a certain number of records from the log file to a temporary file, which is then read by both `stats` and `plot`. Both the `tail` command and the temporary file are optional: in principle, you could rely entirely on the plot range to restrict the data and run both `stats` and `plot` directly on the log file. But if the log file is long (and what log file is ever short?), then the `tail` command is likely to be faster and less resource-intensive than gnuplot's own (and very complex) data-file parser (which is invoked by both `stats` and `plot`). Furthermore, by creating a temporary file, it's guaranteed that both `stats` and `plot` see exactly the same data: if you let both commands run on the log file in sequence, you introduce a possible race condition wherein the log file is being updated between the two commands. You decide whether any of these precautions are necessary or desirable in your situation!

11.6.2 *Using a driver to monitor arbitrary data sources*

The solution in the previous section is nice because it's implemented entirely using gnuplot's own facilities. But it also has some drawbacks:

- It's a *polling* solution. Gnuplot polls its input source constantly, whether there is new data or not. As a rule, polling architectures are more resource-intensive than other solutions.
- It's *not flexible*, because any processing of the data would have to be accomplished using gnuplot's limited scripting capabilities.

The approach presented in this section is different, because it relies on an intermediate layer, written in a general-purpose programming language (Python, in this case, but it could, of course, be anything). The script in listing 11.10 acts as a *filter*: it gathers data from standard input and passes it on to gnuplot *only* if new data is available. It therefore implements a "push" model, rather than the "pull" model of the previous solutions. The script also acts as a *buffer*, storing the most recent set of points and passing the points to gnuplot. This is necessary because now the file system is skipped entirely, and gnuplot itself doesn't retain data between plots.

Listing 11.10 A data monitor in Python (file: monitor.py)

```
import sys
import subprocess

buflen = 200                          ⟵——— Number of data points to buffer

gnuplot = "/usr/bin/gnuplot"

gp = subprocess.Popen( gnuplot, stdin=subprocess.PIPE )
gp.stdin.write( "set t wxt\n" )

buf = []
while 1:
    line = sys.stdin.readline()       ⟵——— Reads unbuffered from standard input
```

```
t, u = line.strip().split()
buf.append( (float(t), float(u)) )

buf = buf[-buflen:]

mx = buf[-1][0]
gp.stdin.write( "set xrange [%f:%f]\n" % ( mx-10, mx+1 ) )

gp.stdin.write( "plot '-' u 1:2 w lp\n" )
for d in buf:
    gp.stdin.write( "%f %f\n" % d )
gp.stdin.write( "e\n" )
```

Drops the oldest data points, retaining at most buflen points

Most recent timestamp

The script runs gnuplot as a subprocess and communicates with gnuplot by writing to gnuplot's standard input via a pipe. The script reads incoming data from *its own standard input* (not from a file), splits and parses the intake, and pushes it onto a buffer, before passing it on to gnuplot. In this case, there is no need for gnuplot's stats command, because the determination of the plot range is now done in the Python script.

The script is straightforward. Only two Python-specific implementation details require comment:

- To achieve real-time behavior, all input and output must be unbuffered. This is the reason the script uses sys.stdin.readline() in a while loop, instead of the more familiar idiom for line in sys.stdin:. Also, make sure to run this script using Python's -u command-line option.
- The line buf = buf[-buflen:] retains the last buflen elements in the array. If the array contains fewer than buflen elements, then all of them are retained.

It should be clear that listing 11.10 is only the beginning. Not only are all features missing that would make the program more robust (error-handling, and so on), but the function of the script could be extended in many ways:

- Observe and combine multiple input sources. The script in listing 11.10 listens to its own standard input—instead, it could keep tabs on a range of different log files, combine data from all of them, and generate a unified status plot.
- Conversely, split a single, heterogeneous input source into multiple data streams, and maintain several gnuplot processes, each of which is used to plot only a single output channel.
- Statistically manipulate the data, remove outliers, and identify trends. The results of a predictive model could be combined with the raw output so that both can be plotted together and compared easily.
- Make any other modifications you can think of …

The options are endless but have little to do with our primary topic, which is gnuplot. Listing 11.10 is only intended to get you started.

11.7 *Summary*

In this chapter, you learned about ways to *automate* gnuplot tasks. Gnuplot provides some programming constructs, such as loops and conditionals, but for more complicated tasks, it may be more convenient to "drive" gnuplot from an external programming language as a subprocess.

Another way to automate tasks is to capture all the required gnuplot commands in scripts that can be run by themselves, as batch processes, or even as part of Unix pipelines. In chapter 12, you'll learn about additional uses for script files.

Gnuplot's automation capabilities enable the creation of *animated* graphics. We demonstrated the basic principles and then used them in a particularly interesting application: the real-time monitoring of a live data stream.

Beyond the defaults:
workflow and styles

12

This chapter covers

- The standard workflow and some variations
- Using external editors and viewers
- Hotkeys and mousing
- Configuring your workspace
- Defining and using stylesheets

Overall, gnuplot's defaults are pretty good—so good, in fact, that it's easy to forget that they aren't set in stone, and that there may be good reasons for changing them. This applies not only to the visual appearance of plots, but also to the way you interact with gnuplot.

In this chapter, we'll explore some ways to customize your gnuplot experience. First, we'll consider workflow alternatives: ways to organize your interaction with gnuplot beyond the regular, interactive session. We'll discuss various shortcuts and conveniences, along with the possibility of using an external editor for authoring gnuplot command files rather than typing in commands interactively at the gnuplot prompt. Next come various topics that are more typically associated with user

262

customization: custom key (and mouse) bindings, and the definition of private option settings and configurations.

Finally, I'll introduce the idea of *stylesheets*: collections of appearance options for lines and points that can be applied to graphs to give them a different appearance. In particular, defining and choosing stylesheets judiciously makes it possible to create graphs with a specific purpose in mind, be it a computer screen, a black-and-white publication, or an overhead presentation.

12.1 The standard interactive workflow

Gnuplot was originally conceived as an interactive tool, and this is still the most natural way to use it: you start an interactive session, enter commands at the gnuplot prompt, examine the resulting graph, and iterate. Frequently, such an interactive session results in a saved command file that can be used to re-create the final state, as well as a final graph, exported in some graphics file format. This section reviews some ideas for improving this straightforward workflow.

12.1.1 Extracting specifics from command files

I've pointed out several times before that it isn't enough to just export the final graph as a PNG or PDF—it's crucial to also save the final set of options (using `save`). Only then can you re-create the graph in gnuplot to examine it again, to modify an aspect, or to update it with new data.

The files that the `save` command generates are long: they contain the values of *all* of gnuplot's options,[1] as well as all user-defined functions and variables. Most of this information is redundant: options are written to file even if they still have their default values. The advantage is, of course, that the resulting file is guaranteed to be self-contained. But the information that is *specific* to the current graph is hard to discern.

You can recover the plot-specific information by comparing the saved file against a file that contains *only* default values. This *baseline* file can be obtained by starting gnuplot and—without doing anything else—issuing a `save` command.

Because the file format of gnuplot's command files is so simple (line-oriented text files, with one command or option per line), it's easy to write a short program in your language of choice to compare the two files.[2] Wherever the GNU `diff` utility is available, it can be used to good effect as well (the cryptic command-line options instruct `diff` to print only lines that are new or different in the second file):

```
shell> diff --new-line-format="%L" --old-line-format=""
➥          --unchanged-line-format="" default.gp graph.gp
```

The resulting file is short and neat, but it's less self-contained than the original file was, because it doesn't contain a complete set of options. If the shortened file is loaded into

---

[1] With three exceptions: the `set output`, `set terminal`, and `set linetype` settings.
[2] For example, you can find one such program among the contributed scripts on the gnuplot homepage at www.gnuplot.info.

a gnuplot session that's configured differently from the baseline file, the resulting graph will be different than the original. Also, if gnuplot's defaults change, the graph created by the shortened file will be affected. This is something to be aware of.

12.1.2 *Extending the command set*

Sometimes it takes several gnuplot commands to achieve a common task (exporting to file, for instance), sometimes the syntax is awkward and hard to remember (setting up a palette for false-color plots), and sometimes both (time-series scaling for the x axis). It would be nice if we could define our own commands to take the drudgery out of it. There are basically two mechanisms available:

- Build up a string of commands, and execute it using `eval` (see section 5.2.2).
- Capture commands in a command file, and run it using either `load` or `call` (see section 11.2.1).

These methods are almost equivalent—the main difference between them lies in the way they're called. Let's look at some examples. (Either method can also be bound to a *hotkey*—see section 12.4.3 later in this chapter.)

USING SCRIPTLETS

It's important to remember that command files don't need to be complete: a command file that only defines some functions or sets some options is perfectly legitimate and often useful. Listings 12.1 and 12.2 show two examples that do just that: listing 12.1 defines two distributions that I frequently need in my work; and listing 12.2 switches on time-series mode, using a specific timestamp format. The real benefit of the latter example is that I don't need to remember the conversion specifiers! (I can never remember whether it's minutes or months that go with an uppercase *M* as opposed to a lowercase one.)

You can also collect style definitions and appearance options (line widths and colors) together to form *stylesheets*. This idea is so important that it's given its own section (section 12.6) later in this chapter.

Gnuplot looks for scripts invoked with `load` or `call` first in the current working directory and then in the directories listed in the `loadpath` option.[3] The `loadpath` option can be set using `set loadpath`; if the GNUPLOT_LIB environment variable is set, its contents are appended to `loadpath` (see section 12.5.1). It's therefore possible to create a library of useful scripts in a personal collection.

Listing 12.1   Scriptlet defining common functions (file: normdist.gp)

```
sqrt2pi = sqrt(2.0*pi)

gauss(x,m,s) = exp(-0.5*((x-m)/s)**2)/(sqrt2pi*s)
lognorm(x,m,s) = exp(-0.5*((log(x)-m)/s)**2)/(sqrt2pi*x*s)
```

---

[3]   Gnuplot uses `loadpath` for scripts and also for data: it searches first in the current directory and then in the directories listed in `loadpath` for data files named in calls to `plot` or `splot`.

```
set xdata time
set xtics format "%T"
set timefmt "%H:%M:%S"
```

USING EVAL

Strings containing valid gnuplot commands can be executed using `eval`. Furthermore, you can build up strings dynamically for a specific purpose or—even better—write functions to do this for you.

Here's a simple example. Switching to polar mode usually requires several commands (see section F.2.1). Combining them all into one string, you can achieve the same effect with a single call:

```
polar = "set polar; set size sq; unset border; unset tics; set rtics; \
set grid polar; set key outside"
```

Now, `eval polar` will not only turn on polar mode, but also set up suitable appearance options.

Switching on time-series mode can be accomplished in a similar fashion. If the following strings exist in the current gnuplot session

```
times = "set xdata time; set xtics format '%T'; set timefmt '%H:%M:%S'"
dates = "set xdata time; set xtics format '%F'; set timefmt '%Y-%m-%d'"
```

then time-series mode can be turned on by using `eval times` or `eval dates`. (Of course, the format of the timestamps in the input file must match the format specified in these strings.)

Finally, and as an example of a *function* that generates a string dynamically, consider again the `export` function introduced in section 5.2.2 (see listing 12.3). It generates a string that—when executed using `eval`—exports a graph, using a user-supplied filename and terminal. The function can be called directly from within the `eval` command:

```
eval export( "graph.pdf", "pdfcairo" )
```

Listing 12.3   A function that generates a command string dynamically (file: export.gp)

```
export( file, terminal ) = sprintf(
➧       "set t push; set t %s; set o '%s'; replot; set o; set t pop",
➧       terminal, file )
```

ADDITIONAL CONSIDERATIONS

On startup, gnuplot reads an initialization file (see section 12.5.1), and it may seem natural to place as many definitions as possible in this file so they're always available. Over time, I've moved away from this approach and instead suggest that you use many

smaller files, each containing only closely related definitions that can be loaded as needed; this way, you avoid polluting the gnuplot session with all kinds of rarely used functionality.

This highlights one of the main differences between scriptlets and dynamically evaluated strings. The strings (or functions that generate them given a set of arguments) must exist in the gnuplot session, whereas scriptlets don't. Furthermore, anything that lives in the gnuplot session may be overwritten at any time and without warning; scriptlets, by contrast, offer a degree of *encapsulation*.

Finally, you can *bind* either scriptlets or strings to be evaluated with eval to keyboard events. In that case, pressing the appropriate key invokes the desired action. (We'll discuss this later in this chapter, in section 12.4.3.)

12.1.3 *Session variables, loops, and macros*

Don't forget that it's often possible to streamline your workflow using the techniques introduced in chapter 5: session variables, string macros, and inline loops. Inline loops don't need any further explanation (you've seen plenty of examples), but here are some ideas for using session variables and macros.

USING VARIABLES

Imagine that you want to work with a data set that's in a file with an inconveniently long filename (possibly not even in the current directory). Assigning the name to a variable helps to keep command lines much shorter:

```
FILE = "../path/to/directory/long-and-painful-filename.dat"
stats FILE u 2 nooutput
plot FILE u 1:($2/STATS_mean) w lp
```

Another possibility is to assign just the base name of the file to a variable, and add extensions on the fly:

```
BASE = "long-and-painful-filename"

plot BASE . ".dat" u 1:2

save BASE . ".gp"

set terminal pdfcairo
set output BASE . ".pdf"
replot
```

You get the idea.

USING MACROS

Macros are another way to shorten command lines (see section 5.2). Imagine that you want to plot several files using the with xyerrorbars style. That style requires *six* columns to be named in the using clause. A macro may come in handy:

```
cols = "using 1:2:3:4:5:6"
plot "data1" @cols w xyerrorbars, "data2" @cols w xyerrobars
```

In fact, you could make with xyerrorbars part of the macro! Macros can also help straighten out complicated inline transformations:

```
trsf = "using ($1-$2/$3):(sqrt((abs($4-$5)/$6)))"
plot "data1" @trsf, "data2" @trsf
```

A third idea is to use macros to shorten appearance options, in particular colors (see section 9.1). Compare this command

```
plot "data" u 1:2 lc rgb "red", "" u 1:3 lc rgb "green",
➡   "" u 1:4 lc rgb "blue"
```

with this:

```
r = "lc rgb 'red'"
g = "lc rgb 'green'"
b = "lc rgb 'blue'"

plot "data" u 1:2 @r, "" u 1:3 @g, "" u 1:4 @b
```

Defining line styles is an alternative, of course.

12.2 *Using external editors and viewers*

In everything so far, I've assumed that you're using gnuplot *interactively*: that you started gnuplot and are typing commands at gnuplot's own command prompt and that you view the generated plots in one of gnuplot's interactive terminals (such as the wxt or qt terminal). But it turns out there are other ways to interact with gnuplot, even for interactive, non-batch work. (Batch-oriented operations are treated in chapter 11.)

Rather than entering commands at gnuplot's prompt, you may find it convenient to use your favorite text editor to type commands into a command file and then let gnuplot run this file as a script. One way to do this is to run an interactive gnuplot session in another window.[4] You then load the script using load and examine the graph in one of the interactive terminals.

Alternatively, you may choose to work with one of the file-based terminals and export graphs in PNG or PDF format directly. If you do this, you need to specify the output terminal and filename in your script before the plot command:

```
set terminal pdfcairo
set output "graph.pdf"
plot sin(x)
```

Now you don't even need to start an interactive session. Instead, you run gnuplot at the shell prompt, giving the name of your script as a command-line argument to gnuplot

```
shell> gnuplot script.gp
```

---

[4] Depending on your text editor, you may not even need a separate window. If you use Emacs, for instance, an internet search will turn up not one, but *two* gnuplot modes that let you run gnuplot from within your Emacs window.

and view the resulting graph in a separate viewer (such as okular or evince). Modern viewers usually detect when the file they're displaying changes on disk; they automatically reload and redisplay the graph after you change your script and run gnuplot again, leading to a comfortable workflow experience.

Alternatively, you can load the command file into a gnuplot session using `load`, but in this case your script must contain the command `set output` *after* the `plot` command to ensure that all output buffers are flushed. Running gnuplot from the shell prompt avoids this need: the gnuplot process terminates after completion of the script, and that forces all open files to be flushed.

Using an external editor (and, possibly, an external viewer) has some advantages compared to running gnuplot in an interactive session:

- All commands and option values are simultaneously visible in the command file.
- Editing may be more convenient in a proper editor.
- You can never forget to save your commands (not just your graph).
- For file-based formats, there can be no discrepancy between the appearance of the graph during the interactive session and the final graph saved to disk. (This can be a problem when you prepare a graph using an interactive terminal but then export the final version as PNG or PDF.)
- Because multiplot mode (see section E.1) doesn't permit you to change the `output` and `terminal` settings, the command-file-based approach is the only practical way to create a multiplot graph.

The primary disadvantage of using an external editor is the need to run a separate window (which can be mitigated if a gnuplot plug-in is available for your editor). When using file-based output formats and external viewers, you lose the ability to interact with the graph using the mouse—which often doesn't matter but can be painful at times.

> **TIP** Consider using an external editor and viewer. You might like it!

12.3   *Invoking shell commands from gnuplot*

Gnuplot can't do everything alone. It's occasionally useful to invoke other programs within a gnuplot session. There are four ways to do this:

- The `system` command executes a command in a subshell, directing its output to the interactive gnuplot terminal. For example, you can list the contents of the current directory with `system "ls"`.
- The `system()` function is similar to the `system` command. It takes a command as argument, executes it as a subprocess, and returns its output as a string (in contrast to the `system` command, which always directs any output from the subprocess to the interactive terminal).

- Backticks or back quotes (decimal ASCII code 96—typically bound to the left-most key in the top row on the keyboard) may be familiar from Perl and some Unix shells. The command in back quotes is executed, and its output replaces the back-quoted text (including the back quotes). Back-quote substitution can be used anywhere except inside single-quoted strings.
- The `shell` command suspends the current gnuplot session and starts an interactive shell. Quitting the shell (typically by typing `exit` or pressing Ctrl-D) resumes the gnuplot session. (The `shell` command is somewhat of a legacy from days before windowing systems—today, we're more likely to pop up a shell in another terminal window, rather than suspend the process in the current window.)

In general, the `system()` function is more versatile than backticks and is the preferred way to capture output from a subprocess. But backticks have the advantage that they can occur inside double-quoted strings, whereas the output from the `system()` function must be concatenated explicitly to the string. In other words, the following two commands are equivalent:

```
set label 1 "Today: `date`" at 0,0
set label 2 "Today: " . system( "date" ) at 0,1
```

For simple string substitutions such as these, backticks can be a convenient shorthand. For more complicated operations, the `system()` function is the better choice.

12.3.1 *Worked example: plotting each file in a directory*

Shell commands can be combined with loops. To plot every file in the current directory, either of the following two commands will work:

```
files = system( "ls" )                    ⟵——— Uses the system() function
plot for[f in files] f u 1:2

plot for[f in "`ls`"] f u 1:2             ⟵——— Uses backticks
```

12.4 *Hotkeys and mousing*

When you issue a `plot` command, gnuplot pops up a new window containing the plot (assuming, of course, that you're working with one of the interactive terminals and not currently exporting graphs to file). The new window containing the graph is automatically *raised* or active.

When a graphics window is active like this, you can invoke gnuplot commands through keyboard shortcuts or mouse clicks. In this section, we look at some of the default bindings provided by gnuplot, and also see how to define custom keyboard or mouse shortcuts.

12.4.1 *Default hotkeys*

Pressing the space bar when the plot window is active raises the gnuplot command window. This is convenient when you're doing interactive work: you can iteratively continue working on a single graph without ever having to take your hands off the keyboard!

Grid lines can be toggled on or off through the hotkey g. Logarithmic plotting is bound to l (y axis only) and L (in this case, gnuplot scales only the axis closest to the mouse pointer logarithmically). The + and - keys can be used to zoom in and out of the plot. Pressing q closes the current plot window.

These and some other interesting default bindings are listed in table 12.1. You can get a display of *all* currently defined key bindings by pressing h while the plot window is active.

I don't find most of the default bindings all that useful (and some are outright annoying), but custom hotkeys (see section 12.4.3) can be very useful indeed.

Table 12.1   Selected default hotkey bindings

Key	Function
Space bar	Raises the command window and switches keyboard focus to it
q	Closes the current plot window
g	Toggles grid lines on the plot
r	Toggles the crosshair (ruler) at the current mouse position
l (lowercase letter *L*)	Toggles logarithmic scaling for the y axis
L	Toggles logarithmic scaling for the axis closest to the mouse pointer
+	Zooms in
-	Zooms out
u	Unzooms (after zooming using the mouse)
h	Help: shows all key bindings
Arrow keys	Scrolls the graph in the indicated direction

12.4.2 *Mousing*

In interactive terminals, you can use the mouse to navigate the graph. By default, mouse actions should be enabled, but in case they aren't, you can enable them using the command

```
set mouse
```

When the mouse is active, the current coordinates are always displayed at the bottom of the plot window. You can place a temporary crosshair at the current mouse position

by pressing the r key. Pressing r again switches off the crosshair. While the crosshair is active, its location and the relative distance of the mouse pointer from the crosshair coordinate are shown in the plot window, together with the absolute coordinates of the mouse pointer.

You can use the mouse to zoom in on a graph by dragging the mouse while holding mouse button 3 (often the right mouse button). Click into the graph with mouse button 3, drag while holding the button down, and then click with mouse button 1 to replot only the section of the graph in the indicated region. Press u (unzoom) to return to the previous setting. When you use the mouse to set the plot range in this way, *both* the x and the y range are fixed according to the mouse coordinates (in particular, the y range is no longer autoscaled).

There are other ways to use the mouse in current interactive terminals. For example, clicking a key entry toggles the corresponding curve in the graph on or off.[5] The set mouse command takes some additional options that mostly control the display format of the mouse coordinates in the plot window. Check the standard gnuplot reference for details.

12.4.3 Custom hotkeys

The standard hotkey bindings are nothing to write home about, and neither is the default mouse behavior. But you can also define your own hotkey and mouse bindings, and that allows for some exciting possibilities.

The bind command is the essential facility to do this. It's used to bind gnuplot commands to key strokes or mouse clicks. Here's a trivial example:

```
bind "r" "replot"
```

This binds the r key to the replot command. Now, pressing the r key (while the plot window has focus) will redraw the most recent plot.

THE BIND COMMAND IN DETAIL

The bind command takes two string arguments: the key (possibly in combination with one or several of the control keys) and the command to be executed (when invoked without arguments, bind shows all currently defined bindings):

```
bind                                          ⟵——— Shows current bindings
bind [allwindows] "{str:keys}" "{str:command}"   ⟵——— Defines a new binding
```

You can bind commands to special keys—see table 12.2 for the symbolic names of some function keys you may want to know, and see the standard gnuplot reference documentation for the bind command for the complete list.

---

[5] This feature is new with gnuplot 5.

Table 12.2 Symbolic names of some of the function keys available

Location	Name
Control keys	`"ctrl-"`, `"alt-"`
Main key block	`"Tab"`, `"Escape"`, `"Return"`
Arrow keys	`"Left"`, `"Up"`, `"Down"`, `"Right"`
System key block	`"Home"`, `"End"`, `"PageUp"`, `"PageDown"`
Function keys	`"F1"`, …, `"F12"`
Window events	`"Button1"`, `"Close"`

Usually, the commands bound to the hotkeys are executed only *if the plot window has keyboard focus*; hotkeys don't apply when the command window has focus. This behavior can be overridden by the optional `allwindows` option to `bind`: if it has been given, then the binding applies to all plot windows, active or not. In this case, the variable `MOUSE_KEY_WINDOW` is set to the ID of the window generating the event.

Here are some additional points to watch out for:

- Neither the space bar nor the q key can be rebound from its default settings.
- Only mouse button 1 can be bound.
- Running `bind` without arguments displays a list of current bindings. So does `show bind` or pressing the h key while the plot window has focus.
- The command `reset bind` returns all key bindings to their defaults.

WORKED EXAMPLES

Some of the custom additions to the standard command set that were discussed in section 12.1.2 can be bound to keys. Assuming that the string variables `polar`, `times`, and `dates` exist in the gnuplot session and have the values shown earlier, you can bind them to keys, like so:

```
bind "F1" polar
bind "F2" times
bind "F3" dates
bind "F4" "reset session"
```

For convenience, I've bound these commands to function keys here; you can bind them to any other key as well. The binding for F4 resets gnuplot to its state at the beginning of the session, presumably resetting the configuration changes brought about by any of the other keys.

You can also bind the loading of a scriptlet to a key. In this case, the effective command is `load`:

```
bind "F5" "load 'normdist.gp'"
```

One word of warning: if you use string variables in `bind` (as in the bindings for F1, F2, and F3 in the earlier example), then the variables must contain the string to evaluate *at the time the* `bind` *command is executed* (early binding). You can't bind a key to a variable first and then dynamically change the value of that variable later.

12.4.4 *Capturing mouse events*

Whenever a mouse or keyboard event occurs while the graph window has focus (and provided mousing is enabled using `set mouse`), gnuplot populates several variables with information regarding the details of the most recent events. The names of these variables all begin with the prefix `MOUSE_`. See table 12.3 for a complete list.

You can use the information contained in these variables, in combination with the `bind` command, to write gnuplot code that responds to user actions. In this way, it's possible to achieve a limited form of programmable interactivity.

Table 12.3 These variables are set whenever a mouse or keyboard event occurs when a plot window is active, provided that mousing has been enabled using `set mouse`.

Variable name	Description
MOUSE_X	x coordinate at the time of the mouse event (measured in the `first` coordinate system)
MOUSE_Y	y coordinate
MOUSE_X2	x coordinate (measured in the `second` coordinate system)
MOUSE_Y2	y coordinate
MOUSE_BUTTON	ID of the mouse button clicked (1, 2, or 3)
MOUSE_SHIFT	Nonzero if the Shift key was pressed when the event occurred
MOUSE_CTRL	Nonzero if the Ctrl key was pressed when the event occurred
MOUSE_ALT	Nonzero if the Alt (Meta) key was pressed when the event occurred
MOUSE_KEY	ASCII code of the key that was pressed
MOUSE_CHAR	Character value of the key that was pressed
MOUSE_KEY_WINDOW	ID of the plot window that received the event

WORKED EXAMPLE

Here's an intentionally simple example, mostly to demonstrate the principle. I'll present a much more interesting and useful application in the following section.

If you execute the following command

```
bind "Button1" "set label at MOUSE_X, MOUSE_Y center point pt 1 ps 4; \
set label at MOUSE_X, MOUSE_Y center point pt 6 ps 3; replot"
```

and then click into the plot window, a point symbol is placed at the current mouse position—for example, to draw attention to an oddity in the data. To make things a little

more interesting, it's a symbol that gnuplot doesn't provide as a built-in: a crosshair, here generated from a separately drawn cross and circle. To remove the symbols, you need to find their tags first using `show label` and then remove them individually with `unset label`, followed by the tag.

This code snippet is a straightforward application of the `bind` command. Let me draw your attention to the `replot` command at the end of the command string: this is common practice when using `bind` and ensures that the result of the action is visible immediately. You'll see more examples of this technique in the next section.

12.4.5 *Case study: placing arrows and labels with the mouse*

Truth be told, I don't find most key bindings terribly useful: the whole purpose of gnuplot is graphs, and I want to interact with them using the mouse, not the keyboard (for a stunning exception to this rule, see section D.7). Fortunately, the `bind` command can be used to create some special mouse bindings as well.

Wouldn't it be nice if you could place arrows and labels onto the graph with the mouse: just point and click? Turns out you can. Here are some ideas.

LABELS

The problem with placing labels is that there isn't a way to prompt the user for the text of the label. Hence, you must rely on the user to prepopulate an agreed-on variable, or you must rely on the user to call an appropriate function later.

The following snippet follows the former approach:

```
bind "ctrl-Button1" 'eval sprintf( "set label %d \"%s\" at %f,%f; replot", \
labeltag, labeltext, MOUSE_X, MOUSE_Y )'
```

It assumes that the variables `labeltext` and `labeltag` have been populated appropriately. Then, clicking into the plot while pressing the Ctrl key will place the label at the current position of the mouse pointer.

Two notes on implementation: in the second argument to `bind`, you must escape the innermost quotation marks. Also note the `replot` command, which ensures that the plot automatically updates when the mouse button is clicked. This is common practice when using `bind`.

The next code snippet takes the other approach. You click first, and then you supply the text (note that the second string supplied to the `bind` command is continued on the next line):

```
bind "ctrl-Button1" "label_x=MOUSE_X; label_y=MOUSE_Y; \
set label 1000 at label_x,label_y point; replot"

label( tag, text ) =
➥    sprintf( "set label %d '%s' at %f,%f; unset label 1000; replot",
➥            tag, text, label_x, label_y )
```

Now, if you Ctrl-click anywhere in the plot window, a point symbol is placed at the position of the mouse cursor, to remind you to supply the label text. You do the latter

using the `label()` function, which removes the point from the graph and replaces it with the supplied text. Don't forget to use `eval` with this function:

```
eval label( 2, "Hello, World!" )
```

ARROWS

Arrows are trickier, because you need to capture *two* positions. The problem is that the only accessible mouse event is the click: this means you can't use drag-and-drop as in a drawing program. Instead, you have to capture two clicks separately. This leads to a problem: what if you change your mind between the first click and the second?

Listing 12.4 shows the solution. You bind both the mouse click and the Esc key. When the mouse key is clicked once, a temporary marker is placed on the plot; when a second click occurs, the marker is removed and an arrow is drawn. If you change your mind after the first click, you can press the Esc key, which removes the temporary marker and leaves gnuplot in a clean state. Pretty cool! (Try it!)

> **Listing 12.4   Commands to place arrows using the mouse (file: mouse-arrows.gp)**

```
bind "Button1" \
"if(exists('arrow')==0) { \
   arrow=1; arrow_x=MOUSE_X; arrow_y=MOUSE_Y; \
   set label 1000 at arrow_x,arrow_y point; replot; \
 } else { \
   unset label 1000; \
   set arrow from arrow_x,arrow_y to MOUSE_X, MOUSE_Y; replot; \
   undefine arrow,arrow_x,arrow_y \
}"
bind "Escape" "unset label 1000; replot; undefine arrow,arrow_x,arrow_y"
```

A crucial element of the logic in listing 12.4 is the variable called `arrow`. If it exists in the gnuplot session, then you know the first point has been selected and the temporary marker has been placed; if it doesn't exist, then the first point has not been selected yet. You can test for the *existence* of a variable with this name using the `exists()` function. The reason to test for the existence of this variable (rather than for its value) is that a test for value would result in an error if the variable hadn't yet been initialized. The existence test is more robust.

> **TIP**   The snippet in listing 12.4 works extremely well. It takes all the pain out of gnuplot's `set arrow` facility.

12.5   *Startup configurations and initialization*

You can pass commands and information to gnuplot on startup. All three typical ways of initializing a program are available: initialization files, environment variables, and command-line flags. Let's consider them in turn.

12.5.1 Startup and initialization files

On startup, gnuplot attempts to read the file gnuplotrc from a standard location, such as /usr/share/gnuplot/ on Linux—run `show loadpath` to see the configured location. By default, this file is empty, but you can use it for customizations that should be *global* to your installation.

Gnuplot also looks for a *personal* initialization file, called .gnuplot on Unix (and Mac OS) and GNUPLOT.INI on Windows. Gnuplot searches for this file first in the current directory and then in the current user's home directory (as defined by the corresponding environment variable). Once a file has been located, searching doesn't continue. The file is executed (using `load`) before any other files or commands are read from the command line, standard input, or an interactive terminal. In general, settings in the initialization file override environment variables.

You can suppress reading of both initialization files by giving the `-d` command-line option when invoking gnuplot. This may be useful when you want all of gnuplot's internal settings to assume their default values.

WHAT TO PUT IN YOUR INITIALIZATION FILE

I recommend keeping the default initialization file relatively short, in order to prevent pollution of the gnuplot session (and all command files subsequently saved with `save`). I think it's a better idea to group related settings in individual command files that can be loaded as needed, and to limit the default file to only those settings that you really want to have in effect at all times. Also keep in mind that there is only *one* initialization file that is read automatically by gnuplot. With individual command files or stylesheets (see section 12.6), you have greater control over the state of your gnuplot session. (You can either load them after gnuplot has started, using `load`, or specify them on the command line.)

The following listing shows my personal initialization file. It's relatively short and contains only a handful of settings that I find generally useful.

Listing 12.5   A typical .gnuplot initialization file (file: dot.gnuplot)

```
set encoding utf8

set samples 3000                        ❶ Improves the
                                          default settings
set isosamples 30
set hidden3d
                                                      ❷ Personal library of
                                                        gnuplot scripts
set loadpath "/home/janert/lib/gnuplot/"

load "default-style.gp"

hsv( h, s, v )  = hsv2rgb( h - floor(h), s, v )       ❹ Color-management
pack( r, g, b ) = 2**16*r + 2**8*g + b                  functions

bind "Button1" \
"if(exists('arrow')==0) { \                            ❺ Sets arrows
   arrow=1; arrow_x=MOUSE_X; arrow_y=MOUSE_Y; \          with the mouse
```

❸ Loads personal stylesheet

```
   set label 1000 at arrow_x,arrow_y point; replot; \
 } else { \
   unset label 1000; \5
   set arrow from arrow_x,arrow_y to MOUSE_X, MOUSE_Y; replot; \
   undefine arrow,arrow_x,arrow_y \
}"

bind "Escape" "unset label 1000; replot; undefine arrow,arrow_x,arrow_y"
```

The first few lines fix instances where gnuplot's defaults don't work well ❶. I explicitly enable UTF-8 (see the sidebar at the end of section 5.1) and increase the sampling density for `plot` and `splot` (see sections 3.1 and C.1.2). Next, I define the path for my local library of gnuplot scripts ❷ and load the default style sheet directly from that directory ❸. Then I define two functions for handling color (see section 9.1.1) ❹ and define bindings that let me set arrows with the mouse (see listing 12.4) ❺. That's it—everything else is loaded only as required.

12.5.2 *Environment variables*

Gnuplot examines several environment variables (see table 12.4). Not all of them are still important; some of them provide information for graphics terminals that are no longer current. In general, I don't like to rely on environment variables and instead prefer explicit initialization in a configuration file or scriptlet.

Table 12.4   The environment variables that gnuplot reads on startup. Unless otherwise specified, the variables have the same name on all platforms.

Name (Unix and Mac OS X)	Name (Windows, if different)	Description
GNUTERM		Name of the terminal to be used. (Compare the `set terminal` command.)
GNUHELP		Path to the online help file gnuplot.gih.
GNUPLOT_LIB		Additional directories to be searched for data and command files. (Compare the `set loadpath` facility.)[a]
GDFONTPATH and GNUPLOT_DEFAULT_GDFONT		Search path used by terminals based on the GD library (PNG, GIF, and JPG) to locate font files, and a default font to be used with these terminals, respectively.
GNUPLOT_FONTPATH		Search path used by the PostScript and SVG terminals to locate font files. (Compare gnuplot's `set fontpath` facility.)[a]
GNUPLOT_PS_DIR		Used by the PostScript terminal to locate custom prologue files.

[a] Path entries are separated from one another using a colon (`:`) on Unix and a semicolon (`;`) on Windows.

Table 12.4   The environment variables that gnuplot reads on startup. Unless otherwise specified, the variables have the same name on all platforms. *(continued)*

Name (Unix and Mac OS X)	Name (Windows, if different)	Description
GNUFITLOG		Name of the log file maintained by the `fit` command (discussed in section F.7); defaults to fit.log. If a directory is named, then gnuplot creates a file with the default name in this directory.
HOME	GNUPLOT and USERPROFILE	Directory to search for a .gnuplot initialization script if none is found in the current directory.
PAGER	n/a	Name of the pager program to use.
SHELL	COMSPEC	Name of the shell to spawn in response to the gnuplot `shell` command.

12.5.3 Gnuplot command-line flags

In general, gnuplot executes the contents of command files named on the command line when gnuplot is first started (see section 11.3). Remember that the special file-name - (hyphen) suspends processing of the command line and begins reading interactive commands from standard input.

In addition, gnuplot recognizes six command-line flags. See table 12.5.

Table 12.5   Command-line options recognized by gnuplot

Short form	Long form	Description
-V	--version	Prints version information.
-h	--help	Prints a short usage message.
-p	--persist	Continues to display the display window for interactive terminals, even after the interactive gnuplot session has terminated.
-d	--default-settings	Suppresses loading of both the global (gnuplotrc) and the personal (.gnuplot) initialization files.
-c ...		Treats the rest of the command line as if invoked using `call`. This option should be followed by the name of a command file and, optionally, its arguments.
-e "..."		Executes the commands given as string argument before processing the rest of the command line.

12.6 Stylesheets

The concept of a *stylesheet* is probably most familiar in the context of web development, but it is, in fact, more general: wherever a document is prepared using *markup,* the stylesheet principle can be applied. The markup language is used to indicate *semantics*, whereupon the stylesheet is applied to fix the visual *appearance.*

This notion can be carried over to gnuplot, once you realize that a command like

```
plot "data" u 1:2 w lp lt 1, "" u 1:3 w l lt 2
```

constitutes a form of markup. The appearance of the plot depends on the styles that are in effect when the plot is *rendered* by gnuplot. By supplying suitably chosen definitions for line types and other properties (such as arrow styles and text fonts), you can therefore create a family of graphs, each with a different visual appearance.

The notion of using stylesheets like this with gnuplot may seem unfamiliar (at least, I've never seen this done before), but it's an extremely powerful concept. It enables you to create graphics for specific circumstances and purposes. At their most basic, stylesheets allow you to create both color and black-and-white versions of graphs, but the idea can be taken further. Graphs for interactive work (on a computer screen) place different demands on colors, line widths, point sizes, and so on, than graphs prepared for (say) a PowerPoint presentation (see the sidebar "But is it beautiful?" later in this chapter).

To be clear, I'm not saying you can apply a different stylesheet to an existing command file and obtain a satisfactory graph without tweaking (as you can with CSS or, for example, LaTeX). But by using different sets of predefined line types and other visual-appearance options, you can begin to create graphs that take the specifics of the intended presentation channel or medium (screen, print, or overhead) into account. If nothing else, you can use a personal stylesheet to escape the standard look of gnuplot graphics (although I don't think you should—see the sidebar in section 8.3.2). But it's nice to have the option!

> **TIP** Customize styles to adapt the appearance of graphs to their intended use and medium. Collect related styles together to create stylesheets for different purposes.

12.6.1 Worked example: stylesheets

What does the stylesheet concept look like in practice? In this section, I'll present the same graph four different ways, using four different stylesheets—with *no* changes to the plot command! The plot includes six curves (so there's a need for a variety of distinguishable line styles) and both sparse and dense data sets (so I have the opportunity to use both the with lines and the with linespoints plot styles).

The plot command is shown in the next listing. It's used without modification to generate all of the next four graphs. Only the stylesheets differ.

> **Listing 12.6   Plot command for figures 12.1 through 12.4 (file: style-cmd.gp)**

```
set xlabel "Input"
set ylabel "Response"

plot "sparse" u 1:2 t "Sparse 1" w lp, "" u 1:3 t "Sparse 2" w lp,
➥    "" u 1:2:(1000) t "Smooth" s acs w l,
➥    "dense" u 1:2 t "Dense 1" w l, "" u 1:3 t "Dense 2" w l,
➥    "" u 1:4 t "Dense 3" w l
```

The stylesheets must be loaded (using `load`) before the `plot` command is issued. You should also run `reset` or `reset session` before loading the next stylesheet (otherwise, previous styles will sometimes interfere with current ones).

INTERACTIVE COMPUTER WORK 1: THIN AND BRIGHT

The stylesheet in listing 12.7 is intended as a replacement for the old gnuplot default appearance and is meant primarily for interactive use at a computer screen (although I also used it for most of the color figures in this book). It's characterized by relatively thin lines, in bright colors. To make thin lines visually distinguishable, it's necessary to choose relatively bright colors; but in contrast to the old gnuplot default, the colors are toned down a bit. A dark green is used in place of bright green, and the color yellow (traditionally the sixth style in the default sequence), which is too difficult to see against a white background, is replaced with a drab goldenrod.[6] Moreover, I tried to make sure colors that are adjacent in the sequence exhibit good contrast with each other; hence the two greens and blues are well separated. The choice of point type also warrants a remark: instead of the spidery cross and saltire, this stylesheet prefers filled and open circles and squares. The resulting plot is shown in figure 12.1. (The `set termoption` command changes just one aspect of a terminal, without requiring the terminal to be specified—see chapter 10.)

> **Listing 12.7   Stylesheet for figure 12.1 (file: style1.gp)**

```
set linetype  1 pt 7 lc rgb '0xee0000'       ⟵——— Red2
set linetype  2 pt 5 lc rgb '0x008b00'       ⟵——— Green4
set linetype  3 pt 6 lc rgb '0x0000cd'       ⟵——— MediumBlue
set linetype  4 pt 4 lc rgb '0xff3fb3'       ⟵——— Maroon1
set linetype  5 pt 2 lc rgb '0x00cdcd'       ⟵——— Cyan3
set linetype  6 pt 7 lc rgb '0xcd9b1d'       ⟵——— Goldenrod3
set linetype  7 pt 5 lc rgb '0x8968cd'       ⟵——— MediumPurple3
set linetype  8 pt 6 lc rgb '0x8b8b83'       ⟵——— Ivory4
set linetype  9 pt 4 lc rgb '0x00bb00'       ⟵——— Custom green
set linetype 10 pt 2 lc rgb '0x1e90ff'       ⟵——— DodgerBlue1
set linetype 11 pt 7 lc rgb '0x8b2500'       ⟵——— OrangeRed4
set linetype 12 pt 5 lc rgb '0x000000'       ⟵——— Black

set termoption font "Helvetica,12"
set termoption fontscale 0.5
```

---

[6] Because hex strings aren't intuitive, the X11 color name has been included for each stylesheet color in this section.
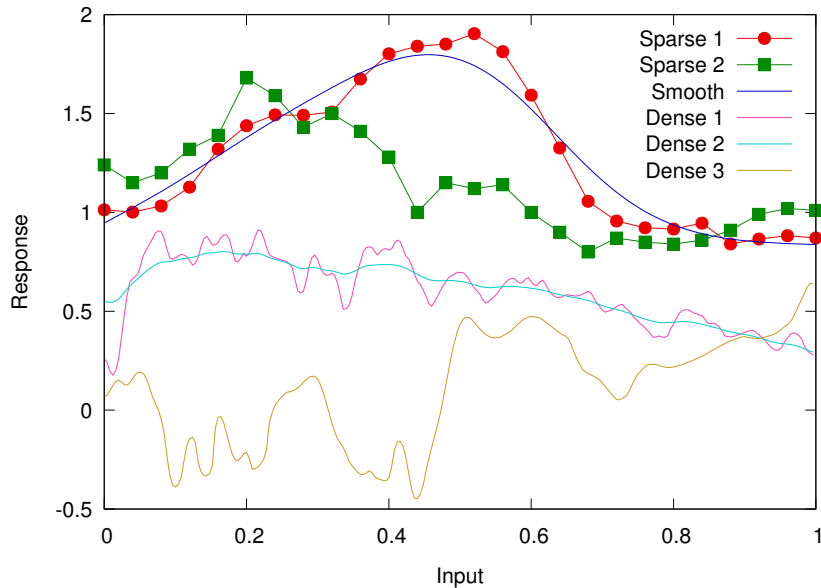
Figure 12.1 The example graph using the stylesheet from listing 12.7: thin lines, bright colors, small symbols

INTERACTIVE COMPUTER WORK 2: THICK AND PASTEL

The second stylesheet (listing 12.8) is also intended for interactive use on a computer screen, but behold how different the resulting graph appears! (See figure 12.2.) Gone are the brightly colored hairlines, and instead the graph uses thick, wide lines in soft pastel colors. (It also uses a boldface font, in keeping with the thick line widths.) If you're used to graphs like figure 12.1, then this style takes some getting used to, but I recommend that you give it a fair try. After a while, you may notice that the muted colors and soft edges have a welcome, soothing effect. But it's quite different than what you saw before.

Listing 12.8   Stylesheet for figure 12.2 (file: style2.gp)

```
set linetype 1 lw 3 pt 7 lc rgb '0x87ceeb'      ⟵——— SkyBlue
set linetype 2 lw 3 pt 5 lc rgb '0x7ccd7c'      ⟵——— PaleGreen3
set linetype 3 lw 3 pt 6 lc rgb '0xab82ff'      ⟵——— MediumPurple1
set linetype 4 lw 3 pt 4 lc rgb '0x6ca6cd'      ⟵——— SkyBlue3
set linetype 5 lw 3 pt 7 lc rgb '0xfa8072'      ⟵——— Salmon
set linetype 6 lw 3 pt 5 lc rgb '0xd2b48c'      ⟵——— Tan

set pointsize 0.5

set termoption font "Helvetica-Bold,11"
set termoption fontscale 0.5
```
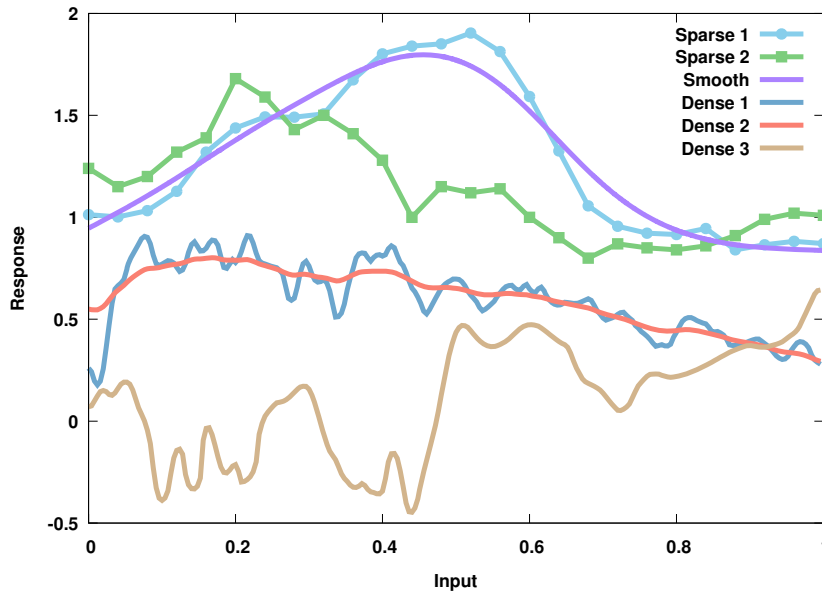
Figure 12.2   The example graph using the stylesheet from listing 12.8: thick lines, pastel colors. Compare figure 12.1.

PRINT PUBLISHING: BLACK-AND-WHITE

For print publishing, it may still be necessary to submit graphs in black and white. For line art, this means you have to resort to *dash patterns* to distinguish lines from each other. This is what the stylesheet in listing 12.9 is for. It uses custom dash patterns that are intended to re-create (and improve on) the dash patterns from gnuplot's Post-Script terminal. In keeping with the preference for hairlines in scientific drawing, it doesn't use filled point symbols and instead gives preference specifically to the cross and saltire—but with a twist! It uses gnuplot's new `pointintervalbox` feature so the continuous line doesn't touch, and thereby interfere with, the point symbols. The resulting graph is shown in figure 12.3.

Listing 12.9   Stylesheet for figure 12.3 (file: style3.gp)

```
set linetype 1 pi -1 pt 1 lc black dt solid
set linetype 2 pi -1 pt 2 lc black dt (8,6)
set linetype 3 pi -1 pt 6 lc black dt (4,3)
set linetype 4 pi -1 pt 4 lc black dt (3,6)
set linetype 5 pi -1 pt 1 lc black dt (12,5,2,5,2,5)
set linetype 6 pi -1 pt 2 lc black dt (16,8)
set linetype 7 pi -1 pt 6 lc black dt (20,6,2,6)
set linetype 8 pi -1 pt 4 lc black dt (30,10)

set pointsize 0.5
set pointintervalbox 1.75

set termoption font "Helvetica,12"
set termoption fontscale 0.5
```
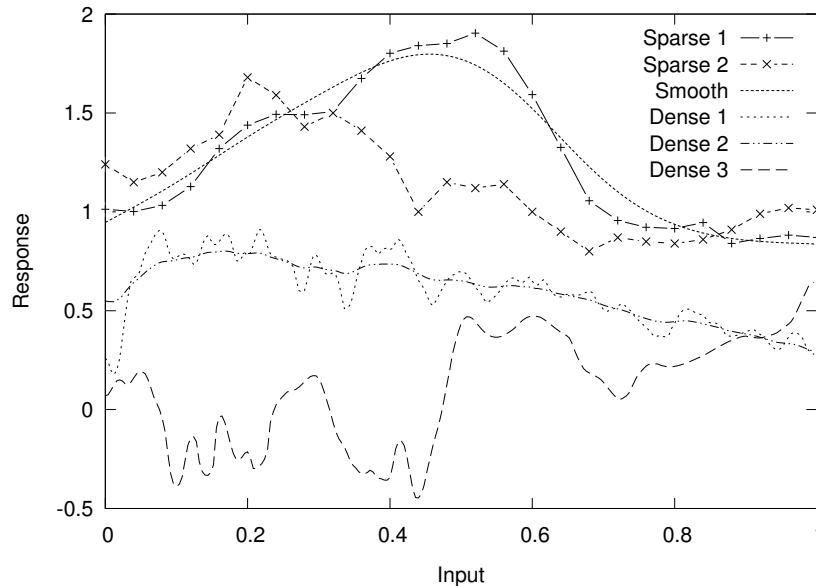
Figure 12.3 The example graph using the stylesheet from listing 12.9: black-and-white, dashed hairlines, cross-shaped symbols

A WALK ON THE WILD SIDE

The last stylesheet has a two-fold purpose: I want to demonstrate some intentionally *questionable* choices for the appearance of a graph—these aren't recommendations! More seriously, I also want to show how to make a graph look entirely *different* from the usual gnuplot appearance. Here are some of the differences: the graph has no frame, but a couple of arrows indicating the coordinate axes; the aspect ratio is square; the font is a serif type; the lines are wide *and* bright; despite being color-coded, they're also dashed; the frame around the key unnecessarily clutters and partially obscures the graph; and the text is either too large or too small. Not all of these choices are poor, but some definitely are. (See the following listing and figure 12.4.)

Listing 12.10 Stylesheet for figure 12.4 (file: style4.gp)

```
unset border
set xtic nomirror
set ytic rotate by 90 center nomirror
set arrow 1000 from graph 0,0 to graph 1.075,0 head filled lw 2
set arrow 1001 from graph 0,0 to graph 0,1.075 head filled lw 2

set size square

set lmargin 7
set rmargin 5
set tmargin 1.5
set ylabel offset 3,0
set xlabel offset 0,0.5
```

```
set key opaque title "Legend" box lw 2 font ",10"

set linetype 1 lw 2 pt  9 lc rgb '0xff4500'          ⟵——  OrangeRed
set linetype 2 lw 2 pt 11 lc rgb '0x00ff00' dt '-'   ⟵——  Green
set linetype 3 lw 2 pt 13 lc rgb '0x0000ee' dt '_'   ⟵——  Blue1
set linetype 4 lw 2 pt  9 lc rgb '0xff1493' dt '.'   ⟵——  DeepPink
set linetype 5 lw 2 pt 11 lc rgb '0x00eeee'          ⟵——  Cyan2
set linetype 6 lw 2 pt 13 lc rgb '0x9400d3' dt '-'   ⟵——  DarkViolet

set termoption font "Times,20"
set termoption fontscale 0.5
```
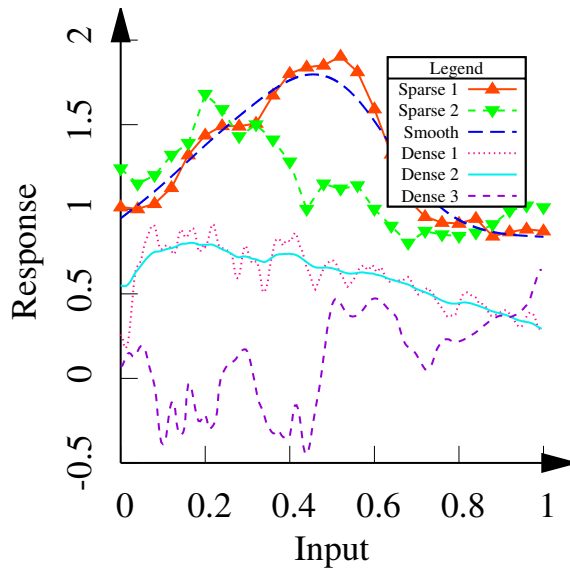

Figure 12.4   The example graph using the stylesheet from listing 12.10: as different from the gnuplot defaults as possible

But is it beautiful?

Gnuplot's defaults owe some to scientific tradition and some to technical limitations on early computer systems. Few of these limitations still remain, which means you're free to choose from the available options for the best results.

The first two of gnuplot's point symbols (or point types—see figure 9.7) come from a scientific tradition: on graph paper, where accuracy needed to be maintained manually, two thin, short lines intersecting in a point provided a good way to fix a location on the graph precisely. Today, when the data is available numerically and the computer takes care of accuracy, this is much less of a concern, and you can choose symbols that are more aesthetically pleasing.

For scientific plots of medium-sized data sets, both cross and saltire may still be adequate. The combination of the two (the third symbol in gnuplot's default ordering) is bulky and shouldn't be used.

For presentation graphics, in particular for sparse data sets, I recommend the filled circle and the filled square. These two symbols, together with a choice of colors, should be enough to distinguish several data sets in a single graph. If you need even more symbols, I recommend the unfilled circle and unfilled square, followed by cross and saltire, and possibly the tip-down triangles (filled and unfilled). The diamond (four-cornered symbol, standing on its tip) doesn't reproduce well and should probably, although regretfully, be avoided.

For large data sets consisting of many points, the unfilled circle has a special advantage: even if points are spaced densely enough that symbols overlap partially, unfilled circles provide the best chance of being recognizable as *separate symbols*. Whereas filled symbols occlude each other, and symbols made up of straight lines (such as crosses and squares) blend together as individual edges align, unfilled circles can still be distinguished from each other.

Traditionally, thin lines have been used in scientific publishing: again, because a thin line fixes a location on a graph more precisely. This is still a good tradition to follow, in particular for print publishing: thin lines often give a publication a classy appearance. (Provided they aren't too thin to render properly—as publishers switch from photo-offset to print-on-demand technology, sturdier lines might be a good idea!)

For other publication channels, this isn't true. In particular for slides and overhead presentations, thicker, rounded lines work better: they can be seen from the back of the room, and they tend to fit better with the overall style of the medium. Graphs for the Web fall between these two extremes.

In general, I recommend using non-serif fonts (such as Helvetica) for labels on a graph, because such fonts are the most neutral. Make sure you choose the font size properly: it's surprisingly common to see graph labels that are too large, leading to an imbalanced appearance of the entire graph.

12.7   Summary

In this chapter, we explored different ways to go beyond gnuplot's default appearance and behavior. First, we considered ways to streamline your gnuplot workflow through scripts and customized key bindings (hotkeys). You may want to personalize your work environment when using gnuplot, and I showed you how to do that.

In the last part of the chapter, I introduced the concept of *stylesheets*. Stylesheets are bundles of appearance options that you can use to create graphs that are optimized for their intended use and medium—be it an interactive computer session, a black-and-white print publication, or a presentation to a live audience.

Part 4

Understanding data

In this final part, we can take the technical aspects of working with gnuplot largely for granted, and instead concentrate on the tasks that can be solved with gnuplot. Chapter 13 introduces several fundamental types of graphs and explains when and how to use them. Chapter 14 is more advanced and offers solutions to some recurring topics in graphical analysis. Chapter 15 closes this book with a reminder regarding the ultimate purpose of all kinds of analysis.

<div align="right">

Basic techniques of graphical analysis

</div>

This chapter covers

- Investigating relationships
- Using logarithmic scales
- Representing point distributions
- Visualizing ranked data
- Organizing your work

This chapter and the next largely take gnuplot and its features for granted and instead concentrate on *applying* gnuplot to specific goals that commonly arise when graphing data. At the same time, I'll take the opportunity to show you how certain tasks can be accomplished using gnuplot's features. In this chapter, I'll discuss some basic activities that are generally useful in graphical analysis; the next chapter will cover some more specialized tasks in greater depth.

When faced with a new data set, two questions usually dominate. The first is, how does one quantity depend on some other quantity—how does y vary with x? The second question (for data sets that include some form of statistical noise) asks, how is some quantity distributed—where are data points located, and what's the

<div align="center">

289

</div>

character of the randomness? These are the two primary questions for this chapter. In the course of it, we'll also revisit logarithmic plots and their uses and examine more applications of smooth approximations to noisy data sets—topics first introduced in chapter 3.

13.1 Representing relationships

One of the conceptually easiest and most familiar types of graphs shows *the dependence of one quantity on another*—so much so that creating a plot of this type is what is commonly meant by "graphing data." You've seen several examples in this book, like figure 2.4, figure 3.1, and many more.

13.1.1 Scatter plots

For many data sets, it's natural to ask whether one quantity depends on another and, if so, how: does y grow as x grows, or does it fall, or does y not depend on x to begin with? A *scatter plot* is the first step in finding the answer. A scatter plot shows unconnected symbols located at the position given by x and y. It's an easy way to get a feeling for an otherwise unknown data set.

A SIMPLE EXAMPLE: CAR DATA

Listing 13.1 shows the first few lines from a sample data set containing 26 attributes for 205 different car models that were imported into the US in 1985.[1] The 14th column gives the curb-weight in pounds, and the last (26th) column the price (in 1985 dollars). A scatter plot like the one in figure 13.1 shows how weight varies as a function of price.

> **Listing 13.1   Incomplete data for figure 13.1 (file: imports-85.data)**

```
1,158,audi,gas,turbo,four,sedan,fwd,front,105.80,192.70,71.40,55.90,
0,?,audi,gas,turbo,two,hatchback,4wd,front,99.50,178.20,67.90,52.00,
2,192,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2395
0,192,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,239
0,188,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2710
...
```

In this case, the input file is not whitespace-separated, but comma-separated. Instead of transforming the input file to space separated, it's more convenient to use gnuplot's set datafile separator option to plot this data file. The open circle (pt 6) is used because it remains easiest to distinguish even when data points overlap:

```
set datafile separator ","
plot [2500:] "imports-85.data" u 26:14 pt 6
```

You can clearly see that weight goes up as the price increases, which is reasonable. You should also note that there are many more low-price/low-weight cars than heavy, premium vehicles. For budget cars, weight seems to increase in step (linearly) with price

---

[1]   This example comes from the "Automobile" data set, available from the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/datasets/Automobile.
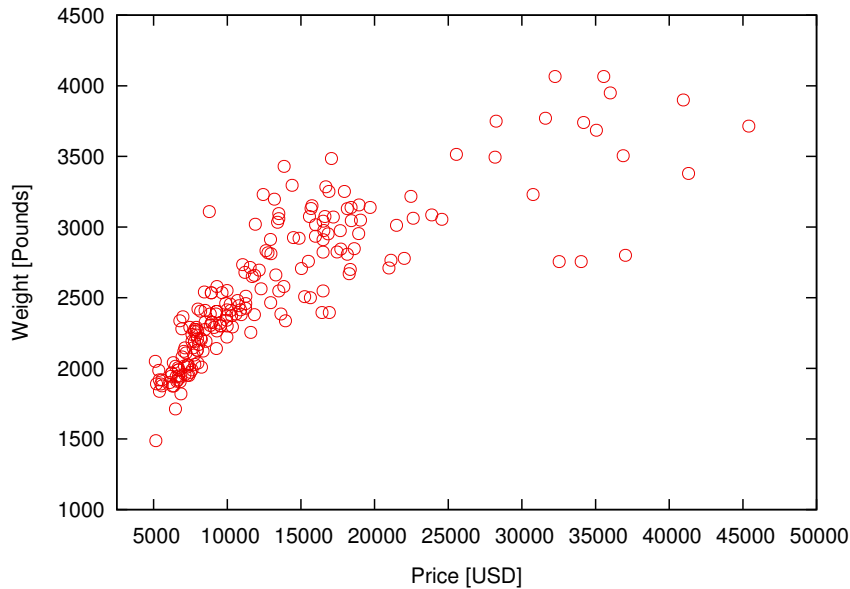
Figure 13.1 Curb weight versus price for 205 different cars. See listing 13.1.

for a while; but for higher-priced vehicles, the gain in weight levels off. This observation may have a simple explanation: the price of mass-market vehicles is largely determined by the cost of materials, so that a car that's twice as big (as measured by its overall mass) is also twice as expensive, whereas the price of luxury cars is determined by higher quality (fancier materials such as leather seats, and additional options such as more electronics) rather than by sheer bulk.

USING SCATTER PLOTS

This example demonstrates what to look for when examining a scatter plot. The first question usually concerns the nature of the relationship between x and y. Does y fall as x grows or vice versa? Do the points fall approximately onto a straight line or not? Is there an oscillatory component? Whatever it is, take note of it.

The second question concerns the strength of the relationship, or, put another way, the amount of noise in the data. Do the data points jump around unpredictably as you go from one x value to the next? Are there outliers that seem to behave differently than the majority of the points? Detecting outliers is important: gone unnoticed, they'll mess up most statistical quantities (such as the mean) you may want to calculate later. And sometimes, outliers indicate an interesting effect—maybe a subgroup of points follows different rules than the majority. Outliers in scatter plots should never go uninvestigated.

A third aspect to look for in a scatter plot is the distribution of points in either dimension. Are points distributed rather uniformly, or do they cluster in a few locations? If so, do you understand the reason for the clustering, or is this something you should investigate further? There may be a lot of information even in a humble scatter plot!

13.1.2 *Highlighting trends*

The primary reason for preparing scatter plots is to determine whether there is any relationship between the plotted quantities, and what the nature of this relationship might be. In particular when the data is noisy, it may be difficult to recognize the dominant behavior from the individual data points alone, and so there is a need for graphical tools to highlight trends that may exist in the data. Gnuplot offers *weighted cubic splines* for this purpose. Let's study an example before discussing splines in more detail.

WORKED EXAMPLE: MARATHON WINNING TIMES

Figure 13.2 shows the finishing times of the winners in a marathon event from when the event was first conducted until 1990.[2] In general, you see that the finishing times have decreased over time—the top athletes are getting better every year. The changes are particularly dramatic for the women's results since they started competing in 1966.
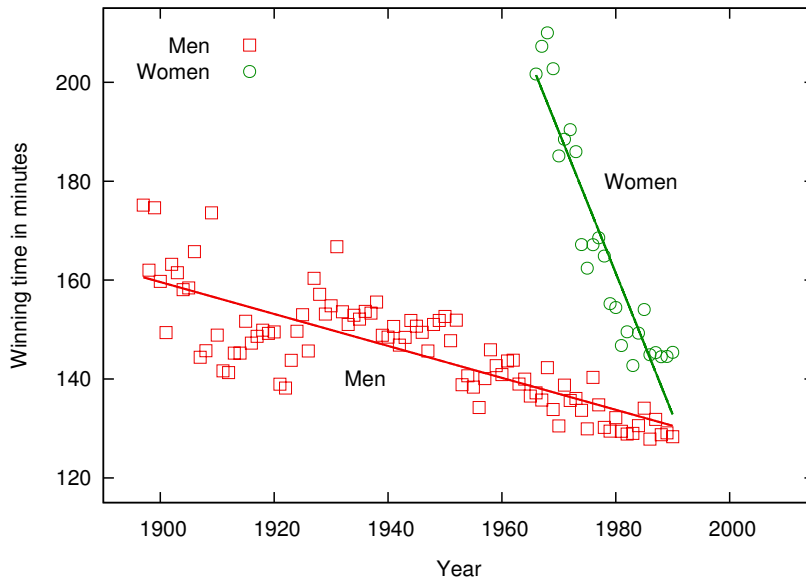
Figure 13.2   Finishing times (in minutes) for the winner of a marathon (up to the year 1990), together with the best straight-line fit. Will women overtake men in the coming years?

Also shown are the best-fit straight-line approximations to the data (as found by the `stats` command), and they seem to represent the data quite well. The only issue is that according to those fits, women should overtake men sometime in the early 1990s—and then continue to get dramatically faster. Is this a reasonable conclusion?

This example attempts to demonstrates two important points when working with data. The first is the need to be sensitive to the structure and quality of the data. For the data in figure 13.2, fitting a straight line provides only a very coarse—and, as you'll see, misleading—approximation of the real behavior.

---

[2] This example was inspired by the book *Graphic Discovery* by Howard Wainer (Princeton University Press, 2005).

You need to remember that by fitting a straight line, you've chosen one specific model to represent the data. At this point, you're no longer analyzing the data with the intention of revealing the information contained in it, but are making a specific statement about its surmised behavior. Before making such a strong statement, you should have sufficient evidence for the applicability of the particular model selected. And coming back to the current example, there certainly doesn't seem to be any strong theoretical reason why athletic performance should follow a straight line as a function of time.

To understand the structure of the data, it might be a better idea to represent the data by using "soft" local approximations, such as weighted splines. Some experimentation with the weights will tell you much about the structure of the data: does the overall shape of the approximation curve change drastically as you vary the weighting? Which features are the most robust, and which disappear most quickly? Typically, significant features tend to be rather robust under transformations, whereas less relevant features are more transient.

Figure 13.3 shows the same data as figure 13.2, but instead of a straight line, a soft spline is used to represent the data, like so:

```
plot [1890:2017.5][115:215] "men-women" u 1:2 t "Men" w p pt 4,
➡                           "" u 1:3 t "Women" w p pt 6,
➡                           "" u 1:2:(1e-2) t '' s acs lt 1 lw 2,
➡                           "" u 1:3:(1e-2) t '' s acs lt 2 lw 2
```
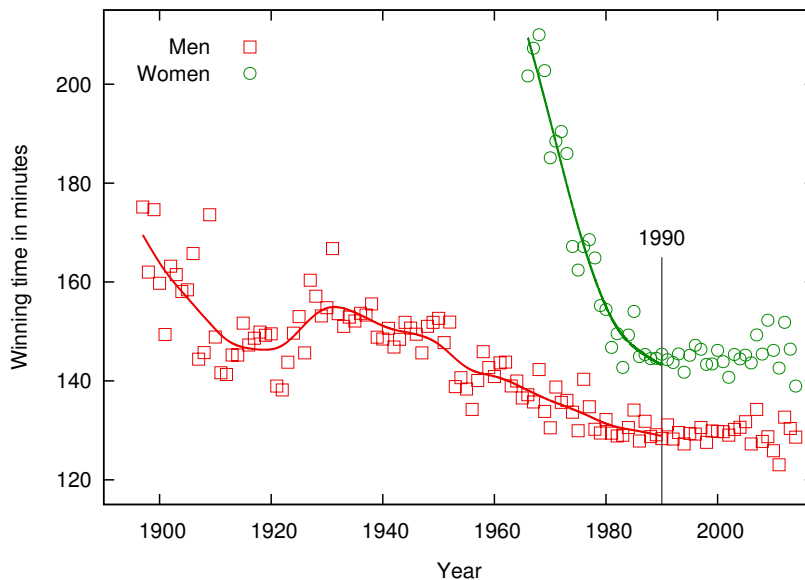

Figure 13.3   The same data as in figure 13.2, together with a weighted-splines fit. The fit is based only on points prior to 1990, but the actual finishing times for the following years are also shown. The softer spline clearly reveals the leveling off of the women's results well before 1990.

This approximation suggests that women's performance starts to level off in the late 1980s, and the results from years after 1990 corroborate this observation. Note that the spline approximation in the graph is based *only* on years up to and including 1990, but not on later data points.

More on splines

Splines are a way to provide a smooth approximation to a set of points.[a] Splines are constructed from piecewise polynomial functions, which are joined together in a smooth fashion; the joints are referred to as *knots*. In the case of *interpolating* splines, the knots are made to coincide with the actual data points; the resulting curve is therefore forced to pass *exactly* through all data points. In the case of *smoothing* or *approximating* splines, the vertical position of each knot is left to vary, so the resulting curve in general does *not* pass through the individual data points. Because in the latter case the curve doesn't have to pass through any data points exactly, it can be less wiggly.

Both interpolating and approximating splines must fulfill the same smoothness conditions, but in addition, the approximating spline must strike a balance between the following two conditions:

- Passing close to the data points
- Not being too wiggly

These conditions are expressed in the following functional, which is minimized by the approximating spline $s(x)$

$$J[s] \; = \; \int \left(s''(x) \right)^2 \mathrm{d}x + \sum_{i}^{N} w_i \left(s(x_i) - y_i \right)^2$$

where (x_i, y_i) are the coordinates of the data points, the w_i are the weights attached to each data point, and the prime indicates a derivative with respect to x. In this functional, the first term is large if $s(x)$ is wiggly, and the second term is large if $s(x)$ doesn't pass close to the data points. (The form of the first term comes from a physical analogy: if the spline were made out of a real material, such as a thin strip of wood or metal, the first term would be related to the total bending energy of the strip.)

The balance between these two terms is controlled through the weight parameters w_i: if the w_i are small, the first term dominates, and the resulting spline approaches a straight line (which happens to coincide with the least-squares linear regression line for the data set). If the weights are large, the second term dominates, and the spline approaches the interpolating spline (which passes exactly through all data points).

---

[a] I'd like to thank Lucas Hart and Clark Gaylord for helpful correspondence regarding this topic.

Another way to think about the weights is to write $w_i = 1/d_i^2$, where d_i is a measure for the uncertainty in the data of point i (such as the standard deviation in this point). One would expect the spline to pass through the interval $[y_i\text{-}d_i, y_i\text{+}d_i]$ at x_i. The higher the confidence in one point, the smaller you can choose to make this interval, and therefore the larger the weight w_i will be. By choosing $d_i = 0$ for one of the points, you can even force the curve to pass through this point exactly, although you may let the spline float more freely for the other points. In principle it's possible to choose a different weight for each point, but if all points are known to the same accuracy, then all weights should be the same. This is what was done for all examples in this book.

One important remark: the way $J[s]$ is written in the example, the size of the second term depends on the number of data points—if you double the size of the data set, the size of the second term will be (approximately) twice as large. By contrast, the first term does *not* depend on the number of data points. If the number of data points grows, the second term will therefore become larger relative to the first one, and the resulting spline will be more wiggly.

To maintain the original balance between wigglyness and closeness of approximation, the weights must be increased accordingly for data sets containing a larger number of points. Equivalently, you might want to take the number of points into account explicitly by writing $w_i = u_i/N$, where u_i is the actual weight and N is the number of knots. With this choice for w_i, the balance between both terms will be maintained regardless of the number of knots in the data set.[b]

---

[b] You can find more information on splines in chapter 1 of *Handbook on Splines for the User* by E. V. Shikin and A. I. Plis (CRC Press, 1995).

13.2   Logarithmic plots

Logarithmic scales are one of the most versatile tools in the graphical analyst's toolbox. I introduced them in section 3.4 and discussed how they work. Now let's put them into action.

Logarithmic scales serve three purposes when plotting:

- They rein in large variations in the data.
- They turn multiplicative deviations into additive ones.
- They reveal exponential and power-law behavior.

13.2.1   Large variations in data

To understand the meaning of the first two items, let's study the daily traffic pattern at a website. Figure 13.4 shows the number of hits per day over approximately three months. There's tremendous variation in the data, with alternating periods of high and low traffic. During periods of high traffic, daily hit counts may reach close to half a million hits but then fall to very little traffic shortly thereafter. On the scale of the graph, the periods of low traffic seem barely different from zero, with little fluctuation.
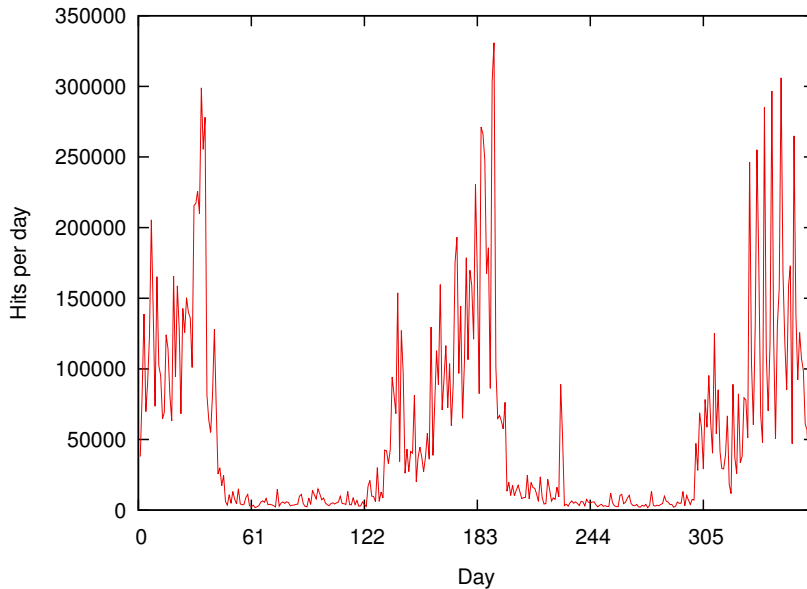
Figure 13.4   Traffic patterns for a website. Daily hit count versus day of the year. Note the extreme variation in traffic over time.

Figure 13.5 displays the same data, but now on a semi-logarithmic scale. The logarithmic scale helps to dampen the extreme variation of the original data set (two orders of magnitude) so you can see the structure during both the high- and the low-traffic seasons. That's the first effect of logarithmic plots: they help to make data spanning extreme ranges visible by suppressing high-value outliers and enhancing low-value background.

Furthermore, you can see that the *relative* size of the day-to-day fluctuations is about equal during both phases. The absolute size of the fluctuations is quite different, but their size as a percentage of the average value is roughly the same (very approximately, during low season, traffic varies between 2,000 and 20,000 hits a day, a factor of 10; whereas during high season it varies between 30,000 and 300,000 hits a day, again a factor of 10). That's the second effect of logarithmic plots: they turn multiplicative variations into additive ones.

Figure 13.6 tries to demonstrate the last point in a different way. The bottom panel shows the web traffic on consecutive days (like figure 13.4), displaying great seasonal variance, but the top panel shows the ratio of the difference in traffic on consecutive days divided by the actual value—(*current day* − *previous day*)/*current day*—which does *not* exhibit a seasonal pattern. This is further proof that the daily fluctuation, viewed as a percentage of the overall traffic, is constant throughout. The calculation for the top panel can be performed as an inline transformation, by the way:

```
prv=40000
plot [0:365][-1:1] "webtraffic" u (y=($2-prv)/($2+prv), prv=$2, y) w l
```
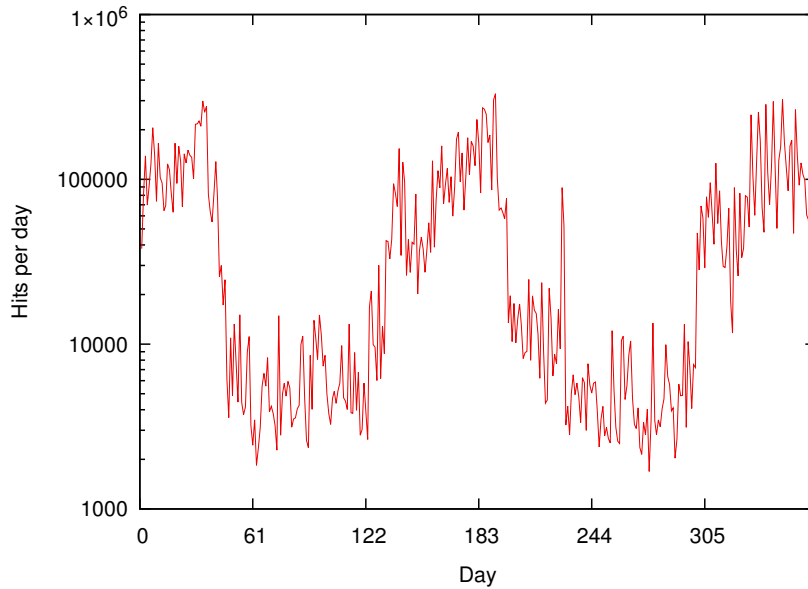
Figure 13.5   The same data as in figure 13.4, but on a semi-logarithmic scale. Note how the high-traffic outliers have been suppressed and the low-traffic background has been enhanced. In this presentation, data spanning two orders of magnitude can be compared easily.
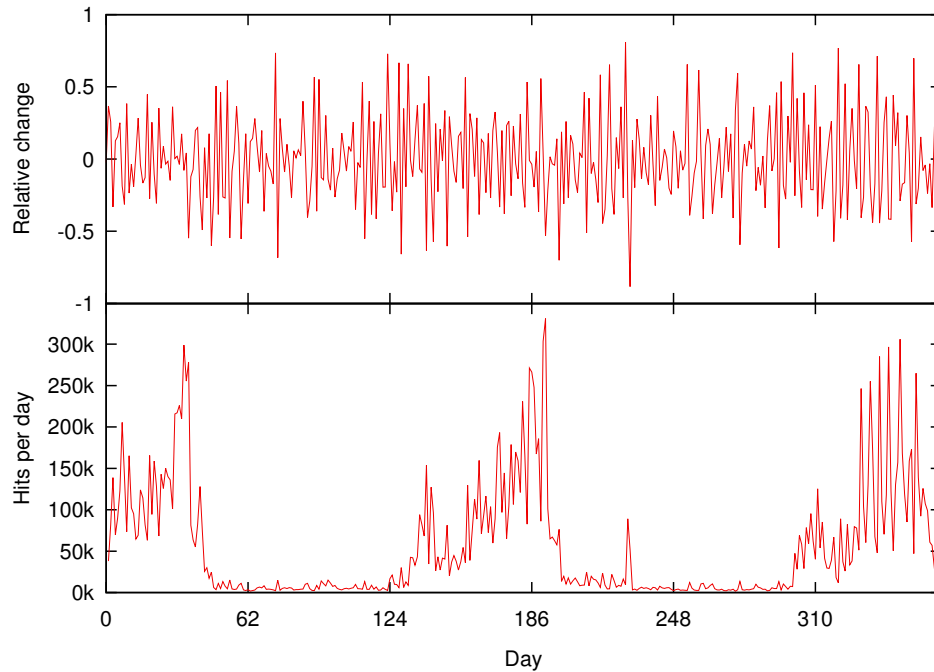
Figure 13.6   Bottom panel: hits per day over time (as in figure 13.4); top panel: change in traffic between consecutive days, divided by the total traffic. Note how the relative change (top panel) doesn't exhibit any seasonal pattern, indicating that the relative size of the variation is constant.

13.2.2 *Power-law behavior*

Finally, let's look at a curious example that brings together two benefits of logarithmic plots: the ability to display and compare data of very different magnitude, and the ability to turn power-law behavior into straight lines. Mammals come in all shapes and sizes, from tiny rodents (the smallest known land mammal is the Pygmy Shrew, which weighs only a few grams, but some bats found in Thailand are apparently smaller still) to the largest of whales (weighing several hundreds of tons). It's a curious empirical fact that there seem to be fixed relationships between different metabolic quantities— basically, the larger an animal is, the slower its bodily functions progress. Figure 13.7 shows an example: the duration (in seconds) of a single resting heartbeat as a function of the typical body mass. The regularity of the data is remarkable, spanning *eight orders of magnitude* for the mass of the animal. What's even more amazing is how well the data is represented by the simple function $T \sim m^{1/4}$. This law isn't limited to the examples shown in the graph: if you added animals to the list, they'd also fall close to the straight line (I didn't just pick the best ones).

The existence of such scaling relations in biological systems has been known for a long time and seems to hold generally. For example, it turns out that the typical lifetime of a mammal also obeys a quarter-power scaling law relation against the body mass, leading to the surprising conclusion that the total number of heartbeats in the life of a single organism is fixed—no matter what the typical resting heart rate is. (In case you care, the number comes out to about 1.5 billion heartbeats during a typical lifetime.)
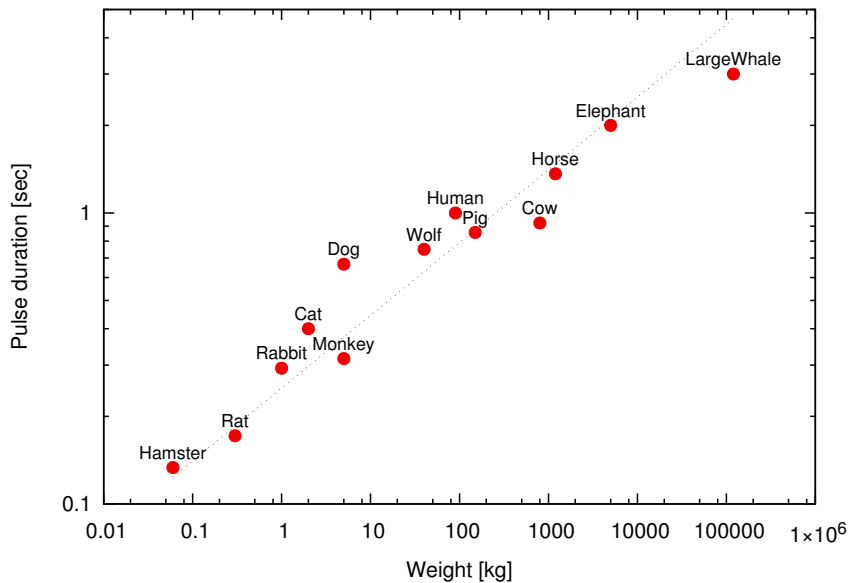
Figure 13.7   Allometric scaling: the duration of an average resting heartbeat as a function of the typical body mass for several mammals. Note how the data points seem to fall on a straight line with slope 1/4.

These observations have been explained in terms of the geometrical constraints that must exist in the vascular networks (the veins and arteries), which supply nutrients to all parts of the organism.[3] As it turns out, you can derive the quarter-power scaling laws starting from only three simple assumptions: that the support network must be a space-filling fractal, reaching all parts of the organism; that the terminal capillaries where nutrients are exchanged are the same size in all animals; and that organisms have evolved in such a way that the energy required for the transport of nutrients through their bodies is minimized. I think it's amazing how such a powerful result can be derived from such simple assumptions. But, on the other hand, we shouldn't be surprised: generally applicable laws (such as the quarter-power scaling in this example) must stem from very fundamental assumptions, disregarding any specifics.

Let's come back to figure 13.7. The double-logarithmic scales make it possible to follow the data over eight orders of magnitude. (Had I used linear scales, all animals except for the whale would be squished against the left side of the graph—literally crushed by the whale.) So again, logarithmic scales can help to deal with data spanning a wide range of values. In addition, the double-logarithmic plot turns the power law relationship $T \sim m^{1/4}$ into a straight line and makes it possible to read off the exponent from the slope of the line. I explained how this works in detail in section 3.4 and won't repeat it here.

Finally, figure 13.7 is a nice example for the power of gnuplot's `with labels` plot style. The graph was generated using

```
plot [][:5] "mammals" u ($2/1000):(60/$3) w p pt 7,
➡         "" u ($2/1000):(60/$3*1.125):1 w labels
```

The first part of the command draws the symbols (`with points`); the second adds the labels. Inline transformations are used to express the weight in kilograms (instead of grams) and to transform the heart rate (given in beats per minute) into the duration of each beat (measured in seconds). All the labels are shifted a bit upward so as not to obscure the symbols. In this example, the vertical offset is *multiplicative*, because of the logarithmic scale of the graph (remember: logarithms turn multiplicative offsets into linear ones).

13.3 *Point distributions*

Besides detecting relationships between quantities, you may want to understand how data points that are somehow random are distributed. Are data points spread out evenly, or are they clustered in a few spots? Are distributions symmetric, or are they skewed? How much weight is contained in the tails of a distribution, compared to its center?

Let's say you have a file containing a set of measurements. They can be anything: interarrival times for requests at a web server, completion times of database queries, weights of potatoes, heights of people—whatever. What can be said about them?

---

[3] The original reference is the paper "A General Model for the Origin of Allometric Scaling Laws in Biology" by G. B. West, J. H. Brown, and B. J. Enquist, *Science* 276 (1997): 122. Additional references can be found on the web.

13.3.1 *Summary statistics and box plots*

Two commands will give a first answer: the `stats` command and the `with boxplot` style (see sections 5.6 and E.2, respectively). Both share the advantage that they have *no* adjustable parameters—they are as simple as can be. Both also share the disadvantage that they lose a lot of information about the distribution of individual points, because both aggregate the entire distribution into a handful of summary statistics. (A box-and-whisker plot is after all nothing more than a graphical representation of the results from running the `stats` command.)

In short: to get a *first* idea how the values in a new dataset are distributed, both of these commands are convenient, because of their extreme simplicity. But both hide too much information to develop a detailed understanding.

13.3.2 *Jitter plots and histograms*

One easy way to get a visualization of a collection of random points is to generate a *jitter plot*, which is really a one-dimensional scatter plot, but with a twist. The bottom part of figure 13.8 was created by shifting each data point vertically by a random amount. (The `rand(0)` function returns a random number in the range [0:1].) If I'd plotted the data in a true, one-dimensional fashion, too many of the points would've overlapped, making it difficult to detect clustering. Such jittering by a random amount is a good trick to remember whenever you're creating scatter plots of larger data sets:

```
plot "jitter" u 1:(0.25*rand(0)-.35)
```
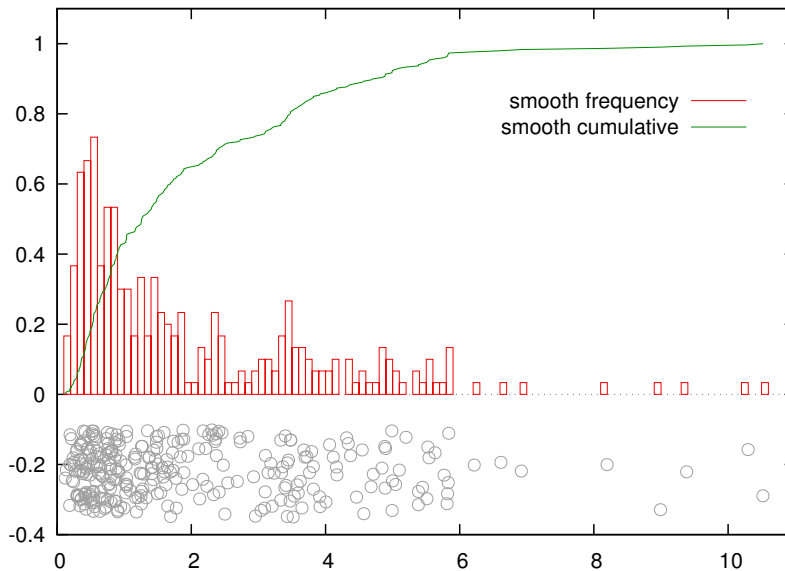

Figure 13.8   Three ways to represent a distribution of random points: jitter plot (bottom), histogram (with boxes), and cumulative distribution function (dashed line)

You can see that the distribution of points is skewed. It's strictly bounded by 0 on the left, with points clustering around 1; and as you move to the right, points become increasingly sparse. But it's hard to say something more definite just by looking at the jitter plot. For instance, is there a second cluster of points between 3 and 4? This does seem possible, but it's hard to tell for sure using this representation.

The next step when investigating the properties of a distribution usually involves drawing a histogram. To create a histogram, you assign data points to buckets or *bins* and count how many events fall into each bin. It's easiest to make all bins have equal width, but with proper normalization per bin, you can make a histogram containing bins of differing widths. This is sometimes useful out in the tails of a distribution where the number of events per bin is small.

Gnuplot doesn't have an explicit histogramming function, but you can use the `smooth frequency` functionality (see section 3.5.3) to good effect. Recall: `smooth frequency` sorts the x values by size and then plots the sum of y values per x value. That's what you need to build a histogram.

The next example introduces a function `bin(x,s)` that takes two parameters. The first parameter is the x value you'd like to bin, and the second parameter is the bin width. The return value of the function is the position of the *left edge* of the bin—you can use the `binc(x,s)` function for bins centered on the x value.

The `smooth frequency` feature forms the sum of all y values falling into each bin. If all you care about is the overall shape of the histogram, you may supply any constant, such as `(1)`; but if you want to obtain a normalized histogram (one with a total surface area equal to unity), you need to take into account the number of points in the sample and the bin width. You can convince yourself easily that the proper y value for a normalized histogram is

```
1/(bin-width · total-number-of-points-in-sample)
```

You can use the `with boxes` style to draw a histogram (see figure 13.8), but you want to fix the width of the boxes in the graph to coincide with the bin width. (By default, the boxes expand to touch their neighbors, which leads to a faulty graphical representation if some of the internal bins are empty.) Choose a bin width of 0.1, and use the `stats` command to find the number of points in the data set. The `plot` command uses the `binc()` function, because the `with boxes` style positions its boxes *centered* at the supplied position:

```
bin(x,s)  = s*floor(x/s)
binc(x,s) = s*(floor(x/s)+0.5)

set boxwidth 0.1

stats "jitter" u 1 noout
plot "jitter"
➥    u (binc($1,0.1)):(1./(0.1*STATS_records)) smooth frequency with boxes
```

Figure 13.8 also includes a curve for the *cumulative distribution function* of the data set. We'll come back to it in section 13.3.4.

13.3.3 *Kernel density estimates and rug plots*

The apparent simplicity of the histogramming method hides some pitfalls. The first concerns the width of the bins: make them too narrow, and the resulting histogram will be bumpy; make them too wide, and you lose relevant features. There's also ambiguity in regard to the placement of the bins: is the first bin centered at 0 (or any other value) or flush left there? In particular for sparse data sets, the overall appearance of the histogram can depend quite sensitively on these details!

A better method to generate distribution curves from individual data points goes under the name *kernel density estimation*. Rather than count how many data points fall into each bin, a strongly peaked but smooth function (a *kernel*) is placed at the location of each data point. Then the contributions from all these curves are summed up and the result is plotted. Mathematically, the kernel estimate $f(x)$ for a data set consisting of N points x_i is

$$f(x) \; = \; \frac{1}{N} \sum_i^N \frac{1}{h} \, K\!\left(\frac{x - x_i}{h}\right)$$

Here, $K(x)$ is any smooth, peaked, normalized function, and h is the *bandwidth*: a measure of the width of the kernel function. A popular example is the Gaussian kernel:

$$K(x) \; = \; \frac{1}{\sqrt{2\pi}} \, e^{-\frac{1}{2} x^2}$$

As you already saw in section 3.5.2, gnuplot can generate such curves using the `smooth kdensity` functionality. It works in much the same way as the `smooth frequency` feature discussed earlier:

```
plot "jitter" u 1:(1./STATS_records) smooth kdensity bandwidth 0.05
```

The first column specifies the location; the second gives the weight each point should have. For a normalized histogram, this should be the inverse of the number of data points—because the kernel functions are normalized themselves, you don't have to worry about the bandwidth at this point as you did for histograms using `smooth frequency`. The bandwidth parameter (preceded by the `bandwidth` keyword) is optional. If it's omitted (or negative), gnuplot calculates a default bandwidth, which would be optimal if the data were normally distributed. This default bandwidth tends to be conservative (which means, rather broad).

Figure 13.9 shows several curves drawn using `kdensity` for the same data set you saw in figure 13.8, for a variety of bandwidth parameters. Studying this graph carefully, you may conclude that there is indeed a second cluster of points, located near 3.5. Note how the choice of bandwidth can hide or reveal features in the distribution of points.

One more thing: figure 13.9 uses a different representation for the individual data points. Instead of using symbols as in the jitter plot included in figure 13.8, this figure
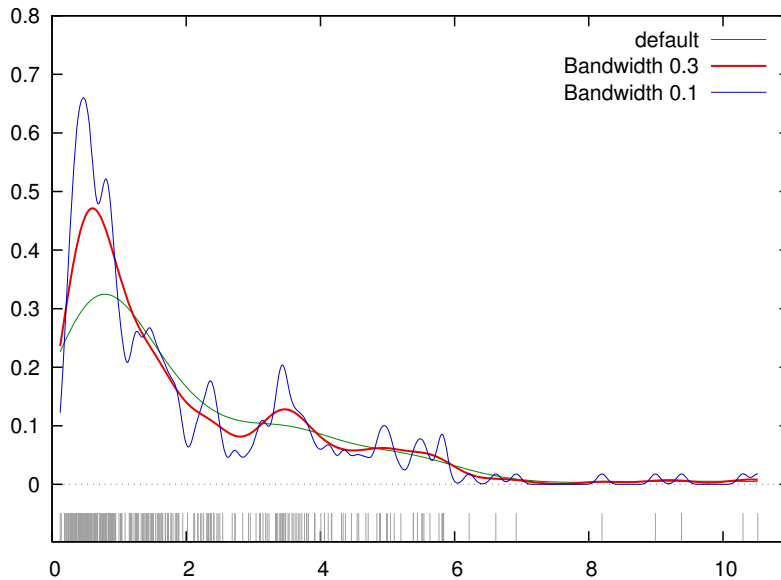
Figure 13.9   An alternative to histograms: kernel density estimates using `smooth kdensity`. Curves for three different bandwidths are shown. A bandwidth of 0.3 seems to give the best trade-off between smoothing action and retention of details. Note how it brings out the secondary cluster near *x*=3.5. Individual data points are represented through the rug plot along the bottom.

uses a *rug plot* along the bottom edge. Each data point is represented through a short, vertical line. In gnuplot, this can be accomplished through the `with vectors` style:

```
plot [:11][-0.1:0.8] "jitter" u 1:(-0.1):(0):(0.05) t '' w vec nohead
```

I prefer jitter plots, because lines in a rug plot tend to obscure each other, making it difficult to identify individual data points. But rug plots may give a somewhat neater appearance to your graphs and appear less noisy.

13.3.4 *Cumulative distribution functions*

Histograms and density estimates have the advantage of being intuitive: they show you directly the probability for a certain value to occur. But they have some disadvantages when it comes to making quantitative statements. For example, based on the histogram in figure 13.8, it's hard to determine how much relative weight is in the tail of the distribution: how likely are values larger than 4 to occur? How about values larger than 6? You can guess that the probability is small, but it's hard to be more precise. To answer such questions, you need to know the *area under the histogram* within certain bounds. In other words, you want to look at the *cumulative distribution function* (or *distribution function* for short).

The value of the cumulative distribution function at position *x* is the fraction of events that have occurred with x_i less than *x*. In figure 13.8, I showed the distribution function together with the histogram. To repeat: the value of the cumulative distribution function at position *x* is equal to the area under the (normalized) histogram from its left border to position *x*.

Cumulative distribution functions are accessible using `smooth cumulative`. The `smooth cumulative` feature is similar to `smooth frequency`: first all points are sorted in order of ascending x value, and then the sum of all y values to the left of the current position is plotted as a smoothed value. To obtain a normalized distribution function, you must supply 1/*number-of-points* as the y value. In contrast to histograms or density estimates, distribution functions don't depend on a width parameter:

```
stats "jitter" u 1 noout
plot "jitter" u 1:(1./STATS_records) smooth cumulative
```

Cumulative distribution functions can be a little unintuitive at first, but they're well worth becoming familiar with. They make it easy to answer questions such as those raised at the beginning of this section. From figure 13.8, you can immediately see that there's a 3% chance of finding a point at *x* > 6 and about a 15% chance for *x* > 4. You can also find more proof for the second cluster of points between 3 and 4: here the distribution function seems to make a jump, indicating an accumulation of points in this interval.

The cumulative distribution function is also very important theoretically and often the basis for further tests and calculations. The discussion of probability plots in section F.6 is a good example and also shows how the results from `smooth cumul` can be captured into a heredoc for further processing.

13.4   Ranked data

Imagine that I give you a list of the countries in the European Union, together with their land area (in square kilometers) and population numbers. How would you represent this information? How would you represent it *graphically*?

The particular challenge here is that *the independent variable has no intrinsic ordering*. What does this mean?

Given the name of a country, the value of the area measure is fixed; hence the name is the independent variable, and the area is the dependent variable. Usually, the independent variable is plotted along the x axis and the dependent variable is observed with it. But in this case, there's no natural ordering of the independent variable. Sure, you can order the states alphabetically by their names, but this ordering is arbitrary and bears no relationship on the data. (You wouldn't expect the size of a state to change if you gave it a different name, would you?) Also, the ordering might change if you were to translate the names to a different language. But the information that is to be displayed depends on the areas and shouldn't be affected by the spelling of the country names.
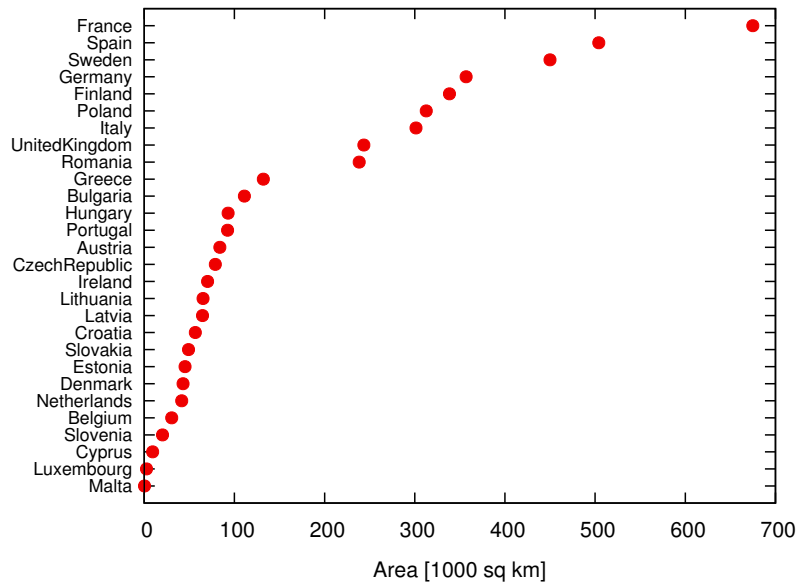
Figure 13.10   A rank-order plot. Because there's no natural ordering in the independent variable (in this case, the country names), the data has been sorted by the dependent variable to emphasize the structure in the data.

For data like this, the only ordering that's intrinsic to the data is in the values of the dependent variable. Therefore, a graphical representation of this data should be ordered by the dependent variable, not the independent one. Such plots are often called *dotplots*, but I tend to think of them as *rank-order plots*. Figure 13.10 shows an example.

   If the input file is sorted by the appropriate quantity, you can generate such plots easily using gnuplot's facility for reading tic labels from the input file. Given an input file containing the names and areas in two columns, such as this[4]

```
Malta              316
Luxembourg        2586
Cyprus            9251
Slovenia         20273
...
```

the entire plot can be generated using the following command:

```
plot "europeanunion" using ($2/1000):0:ytic(1) w p pt 7
```

The `ytic(1)` function selects the values in column 1 as tic labels for the y axis (see section 8.3.5); and the pseudocolumn 0, which evaluates the line number in the input file, is used as the corresponding vertical coordinate (see section 4.4). The inverted y range places the first line in the file at the top of the graph instead of the bottom.

---

[4] The data can be found on the Wikipedia page for the European Union.

This is the basic idea. You could plot the state names along the x axis instead, but then you'd need to rotate the labels to make sure they didn't overlap. Unfortunately, rotating the labels by 90 degrees (so that they run vertically) makes them hard to read. A good trick is to rotate them by some angle so that they run diagonally; but the initial layout, with the names running down the y axis, is the easiest to read.

What if you want to show and compare multiple data sets, such as the land area and the population? The best strategy is to declare a primary data set, which determines the ordering for all others. Figure 13.11 shows an example. The points of the secondary data set (the population in millions) have been connected with lines to make them stand out more. Additionally, the x axis has been scaled logarithmically, which is often useful with dotplots of this sort. You can see that overall the population count follows the area, but there are some notable outliers: the northern Scandinavian countries Sweden and Finland are thinly populated, whereas the so-called Benelux countries (Belgium, Netherlands, and Luxembourg) have an exceptionally high population density.

Dot- or rank-order plots are useful whenever the independent variable has no natural ordering. On the other hand, if the independent variable can be ordered—even if it's nonnumeric, such as the categories *Strong Dislike*, *Dislike*, *Neutral*, *Like*, and *Strong Like*—you should use the information in your graphs and order data points by the independent variable.
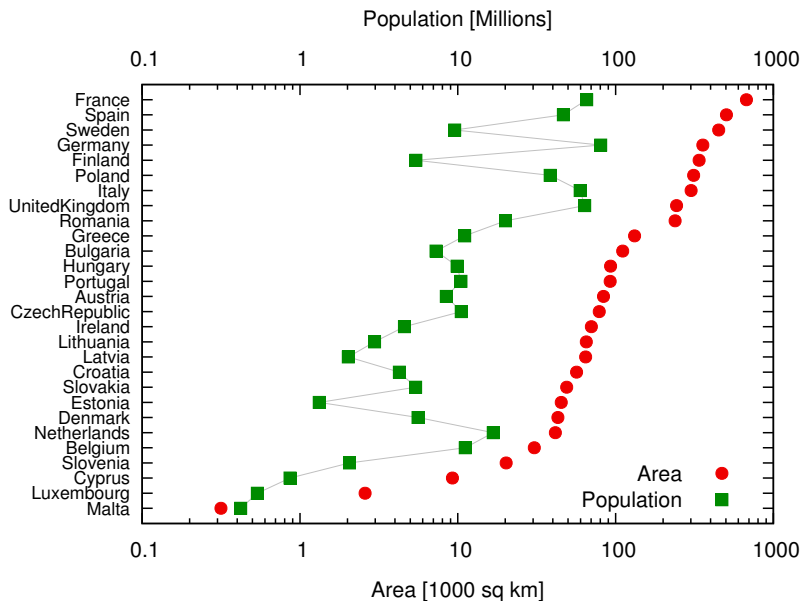
Figure 13.11   A rank-order plot displaying a primary and a secondary data set for comparison. The country names are sorted according to the primary data set (the area); the points in the secondary data are connected by lines to make them easier to distinguish. Note the logarithmic scale for the horizontal axes.

13.5 *Pie charts*

Pie charts are a way to visualize how an entity breaks down into its constituent parts. Their strongest point (and the only reason to use them) is that, by construction, they represent *all* components of the overall entity, without any pieces that might possibly be missing. Aside from that, pie charts have many well-known problems. Specifically, comparing the size of different components with similar magnitudes is difficult, as is the representation of more than a handful of components. For such tasks, some variant of rank-order plot (see section 13.4) is usually a better solution.

Gnuplot doesn't have pie charts built in, but they can be constructed through judicious use of existing gnuplot features. Given the following data set

```
# Name   Percent
A        55
B        25
C        10
D        7
E        3
```

you can generate the pie chart in figure 13.12 using the following commands:

```
set size square
set angles degrees
unset key; unset tics; unset border

beg(x) = (b=a, a=a+x, 360.*b)
end(x) = (c=c+x, 360.*c)
med(x) = (u=v+x/2, v=v+x, 360.*u)

a=b=c=u=v=0
plot [-1:1][-1:1]
➡    "pie-data" u (0):(0):(1):(beg($2/100.)):(end($2/100.)) w circ,
➡    "" u (0.75*cos(med($2/100.))):(0.75*sin(360.*u)):1 w labels
```

The challenge is that the `with circles` style requires the beginning and end angle of each pie segment (each wedge). But the data set only gives the percentage for each component, which corresponds to the angle enclosed (or subtended) by the wedge. The function `end(x)` therefore keeps a running total of all percentages encountered so far: this is the leading edge of the current slice. The function `beg(x)` does the same, but it always reports the most recent value: this is the trailing edge of the current slice. The function `med(x)` reports a value halfway between the leading and the trailing edges: this is where the label is placed. Don't forget to set (or reset) all variables to 0 before each plot!
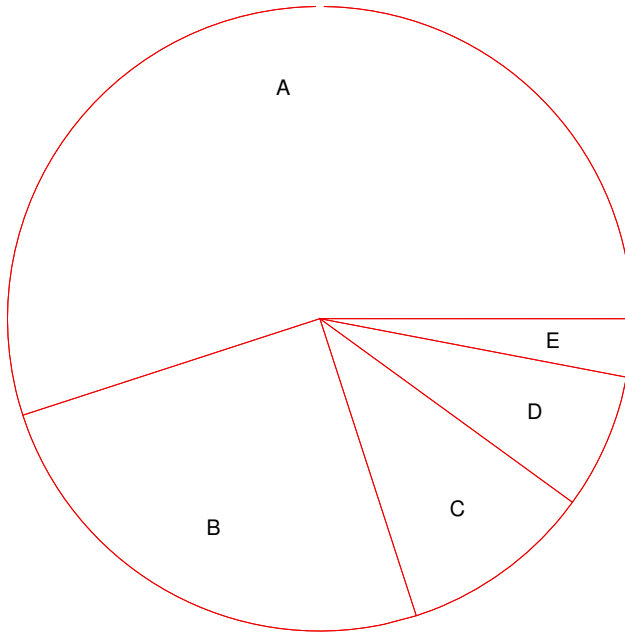
Figure 13.12   A pie chart

13.6   *Organizational issues*

The basic techniques of graphical analysis don't just range over types of graphs and when to use them, but also include some recurring administrative tasks and organizational questions, mostly relating to the handling of data sets and the files containing them. In this section, I'll make some recommendations on that.

I also want to stress that graphs are prepared for different purposes. Graphs that are prepared merely for personal use and discovery require little polishing, but this isn't true for presentation graphics that are intended to communicate results to an audience. Presentation graphics are important: they document completed work. I'll close this section with some recommendations for the preparation of attractive and valuable presentation graphics.

13.6.1   *The lifecycle of a graph*

It's helpful to have a sense for the life expectancy of your graphs: short (seconds to days, for interactive exploration and ongoing work), intermediate (days to weeks, for intermediate results), and long (weeks to infinity, for final results and public consumption). Treat graphs differently, based on their expected lives: for short-lived graphs, ease of use is the predominant concern. Nothing should stop you from redrawing the graph, changing the range or plotting style, or applying a transformation. Any amount of polishing is too much.

For graphs in the intermediate bracket, polishing is still not required, but contextual descriptions (title, units, key) become important: not necessarily to communicate such details to others, but to serve as reminders to yourself, should you come back to your work after a few days or weeks of absence. (It's amazing how quickly even essential details can be forgotten.)

For long-lived graphs, and those published or presented publicly, different rules apply. Such graphs belong to *presentation graphics* proper, and I'll have a bit more to say about that topic in section 13.7.

13.6.2 Input data files

Data files should be reasonably *self-contained* and *self-explanatory*. Let me explain.

When I was looking for data sets to use as examples for this book, I checked out quite a few publicly accessible data-set libraries on the web. In one of them, I found a data set that, according to the description on the website, contained annual sunspot numbers for a certain specified range of years. I downloaded the corresponding file together with several others data sets from the website, and only then started to examine the contents of each file in detail.

Each file consisted of only a single column, containing the dependent variable—and nothing else! Looking at these files alone, it was no longer possible to determine which one contained what, be it sunspot numbers or the price of pork bellies, or some other data set. Because the independent variable was missing as well, I couldn't even tell whether I was looking at monthly or yearly data, and, in any case, for what time frame. In other words, the sheer act of downloading a file turned it into instant garbage, by irrevocably separating it from the information that gave it meaning!

To avoid such embarrassments, it's generally a good idea to keep the basic information that's necessary to understand the data contained in the data set as part of the data file, typically as a block of comments near the beginning of the file. (I find it more convenient to have this information at the top of the file than at the end.) The information contained in such a header is more or less the same information you'd put onto a graph in the form of textual labels and descriptions. Here are some pointers:

- Most important is a description of the *file format*. It should at least include a description of the content of all columns, together with the units used. It's nice if it also contains a brief overall headline description of the file. If ancillary information would be required to re-create the data set, it should also be included: things such as parameter settings for the experimental apparatus, or starting values and the version number for the simulation program used to calculate the data. If the data was downloaded from the web, I always make sure to keep a reference to the source URL in the file.

- I also recommend being generous when it comes to the inclusion of *redundant* information. The sunspot data I mentioned earlier is an interesting example. The lack of data for the independent variable made it more difficult to use and understand the contents of the file. Given the starting value and the increment,

it's trivial to reproduce the values, but it's generally much more convenient to have all this information available already.

- The ability to *reproduce* a data file if necessary is critical. If you combine data from several sources into a single data set, or manually edit a data set to remove glitches, keep the originals. And stick a note in the combined file explaining its origins. More likely than not, you'll have to come back and do it all over again at some point. Remember: if you delete the originals, they're gone forever.

- Unless there are extenuating circumstances (and even in most cases when there are), *prefer plain text* files over binary formats. The advantages of plain text are just too numerous: plain text is portable across all platforms (despite the annoying newline character issue) and can be manipulated using standard tools (editors, spreadsheets, scripting languages). A corrupted text file can be repaired easily—not so for a binary file. A text file is by construction more self-explanatory than a binary file will ever be. And finally, text files compress nicely and therefore don't take up much disk space.

Some statistics packages keep their data sets in proprietary binary formats. I find the constant need to invoke a special, proprietary program just to view the contents of a file a major inconvenience, and the dependence on the operations provided by the package for all data-manipulation tasks is an unwelcome hindrance to my work.

If you have a legitimate need for a binary file format, at least use an existing standard, for which quality libraries are easily available. Ad hoc binary formats are almost certainly a bad idea.

Something we don't tend to think of often are clerical errors: typos, incorrectly named files, data entered into the wrong column, wrongly recorded units—that sort of thing. They're apparently more common than you might think: Cleveland and Wainer[5] give interesting examples of odd entries in well-known data sets and provide convincing arguments that explain these entries as due to clerical errors, such as the interchange of digits or the inadvertent switching of columns for the data points in question.

Finally, make sure critical data is backed up frequently. (I once accidentally deleted data that had taken weeks to compute. Praise your system administrators!)

13.6.3   *Output files*

The most important advice (again) is to make plots reproducible. Don't just export to a printable format and move on. It's almost guaranteed that you'll want to redraw the graph, with minor modifications, as more data becomes available or the understanding of it grows. Always save the plotting commands and options to a file before moving on.

Use an appropriate file format. Bitmaps (such as PNG) are simple and portable but can't be resized without loss of quality. Whenever there is a chance that you (or

---

[5]   See section 6.4 in *Visualizing Data* by W. S. Cleveland (Hobart Press, 1993) and the introduction to *Graphic Discovery* by H. Wainer (Princeton University Press, 2005) for details.

whoever is using your graphs) might need to resize a graph, make sure to use a vector format (such as PDF). This is particularly true for print publishing: not only will graphs routinely be resized, but only vector formats will take advantage of the higher resolution offered by printing devices.

Decide whether you need to prepare your graphs in color, black and white, or half-tone (grayscale). Consider the use of stylesheets (see section 12.6) to target graphs to different display channels. I routinely create different versions of the same graph (using different stylesheets) so I'm set for all possible applications. Adopting a naming convention helps to keep different versions organized.

13.7 *Presentation graphics*

This book isn't primarily about presentation graphics, but about graphical analysis. There's already plenty of advice on presentation graphics, and you'll have no difficulty finding it, but not much of it appears to be based on rigorous studies. Nevertheless, the advice is often worded assertively, if not emotionally, and there's an unfortunate (and ultimately unhelpful) tendency toward the derision of work considered inadequate. Given the lack of rigorous evidence, tastes and personal opinions naturally play a large role.

I don't intend to add to this debate. Instead, I'd like to present a list of reminders concerning details that are easily (and inadvertently) forgotten when preparing a graph for publication. Most of them concern the inclusion of *contextual* information, which can't be inferred from the graph itself, but which is nevertheless important for understanding. By *publication* I mean any form of wider distribution of the graph: anything leaving the immediate group of coworkers who were involved in its creation, and in particular any use with a long expected lifetime.

These aren't commandments, but reminders. Use your own good judgment:

- Axes should be labeled. The labels should describe the quantity plotted and should include the units of measurement. Don't leave this information to a separate figure caption. Keep in mind that the graph may be separated from its context, in which case information contained only in the caption may be lost. The caption should be used to describe salient features of the graph, not to give vital information about the plotted data.
- Choose meaningful, self-explanatory labels. If you must use abbreviations that aren't in common use, explain them, ideally on the graph itself, not in the caption (see previous item). (In a recent book on data graphing, I found a figure of a histogram in which the buckets were labeled *Married*, *Nvd*, *Dvd*, *Spd*, and *Wdd*. I marveled about the possible meaning of *Nvd* for quite a while. The abbreviations were explained neither in the text nor in the figure caption.)
- If there's more than one line in a graph, explain what each of the lines represents, either through a key or by using an explicit label referring to each line. If you use a key, make sure the key entries are meaningful.

- If there's ancillary information, consider placing it onto the graph, rather than supplying it only in the caption.
- If at all possible, make sure text labels, including the key, don't obscure the data. If necessary, move or rewrite them.
- When publishing a false-color plot, *always* include the associated color scale in the graph. No matter how intuitive the chosen palette may appear to you, remember that there's no universal and unambiguous mapping from numbers to colors and vice versa.
- Describe the meaning of error bars. Do they show the calculated standard deviation of the sample population? Do they represent interquartile ranges? Or do they indicate the limits of resolution of your experimental apparatus? This information can't be inferred from the graph but must be explained through textual information.
- Use an appropriate measure of uncertainty. Don't show standard deviations for highly skewed or multimodal populations just because they're easy to calculate.
- Choose meaningful plot ranges. The graph should display the *relevant* part of the data, and the data should make use of the entire available space. It may be necessary to use more than one graph to convey all the relevant detail: it isn't unreasonable to have one graph to exhibit the overall shape of the data and one that zooms in on a particular region at greater resolution. Keep in mind that the relevant information in a data set is sometimes the trend and sometimes the absolute value. Make sure the chosen plot range is appropriate given the context.
- Consider logarithmic scales if the variation in the data is too great to be represented otherwise.
- Don't be shy about choosing a different font if the default font looks ugly. Given the importance of textual information on a graph (labels, tic marks, keys), make the necessary effort to ensure that all text is clearly readable even after the graph has been reproduced and possibly reduced in size a few times. (On the other hand, making labels too big or too bold can easily ruin the overall appearance of a plot. Experiment!)

  In general, sans serif fonts (such as Helvetica) are preferred for standalone pieces of text, whereas serif fonts (such as Times Roman) are considered more suitable for body text. Because labels on a graph tend to be short, this suggests using a good sans serif font in plots. (I also find that sans serif fonts enhance the visual clarity of a graph, whereas serif fonts don't, but others may disagree. Judge for yourself.)

- Use an appropriate aspect ratio. Human perception seems to have an affinity to figures with an aspect ratio of roughly 10:7, in *landscape* orientation (that is, wider than tall). In particular, it's generally not a good idea to use a square graph unless the symmetry of the data demands it.

- Don't use bitmapped graphics formats (PNG, GIF, JPG) in print publications. Use vector formats such as PDF, PostScript, and EPS instead.
- Proofread your graphs. Common error spots include typos in textual labels, switched data sets or interchanged labels, and omitted qualifiers (milli-, kilo-, and so on) for units.

13.8 Summary

Understanding data with graphs typically involves certain recurrent tasks and questions. How one quantity depends on another is such a question; how a bunch of data points are distributed is another. Other recurrent problems have to do with data that spans many orders of magnitude, and data that isn't strictly numeric.

In this chapter, we discussed the standard types of graphs to investigate such questions, and showed how to create them using gnuplot. All these methods and plots are very general—in the next chapter, we'll concentrate on some more specific problems and explore them in depth.

14

Topics in graphical analysis

This chapter covers
- Techniques for time-series plots
- Techniques for multivariate data sets
- Techniques to improve visual perception

The previous chapter discussed some fundamental graphical methods that are very generally useful. In this chapter, I'll select a few tasks that are more specific and explore them in greater depth.

Because of their immense practical importance, our first topic will be the handling of time series. In particular, you'll see some clever ways to smooth a time series using only gnuplot commands. The next topic concerns multivariate data sets: data sets with more than two or three different attributes. A variety of techniques are available for such data, and some of gnuplot's relatively new features facilitate the visual exploration of such data.

Next, we'll discuss ways to enhance the visual perception of graphs. You can aid analysis and discovery by preparing graphs such that noteworthy features are more easily recognized. This topic isn't frequently discussed, but some awareness of the issues involved can greatly improve your results.

14.1 Techniques for time-series plots

Time-series plots are probably *the* most common graph in practice: whenever you want to monitor a quantity, you're dealing with a time-series plot of some form. This section introduces some operations that are frequently useful when dealing with time series and shows you how to coax gnuplot into performing those operations.

14.1.1 Plotting an Apache web server log

Technically, a *discrete time series* consists of measurements at *equally spaced* time intervals. The requirement that all time steps are of the same size is important: several time-series operations and transformations rely on this property.[1]

But raw data is frequently not of that form. Web server logs (and other event logs, generally) merely record the timestamp when an event occurred—they're *asynchronous*. Turning this data into a (discrete) time series means counting the number of events per (equally sized) time interval.

In general, I strongly suggest that you do this kind of data munging outside of gnuplot, using the programming language of your choice. But in a pinch, it *can* be done with gnuplot. The following listing shows the beginning of an Apache web server log file (lines are truncated on the right).[2]

Listing 14.1   Part of a web server log; incomplete data for figure 14.1

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/Main/Doub
64.242.88.10 - - [07/Mar/2004:16:06:51 -0800] "GET /twiki/bin/rdiff/TWiki/Ne
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivis
64.242.88.10 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/TWiki/Wik
64.242.88.10 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/Main/DCCA
...
```

The timestamp is in the fourth column, and gnuplot's time-series mode is required to parse it properly. The timestamp must be truncated to the desired interval for the time series. Then the number of events in each interval must be counted. You can use `smooth frequency` (see section 3.5.3) for the latter. When truncating the timestamp, keep in mind that gnuplot treats it as a *string*, so you need to use string operations for all manipulations. The next listing truncates the timestamp to the hour: that is the step size of the resulting time series (see figure 14.1).

Listing 14.2   Commands for figure 14.1 (file: access-log.gp)

```
set xdata time                              ◁—— Turns on time-series mode
set timefmt "[%d/%b/%Y:%H:%M:%S"           ◁—— The left bracket is part of the format.
set xtics format "%d%b\n%H:00" rotate

set ylabel "Hits per hour"                          Truncates the
                                                    timestamp to the
plot "access_log" u (strcol(4)[1:15] . ":00:00"):(1) s f w l  ◁— first 15 characters
```

---

[1] If observations occur asynchronously, so that the intervals are of varying sizes, the data is said to form a *continuous* time series. I recommend reserving the term *time series* for sets of equally spaced observations only, and speaking of an (asynchronous) *event stream* otherwise.

[2] You can find the complete file at www.monitorware.com/en/logsamples/apache.php.
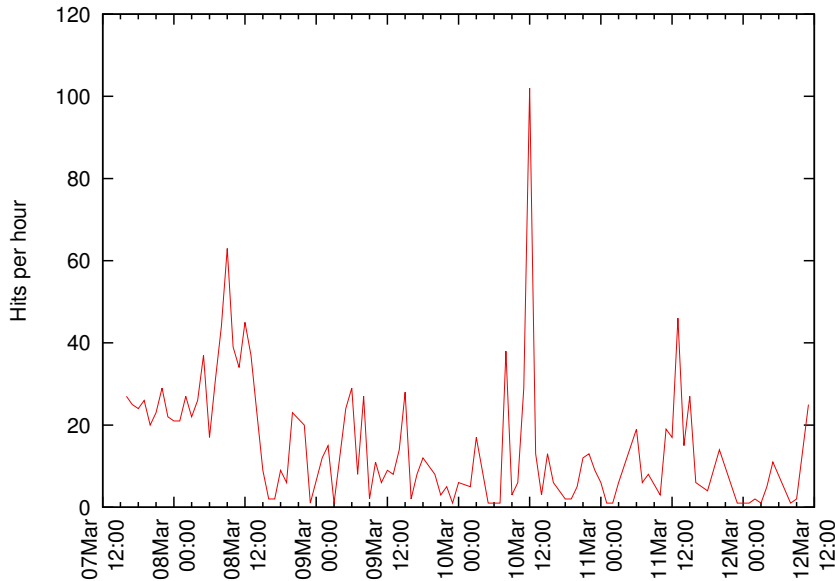
Figure 14.1   Web traffic (hits per hour). Compare listing 14.2.

Because this method relies on gnuplot's timestamp parsing, it's rather brittle and doesn't deal well with missing or malformed entries in the log file. If you encounter such problems, it's probably easiest to preprocess the raw data in a general programming language.

14.1.2 Smoothing and differencing

The transformation most frequently applied to a time series is probably some form of *smoothing*, to remove noise and bring out the underlying trend more clearly. Another often useful operation is the *differencing* of a time series, wherein the original data is replaced by the difference between any two consecutive observations. Using session variables and the comma operator, these tasks can be accomplished as part of a `plot` command.

EXPONENTIAL SMOOTHING IN GNUPLOT

Probably the most familiar smoothing methods are the *floating* and *trailing averages*, which replace each data point with the average over some number of adjacent data points. In the first case, the averaging interval is symmetrical, which means the smoothed curve can never extend all the way to the most recent value in the time series; for this reason, the trailing average uses a one-sided interval. Both methods have some problems, not the least of which is that they need to keep track of all the data points in the averaging interval individually.

An often more favorable and much simpler smoothing method is known as *exponential smoothing* or the *Holt-Winters method*. It replaces each data point not with the average, but with the following quantity:

```
s[t] = a * x[t] + (1-a) * s[t-1]
```

Here, `s[t]` is the smoothed value at time step `t`, and it's calculated as a mixture between the observed value `x[t]` at that time and the most recent smoothed value `s[t-1]`. The smoothing parameter `a` is a number between 0 and 1: the closer to 1, the less the data is smoothed.

Exponential smoothing can be implemented as an inline transformation in gnuplot:

```
s=0; a=0.5
plot "data" u 1:(s=a*$2 + (1-a)*s) w l
```

This works because the value of an assignment expression is the assigned value. It may be convenient to bundle the operation into a function, as in the following listing (also see figure 14.2).

Listing 14.3   Single exponential smoothing: see figure 14.2 (file: smoothing.gp)

```
hw(x,a) = a*x + (1-a)*s
s=1; plot "unsmoothed" u 1:2 w l, "" u 1:(s=hw($2,0.2)) w l lw 2
```

It's important that the variable `s` is initialized before the `plot` command. For best results, it should be initialized to a value that is similar to the first value in the data set.
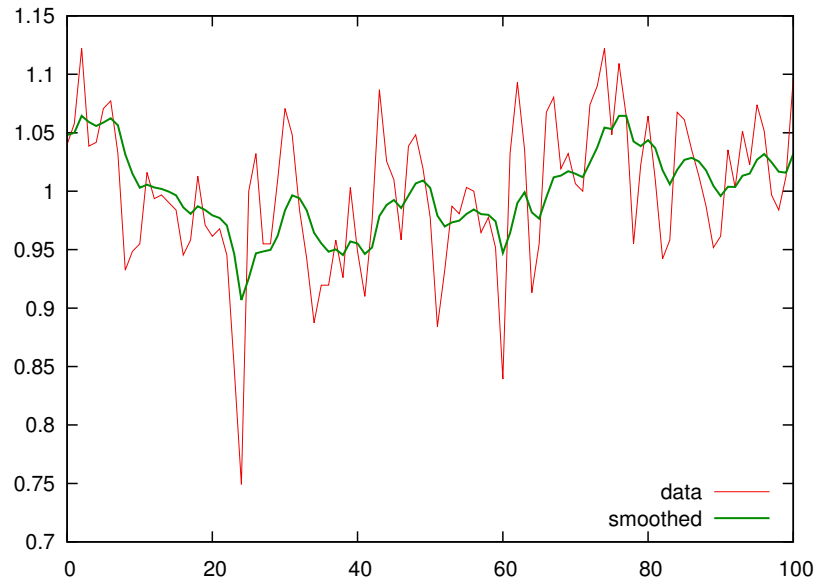
Figure 14.2   A noisy time series and a smoothed trend line. The trend was calculated using single exponential smoothing: `s=a*x + (1-a)*s`, with a smoothing parameter of `a = 0.5`.

TIP   Exponential smoothing is an exceedingly simple yet robust and versatile method to smooth a time series and bring out an underlying trend. It can be implemented using a convenient inline transformation in gnuplot. Don't forget that the temporary variable must be initialized before invoking the `plot` command.

DOUBLE EXPONENTIAL SMOOTHING

Single exponential smoothing (as in listing 14.3) works best for stationary time series that don't exhibit a long-term trend. If a trend is present, the smoothed curve tends to lag behind and doesn't constitute a good approximation to the raw data. For data sets that include a trend, it may therefore be necessary to use a smoothing scheme that explicitly includes a trend. One such method is *double exponential smoothing*:

```
s[t] = a*x[t] + (1-a)*(s[t-1] + u[t-1])
u[t] = b*(s[t] - s[t-1]) + (1-b)*u[t-1]
```

Here, `s[t]` is the smoothed value at time step `t`, and `u[t]` is the smoothed trend at the same time step. There are now two smoothing parameters, `a` and `b`, for the value and the trend, respectively. This method, too, can be implemented as an inline transformation in gnuplot:

```
a=0.5; b=0.5
s=0; u=0; q=0

plot "data" u 1:(s=a*$2+(1-a)*(s+u), u=b*(s-q)+(1-b)*u, q=s) w l
```

Obtaining a good approximation now requires finding suitable values for both smoothing parameters, `a` and `b`. This may involve a fair bit of trial-and-error experimentation!

Again, the formulas can be wrapped into a function:

```
hw2(x,a,b) = (s=a*x+(1-a)*(s+u), u=b*(s-q)+(1-b)*u, q=s)

s=0; u=0; q=0
plot "data" u 1:(s=hw2($2,0.5,0.5)) w l
```

Keep in mind that a gnuplot function doesn't create a scope for local variables: all variables are global. The function finds the variables it writes to (such as `s`, `u`, and `q`) as session variables in the current gnuplot session. It's a mistake to supply such variables as arguments to the function. But because the function expects to find these variables in the gnuplot session, it's essential that the variables be created (that is, initialized) before the function is invoked.

DIFFERENCING A TIME SERIES IN GNUPLOT

Another transformation that is often useful is to *difference* the time series. This amounts to replacing each entry in the series with the difference between two consecutive values; the resulting series is shorter by one record.

In gnuplot, this can be accomplished through an inline transformation as well:

```
u=0
plot "data" u 1:(v=$2-u, u=$2, v) w l
```

This works because gnuplot treats the comma (,) as an operator, similar to the way the C programming language does. It separates different expressions, evaluating each in turn (left to right), and returns the value of the last (rightmost) one.

Using the comma operator, it's possible to define gnuplot functions comprising multiple statements. Traditionally, the body of a gnuplot function had to consist of a single expression, but the comma operator provides a way to circumvent this limitation. You saw an example in the previous section on double exponential smoothing, in the definition of the function `hw2(x,a,b) = (s=a*x+(1-a)*(s+u), u=b*(s-q)+(1-b)*u, q=s)`.

> **TIP** The comma operator lets you combine multiple statements into a single expression. Statements are evaluated left to right, and the value of the rightmost statement is returned as the value of the entire expression.

14.1.3 *Monitoring and control charts*

One of the most common applications of time series and time-series graphs is the monitoring of a metric over time—usually to detect whether the observed system is behaving as expected. Two tasks commonly arise in such applications: data needs to be normalized, so that the behavior can be compared across different metrics. Additionally, there's often an interest in identifying unexpected behavior, such as outliers and drifts. Let's discuss them in turn.

MAKING DATA COMPARABLE

Imagine that you're in charge of some operation. It may be a factory or production plant, it may be a data center, or it might even be a sales department. What matters is that you want to monitor the most relevant metrics constantly.

To keep things simple, let's assume it's a manufacturing operation where just three parameters really matter: overall productivity (units per hour), the completion time for each unit (in minutes), and the defect rate (the ratio of defective units to the total number of units produced). You might want to plot them together on a control chart so you can immediately see if one of them starts running out of the allowed zone. But most of all, you want to be able to *compare* them against each other: is one parameter consistently performing better than the others? Is one of the parameters on a slippery slope, getting worse and worse, relative to the others? And so forth.

A naive way to achieve this effect is to plot the three parameters in a single chart. The result is shown in figure 14.3 and is probably *not* what you want!

The problem is that the three parameters assume very different values: productivity is typically around 10,000 units per hour, assembly time is on the order of an hour, and the defect rate should be very, very small. Before you can compare them, you therefore need to *normalize* them.

Making metrics comparable by normalizing them usually involves two steps: first, they are *shifted* by subtracting a constant offset (such as a long-term average), and then they are *rescaled* by dividing them by their typical range of values (as expressed in the standard deviation, for example). The resulting normalized metrics are all centered on or near zero, and their fluctuations are of roughly unit size. (Also see figure 14.4.)
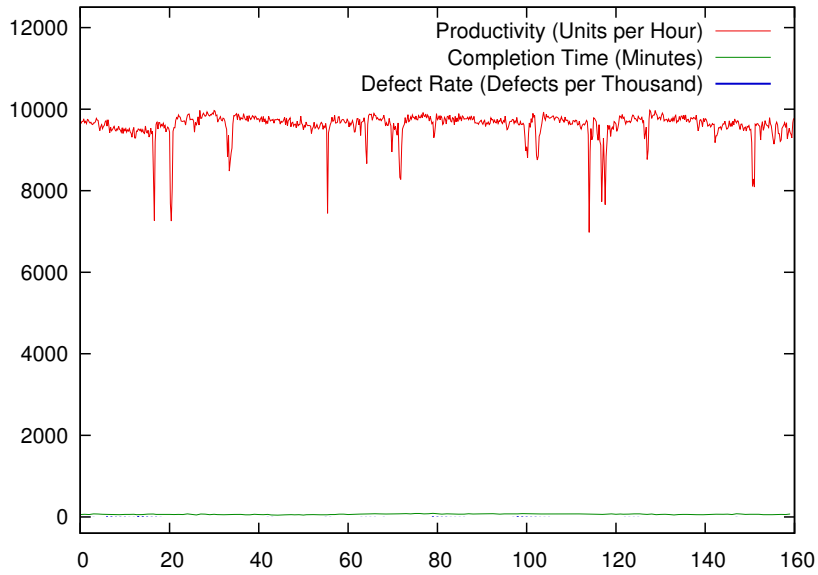
Figure 14.3   A control chart showing three very different quantities simultaneously. What's wrong with this picture?
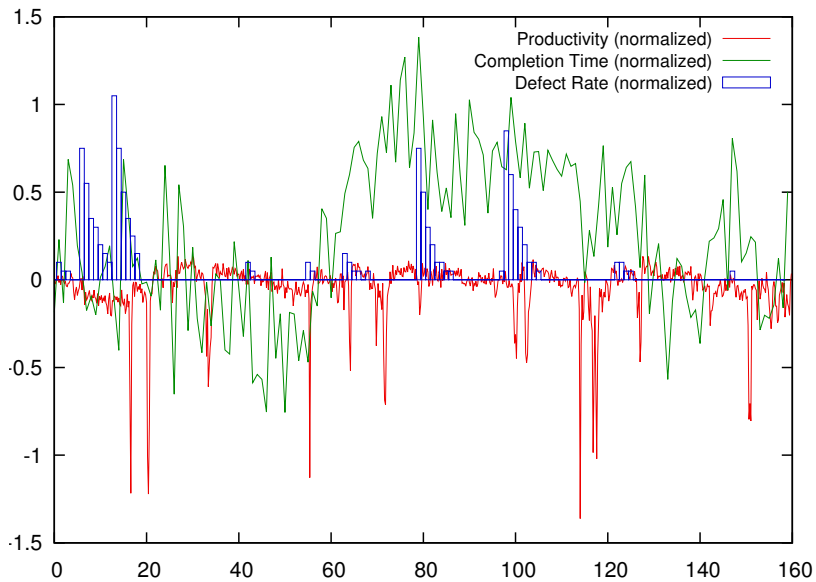
Figure 14.4   A control chart showing normalized metrics. The data is the same as in figure 14.3.

More generally, normalizing metrics always involves a linear transformation of the form

$$z \ = \ \frac{x - m}{s}$$

where *m* is a measure for the typical *value* of the metric in question, and *s* is a measure for the typical *spread* or range of values. If the metric is well-behaved, with relatively regular and symmetric fluctuations, then the mean and standard deviation are popular choices for *m* and *s*; but if the metric is strongly affected by outliers, then the median and the interquartile range may be more robust choices. Another possibility is to use the smallest possible value for *m* and the range between the greatest and the smallest value for *s*: the resulting normalized quantity isn't centered on zero but is restricted to the unit interval.

Because normalization involves rescaling by the typical range of values, the resulting quantities are all *dimensionless* and are therefore directly comparable (in contrast to the original data, whose numerical values depended on the units of measure that were used).

IDENTIFYING OUTLIERS

Identifying outliers is in some ways a complementary activity to normalizing data, because it amounts to finding data points that are *different* than the rest. When you're dealing with time series, there is one additional issue: the baseline itself is changing constantly!

In the context of time-series analysis, an *outlier* is a data point that is further away from the trend than is typical. In the absence of other information about the data, the smoothed data is the best guess at the true trend, and is therefore used as a baseline.

Identifying outliers in a time series thus involves three steps:

1 Establish a baseline by finding a smooth trend for the time series.
2 Find the range of typical deviations of the raw data from the trend.
3 Tag as outliers those records that differ by more than the typical range from the trend.

Figure 14.5 shows the situation after the first step. The data consists of the daily number of calls made to a call center, and a trend has been added using single exponential smoothing:

```
s=0; a=0.25
plot "callcenter" u 1:2 w l, "" u 1:(s=a*$2+(1-a)*s) w l lt 3 lw 2
```

The next step involves finding the typical range of the fluctuations of the raw data around the trend. It can be found using the `stats` command:

```
s=0; a=0.25
stats "callcenter" u 1:(s=a*$2+(1-a)*s, $2-s)
```

It's important that the last expression in the `using` expression is `$2-s`: this is the momentary *deviation* of the actual data from the smoothed trend. For simplicity, I used
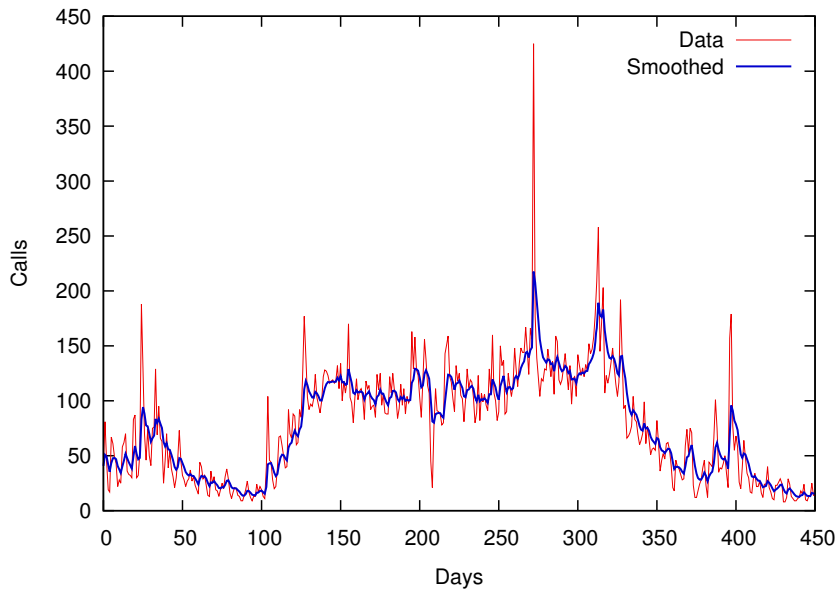
Figure 14.5   **Number of daily calls to a call center. The figure shows the raw data and the smoothed trend line.**

the standard deviation (as reported by `stats`) as a measure for the range—for this data set, it doesn't differ much from the interquartile range. The value of the standard deviation is roughly 21, and I chose to label all records that differ by more than twice that as outliers.

Figure 14.6 shows the data together with the trend, as before. Outliers as identified by the foregoing procedure are marked with symbols, and the typical range of fluctuations is indicated using a grey band. The complete `plot` command is as follows:

```
s=0; a=0.25                          ⟵——— Initializes temporary variables
plot "callcenter" u 1:2 w l,
     "" u 1:(s=a*$2+(1-a)*s) w l lt 3 lw 2,                    ⟵ Smooth trend
     "" u 1:(s=a*$2+(1-a)*s, abs($2-s)>2*STATS_ssd_y?$2:NaN) w p lc 2 pt 7,
     "" u 1:(s=a*$2+(1-a)*s, s+2*STATS_ssd_y):(s-2*STATS_ssd_y)
        w filledc fs transparent solid 0.5 lc rgb "dark-grey"   Grey band
```

Raw data ⟶ (second line)
Outliers ⟶ (fourth line)

CUSUM PLOTS

Occasionally, you may find yourself monitoring a quantity that has a well-defined target value (or *setpoint* in control-theory lingo): for instance, you may want to keep a power supply at a precisely determined temperature. In such cases, it's essential to detect *early* whether the metric has drifted away from its setpoint. This may be difficult, because any noise in the data is likely to obscure a small but systematic change quite effectively. Consider figure 14.7. Would you say that the system has undergone any change from its
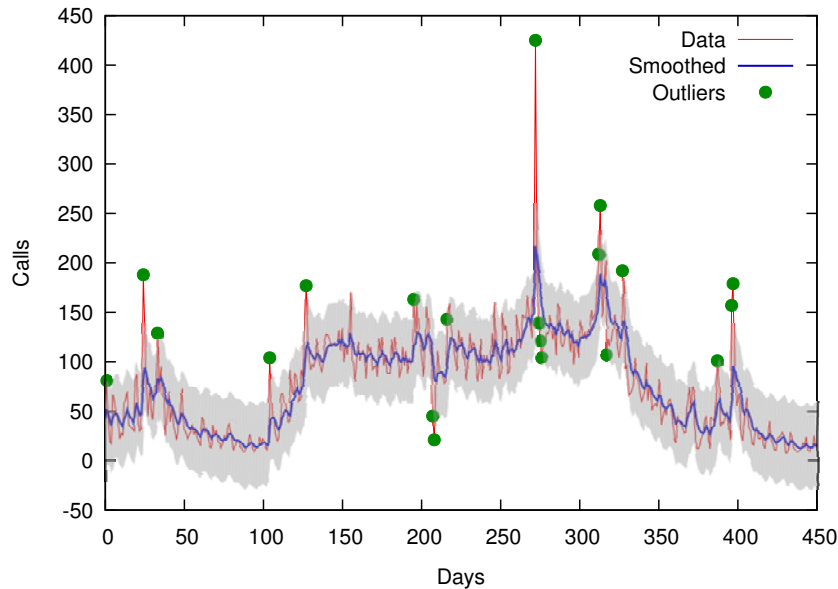
Figure 14.6   Same data as in figure 14.5. In addition to the smoothed trend line, the region of normal fluctuations is indicated (shaded); outliers outside this region are indicated with symbols. (See the text for details.)

setpoint at $y=5$ during the observed interval? (The horizontal lines at $y=3$ and $y=7$ are intended to indicate the *permissible range* for the observed metric.)

The trick is to amplify the effect by keeping an accumulating sum of all deviations from the intended setpoint. As long as the system is centered on the correct value, the deviations will cancel each other out. But a persistent deviation, even if it's small, will quickly lead to the build-up of noticeable error in the *cumulative sum*. Plotting this quantity in a *cusum chart* provides a low-tech yet remarkably sensitive diagnostic instrument to spot even small but persistent deviations from the setpoint.

In gnuplot, you can obtain a cusum chart through a simple inline transformation. I used the following commands to create figure 14.8, which clearly shows that *something* changed near time step 15:

```
setpoint = 5
s=0; plot [][-1:11] "cusum" u 1:2 w lp pt 7, "" u 1:(s=s+($2-setpoint)) w l
```

In this case, the data has been created synthetically as Gaussian noise, with mean $\mu=5$ and standard deviation $\sigma=1$ for the first 15 observations, but with mean $\mu=5.5$ for the subsequent data points. Looking at the raw data in figure 14.7 doesn't reveal this permanent shift; even comparison with the reference lines above and below doesn't suggest any trouble. But the cumulative error in figure 14.8 is a dead giveaway!
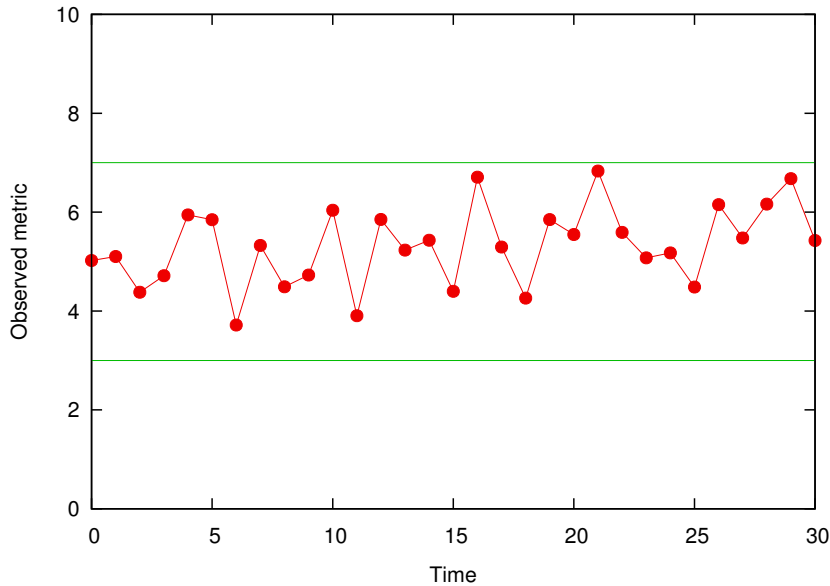
Figure 14.7   A control chart. The target value for the observed metric is 5. The horizontal lines indicate the permitted range around the setpoint.
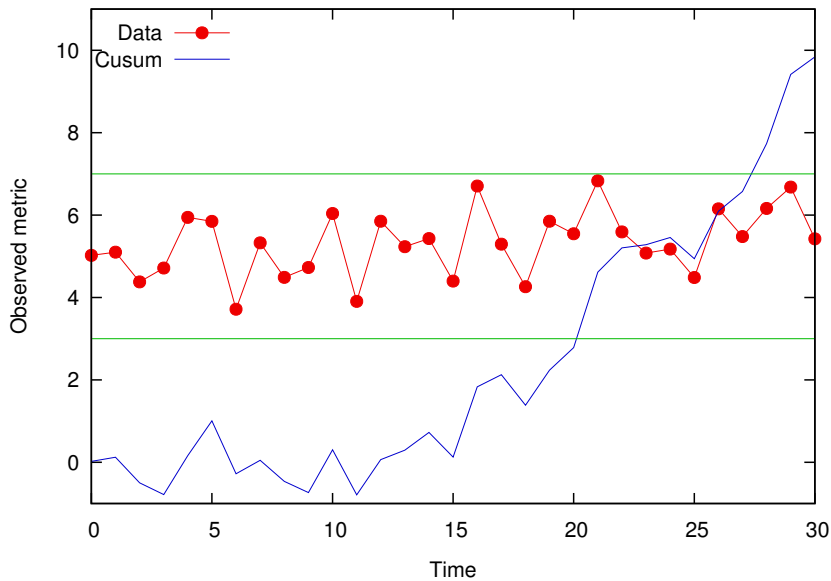
Figure 14.8   Same data as in figure 14.7, but shown together with the cumulative deviation from the intended setpoint of 5. The sudden and persistent increase in the cumulative error indicates that the system is shifted away from its target value—even though this isn't immediately visible in the raw data.

When using cusum charts, the cumulative error shouldn't be calculated "since the beginning of time." Even when the system is perfectly centered on its setpoint, the cumulative error will exhibit arbitrarily large deviations in the long run. For this reason, the cumulative error should be reset frequently and calculated over only a relatively small number of recent observations (a few dozen).

14.1.4 *Changing composition and stacked curves*

As a final example of a time-series problem, let's consider a situation where the composition of an aggregate changes over time, while at the same time, the magnitude of the entire aggregate also changes. The total human population, for example, changes over time—but so does its breakdown by continent. Representing both aspects (overall size and composition) simultaneously is a difficult problem. This section introduces four complementary solutions. This will also give me an opportunity to demonstrate gnuplot's for and sum constructs. (See the next listing—multiplot mode will be discussed in section E.1.)

Listing 14.4   Commands for figure 14.9 (file: composition.gp)

```
set xtics 26
set mxtics 2

set xrange [0:150]

set multip layout 2,2 rowsfirst margins 0.1,0.95,0.15,0.9 spacing 0.1,0.05
```
Turns on multiplot mode ◁

Upper left: individual product numbers
```
set key horizontal samplen 2 at screen 0.55,0.95 width -7 center
set xtic format ""
set ylab "Absolute" offset 1,0
tx(k) = (k==2 ? 'Product 1' : ''.k-1)
plot for[k=2:5] "composition" u 1:k t  '' . tx(k) w l
unset ylab
unset key
```
Uses a function to supply a description for key entry ◁

```
plot for[k=2:5] "composition" u 1:(sum[j=2:k] column(j)) w l
```
◁ *Upper right: stacked product numbers. Uses the sum feature.*

Lower left: individual fractions
```
set xtic format "%g"
set xlab "Individual"
set ylab "Fractional" offset 1,0
plot [][0:.6] for[k=2:5] "composition"
➡    u 1:(column(k)/(sum[j=2:5] column(j))) w l
unset ylab
```
◁ *Divides by the sum over all products*

Lower right: stacked fractions
```
set xlab "Stacked"
plot [][0:1.025] for[k=2:5] "composition"
➡    u 1:((sum[j=2:k] column(j))/(sum[j=2:5] column(j))) w l

unset multiplot
```
◁ *Turns off multiplot mode*

◁ *Divides the sum by the overall sum*

Let's consider a company that manufactures four products, numbered 1 through 4. The absolute weekly production numbers (in thousands of units) for each product separately are shown in figure 14.9, upper left. Such a plot makes it easy to follow the behavior of each product individually, but as the number of products grows, such a plot can become cluttered. Moreover, it isn't easy to see how *total* production changes over time. You could add a curve, but it would obviously be on a very different scale than the individual contributions. Using a second vertical scale (one for the individual numbers and one for the totals) is possible but leads to even more clutter.
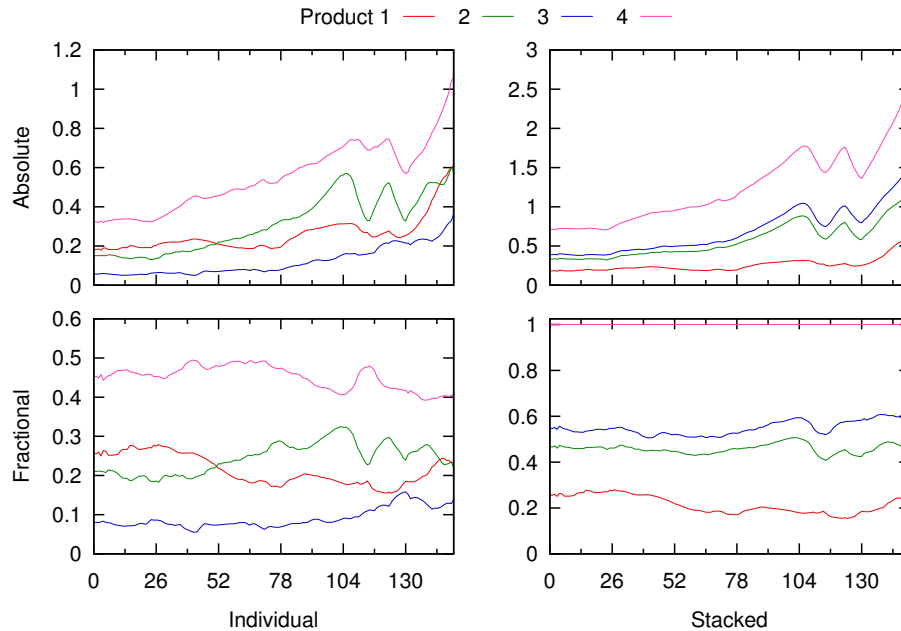
Figure 14.9   Four different ways to present changes in composition. Each line corresponds to the number of units produced for one of four different products. The top row shows absolute numbers of units produced; the bottom row shows fractions relative to total production. In the left column, each line is drawn separately; in the right column, the lines are stacked, giving a cumulative view into the data. See listing 14.4.

For these reasons, you occasionally see the data represented in a *stacked* plot, as in figure 14.9, upper right. Here, the numbers for all products have been stacked on top of one another, so that the curve for (say) product 2 represents the total manufactured output for products 1 and 2 combined. By construction, lines in such plots never cross, and hence the graphs always have a neat appearance—which makes them popular in general-interest publications. Nevertheless, they have significant problems. Only two curves are easy to interpret: the bottom one (product 1) and the top one (total production). It's exceptionally hard in such a stacked graph to estimate how the contributions from the individual components change over time. For

example, how much of the sudden increase in total production after week 130 is due to product 4? It's basically impossible to tell from this graph—you have to go back to the previous panel. Similarly, from this graph, it isn't easy to see that the double-hump configuration starting in week 100 is due exclusively to product 2. (Judging the distance between closely spaced curves that vary a lot over the plot range is exceptionally difficult—see section 14.3.2.)

If you're interested in the relative changes in composition, then—following the principle to "plot what's relevant"—you shouldn't plot the absolute numbers but rather their fractional contribution to the total. That's what the panel at bottom right in figure 14.9 does. This panel is also a stacked graph, but it shows the relative contribution of each component, not the absolute numbers. It shares with the previous panel that it's guaranteed to look clean and that the individual contributions are hard to estimate. In addition, this diagram completely *hides* any information about total production. In particular, the dramatic increase toward the end of the observation time frame is invisible! Although diagrams of this form have a lot of intuitive appeal, they convey information poorly.

Finally, the panel at bottom left shows the relative contributions, separately for each component again. It shares with the first panel (top left) that it appears cluttered but that the information on individual products is easy to discern.

The appearance and information content of stacked graphs (like the ones on the right side of figure 14.9) depend strongly on the *order* in which the individual curves (here: product lines) are stacked. If there is some natural ordering, then such stacked graphs may, in fact, be the best way to represent the information. Figure 14.10 shows one such attempt. In this figure, the products have been ordered by their contribution to the total output, based on the (reasonable) assumption that products that contribute more are more important. It makes sense to include curves that show less variation and are flatter earlier in the graph, to retain a more stable baseline for the curves that are added later to the stack. For this reason, product 1 has been included *before* product 2, although it contributes overall slightly less: this way, the infamous double-hump affects fewer curves in the overall stack.

Because the components for figure 14.10 had to be picked individually, the `sum` facility was much less useful:

```
plot [0:150] "composition" u 1:5 t "Product 4" w l lt 4,
➥            "" u 1:($5+$2) t "1" w l lt 1,
➥            "" u 1:($5+$3+$2) t "2" w l lt 2,
➥            "" u 1:($5+$3+$2+$4) t "3" w l lt 3
```

> **TIP** Stacked graphs look appealing but often convey information poorly, unless you can find a natural order in which to add curves to the stack.
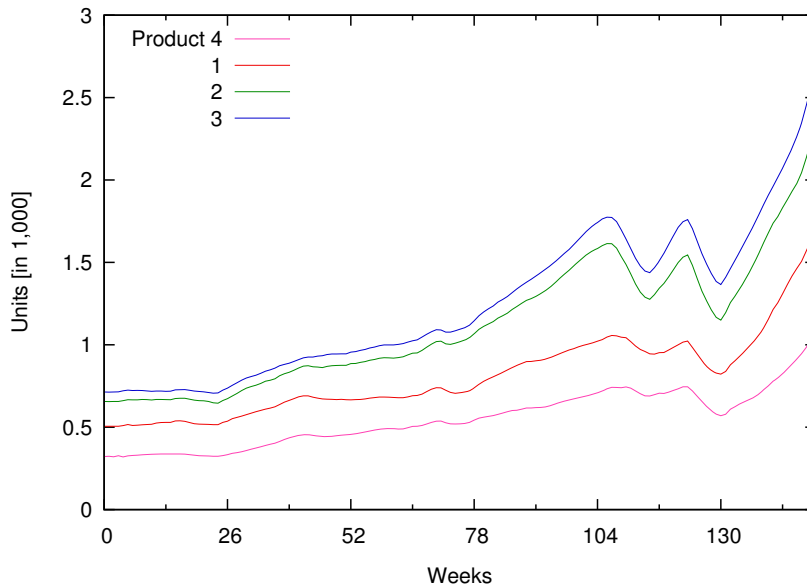
Figure 14.10   A stacked graph of absolute production numbers. The curves have been sorted in such a way that lines showing the smallest variation are lower in the stack, leading to a more stable baseline to which the subsequent products are added.

14.2   *Graphical techniques for multivariate data sets*

Data sets with more than two (or maybe three) columns are called *multivariate* data sets. In principle, the number of columns can be arbitrarily large. Multivariate data sets pose special challenges for graphical methods: it's difficult to represent more than two dimensions in a necessarily flat graph!

There are basically two approaches to overcome the limitation posed by the two-dimensional nature of the canvas:

- Juxtapose several graphs, each representing only a small number of the available attributes.
- Use complex symbols or glyphs that convey information not only through their position on the graph, but also through their appearance (shape or color).

In this section, I'll demonstrate some of the most common graphical techniques for multivariate data sets. My intent here is twofold: I want to describe these graphical methods, but I also want to show you how to realize them in gnuplot. Given that we're dealing with a *multivariate* problem, this will be an exercise in the use of loops and multiplot mode![3]

---

[3] This section uses several gnuplot features that are explained in appendix E.

14.2.1 *Introduction*

Our case study for exploring multivariate graphics is a data set describing the composition of a little more than 200 individual samples of glass.[4] The data set consists of 11 columns: the first is an integer to identify the record, followed by 9 columns of actual measurements, and finally a column that specifies the type of glass as an integer between 1 and 7.[5] Hence, this data set has 9 (or, if you count the glass type as well, 10) attributes. The data set is therefore neither particularly large nor excessively high-dimensional—it's just the right size to introduce the topic.

The data file is *comma-separated*, so it's necessary to use `set datafile separator ','`. The following listing does this and also includes a bunch of other definitions that will turn out to be useful. In all subsequent listings, this file with common definitions is loaded first.

Listing 14.5　Common definitions for the glass data set (file: glass.def)

```
set datafile separator ','                          Name of the input data file
data = "glass.data"
names = "ID RfIx Na Mg Al Si K Ca Ba Fe Type"       Column names
types = "Bldg1 Bldg2 Vhcl1 Vhcl2 Cont Tbl Lamp"     Types of glass (column 11)
```

14.2.2 *Distribution of values by attribute*

The first question concerns the distribution of values for the various attributes. The nine observed values are numerical, but the glass type is categorical—it can take on only seven distinct values. Figure 14.11 shows a histogram of the latter; the commands are in the next listing.

Listing 14.6　Commands for figure 14.11 (file: glass1.gp)

```
load "glass.def"

set boxwid 1
set xtic format ""
set for[k=1:7] xtic add ( word(types,k) k )
plot data u 11:(1) smooth frequency w boxes
```

Clearly, window glass as used in buildings dominates this data set; moreover, there are *no* records in this data set of type 4 (vehicle windows of variety 2). For this reason, it's important to fix the boxwidth: otherwise, the two adjacent boxes will extend to fill the gap.

---

[4] This example comes from the "Glass Identification" data set, available from the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/datasets/Glass+Identification.

[5] The measurements consist of the refractive index (RfIx) and the relative concentration of various chemical elements in each glass sample. The types of glass include two types each for building and vehicle windows, glass for containers, tableware, and headlamps.
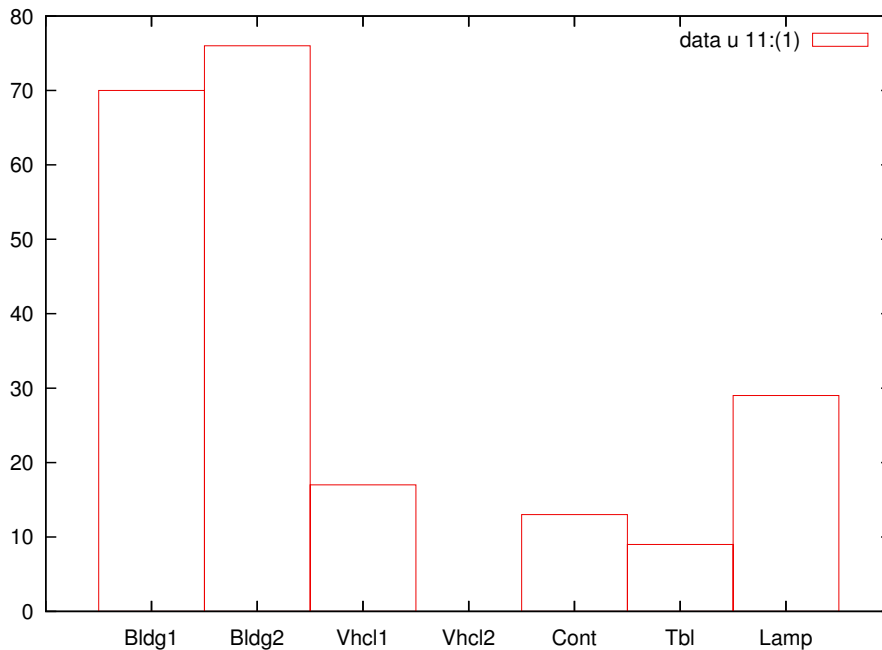
Figure 14.11   Number of data points for each type of glass. Compare listing 14.6.

Next let's investigate the distribution of the nine numerical attributes. Keep in mind that, at this point, you have no idea what their numerical ranges are. That suggests box-and-whisker plots, because they don't require any adjustable parameters. See figure 14.12 and the next listing.

Listing 14.7   Commands for figure 14.12 (file: glass2.gp)

```
load "glass.def"

unset xtics; unset key
set ticscale 0; set grid lt 1 lc rgb "grey" lw 2        Moves the tic marks
set ytic offset 0.5,0                                    closer to their boxes
set style fill solid 0.5

set multiplot layout 1,9 margin 0.075,0.975,0.1,0.95 spacing 0.075,0
do for[k=2:10] {
  set xlabel word(names,k)
  plot data u (1):k:(0.5) w boxplot pt 6
}
unset multi
```

The graph is a bit cluttered, mostly because each panel requires its own set of tic marks. For this reason, individual tic marks are replaced with grey grid lines to provide a quieter visual appearance. Empty, instead of filled, circles are used for the outliers, because those are easier to distinguish when they overlap partially.
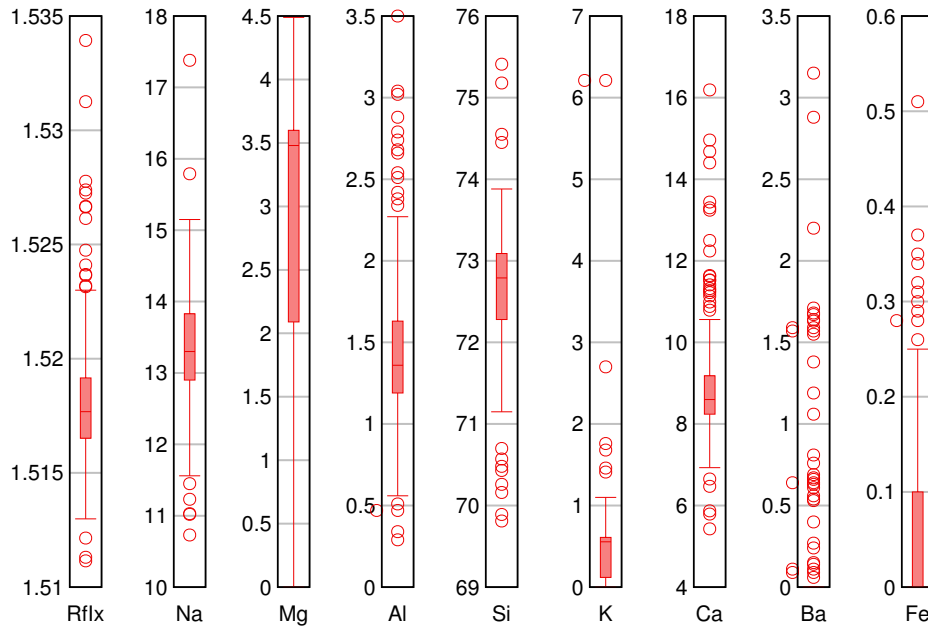
Figure 14.12   Distribution of values for each attribute. Compare listing 14.7.

14.2.3 *Distribution by level*

It's natural to ask whether different glass types can be distinguished by the observed properties of the sample. Figure 14.13 therefore investigates the distribution of values separately for each type of glass, which in the present case constitutes the *level*. (Also see the next listing.) The situation certainly isn't as clear cut as in section E.2.2!

Listing 14.8   Commands for figure 14.13 (file: glass3.gp)

```
load "glass.def"

unset xtic; unset key

set multi layout 3,3 margin 0.075,0.99,0.015,0.975 spacing 0.055,0.035
do for[k=2:10] {
  set label word(names,k) at graph 0.95,0.9 right
  plot data u (1):k:(0.5):11 w boxplot pt 6
  unset label
}
unset multi
```

SAMPLE SIZE IN A SERIAL BOX-AND-WHISKER PLOT

Figure 14.13 is great to get an overall view, but a graph showing only a single attribute allows greater detail—see figure 14.14. An interesting question when looking at a serial box-and-whisker plot concerns the number of data points for each level: that would be good information to have in order to evaluate the relative importance of
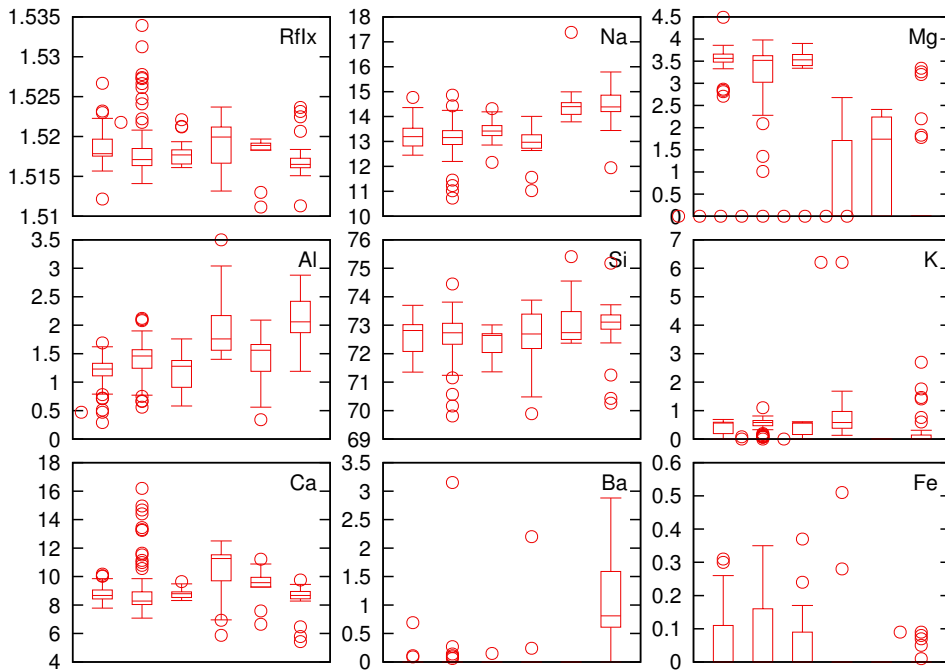
Figure 14.13   Distribution of values for each attribute, shown separately for each type of glass. The sequence of glass types is the same as in figure 14.11; type Vhcl2 is ignored because there are no records for it. Compare listing 14.8.

each level. One way to represent this quantity is via the *width* of the central box for each level, and that's what's done in figure 14.14.

Gnuplot's `with boxplot` style doesn't support this feature out of the box, but because it *does* support level-specific boxwidths, you can accomplish this goal with a little effort. In the following listing, the `stats` command is used to count the number of records for each level. A string is built up holding all the counts as whitespace-separated tokens; the `word` function extracts the appropriate token when needed. (See section 11.1.2 for a more detailed discussion of this technique.)

Listing 14.9   Commands for figure 14.14 (file: glass4.gp)

```
load "glass.def"

col = 6                                   Selects the column
                                          to display
get( s, k ) = word( s, int(k) )           Accesses the k-th
                                          element in string s
unset key
set title "Attribute: " . get(names,col)

s = ''
do for[k=1:7] {                           Builds up the string
  stats data u ($11==k?1:0) noout         of record counts for
  s = s . sprintf( " %d", STATS_sum )     each level
}
plot data u (1):col:(0.01*get(s,$11)):(get(types,$11)) w boxplot
```
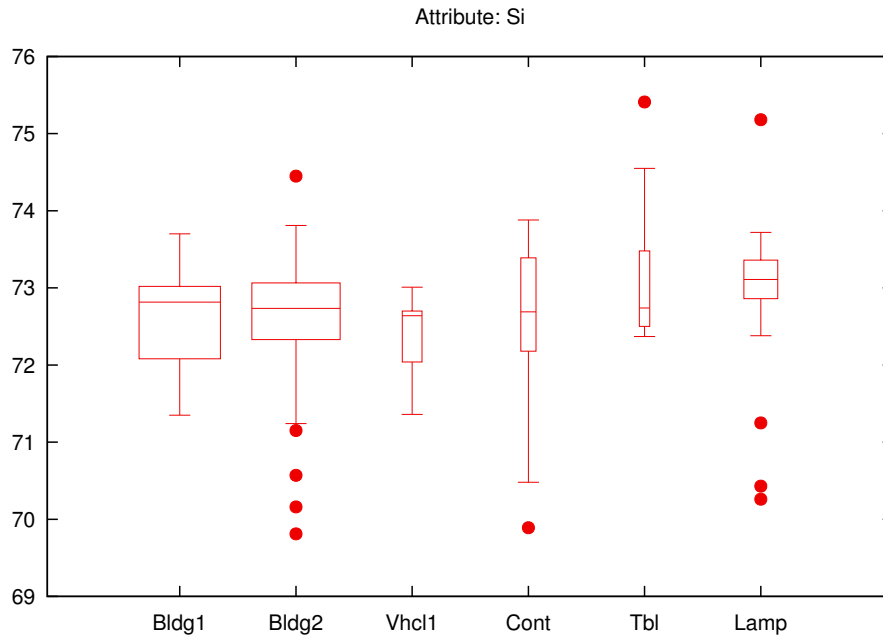
Figure 14.14   Distribution of values for attribute Si, shown separately for each type of glass. The width of the central box is proportional to the number of data points for each level. Compare listing 14.9.

14.2.4   *Scatter-plot matrix*

Box-and-whisker plots help us understand the *distribution* of data points for each attribute, but they tell us nothing about the *relationships* between different attributes. That's what the *scatter-plot matrix* is for.

A scatter-plot matrix consists of scatter plots for all possible pairs of attributes in a data set. In the present case, there are nine numerical attributes, and hence the scatter-plot matrix forms a 9×9 array (see figure 14.15 and listing 14.10). You can inspect this collection of graphs for correlations between attributes. For example, the index of refraction grows quite regularly with the calcium (chemical symbol: Ca) content of the sample.

The upper and lower half of a scatter-plot matrix aren't independent; each pair of attributes appears twice, with the horizontal and vertical axes interchanged. For this reason, I use the bottom half of the array to visualize clusters of points. The symbols are larger than in the top half, but the color is so transparent as to be almost invisible. Only where many points overlap is the density high enough to form recognizable spots. (See section 9.1.2 for information about transparency and alpha shading.)
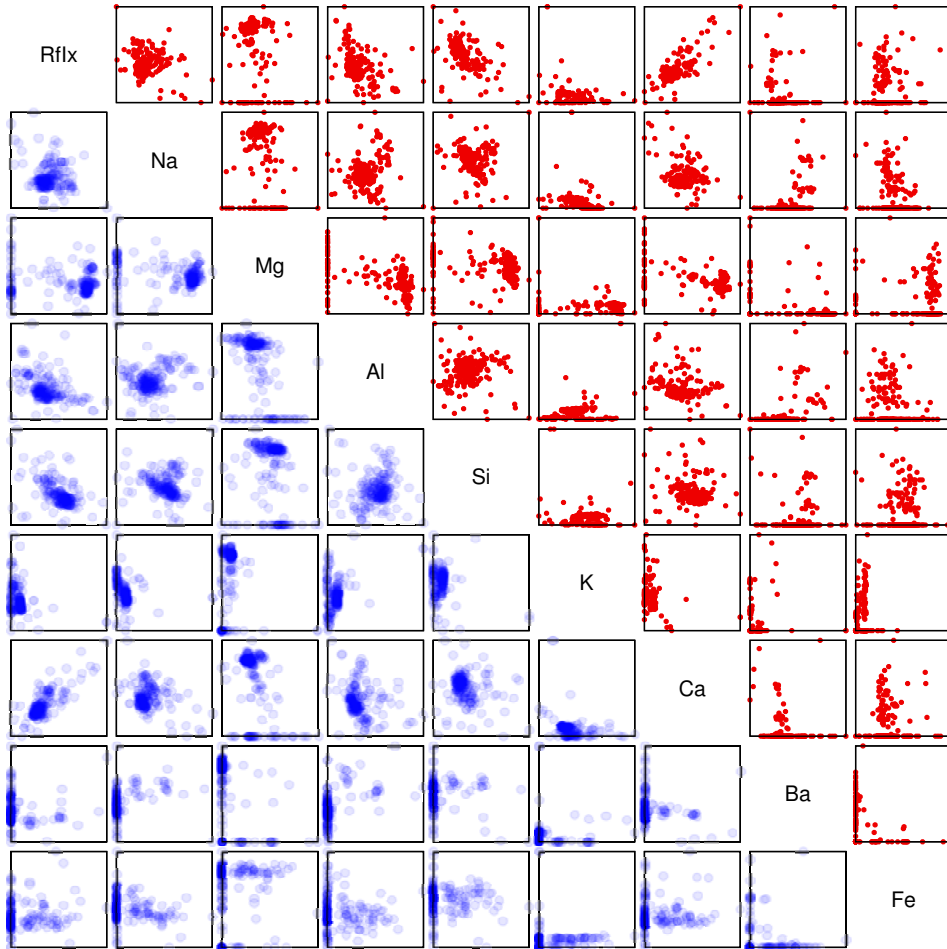
Figure 14.15   Scatter-plot matrix of all attributes, ignoring the type of glass. The upper half of the matrix shows individual points, and the lower half shows smeared-out and partially transparent symbols to facilitate the recognition of clusters. Compare listing 14.10.

Listing 14.10   Commands for figure 14.15 (file: glass5.gp)

```
load "glass.def"
unset key; unset tics
set size square
set multi layout 9,9 margin 0.01,0.99,0.01,0.99 spacing 0.01
do for[i=2:10] {
  do for[j=2:10] {
    if( i!=j ) {
      if( i<j ) {
```

```
      plot data u i:j w p ps .25 pt 7
    } else {
      plot data u i:j w p ps .5 pt 7 lc rgb "#e60000ff"
    }
  } else {
    unset border
    set label word(names,i) at 0,0 center
    plot [-1:1][-1:1] 1/0
    unset label
    set border
  }
  }
}
}
unset multi
```

The command `set size square` ensures that each individual *panel* of the 9×9 matrix is square. For best results, you may want to make sure that the shape of the overall canvas forms a square as well, using `set terminal ... size ...` with your preferred terminal.

14.2.5 *Parallel-coordinates plot*

The scatter-plot matrix is a method to visualize the relationship between any *two* attributes, but it provides no help at all when it comes to relationships across more than two attributes at the same time. For that, we turn to *parallel-coordinates plots.* They're formally introduced in section E.3, and there it's also explained that gnuplot's current implementation is subject to some limitations: in particular, the number of attributes is constrained to a small integer number. Among other things, you'll now learn how to work around this issue.

The trick is to use multiplot mode (again)! Using multiplot mode, you can include as many attributes as desired by placing several graphs side by side, each of which doesn't exceed the maximum number of columns supported (see listing 14.11).

Multiplot mode deals with the number of attributes; alpha shading is used to cope with the number of *records*. In the present case, the number of records is sufficiently high that lines obscure each other, making it impossible to discern any dominating structure in the graph. By using partially transparent lines, isolated lines fade into the background, but areas where lines cluster stand out strongly. (See figure 14.16.)

> **Listing 14.11   Commands for figure 14.16 (file: glass6.gp)**

```
load "glass.def"

unset key; unset ytics
set xtics format ""

set style parallel lw 1 lc black

set linetype 1 lc '#ddff0000' lw 1

set multi layout 1,2 margin 0.05,0.9,0.1,0.95 spacing 0

set for[k=1:5] xtic add ( word(names,k) k )
plot [1:6] data u 1:2:3:4:5:6 w parallel lt 1
```

Turns on multiplot mode

Left side

```
set for[k=1:6] xtic add ( word(names,k+5) k )
set paxis 6 tics format ""                                    Right side
set for[k=1:7] paxis 6 tics add ( word(types,k) k ) offset 7,0
plot [1:6] data u 6:7:8:9:10:11 w parallel lt 1

unset multi                                    ←——— Turns off multiplot mode
```

HIGHLIGHTING A SUBSET OF RECORDS

It's often desirable to highlight a subset of records in a parallel-coordinates plot, based on some condition. In figure E.12, records are highlighted based on their class. In the present case, all records are highlighted that belong to a particular cluster. If you examine figure 14.16, you'll notice that there is a reasonably well-separated cluster of records with an intermediate concentration of Ca. In figure 14.17, those lines are highlighted, and you can observe two things: records with an intermediate Ca concentration tend to have an unusually high or low aluminum (symbol: Al) concentration. And they tend to correspond to glass made for lamps or tableware, but not for containers or windows. That's an interesting finding.

Highlighting makes use of data-dependent coloring (see section 9.1.5): an additional column is specified at the end of the using declaration, and its value is evaluated to the linetype to use. Listing 14.12 defines the function c() that selects the linetype, depending on whether the argument to the function falls into the desired numerical range. (Tic marks have been turned on for the attribute in question.)
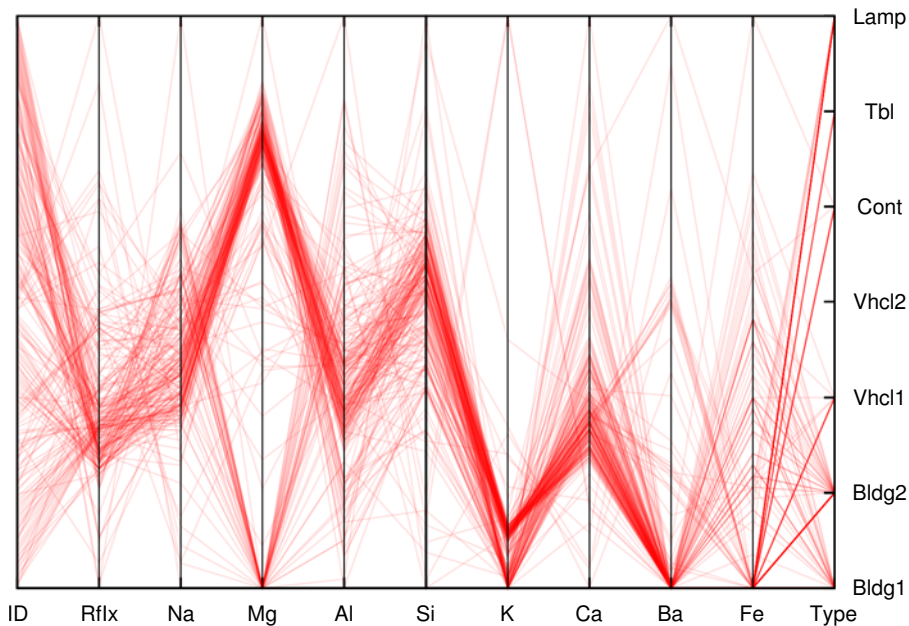
Figure 14.16   Parallel-coordinates plot of the entire glass data set. Because of the relatively large number of records, all lines are drawn partially transparent, so as not to obscure each other. To accommodate all attributes, this graph was composed using multiplot mode (see listing 14.11).
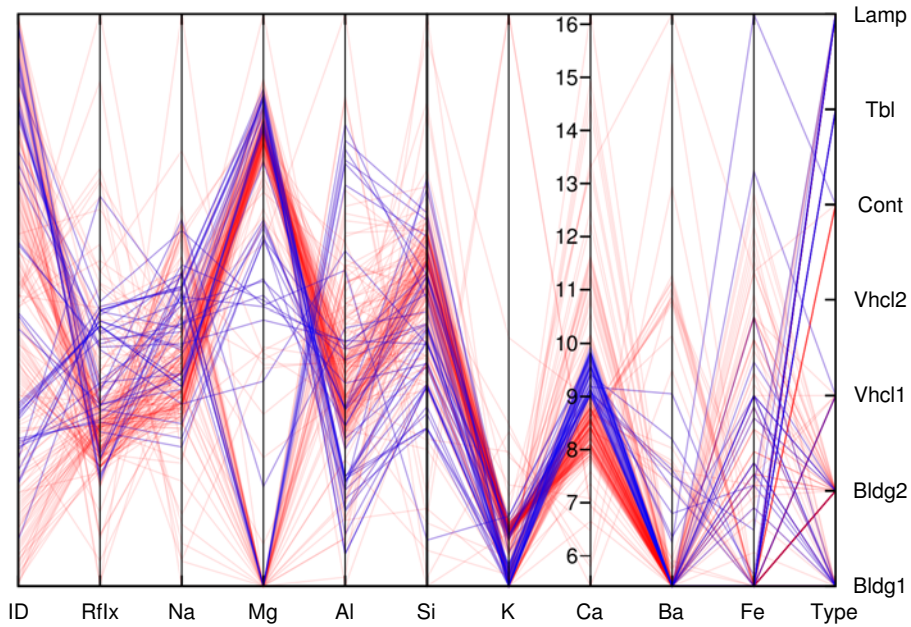
Figure 14.17   Same as figure 14.16, but with a subset of records highlighted. Records with a Ca content between 9 and 10 have been selected for highlighting. Compare listing 14.12.

Listing 14.12   Commands for figure 14.17 (file: glass7.gp)

```
load "glass.def"

unset key; unset ytics
set xtics format ""

c(x) = 9<x && x<10 ? 2 : 3

set style parallel lw 1 lc black

set linetype 1 lc '#ddff0000' lw 1
set linetype 2 lc '#aa0000ff' lw 1
set linetype 3 lc '#ff000000' lw 1

set multi layout 1,2 margin 0.05,0.9,0.1,0.95 spacing 0

set for[k=1:5] xtic add ( word(names,k) k )
plot [1:6] data u 1:2:3:4:5:6 w parallel lt 1,
           data u 1:2:3:4:5:6:(c($8)) w parallel lc var

set for[k=1:6] xtic add ( word(names,k+5) k )
set paxis 3 tics 1
set paxis 6 tics format ""
set for[k=1:7] paxis 6 tics add ( word(types,k) k ) offset 7,0
plot [1:6] data u 6:7:8:9:10:11 w parallel lt 1,
           data u 6:7:8:9:10:11:(c($8)) w parallel lc var

unset multi
```

Default style → `set linetype 1 lc '#ddff0000' lw 1`

Highlight style → `set linetype 2 lc '#aa0000ff' lw 1`

Invisible (transparent) style → `set linetype 3 lc '#ff000000' lw 1`

Left side

Right side

Turns on tic marks for one column → `set paxis 3 tics 1`

The listing defines two line types in addition to the default: one for highlighting and one that is completely transparent and therefore *invisible*. Because it doesn't obscure any of the existing graphical elements, it's used for the records that aren't selected for highlighting.

FURTHER SUGGESTIONS

Figure 14.17 explores only one aspect of the data set; many similar clusters can be found and traced across all attributes. You can also do what I did in figure E.12 and highlight all records belonging to a particular type of glass. Does this reveal any interesting structure in the data set? Another possibility is to draw all lines belonging to the same type of glass in the same color, using six different colors simultaneously.

Another aspect affecting the appearance of a parallel-coordinates plot is the *order* or *sequence* in which attributes are plotted. In figures 14.16 and 14.17, I used the (arbitrary) arrangement of columns in the original data file, but different permutations may give quite different results. Unfortunately, gnuplot doesn't make it easy to shuffle columns in a parallel-coordinates plot: you have to create different `using` declarations to rearrange attributes. Here are two permutations that you may want to try:

```
1,4,6,2,5,3,8,7,10,9,11
1,4,6,3,5,8,2,10,7,9,11
```

14.3   *Visual perception*

Even if you've found the most appropriate combination of quantities to look at, the amount of information you can extract from a graph still depends on the way the data is presented. The way people perceive a plot depends on the human, visual-psychological perception of it. To get the most out of a graph, you should therefore format it to improve the way it will be perceived.

Whether aware of it or not, in looking at a plot, humans always tend to engage in *comparisons*. Is this line longer than that? Is this part of the curve steeper than that? Is the area underneath this function larger than the area underneath that function? And so on.

To gain the maximum insight from a plot, you therefore want to find a representation of the data that facilitates such comparisons as much as possible. In this section, I give you some ideas for how this can be achieved.

14.3.1   *Banking*

The idea of *banking* (or *banking to 45 degrees*) was proposed by W. S. Cleveland, who demonstrated in a series of controlled experiments that our ability to compare angles is greatest for angles that are close to 45 degrees.[6] In other words, if you want to assess and compare the slopes of lines on a graph, you'll do best if the graph is sized in such a way that most of the lines are roughly going diagonally across it.

In a way, this is a confirmation of something you've probably been doing intuitively already. Whenever you've found yourself adjusting the plot ranges of a graph to get its aspect ratio right (as opposed to narrowing the plot ranges to zoom in on a

---

[6] See *The Elements of Graphing Data* by William S. Cleveland (Hobart Press, 1994) and references therein.
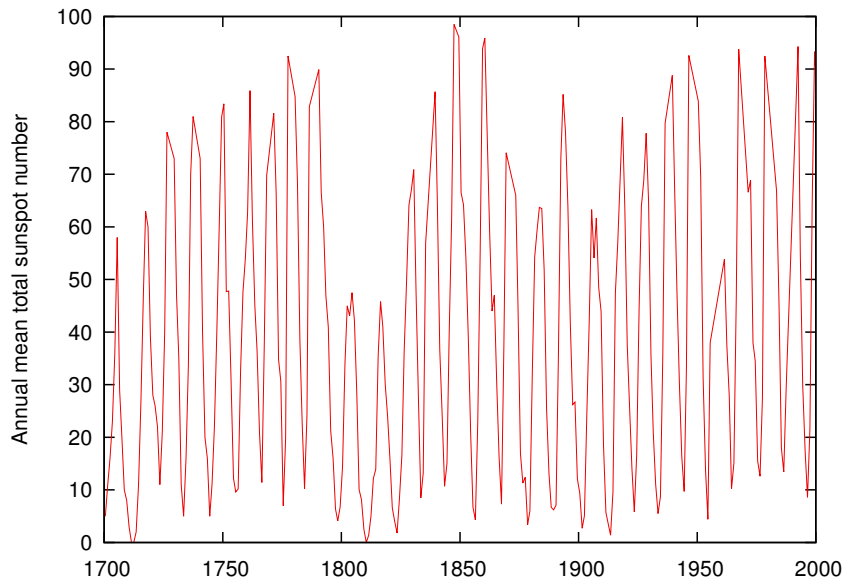
Figure 14.18   Annual sunspot numbers for the years 1700 through 2000. What can you say about the shape of the curve in this representation?

region), you've probably been *banking*. Let's consider an example: figure 14.18 shows the number of sunspots observed annually for the 300 years from 1700 to 2000.[7] The number of sunspots observed varies from year to year, following an irregular cycle of about 11 years.

Because of the large number of cycles included in the graph, vertical (or almost vertical) line segments dominate the graph, making it hard to recognize the structure in the data. In figure 14.19, the plot range has been split into thirds, and each third is displayed in a separate panel. The aspect ratio of the panels has been chosen in such a way that the rising and falling edges of the curve are close to 45 degrees. The graphs alone (without the labels and decorations) can be created easily:

```
set multiplot layout 3,1 margins 0.1,0.95,0.1,0.95 spacing 0
plot [0:100] "sunspots" u ($1-1700):2 w l
plot [0:100] "sunspots" u ($1-1800):2 w l
plot [0:100] "sunspots" u ($1-1900):2 w l
unset multi
```

Now an interesting feature in the data can be recognized: the number of sunspots during each cycle rises quickly but tends to fall more slowly. (There's the visual comparison I referred to earlier!) You wouldn't have been able to see this from the representation of the data in figure 14.18.

---

[7] The data was obtained from WDC-SILSO, Royal Observatory of Belgium, Brussels, www.sidc.be/silso /datafiles.
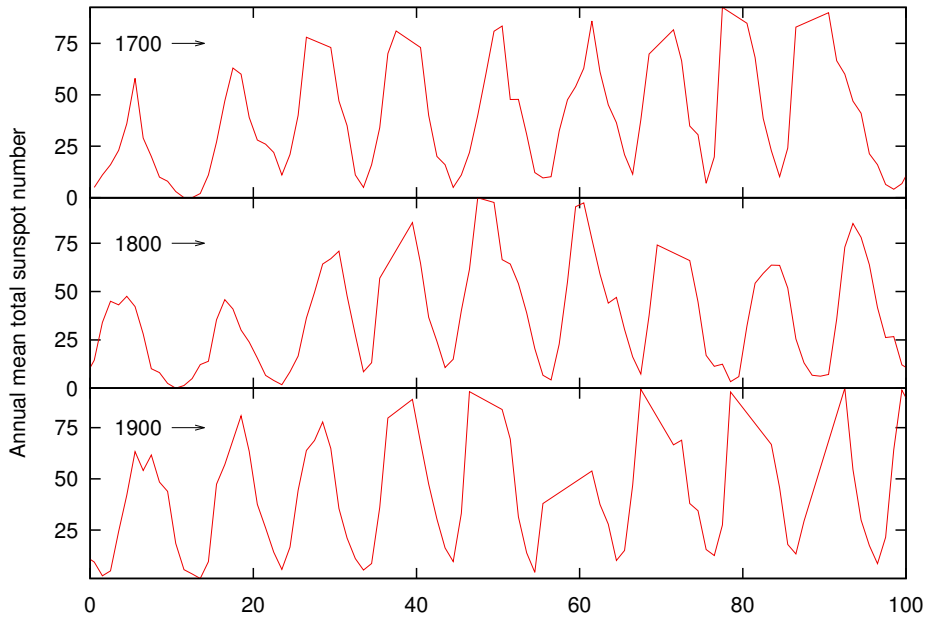
Figure 14.19   A different representation of the sunspot data from figure 14.18. The aspect ratio of each segment in the cut-and-stack plot is such that it banks lines to 45 degrees.

Banking is a valuable tool. In particular, I find it helpful because it draws attention to the importance of the apparent slopes of lines on a graph. Nevertheless, it must be used with judgment and discretion. Taken by itself, it can lead to graphs with strongly skewed aspect ratios. Figure 14.19 is a good example: if the three segments hadn't been stacked on top of each other, the resulting graph would have had a very awkward shape (extremely long and flat).

> **TIP**   To improve the perception of slopes, choose the plot range and aspect ratio of your graphs such that most of the curves are at approximately 45 degrees to the horizontal. If necessary, chop up the data set into segments and arrange them in a cut-and-stack plot, to keep the overall aspect ratio of the plot at a convenient value.

14.3.2   *Judging lengths and distances*

Look at figure 14.20. It shows the flows to and from a storage tank as a function of time. For the time interval considered here, the inflows are always greater than the outflows, so the tank tends to fill up over time, but that's not our concern right now. (Let's say the tank is large enough.)

   Instead, let's ask for the *net inflow* as a function of time—the inflow less the outflow at each moment. Could you draw it? Does it look at all like the graph in figure 14.21? In particular, does your graph contain the peak between 6 and 7 on the horizontal axis? How about the relative height of the peak?
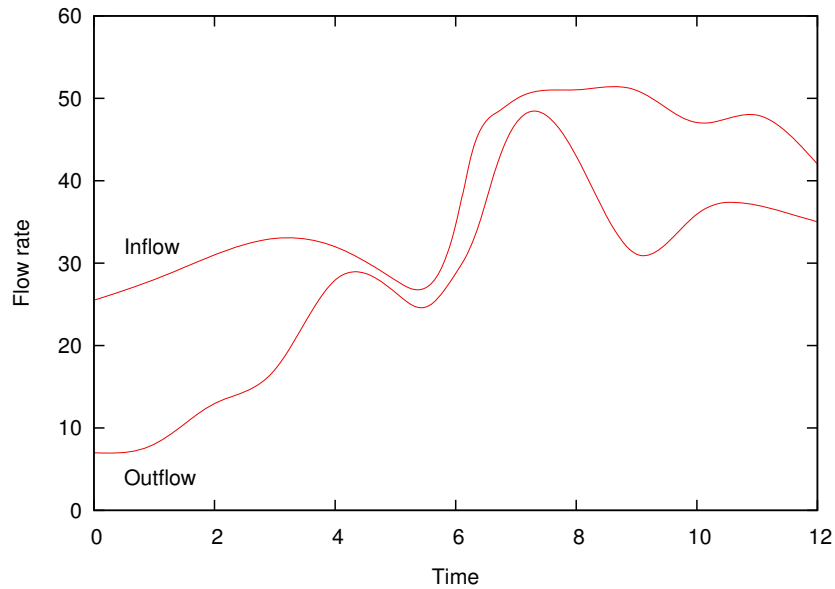
Figure 14.20   Inflow to and outflow from a storage tank. What's the net flow to the tank?
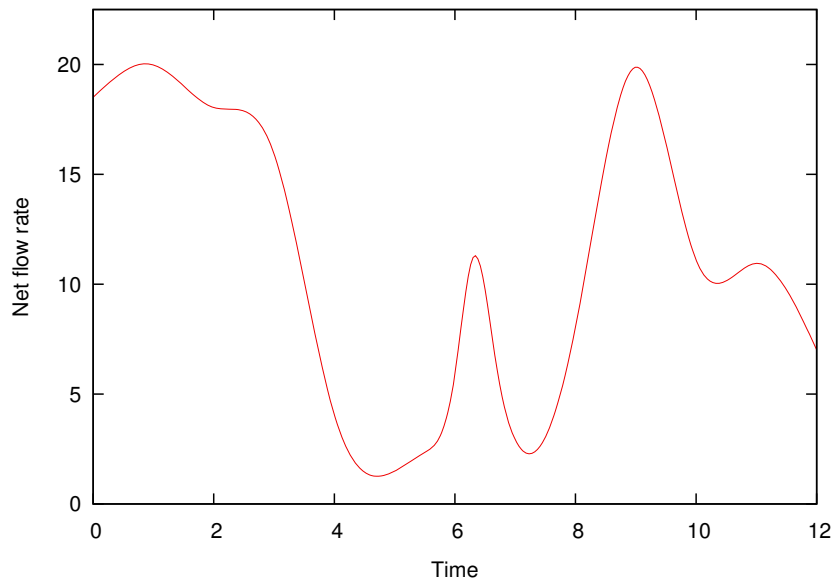
Figure 14.21   Net flow to the storage tank. This is the difference between the inflow and the outflow (see figure 14.20).

This example shows how hard it is to estimate accurately the vertical distance between two curves with large slopes. The eye has a tendency to concentrate on the *shortest* distance between two curves, not on the *vertical* distance between them. The shortest distance is measured along a straight line perpendicular to the curves. For nearly horizontal curves, this is reasonably close to the vertical distance; but as the slopes become steeper, the difference becomes significant. (Because the net flow is the difference between the two flow rates at the same point in time, you're looking specifically for the vertical distance between the two curves.)

You may also want to compare the observations made here with the discussion in section 14.1.4—the problems are similar. As a general rule, it's difficult to estimate changes against a baseline that itself isn't constant; and the more rapidly the baseline changes, the harder it is to use.

> **TIP**   The eye has difficulty accurately judging the vertical distance between two curves with large slopes. If the nature of the difference between two quantities is important, then plot it explicitly. Don't rely on observers to recognize it from the contributing components.

14.3.3   *Plot ranges and whether to always include zero*

Plots should have meaningful plot ranges, showing those parts of the data that the viewer is most likely going to be interested in at a reasonable resolution. It may not be possible to convey all the meaning appropriately with a single choice of plot ranges, so don't be shy about showing two or more views of the same data set—for instance, an overview plot and a detailed close-up of only a small section of the entire data set.

A long-standing (and largely unfruitful) discussion concerns the question of whether zero should be included in the range of a graph. The answer is simple: it depends.

If you're interested in the total value of a quantity, you probably want to include zero in your range. If you care only about the variation relative to some baseline other than zero, then don't include zero.

Figure 14.22 demonstrates what I mean. Both panels of the graph show the same data. The main graph tells us that the total variation is small, compared to the overall value. The inset tells us that there has been a steady increase from left to right. Both views are valid, and each gives an answer to a different question.

Plot ranges are a bit more of a concern when you need to compare several graphs against each other. In such a situation, all graphs should have the same scale to facilitate comparison; in fact, using different scales for different graphs is a guaranteed path to confusion (because the difference in scales will go unnoticed or be conveniently forgotten). And if one of the graphs legitimately includes zero, then all of them must do the same.
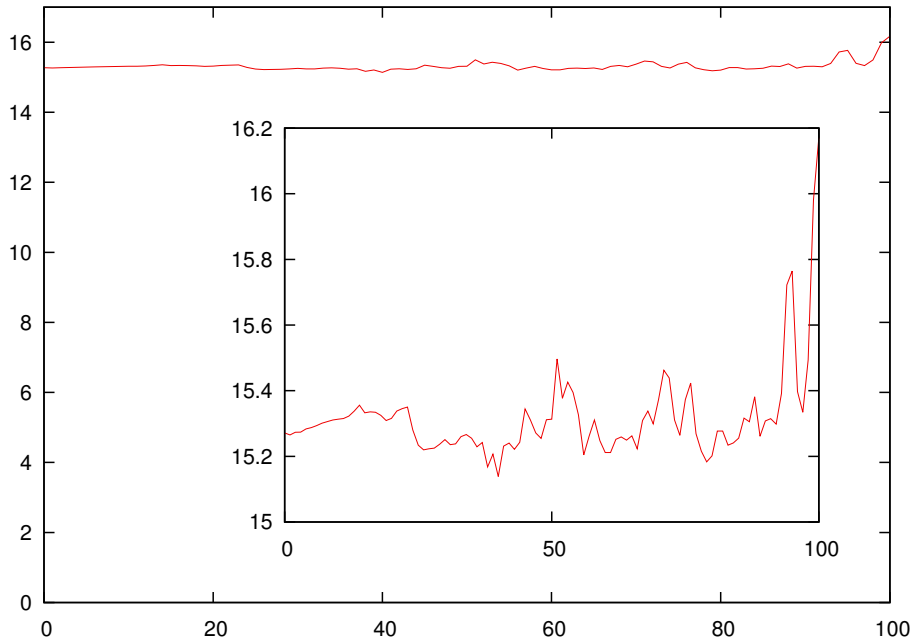
Figure 14.22   Two different views of the same data set. The outer graph includes zero in the vertical plot range, and the inner graph displays only that part of the vertical axis that corresponds to data points. Both views are valid: the outer graph shows that the relative change in the data is small compared to its absolute value; the inner graph shows that the variation increases dramatically from left to right.

14.4   Summary

In this chapter, I chose three recurring topics and discussed them in greater depth. The first was the graphical analysis of time series; we paid particular attention to finding a smooth approximation to the raw data's overall trend. I then described a variety of methods for the graphical treatment of multivariate data sets and how to realize them in gnuplot. Finally, I pointed out the importance of visual perception and described some ways to take advantage of it.

15

Coda: understanding data with graphs

But in the end, the most important thing is to draw the right conclusions from the available evidence.

During World War II, a program was launched to explore the possibility of equipping fighter planes with additional armor as protection against ground-based anti-aircraft fire.[1] To determine where on the airplane to place the armor, fighter planes returning from combat missions were investigated for bullet holes. In figure 15.1, the areas where bullet holes were found are shown shaded; areas without bullet holes are left white.

Given this evidence, where would *you* put additional armor?
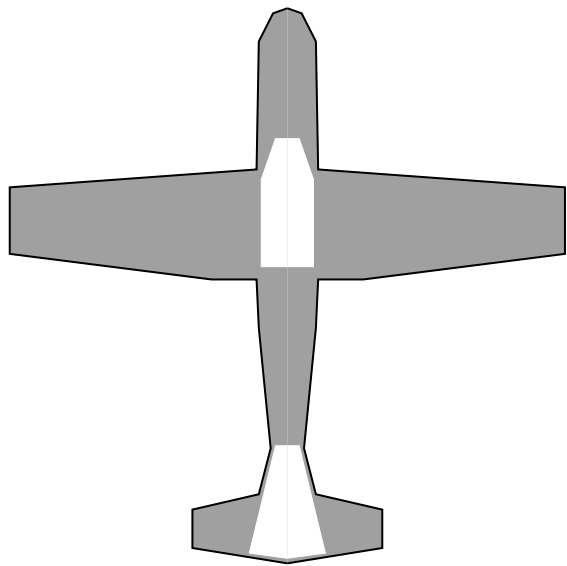
Figure 15.1   Schematic outline of a fighter airplane. Areas where bullet holes were found on machines returning from combat missions are shaded. Where would *you* recommend placing additional armor?

---

[1]   The inspiration for this story stems from the book *Graphical Discovery* by Howard Wainer (Princeton University Press, 2005).

344

The not-so-obvious obvious answer is to add the armor in those areas where *no* bullet holes were found. Why? Because airplanes are subject to hits everywhere, but if the hits strike in the white areas in figure 15.1, *the airplane doesn't come back* from its mission. (Statisticians speak of *survivorship bias.*) Therefore, those are the most vital areas of the machine and should receive the best possible protection.

So, let this be our final lesson. Evidence, be it graphical or otherwise, is just that: mere data. But actual *insight* arises only through the correct *interpretation* of those facts.

Obtaining, building, and installing gnuplot

The easiest way to install gnuplot on your local computer is to download and install a precompiled package. If you're running Linux, there's a good chance gnuplot is already installed; if not, you'll have no difficulty finding an RPM or Debian package on the internet. There are MacPorts packages for Mac OS X and precompiled binaries for Windows as well. In section A.3, we consider some of these options in more detail.

If you'd like to be totally up to date and have access to the newest features, or if you want to start hacking on gnuplot yourself, you'll have to build from source. Section A.4 in this appendix will help you get started.

A.1 Inspecting compile-time options

You can inspect what compile-time options your version of gnuplot was built with from within a gnuplot session. The pertinent commands are `show version` and `show version long`. The first version will tell you little more than the version number, but the second version will also print the compile-time options to the screen. A – prefix indicates an option that is *disabled*, a + prefix one that is *enabled*.

Most of this information is primarily relevant if you're building gnuplot from source yourself (see section A.4) or if you want to report bugs to the gnuplot developers' mailing list (gnuplot-beta@lists.sourceforge.net). But even if some feature doesn't work as expected, it may be worth checking whether your gnuplot executable was built with support for that feature to begin with.

A.2 Release and development versions

Gnuplot versions are generally labeled with a three-part version number indicating major, minor, and bug-fix releases. Bug-fix releases are prepared as needed (roughly twice a year). Minor releases introduce new features but preserve backward compatibility for existing gnuplot command scripts. Major releases may break backward compatibility. The development version of a minor release is indicated by an odd version number, which is incremented to the next even number on promotion to a released version.

Gnuplot is a mature project—for the 5.x.x major release series, minor releases are a few years apart, and the core developer team is very conscientious about not

introducing instability into released versions. On the other hand, new and exciting features continue to be added to gnuplot all the time, so there are good reasons for learning how to build gnuplot from source.

A.3 *Installing a prebuilt package*

Installing a prebuilt package is usually straightforward and mostly dependent on your choice of package manager.

A.3.1 *Linux*

Usually, the easiest way to install new software on a Linux box is to use the standard administration tool that comes with your local Linux distribution. On RPM-based distributions, you can also download an RPM file explicitly and then install it (as root) using

```
rpm -i gnuplot-XXX.rpm
```

where you should replace gnuplot-XXX.rpm with the exact name of the package file you've downloaded. RPM files for current gnuplot releases can be found without difficulty on the web.

On Debian-based systems, you can execute the following:

```
apt-get install gnuplot
```

But be warned that there are reports that gnuplot for Debian-based systems is often built without the GNU readline library (due to strict interpretations of the differences in the GNU and the gnuplot licenses), so you might want to build gnuplot from source instead.

A.3.2 *Mac OS X*

The most convenient way to install gnuplot on Mac OS X is to use a package manager, such as Homebrew or MacPorts, which will automatically deal with configuration issues and resolve dependencies.[1] Assuming you already have Homebrew installed, you can install gnuplot by entering the following command in the Terminal:

```
brew install gnuplot
```

If you're using MacPorts, you type

```
sudo port install gnuplot
```

The latter will prompt you for your password, which you need to enter. Once gnuplot has been successfully installed using either method, you can run it by issuing the command gnuplot in the Terminal window.

---

[1] I'd like to thank Mojca Miklavec for help with this section.

Be aware that a choice of build options is available when installing gnuplot, in particular with regard to the selection of terminals that will be included. The MacPorts distribution, for example, includes AquaTerm and the cairo-based terminals (including the `wxt` terminal) by default, but *not* the `qt` terminal.

A.3.3 Windows

The easiest way to install gnuplot on Windows is to use the precompiled package available from the download section of the gnuplot website at www.gnuplot.info.[2] At the time of this writing, the most recent version available was gnuplot 5.0.1. Download the file gp501-win64-mingw.exe, and execute it. The package works with the Windows installer, which takes care of things like permissions and menu entries.

No further installation is required. To run gnuplot, select the entry in the Start menu. A new window will pop up, containing the familiar gnuplot command prompt. The window has some additional menus that provide shortcuts to many of the gnuplot commands.

The bin directory contains additional gnuplot executables, which are intended to emulate Unix pipes and files necessary to run the `qt` and `wxt` terminals on Windows. Refer to the README-Windows.txt file for additional details.

A.4 Building from source

If you want to build gnuplot from source, you have two choices: either you can download and build an officially *released* version of gnuplot, or—if you want to be truly on the cutting edge—you can download and build the latest *development* version. Released versions of gnuplot can be downloaded as source tarballs from the gnuplot home page at www.gnuplot.info. There, you can also find precompiled binaries for several non-Unix platforms.

A.4.1 Obtaining the development version from CVS

The development version of the gnuplot source tree is kept in CVS at www.sourceforge.net. To get a copy of the development version from the repository, follow these steps:

1 Create a directory on your local drive, and change into it. You can name it any way you like.

2 Log in to the CVS repository:

```
cvs
➥ -d:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot
➥ login
```

Press Enter when you're prompted for a password. (This step is only necessary the first time around.)

---

2 I'd like to thank Bastian Märkisch for help with this section.

3 Check out the source tree:

```
cvs
➥ -d:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot
➥ checkout gnuplot
```

You may want to set the environment variable CVSROOT to

```
:pserver:anonymous@gnuplot.cvs.sourceforge.net:/cvsroot/gnuplot
```

(How you do this depends on the shell you're using.) If CVSROOT is set, you don't need to repeat the cvs-directory as part of the CVS command line using the -d option.

The checkout command creates a directory called gnuplot in the current directory. We'll examine its contents next.

A.4.2 *Layout of the source tree*

Whether you downloaded and unpacked a tarball of a released version or checked out the current development version from CVS, you should now have a directory called gnuplot (or similar) on your local drive. Inside this directory, you'll find the following (I'm listing only the most important files and directories):

- *config/*—Files required by the automake/autoconf utility, which is used to configure the build process to the local environment.
- *demo/*—A large set of gnuplot demos and the data files required by them.
- *docs/*—Source files for the gnuplot reference documentation. Documentation for gnuplot is maintained in a markup language that's specific to gnuplot. Help files in common formats (HTML, TeX, GNU Info, and so on) are built from the master file (called gnuplot.doc), as is the gnuplot online help feature. You can find an explanation of the markup format in the README file in the docs/ directory.
- *src/*—The source tree.
- *Copyright*—The gnuplot copyright and license.
- *INSTALL*—Gnuplot build and installation instructions. Very detailed.
- *INSTALL.gnu*—Documentation for the GNU autoconf/configure utility.
- *NEWS*—Overview of new features, changes, and fixes in the current version of gnuplot.
- *README*—The overall gnuplot Readme file.
- *RELEASE_NOTES*—A rather detailed overview concerning the new features in the current version, bugs and limitations, and important configuration options.

A.4.3 *Building and installing*

Specific build instructions are platform dependent. Because of the great variability of Unix platforms, the instructions for building on Unix are the most complex.

UNIX

Building gnuplot from source on Unix requires that the standard Unix development tools are installed. In addition to a C compiler and `make`, this also includes the autoconf/automake utility, which is used during the configuration step to probe many aspects of the local build environment and to create a localized Makefile. Building is then a rather painless process using `make`.

Configuring

When building from source, it's necessary to set up your build script first. To do so, change into the gnuplot directory and execute `./prepare`. This creates a script called configure. (This step is only required when building the development version of gnuplot; the release versions already include a usable configure script.)

In either case, execute `./configure` in the gnuplot directory. The configure script takes a large number of command-line options. You can obtain a complete list using

```
./configure --help
```

Many of these options are described in detail in the INSTALL file. Here, I mention just the ones that you'll most likely want to tinker with. For the majority of options, the defaults are fine:

- `--prefix=PREFIX`—This controls where the gnuplot executable, library, and documentation files will be installed. The default is /usr/local. If you don't have root permission on your box, or if you'd like to maintain a local version of the development version, you may want to provide a different path here.

  There are many more options that control the installation location of executable, library, and documentation files independently. Check the INSTALL file for details.

- `--with-readline`—Gnuplot comes with its own minimal version of a readline library (which provides command-line editing capabilities, command completion, and a history feature). You can select this version using `--with-readline` or using `--with-readline=builtin` (this is the default). If instead you'd like to use the more powerful GNU readline library, use `--with-readline=gnu`, or `--with-readline=DIR`, where `DIR` is the path to a (nonstandard) directory containing the GNU readline library. Note that the GNU readline library is released under a different license than the rest of gnuplot.

- `--with-wx=DIR`—The location of wx-config, the wxWidgets configuration program. This is important if you want to include support for the `wxt` terminal.

- `--with-qt`—This determines whether gnuplot will be built with support for the `qt` terminal, and which version of `qt` to use. If you select `--with-qt=no`, the `qt` terminal won't be supported; other possible values for this option are `qt4` and `qt5`.

The configuration step produces a long list of output describing the local environment that will be used during the build process. It's worth skimming this output: if the configuration utility doesn't seem to find libraries that you know are available, you

may have to change some of the defaults using additional command-line options described in the INSTALL file. Libraries required for some of the terminals are occasionally not found, which means you need to override the search paths by tweaking the corresponding command-line options to configure.

Also be aware that you'll need the *devel* packages for many of the required libraries if you want to build gnuplot from source, not just the *release* packages. In particular, the packages required for the wxt, cairo, and pango libraries may not be installed on your local system by default.

Building and installing

If the configuration step has completed successfully, building gnuplot is very simple: execute `make` from within the gnuplot directory. Compilation (on a reasonably current system) takes only a few minutes.

If the compilation was successful, execute `make install` to install the gnuplot binary, its documentation, and any required auxiliary files into the specified locations. You may want to follow this with `make distclean` to remove many of the temporary files left over from the build process.

WINDOWS

Gnuplot can be built on Windows using a variety of compilers. A major challenge when building on Windows is resolving the dependencies on external libraries. Check the INSTALL file for more information.

MAC OS X

To build gnuplot on Mac OS X, you should have the Xcode development environment installed: the build process is then essentially the same as for other Unix platforms. You may also want to have AquaTerm installed, which you can obtain through a package manager or directly from SourceForge (http://aquaterm.sourceforge.net).

To include support for cairo and wxt, you must have the cairo and pango libraries installed already. See the INSTALL file for details.

Also, there have been reports of problems with the readline library that ships with Mac OS X. You may want to fall back to gnuplot's own (minimalistic) version of the readline library: `./configure --with-readline=builtin`; or replace the default library with the GNU readline version before building.

appendix B
Resources

B.1 Gnuplot

The official website for the gnuplot project is www.gnuplot.info. There you'll find the official documentation, including the extensive and helpful FAQ, as well as links (not always current) to other sites and related projects.

Community support is available through the newsgroup comp.graphics.apps .gnuplot and the mailing list gnuplot-info@lists.sourceforge.net. There's also a mailing list for gnuplot developers: gnuplot-beta@lists.sourceforge.net.

B.1.1 Websites

Several websites are dedicated to gnuplot and often contain very good information, as well as tips and tricks. The following stand out both for the quality of their material and for the range of topics covered:

- *Gnuplotting* at www.gnuplotting.org—Hagen Wierstorf maintains a page with some stunning examples and uncommon gnuplot applications.
- *Impossible Gnuplot Graphs* at www.phyast.pitt.edu/~zov1/gnuplot/html/intro .html—The first of two websites with gnuplot tricks by Zoltán Vörös.
- *Gnuplot Tricks* at http://gnuplot-tricks.blogspot.com—The other site prepared by Zoltán Vörös.
- *Gnuplot—not so Frequently Asked Questions* at http://lowrank.net/gnuplot/index-e .html and at http://mng.bz/5i80—Originally collected by Toshihiko Kawano at Los Alamos National Lab, this is a large collection of techniques. Some of the material is a little dated, but most of it is still relevant and interesting. (The original site has gone away, but the material is now fortunately hosted by volunteers.)

For specific topics, a quick Google search will typically result in additional posts and articles.

B.1.2 Books

To my knowledge, there is only one other book dedicated exclusively to gnuplot:

- *Gnuplot Cookbook* by Lee Phillips (Packt Publishing, 2012)—This book contains a collection of "recipes" for common, and not-so-common, tasks.

352

B.2 Data repositories

I assume that you'll usually have your own data that you want to plot! Still, the need for example data sets arises from time to time. Here are some good collections of freely available data sets on the web:

- *UCI Machine Learning Repository* at http://archive.ics.uci.edu/ml/index.html—A very large and frequently updated collection of data sets.
- *Journal of Statistics Education Data Archive* at www.amstat.org/publications /jse/jse_data_archive.htm—A large collection of often very interesting data sets used in papers published by the JSE.
- *Time Series Data Library* at http://data.is/TSDLdemo—A very large collection of time series data, collected by Rob Hyndman. Unfortunately only sparsely documented.
- *The Data and Story Library* at http://lib.stat.cmu.edu/DASL—An older, medium-sized collection of data sets and "stories," intended to teach basic statistics.
- *StatLib Datasets Archive* at ftp://rcom.univie.ac.at/mirrors/lib.stat.cmu.edu /datasets/.index.html—Many data sets that have been mentioned in textbooks can be downloaded from this site. You usually need the respective textbook to make sense of the downloads.
- *100+ Interesting Data Sets for Statistics* at http://rs.io/100-interesting-data-sets-for-statistics—A large and heterogeneous collection of links and pointers to data archives on the web.

B.3 Books

Here are a few books on scientific graphics and graphical discovery that I've found helpful and interesting:

- *The Elements of Graphing Data* by William S. Cleveland (Hobart Press, 1994)—Probably the definitive reference on graphical analysis (as opposed to presentation graphics). Much of the material in chapter 13 and chapter 14 has been influenced by Cleveland's careful approach. There's also a companion volume, *Visualizing Data* (1993), by the same author.
- *Visual Revelations* by Howard Wainer (Copernicus/Springer, 1997), *Graphic Discovery* by Howard Wainer (Princeton University Press, 2005), and *Picturing the Uncertain World: How to Understand, Communicate, and Control Uncertainty through Graphical Display* by Howard Wainer (Princeton University Press, 2011)—Three collections of short essays on statistical graphics, written in an entertaining, semi-popular style. Many good ideas and interesting case studies can be found here, but also a lot of editorializing about various issues.
- *Data Analysis with Open Source Tools* by Philipp K. Janert (O'Reilly, 2009)—The first several chapters of this book cover graphical methods in data analysis (similar in spirit to chapters 13 and 14, but in more depth). The remainder of the book discusses a variety of other (nongraphical) techniques for extracting insights from data.

index

Note: index entries that begin with the letters C, D, E, and F can be found in the corresponding appendixes in the e-book version of this book.

C

The `plot` command discussed in this book creates graphs that show how one variable (*y*) behaves as a function of another one (*x*). But what if you want to show how some quantity depends on *two* independent variables? In other words, how can you best visualize a single "output" variable as a function of two "input" variables? That's basically the problem cartographers have: show a quantity (the elevation above sea level or the type of ground cover) as a function of two coordinates (the position on the surface). In gnuplot, you have essentially the same options cartographers have:

- Create a rendering of a raised surface. Cartographers call this a *relief map*; I'll refer to it as a *surface plot.*
- Create a *contour plot* by looking straight down onto the coordinate plane and using contour lines (similar to the ones found on topographic maps) to indicate the elevation (or whatever the dependent variable represents).
- Use color to create a *false-color plot* (what is colloquially known as a *heatmap*).

Surface and contour plots are treated in this appendix; false-color plots are the topic of appendix D. Surface and contour plots are created using gnuplot's `splot` command (short for "surface plot"). It's a close relative of the familiar `plot` command; most of what you already know will carry over directly.

Neither surface nor contour plots deal very well with noise. This means it can be difficult to use either of them with real data—smooth functions are easier, and that's why most of the examples in this appendix show functions only. We turn to data later in this appendix and discuss the different file formats available. (Also keep in mind that false-color plots may be a better approach for noisy data altogether.)

C.1   Surface plots

Surface plots (and contour plots) are created using the `splot` command. Its syntax is very similar to the syntax for the `plot` command. The differences are largely due to the need to handle one additional dimension, which I'll refer to as the *z direction.*

Here's an example of the `splot` command in action (also see figure C.1—if your plot doesn't look anything like figure C.1, keep reading; I'll tell you about the options you need to adjust manually to get a satisfactory result shortly):

```
splot [-2:2][-2:2]
➥      exp(-(x**2 + y**2))*cos(x/4)*sin(y)*cos(2*(x**2+y**2))
```

C1
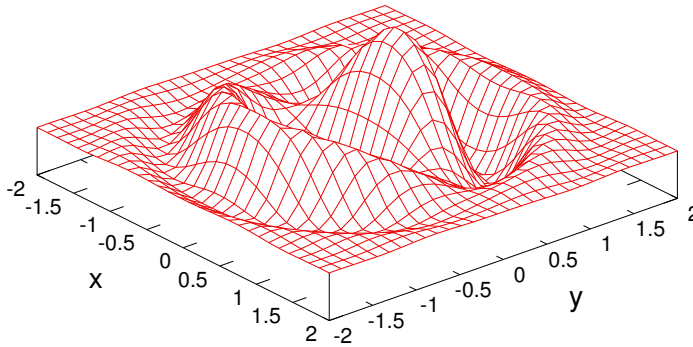
Figure C.1   Creating three-dimensional plots using the `splot` command:
`splot [-2:2][-2:2] exp(-(x**2 + y**2))*cos(x/4)*sin(y)*cos(2*(x**2+y**2))`

You can see how the function must depend on *two* variables, x and y. Corresponding to the two variables, there are two brackets to limit the plot range. A third bracket can be added to restrict the plot range in the new, "vertical" z direction.

C.1.1   *The splot command*

Most of the additional options applicable to the plot command are available for the splot command as well. You can also plot data from a file (see section C.4 later in this appendix) and use many of the directives familiar from plot:

- The title option is available to place a descriptive string into the key.
- The using directive now requires *three* arguments to pick out the columns used for the x, y, and z directions, respectively.
- Similarly, you have to add a third bracket if you want to restrict the z range: `splot [xmin:xmax][ymin:ymax][zmin:zmax] ....`
- Plotting styles are selected in the usual form through the with option. Not all the styles described in chapter 6 are available with splot. Only points, lines, linespoints, and impulses can be used with splot; in particular, this means none of the styles drawing error bars are available.

Although we don't show an example here, nothing prevents you from plotting several functions (or data sets) simultaneously using splot: for example, splot f(x,y), g(x,y).

In addition to using, you can use index and every to plot only parts of a file. The smooth directive (see chapter 3) isn't available for splot, but you can apply data transformations as part of the using directive, as discussed in section 3.3. Furthermore, the dgrid3d option provides a way to calculate and plot a smooth approximation to a noisy data set when using splot (see section C.5).

STYLE, DECORATIONS, AND AXES WHEN USING SPLOT

Global options can be used with `splot` in the same way as with `plot`. Both `set style function` and `set style data` have the desired effect. In addition to the familiar `xrange` and `yrange` options (see section 8.2), there's a `zrange` option to control the plot range globally.

Arrows and labels (but not objects) can be placed onto a three-dimensional graph without problems. Just remember to provide a third value to each coordinate for the z value (for example, `set label "text" at -2,-3,1`).

In contrast to `plot`, graphs generated with `splot` can have only a single coordinate system. Therefore, all the options used to control the secondary coordinate system (see section 8.1) have no meaning for graphs generated with `splot`. Also, the `axes` directive can't be used with `splot`. We'll study tic marks and axes in more detail later, when we discuss ways to choose the viewpoint for a graph created with `splot` in section C.2.

C.1.2 *Special options for surface plots*

There are three options specific to the appearance of a surface plot: `set surface`, `set hidden3d`, and `set isosamples`.

THE SET SURFACE OPTION

The `set surface` option turns rendering of the surface on or off. For surface plots, this is of course not useful; but the `splot` command is also used to generate contour and false-color plots, where an explicitly drawn surface is not desired.

By default, the surface is shown, but it can be switched off using the following command:

```
unset surface
```

Switch the surface back on using `set surface`.

THE SET HIDDEN3D OPTION

In figure C.1, the surface is *opaque*: parts closer to the observer obscure parts of the surface further back. By default, this effect is off, and only a transparent wire mesh of the surface is drawn. Figure C.2 demonstrates what a plot looks like if no opaque surface is drawn.

To switch to an opaque surface rendering, use the `set hidden3d` option. You'll probably want to make opaque surfaces the default—for example, by including the following line in your gnuplot startup file (see section 12.5):

```
set hidden3d
```

The `hidden3d` option can take a number of optional arguments. The most important of these are `offset` and `trianglepattern`:[1]

```
set hidden3d [ [no]offset {int:offset} ] [ trianglepattern {int:mask} ]
```

---

[1] There are a few further options to `hidden3d`, most of which deal with certain edge cases that may arise when making a surface plot containing undefined points or points outside of the z range. Check the standard gnuplot reference documentation if this is relevant to you.
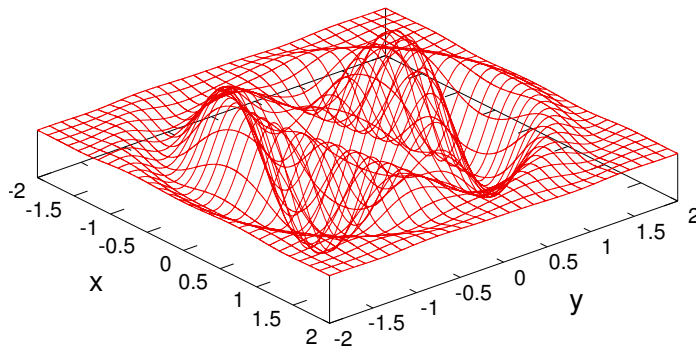
Figure C.2   **The same plot as in figure C.1, but without the opaque surface effect.**
Use `set hidden3d` to enable the drawing of an opaque, nontransparent surface.

You can use the `offset` keyword to control the color and line type that are used for the bottom side of the surface. As you know, gnuplot will cycle through all available plot styles for each new data set (see section 9.3). Through the integer argument to `offset`, you can control how far the internal style counter should be advanced from the style used for the top side of the surface. By default, gnuplot chooses the next available style. An argument of zero (`set hidden3d offset 0`) draws both sides of the surface in the same style. The same effect is achieved through the `nooffset` keyword.

   Using the `trianglepattern` keyword, you can control which lines are drawn to connect neighboring grid points. The argument to `trianglepattern` is an integer, which is interpreted as a bitmask (see table C.1).

Table C.1   **Values used in the mask to the `set hidden3d` option**

Bit position	Value	Description
0	1	Lines parallel to the x axis
1	2	Lines parallel to the y axis
2	4	Diagonal lines, from southwest to northeast

You can select any combination of bits. The default is 3, so that the surface is made up of rectangular surface elements. When you use a value of 7, diagonal lines are also drawn, so that the surface appears to be made out of triangles. Figure C.3 compares different settings of this option.

SAMPLING FREQUENCY FOR PLOTTING FUNCTIONS WITH SPLOT
In section 3.1, you learned to use the `set samples` option to control the number of points where gnuplot evaluates a function (of a single variable) in order to plot it. When plotting a function of two variables using `splot`, things are a bit more complicated.

   If `hidden3d` is on, so that the surface is opaque, then `set isosamples` controls the number of evaluation points, and these points are connected with straight lines to
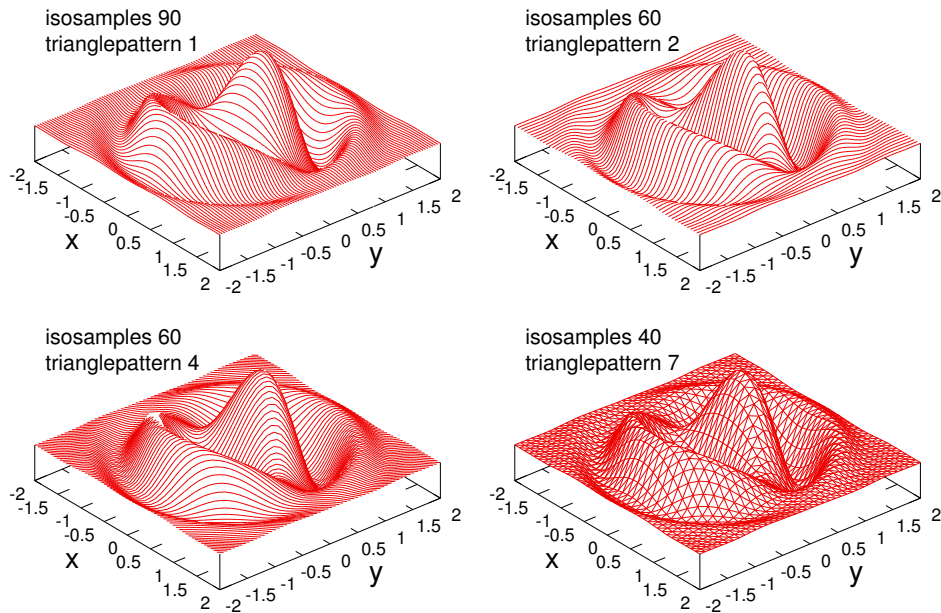
Figure C.3 **Comparing different values of `isosamples` and `hidden3d trianglepattern`. Except for the values of those two options, these graphs are identical to figure C.1.**

form the surface. If `hidden3d` is off, so that the surface is transparent, then `set isosamples` still controls the number of grid points, but `set  samples` controls the number of evaluation points for each line in the grid.

The `set isosamples` command takes one or two integer values, which specify the number of grid points in the x and y direction, respectively. If only a single value is provided, it's used for both directions:

```
set isosamples {int:xlines} [, {int:ylines} ]
```

By default, functions are evaluated on a 10×10 grid, which is too coarse to give a good, smooth appearance of the plotted curve. I find values of around 30 to be ideal—using an even finer grid brings little additional smoothness but leads to grid lines overlapping each other unfavorably. You probably want to include the following line in your gnuplot startup file:

```
set isosamples 30
```

Figure C.4 shows the same function plotted in figure C.1, but now plotted with the default setting of `set isosamples 10`.

> **TIP** Gnuplot's default value of 10 for the `set isosamples` option is much too small to give attractive graphs. Choosing a moderately higher number (such as 30) will lead to much better results.
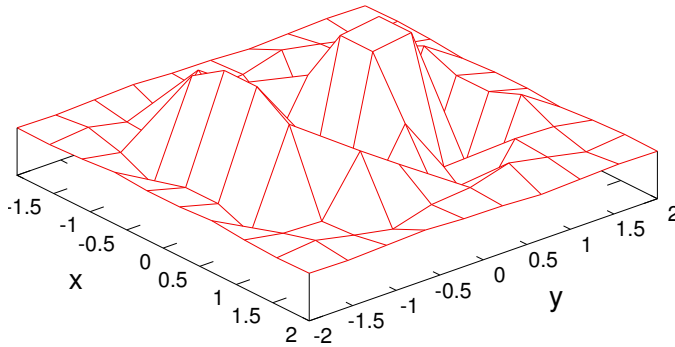
Figure C.4   **The `isosamples` option controls the number of nodes used to draw
the surface. Here, we use the default value of `set isosamples 10`, whereas in
figure C.1, we used a much finer grid of `set isosamples 30`.**

Special rules apply to the pseudofile ++ (see section 4.5). In this case, the number of
grid points used for drawing the surface is controlled by `set samples` for the x direc-
tion and by `set isosamples` for the y direction. To obtain behavior from the pseudo-
file ++ that is equivalent to a regular function plot, you need to assign the same value
to `set samples` and `set isosamples`, as in the following example:

```
f(x,y) = exp(-(x**2 + y**2))*cos(x/4)*sin(y)*cos(2*(x**2+y**2))
set samples 30
set isosamples 30
splot [-2:2][-2:2] "++" u 1:2:(f($1, $2)) w l
```

Finally, let me remind you that `set samples` and `set isosamples` are only relevant
when plotting functions and have no effect when plotting data from a file.

C.2   *View point and coordinate axes*

With three-dimensional plots, there's a new problem that doesn't exist with regular,
two-dimensional plots: they can be viewed from different positions and under different
angles. You must therefore choose a *view point*. Closely related is the issue of coordinate
axes: you need to provide a clear frame of reference so that the viewer can understand
from which position the graph is seen. And finally, you have to provide a scale and tic
marks if you want to convey quantitative information. Let's look at all of that.

C.2.1   *Borders and base plane*

A regular (two-dimensional) plot has four borders that you can modify using the `set
border` option, as you've seen in section 9.5.2. For a surface plot, you can imagine the
entire surface embedded in a cube. You can use the `set border` option to switch each
individual *edge* of this cube on or off.

BORDERS

The set border command takes an integer argument, which is interpreted as a 12-bit bitmask; each bit turns one of the edges on or off. Table C.2 shows the bit positions for each edge of the surrounding cube.

Table C.2   The values of the mask used in the **set border** option. The red lines are the ones switched on by the corresponding bit in the mask.

Bit position	Value	Edge	Bit position	Value	Edge	Bit position	Value	Edge
0	1		4	16		8	256	
1	2		5	32		9	512	
2	4		6	64		10	1024	
3	8		7	128		11	2048	

The default value is 31 when you use splot, turning on the four sides of the base and the leftmost vertical axis. (No matter what's specified in the set border option, borders other than the base are drawn only if set surface is true.)

You can place labels along the borders of the plot in the usual fashion using set xlabel, set ylabel, and (only for splot) set zlabel. In all graphs in this appendix, I've made sure to use explicit labels on both the x and the y axes.

> **TIP**   There is no "obvious" view point for surface plots; it often makes sense to choose a different view point for each function or data set. Because of this, it is *essential* that you tell the reader which axis is which by using explicit labels on all axes.

Tic marks are drawn independently of the borders—see chapter 8 for all the options available to control the drawing and placement of tics. Just keep in mind that the secondary coordinate system (x2tics and so on) isn't available; instead, there's an additional set of options to control the appearance of tic marks in the z direction (ztics and so forth).

THE BASE PLANE

By default, the plotted surface is elevated a certain distance above the base plan of the surrounding box. You can control this elevation through the xyplane option:

```
set xyplane [ at {flt:zvalue} | {flt:frac} ]
```

There are two ways you can use `set xyplane` to fix the position of the base relative to the plot surface. You can specify an explicit z value at which the base plan should be drawn, using the keyword `at` together with `xyplane`:

```
set xyplane at 0.1
```

This is the easiest way to fix the location of the base plane if you use an explicit plot range in the vertical direction.

   If the z range is dynamically chosen, it doesn't make sense to specify a fixed location for the base plane; instead, its position should be chosen as a fraction of the total apparent height of the figure. This is precisely what `set xyplane` *without* the `at` keyword does: it allows you to control the elevation of the plot surface above the base plane as a *fraction of the total z range.*

   A few examples may help to clarify. The command

```
set xyplane 0
```

puts the base plane right at the low end of the z range. Choosing

```
set xyplane 0.5
```

elevates the plotted surface by half the total z range (or half the apparent height of the plot) above the base plane. Using negative arguments to `set xyplane` lifts the base plane so that it intersects with the plot surface. For example, the command

```
set xyplane -0.5
```

places the base plane at the middle of the z range.

   The following formula can be used to convert between the two models. It tells you the z value at which the base plane will be drawn, given the fractional parameter `f` and the z range [`zmin:zmax`]:

```
z = zmin - f * ( zmin - zmax )
```

C.2.2   *View point*

When creating surface plots using `splot`, you can control one aspect of the graph that has no equivalent in two-dimensional plots: the view point—that is, the location (relative to the graph) from which the observer appears to be regarding the plotted surface. You can set the view point in two ways: either programmatically with the `set view` option, or interactively using the mouse.

USING THE MOUSE
The more convenient way to adjust the viewing angle is of course using the mouse! Grab the plotted surface (by left-clicking into the plot window), and drag it to its desired position. This requires a reasonably fast computer (and works much better if the surfaces are opaque—when `hidden3d` is true). When you hold down the Ctrl key

while dragging with the mouse, only the box surrounding the graph is shown: this may facilitate this process on slow computers or for surfaces containing many points.

Holding down the middle mouse button and moving the mouse zooms the graph: moving the mouse left to right increases the size of the entire graph (and vice versa). Moving the mouse upward stretches the graph in the z direction only; moving the mouse downward shrinks the graph in the z direction. Again, you can suppress rendering of the actual plot surface when dragging by holding down the Ctrl key while moving the mouse. Finally, holding down the Shift key together with the middle mouse button lets you move the base plane of the plot (this is equivalent to `set xyplane`).

USING SET VIEW

The `set view` command gives you exactly the same capabilities, but in a non-interactive fashion. The `set view` option takes up to four optional arguments:

```
set view [{flt:theta} [, {flt:phi} [, {flt:scale} [, {flt:z_scale}]]]]
```

The first two arguments are the angles (in degrees) of the view point around the horizontal and the vertical axes, respectively. (In spherical coordinates, these are the polar and azimuthal angles.) The angles are restricted to values from 0 to 360. (See figure C.5.) Figure C.6 demonstrates how a graph can appear from different view points.

The third and fourth parameters correspond to the zooming effect I've already mentioned in the context of mouse interactions. Both default to 1.0: choosing smaller scale factors results in a smaller graph on the canvas. (In particular, the scale factor is the *inverse* of the radial component in spherical coordinates.)
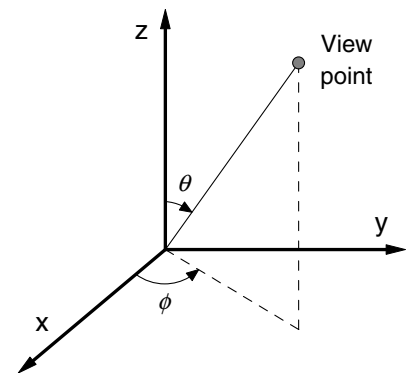
Figure C.5   The angles used when setting the view point with `set view`
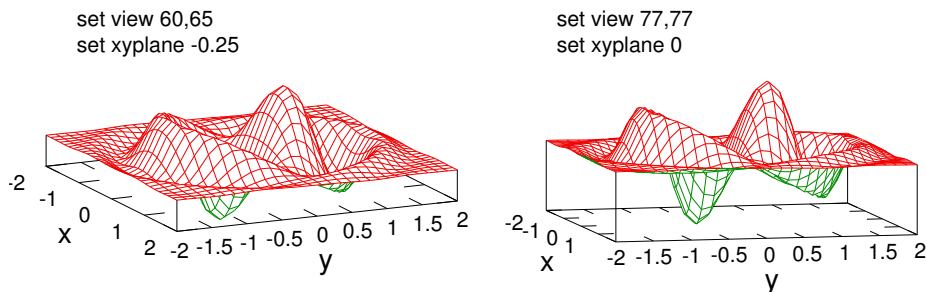
Figure C.6   Different view points. Compare to figure C.1, where the same function is shown with `set view 45,50`.

UNIT LENGTH AND ASPECT RATIO

You can also use `set view` to fix the length of the x and y axis units. (The `set view` command therefore does for `splot` what `set size` does for `plot`—see section 9.5.1.)

The command `set view equal xy` creates a graph within which the (visual) length of one x axis unit equals the length of one y axis unit. The graph is scaled so that it's guaranteed to fit on the canvas.

The command `set view equal xyz` also sets the z axis unit to the same visual length, but in this case, there is no guarantee that the graph will fit on the canvas! The command `set view noequal` removes the constraint between the axis units.

BIRD'S-EYE VIEW

The special command `set view map` places the observer right *above* the plotted surface so they perceive just the base plane of the plot as a regular, two-dimensional plot. This is the proper view for contour plots (see section C.3) and for false-color plots (see appendix D).

But `set view map` is not just a convenient shorthand for `set view 0, 180, 1.425, 1`. It also enables `set size` for `splot`. In particular, you can use `set size square` and `set size ratio -1` to influence the appearance of contour plots and heatmaps.

It's possible to append the keyword `scale` together with a numerical factor to scale the resulting graph on the canvas. The following code snippet turns on map view and also shrinks the graph to half its default size:

```
set view map scale 0.5
```

C.3   *Contour lines and contour plots*

You can add contour lines to a plot generated using `splot`. Contour lines can be a great help when it comes to associating specific z values with surface plots. They can also be used by themselves to make *contour plots*.

Use the `set contour` option to enable contour lines:

```
set contour [ base | surface | both ]
```

By default, contour lines are only plotted on the bottom of the box surrounding the surface plot. But contours can also be drawn on either the plot surface or on both (base and surface), through the use of the appropriate options to `set contour` (as in figure C.7).

C.3.1   *Contour plots*

Contour lines are almost more useful by themselves than in combination with surface plots: a *contour plot* shows only the contour lines, from the bird's-eye perspective. Figure C.8 is yet another representation of the function from figure C.1—this time as a contour plot. Listing C.1 summarizes the commands necessary to go from a three-dimensional surface plot to a flat contour plot.
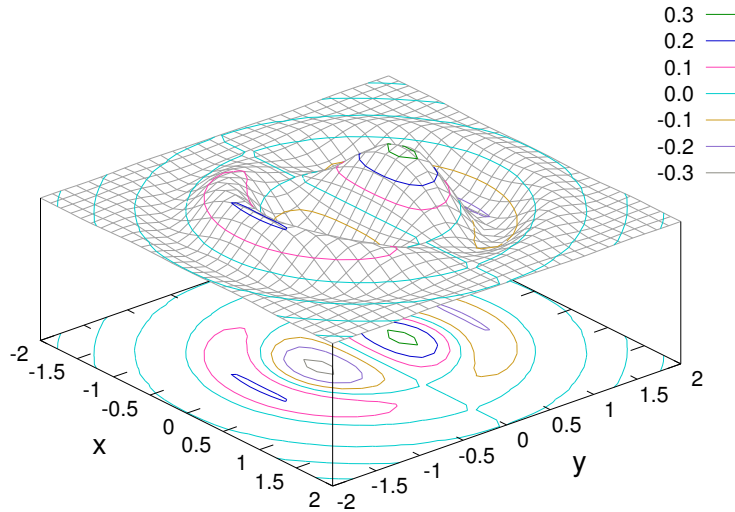
Figure C.7 Adding contour lines at the base and on the surface using `set contour both` and `set cntrparam levels incremental -0.3,0.1,0.4`. The function is the same as in figure C.1.

The quality of the resulting plot depends strongly on the choices for `set isosamples` and `set samples`. The former controls the number of nodes at which the function is evaluated, and the latter controls the number of line segments used to draw the contours. Higher resolution is obviously better, but it requires more computational power. (Values from 30 to 300 for either seem to work well in practice.)
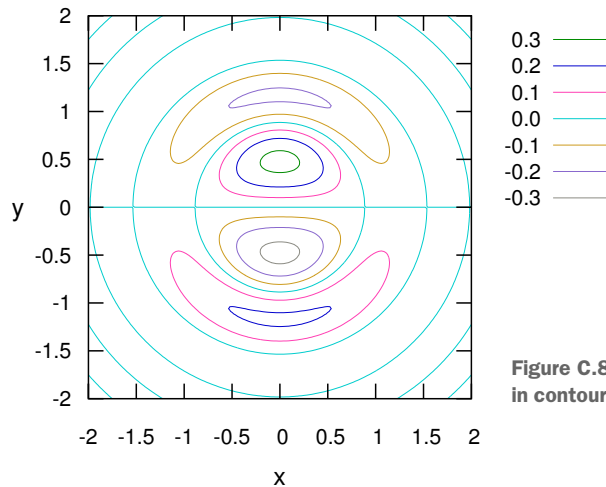
Figure C.8 The function from figure C.1, plotted in contour view. See listing C.1 for the commands.

```
set view map              ◁──────  Chooses bird's-eye view
set size ratio -1         ◁──────  Chooses equal units in both directions
unset surface             ◁──────  Switches off the surface
set contourv              ◁──────  Switches on contour lines
set isosamples 150        ◁──────  Increases the sampling frequency
set cntrparam levels 10   ◁──────  Increases the number of contour lines
set clabel "%.1f"         ◁──────  Chooses the format of the contour labels in the key
```

C.3.2   *Customizing contour lines and their labels*

As you can see in figure C.8, gnuplot assigns a new line type from the predefined sequence of types to each new contour (see section 9.3). To customize the assignment of colors to contours, you need to redefine the appropriate style using `set linetype`. It's also possible to use only a single line type for all contours, using the command `set cntrlabel onecolor`.

A word of warning: gnuplot's commands for customizing contour plots aren't very polished. Most actions are achieved through two commands—`set cntrparam` and `set cntrlabel`—but each is responsible for several, only vaguely related tasks, distinguished by various suboptions. Also, and in contrast to usual gnuplot usage, these commands don't allow you to combine multiple suboptions into a single invocation.

FREQUENCY AND SPACING OF CONTOURS

To influence the *frequency* and *spacing* of contour lines, use `set cntrparam levels`. The different ways to do this are reminiscent of the options used to place tic marks along the axes (see section 8.3.4):

```
set cntrparam levels [ auto [{int:n}]
                     | discrete {flt:z0} [, {flt:z1} [, ...]]
                     | incremental
                         {flt:start}, {flt:inc} [, {flt:end}] ]
```

When using `auto`, approximately five uniformly spaced contour lines are drawn. You can pass an integer argument to `auto` to change the number of contour lines that are generated. All these settings are approximate, because gnuplot attempts to place contour lines at round numbers.

Using `discrete`, you can control exactly where contour lines are placed. The `discrete` suboption takes a comma-separated list of z values as an argument, at which contours should be drawn. All z values for which contours should be shown must be specified in a single call to `set cntrparam levels discrete`; you can't add individual contour lines to autogenerated ones. Finally, you can fix a starting z value and an increment (and optionally an end value) through the `incremental` keyword.

CONTOUR LABELS

By default, gnuplot automatically generates an entry in the key (see section 7.4) for each contour line. The formatting of these labels can be customized using `set`

`cntrlabel format`. The format string can contain any of the familiar conversion specifiers (see section 8.3.3):

`set cntrlabel [format "str:format"] [font "str:font"]`

It's also possible to place the labels directly onto the *graph*, rather than grouping them into a separate key, but (at the time of this writing) support for this feature is still rudimentary. To place labels onto the graph, use the `with labels` style together with the `splot` command. This will draw only the labels; if you want both contours and labels, you need to plot the function of data twice in the same command: once `with lines` and once `with labels`.

Two parameters control the number and placement of labels that are drawn for each contour line: `set samples` and `set cntrlabel`. By default, a label is placed at every 20th line segment—hence a larger value of `set samples` means more labels. You can modify the number and position of labels using

`set cntrlabel [start {int:segment}] [interval {int:segment}]`

Using the `interval` keyword, you can change the frequency of labels: a higher number means more line segments are drawn between consecutive labels. Unfortunately, values higher than 100 are silently truncated to 100. The argument to the `start` keyword determines at which line segment (starting from some arbitrary origin) the first label is drawn. Changing this value, you can slide *all* labels in the graph along their contours.

It's currently not possible to shift or rotate labels. Feel free to experiment, but you may be better off using explicit labels (using `set label`; see section 7.3.3) to annotate a contour plot on the graph.

SMOOTHING CONTOUR LINES

The `set cntrparam` command can also control how contours are generated—this may be useful at times, for example if you want to graph noisy data. The relevant suboptions are as follows:

`set cntrparam [ linear | cubicspline | bspline ]`

`set cntrparam [ points {int:q} ]`
`set cntrparam [ order {int:q} ]`

To draw contours, gnuplot determines a set of points (called *nodes*) at which the function (or data) has the same z value. Through the first set of keywords, you can choose how these points are connected: `linear` (the default) uses straight lines; `cubicspline` uses a smooth curve, which is guaranteed to pass exactly through the nodes; and `bspline` uses a curve that's guaranteed smooth but isn't guaranteed to pass through the nodes exactly. For "crumply" surfaces, or low resolutions (that is, few grid points), one of the spline options may give significantly better results than the default.

You can specify two additional parameters that are relevant to the way splines are drawn (they have no meaning for `linear` mode). The `points` option controls the number of points for which the interpolation between any two neighboring nodes is evaluated when drawing the contour. More points again mean smoother curves. With `order`, you can set the order of the spline to be used (this option is only relevant for `bspline`). The order must be an integer between 2 and 10. The greater the order, the smoother the contour. Note that both `points` and `order` require *separate* calls to set `cntrparam`—they can't be combined with the type of curve (`linear`, `cubicspline`, or `bspline`) in the same call.

C.4   Plotting data from a file using splot

Everything we've done so far has involved the plotting of functions only. Can gnuplot use data from a file to generate surface and contour plots? Of course it can!

Gnuplot supports two data file formats to work with `splot`. For data files in other formats, or for data that's not on a grid, check the `dgrid3d` option described in section C.5.

C.4.1   Grid format

If the data file is in *grid format*, each line must contain both x and y coordinates, as well as the z value that is to be plotted. Data must be organized into data blocks, which are separated from each other by a single blank line. Each block must contain all data points for *a single row of data points*, parallel to the x axis (in other words, within each data block, the x coordinate stays constant, but the y coordinate runs). The following listing shows a short example file demonstrating this format. The corresponding plot is in figure C.9.

> **Listing C.2   Simple data file suitable for `splot`—see figure C.9 (file: grid)**

```
# x       y       z
  0      -1      10
  0       0      10
  0       1      10

  1      -1      10
  1       0       5
  1       1      10

  2      -1      10
  2       0       1
  2       1      10

  3      -1      10
  3       0       0
  3       1      10
```

Keep in mind one very important restriction: *all data blocks must contain the same number of data points*. If the number of data points isn't the same in all data blocks, or if even a single data point is missing or invalid, `splot` can't draw a surface.
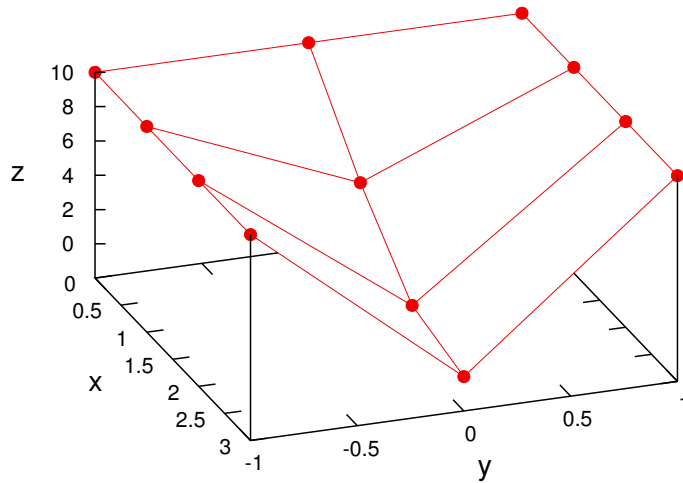
Figure C.9 **The data from listing C.2 plotted using `splot "grid" u 1:2:3 w linesp`**

This is a *formal* requirement, only: there's no constraint that the data *values* (that is, the x and y coordinates) form a regular grid. Small deviations of the x and y positions from regularity are fine, but the plot may look strange or may not be very useful if the underlying grid is too distorted.

As mentioned when we first introduced the `splot` command earlier in this appendix, many of the directives familiar from the `plot` command are available when plotting gridded data using `splot`. In particular, you can have different data sets in a single file, separated from each other by double blank lines (as discussed in chapter 3). You can then select a specific such set using `index`. The `every` directive can be used to pick out subsets of each data set. Finally, the `using` directive can be used to choose the data columns for x, y, and z values and to apply data transformations in the usual fashion (see section 3.3).

MISSING DATA

It's instructive to see what gnuplot does with a defective file that doesn't follow this format. Figure C.10 shows the results of the `splot` command for a file that's the same as in listing C.2, except that the last entry (the point at (3,1,10)) has either been replaced with the not-a-number marker `NaN` or been omitted entirely.

If gnuplot finds a not-a-number marker (as defined by `set datafile missing`—see section 4.3.4), it can still draw a complete surface, just omitting the panels that touch the invalid data point. But if the data point is missing entirely, then no surface can be drawn. Instead, the `splot` command connects all points in one data block consecutively, but doesn't connect data points separated (in the file) by a single blank line—this is exactly what the `plot` command does when it encounters single blank lines in a file. But in addition, `splot` staggers the plots for successive data blocks (front to back) and thus gives the illusion of optical depth. But cross lines (parallel to the x axis) are
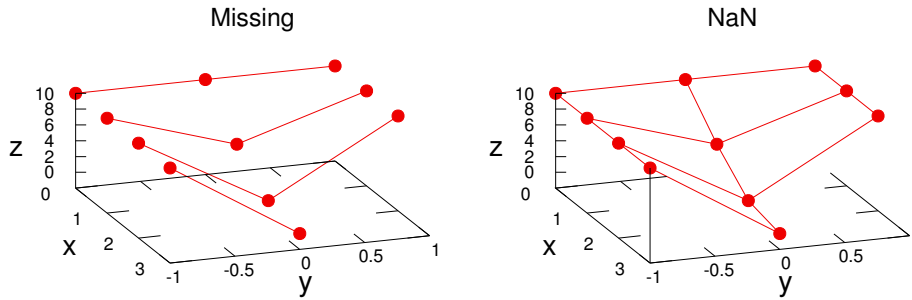
Figure C.10   The `splot` command requires all data points to be supplied on a regular grid, with none missing. This figure shows what happens when the last point (at x=3, y=1) has been replaced with a not-a-number marker or omitted entirely. See listing C.2 for the original data file. The `splot` command is the same as in figure C.9.

only drawn when the data in the input file fulfills the requirements of regular, gridded data blocks, as discussed earlier.

C.4.2   *Matrix format*

The *matrix format* is an alternative file format for data on a regular, rectangular grid. It's more compact, because it doesn't store the (redundant) information on x and y coordinates for every data point—only the z values at the grid locations are kept. There are two permissible matrix formats: one that stores the coordinates of the grid points (without duplication) together with the z value, and a super-compact format that does away with the grid coordinates altogether.

Table C.3 shows the general structure of both formats. In the general matrix format, the first *row* contains the x coordinates of the grid points, and the first *column* contains the y coordinates. The entry in the top-left corner is ignored but must contain a numeric value. In the compact format, the coordinates aren't stored explicitly. Instead, when reading a file in the compact format, gnuplot assumes that the coordinate values consist of pairs of positive integers, starting at `(0, 0)`.

Table C.3   The matrix formats

	General: `nonuniform matrix`					Compact: `matrix`				
Structure	NA	x0	x1	x2	x3	...	z00	z01	z02	z03	...
	y0	z00	z01	z02	z03		z10	z11	z12	z13	
	y1	z10	z11	z12	z13		z20	z21	z22	z23	
	y2	z20	z21	z22	z23		z30	z31	z32	z33	
	y3	z30	z31	z32	z33		...				...
	...					...					
Example	0	0	1	2	3		10	10	10	10	
	−1	10	10	10	10		10	5	1	0	
	0	10	5	1	0		10	10	10	10	
	1	10	10	10	10						

Upon reading a file in matrix format, gnuplot internally transforms the contained information into a three-column grid format, with the x coordinates in the first column, the y coordinates in the second, and the data values in the third. These columns can then be selected in a regular `using` specifier.

> **TIP** Keep in mind that a file in matrix format is internally transformed into the equivalent grid format. Construct a `using` specifier as you would for a file in grid format.

A WORKED EXAMPLE

Let's say that the two example files in table C.3 are called matrix and packedmatrix, respectively. Then the following three commands are equivalent:

```
splot "grid" u 1:2:3 w l                        ⟵——— Using grid format

splot "matrix" nonuniform matrix u 1:2:3 w l     ⟵——— Using matrix format

pos( k, origin, spacing ) = origin + k*spacing                      Using packed
splot "packedmatrix" matrix u (pos($1,0,1)):(pos($2,-1,1)):3 w l    matrix format
```

The only difference is the appearance of the new keywords `matrix` (indicating a file in packed matrix format) and `nonuniform matrix` (for a file in general matrix format: that is, including coordinate information in the first row and column).

A special consideration applies to files in the packed matrix format, because the integer pairs that gnuplot supplies instead of coordinate values must be transformed into true locations. As long as the grid points lie on a regular, rectangular grid, this can always be accomplished through a simple linear transformation, involving the coordinate value of the origin and the lattice spacing, as shown in the listing.

C.5   Smooth surfaces

Surface plots can be fun, but things can go wrong in practice—in particular, when you're graphing data (as opposed to plotting smooth functions):

- Data is scattered (not on a regular grid), so that no regular surface can be drawn.
- Data is sparse, so that the surface consists only of a few elements and is therefore not easy to recognize.
- Data is noisy, so that the surface appears overly bumpy.
- The data file isn't in a suitable format.

C.5.1   The set dgrid3d facility

Gnuplot provides a slick little facility that takes arbitrary input data and generates a smooth interpolation onto a two-dimensional grid: the `dgrid3d` option. If `dgrid3d` is on, `splot` doesn't plot the raw data. Instead, it reads the data and then generates an approximation to this data for every point of a regular grid. The number of grid points and some aspects of the interpolation algorithm can be controlled through `set dgrid3d`:

```
set dgrid3d [ {int:nx} [, {int:ny} [, {int:q} ] ] ]

set dgrid3d [ {int:nx} [, {int:ny} ] ]
            [ splines
              | gauss | cauchy | exp | box | hann
                  [ {flt:dx} [, {flt:dy} ] ] ] ]
```

By default, dgrid3d is disabled. When it's enabled, three-dimensional data read from a file is always treated as a scattered data set. A grid with dimensions derived from a bounding box of the scattered data and size as specified by the nx and ny parameters is created for plotting and contouring. The grid is equally spaced in x rows and in y columns; the z values are computed as weighted averages or spline interpolations of the scattered points' z values. In other words, a regularly spaced grid is created, and a smooth approximation to the raw data is evaluated for all grid points. Only this approximation is plotted, but not the raw data. The number of columns defaults to the number of rows, which defaults to 10.

Several algorithms are available to calculate the approximation from the raw data. Some of these algorithms can take additional parameters. All the algorithms have in common that the closer a data point is to a grid point, the more effect it has on that grid location.

The splines algorithm calculates an interpolation based on thin plate splines. It doesn't take additional parameters, which is convenient but also means the algorithm can't be adapted to take the nature of the data into account.

An alternative and more flexible algorithm is based on weighted averages.[2] Each data point is replaced with a *kernel*: a smooth function that's peaked at the location of the data point. The contributions from all kernels are combined to find the weighted average for the plot:

$$z \;=\; \frac{\sum_i w(d_i) \cdot z_i}{\sum_i w(d_i)}$$

Here, z_i is the value of the ith data point, and d_i is the Euclidean distance between the current grid point and the location of the ith data point. The function $w(d)$ is the smoothing kernel. Several kernels can be selected by supplying the appropriate keyword to set dgrid3d. Table C.4 lists the available choices.

When using one of the five smoothing kernels from table C.4, up to two additional parameters can be specified: dx and dy. These are used to rescale the coordinate differences when calculating the Euclidean distance:

$$d_i \;=\; \sqrt{\left(\frac{x - x_i}{dx}\right)^2 + \left(\frac{y - y_i}{dy}\right)^2}$$

---

[2] The dgrid3d facility allows for some other options as well. Check the standard gnuplot reference documentation for details.

Table C.4   **Smoothing kernels available with `set dgrid3d`**

Keyword	Definition
`gauss`	$w(d) = \exp(-d^2)$
`cauchy`	$w(d) = 1/(1+d^2)$
`exp`	$w(d) = \exp(-d)$
`box`	$w(d) = 1$ if $|d| < 1$; $w(d) = 0$ otherwise
`hann`	$w(d) = \frac{1}{2} \cdot (1-\cos(2\pi d))$ if $|d| < 1$; $w(d) = 0$ otherwise

The value of `dy` defaults to the value of `dx`, which defaults to 1. The parameters `dx` and `dy` make it possible to control the radius over which data points contribute to a grid point *in the units of the data.*

Figure C.11 shows the data from listing C.2 plotted when smoothed using `dgrid3d` with the Gaussian kernel. (The Gaussian kernel is probably the most generally useful and versatile kernel—unless you have specific reasons to use a different one, it'll serve you well in a variety of situations.)
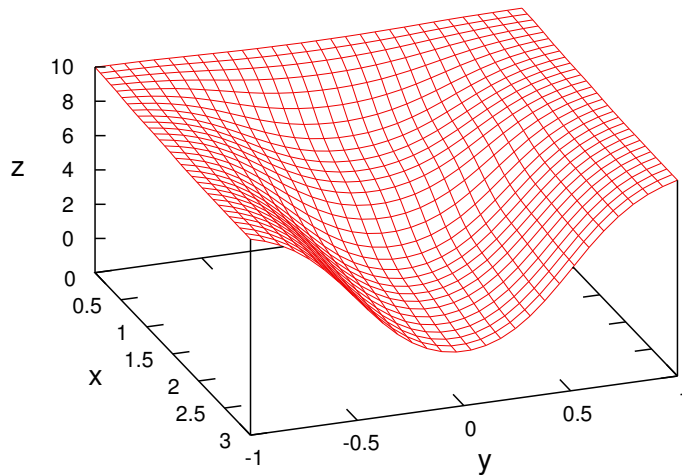
Figure C.11   The same data as in figure C.9, but plotted after turning on smoothing using `set dgrid3d 30,30 gauss 0.6,0.6`

The `set dgrid3d` facility isn't perfect. In particular, it can't be turned on and off for individual data sets—either all data sets are smoothed with it, or none. This makes it impossible, for instance, to plot the raw data together with the smooth surface. But overall, `dgrid3d` is a great tool to generate good-looking two-dimensional graphs from otherwise unsuitable data.

Words of caution

Graphs generated with the `splot` command can be visually very appealing, and you've seen some nice examples in this appendix. Nevertheless, my recommendation is to use them sparingly and to also explore other ways of representing multivariate data (for example, take a look at chapter 14). Surface plots are often stunning, but (because of the additional need to find a suitable view point) getting them "right" is disproportionately more difficult. Reading *quantitative* (as opposed to qualitative) information off of them is often tricky, if not impossible. Finally, they simply aren't suitable for noisy data sets. But they can be effective for conveying the broad aspects of a multidimensional data set, in particular to an audience that has a harder time making sense out of other ways of representing such data (such as false-color plots: for those, see appendix D).

Palettes and false-color plots

Gnuplot provides a facility to map numeric values into a continuous range of colors. This allows you to create *false-color plots* (or *heatmaps*): graphs in which a numeric value is represented by a color, instead of by position.

In this appendix, we'll first study how to set up such mappings between numbers and colors. Because the values in a false-color plot typically form a continuum, the mapping usually consists of a continuous color *ramp* or color *gradient*. The gnuplot command to create and manage such color gradients is `set palette`, and hence I'll refer to these mappings as *palettes*. After introducing the commands to define palettes, we'll look in detail at a catalog of example palettes for a variety of different applications and point out their strengths and weaknesses.

Next, we'll study ways to use palettes to create colored graphs in gnuplot. Gnuplot's palette feature is quite flexible, and you can use it for a variety of purposes, of which heatmaps are only one (but arguably the most important) example.

D.1   Warm-up examples

Before discussing all the various commands and options in detail, let's quickly look at an example that demonstrates the capabilities that'll be introduced in this appendix. Consider again the function that was studied in appendix C. There, you saw surface plots (such as figure C.1) and contour plots (figure C.8) of this function. Now we're going to add some color!

Let's begin simply. The following commands create a colored figure (figure D.1):

```
set isosamples 200

set palette defined ( 0 "blue", 0.5 "light-grey", 1 "red" )

splot [-2:2][-2:2]
➡     exp(-(x**2 +  y**2))*cos(x/4)*sin(y)*cos(2*(x**2+y**2)) w pm3d
```

Two items are new. First, the `set palette` command is used to create a smooth color gradient palette in gnuplot. The `w pm3d` style in the `splot` command then uses this palette to create a colored surface by mapping the surface elevation into the palette. (The abbreviation stands for "palette-mapped three-dimensional" plot—you can see why.)

Of course, other color schemes are possible—that's the point of the `set palette` command. Later in this appendix (in section D.2), you'll learn how to set up your own palettes using the `set palette` command.

D1
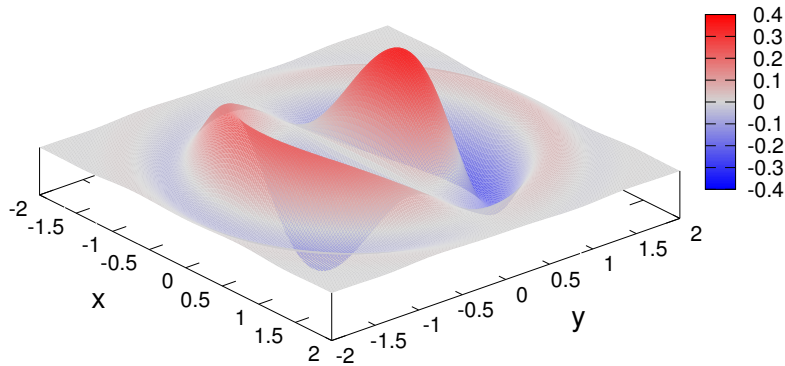
Figure D.1　Adding color from a continuous palette to a surface plot. Compare figure C.1.

But color palettes are not only available for surface plots (as in figure D.1), but also for many other plots. Figure D.2 shows the same function again, but instead of using contour lines, it now uses *only color* to indicate the local value of the function at each point.

　　The graph in the left panel was constructed with a very simple palette, using the following commands:

```
set isosamples 400
set view map; set size square

set palette defined (-1 'white', 0 'blue', 0 'red', 1 'white' )
splot [-2:2][-2:2]
➡      exp(-(x**2 + y**2))*cos(x/4)*sin(y)*cos(2*(x**2+y**2)) w pm3d
```

The palette for the other graph is a tad more complicated; you'll see its definition later, in section D.2.5.
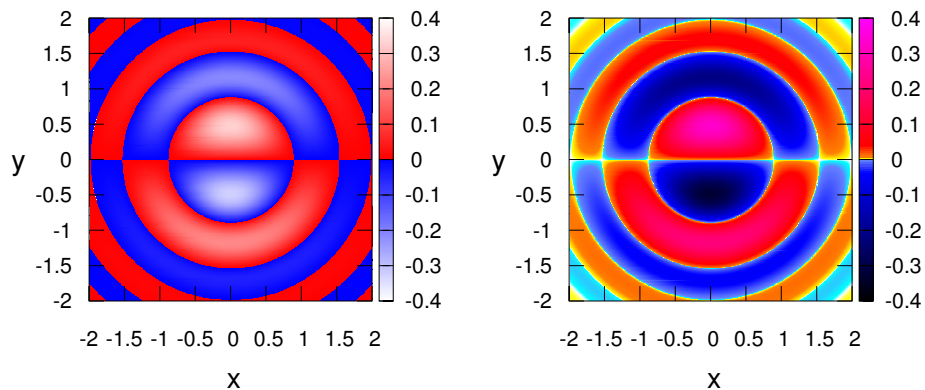
Figure D.2　Two examples of false-color plots. The function is the same as in figure D.1. The graph on the left was drawn using a simple palette defined by `set palette defined (-1 'white', 0 'blue', 0 'red', 1 'white' )`; the graph on the right uses the palette from panel N in figure D.4.

For yet another kind of graph that used a color palette, go back to figure 9.5, when we discussed data-dependent colors. The time has come to introduce all the detail that we skipped back then!

D.2 Creating palettes

Gnuplot's command to create color gradients is `set palette`. The command is pretty smart and does a lot of legwork for you. It's also fairly complex and combines many different bits of functionality. In fact, one of its problems is its almost promiscuous flexibility: the command places few limits on user input, which can be both very convenient and very confusing, depending on whether you understand how gnuplot processes the arguments.

There are basically two different ways to define a color gradient using `set palette`:

- A gradient can be defined through a discrete set of colors at fixed positions within the numerical range. The `set palette` command fills in the gaps between these nodes through linear interpolation, separately for each color component.
- A gradient can be defined through a set of three functions (one for each color component) that map the numerical range to a color component.

D.2.1 Color models and components

The `set palette` command can handle five different color models: RGB (red-green-blue), HSV (hue-saturation-value), CMY (cyan-magenta-yellow; mostly used in print publishing), YIQ (US color television standard), and CIE XYZ (a model based on color perception).[1] For computer graphics, RGB and HSV are by far the most important.

You can select the color model using the `model` keyword. For example:

```
set palette model RGB
set palette model HSV
```

The color model selection is *sticky,* meaning it remains in effect until explicitly changed: you don't need to include the `model` specification in every call to `set palette`. The default is the RGB model.

> **TIP** The color model in `set palette` is sticky and silently stays in effect until explicitly overridden. Unless the color model is RGB, I always include the explicit choice of color model in each call to `set palette`.

All color models represent color through exactly *three* components, and formulas exist to translate between the different color models. (Transparency would require a fourth component, but `set palette` doesn't handle transparent colors, so three components are sufficient.)

---

[1] Check the respective Wikipedia pages for details.

To specify the three components, the `set palette` command accepts three differ-ent formats:

- A named color, such as `"red"`. Use `show colornames` for a list of permissible colors.
- A hex string, such as `"#ff0000"` or `"0xff0000"`. Both formats are equivalent.
- A numeric triple, such as `1 0 0`. The numbers must be between 0 and 1 and be separated by whitespace, without punctuation or delimiters.

Here are two colors in all three formats:

```
"red"       "#ff0000"      1 0 0
"grey"      "0xc0c0c0"     0.75 0.75 0.75
```

Here's the first potential for confusion: the `set palette` command separates the input into three components (let's call them A, B, and C) and *interprets them according to the currently selected color model*. Every format is permissible for every color model, and gnuplot makes no attempt to "guess" the intended color model from the input format.

Let's consider an example. Regardless of whether your input was `"red"` or `"#ff0000"` or `1 0 0`, gnuplot will interpret it as follows: first channel (component A) at maximum, and the other two channels (components B and C) at minimum. If the chosen color model is RGB, then this amounts to "full red, no green or blue." But if the chosen color model is HSV, this means "full hue, no saturation or value" (that is, black, by virtue of the zero value in the third channel). Similarly, the arguments `"grey"`, `"0xc0c0c0"`, and `0.75 0.75 0.75` will all be interpreted as "all three channels at three-quarters of their maximum." In the RGB space, this amounts to a light gray; but in the HSV space, the color will be a medium purple! In every case, the input is split into its three components *first*, and only then are the components interpreted accord-ing to the currently active color model. (In fairness, `set palette` warns you if you try to use named colors with color spaces other than RGB.)

> **TIP**   The `set palette` command doesn't restrict the input format to appro-priate color models. Instead, all input is *first* broken into components, which are only then interpreted according to the currently selected color model.

D.2.2   *Defining palettes through nodes*

You can define a palette by specifying a set of colors at fixed locations within the plot range, and gnuplot will fill in the gaps in between. This is, in my opinion, the most useful way to create palettes in practice.

You use the `defined` keyword to specify the list of colors and positions. You can use any of the color formats discussed in the previous section, and hence the following four commands are equivalent:

```
set palette defined ( -1    "red", 0   "white", 1    "blue" )
set palette defined ( -1 "#ff0000", 0 "#ffffff", 1 "#0000ff" )
set palette defined ( -1   1 0 0, 0    1 1 1, 1    0 0 1  )
set palette defined ( -1    "red", 0 "#ffffff", 1    0 0 1  )
```

The `defined` keyword must be followed by a comma-separated list in parentheses. Each of the entries between the commas consists of a position, followed by a color. Don't get confused when using the third format (numeric triples): each entry indeed consists of four numbers, without any punctuation between them.

There's no limit on the number of entries in the list of colors. The values used as positions aren't restricted: they can be positive or negative, integer or floating-point. The only constraint is that they must form a non-decreasing sequence of numbers. It's possible to repeat a position, and doing so can be useful to create palettes with sharp transitions, as we'll discuss later. Two examples:

```
set palette defined ( -1 "red", 0 "red", 0 "blue", 1 "blue" )    ◄——————  Valid
set palette defined ( 0 "red", -1 "white", 1 "blue" )    ◄─┐
                                                          │ Not valid: positions
                                                          └ are out of order!
```

This brings us to the second potential source of confusion when using `set palette`: the positions need not correspond to actual values from the plot range! In fact, they aren't absolute positions at all—they're merely *relative* positions in the interval spanned by the smallest and the greatest of the positions (that is, by the first and the last entries between the parentheses). The following four commands all define exactly the same palette:

```
set palette defined (  -1 "red", 0   "white",  1 "blue" )
set palette defined ( -10 "red", 0   "white", 10 "blue" )
set palette defined (   0 "red", 0.5 "white",  1 "blue" )
set palette defined (   7 "red", 14  "white", 21 "blue" )
```

In every case, the total range of values spanned by the positions will be shifted and scaled to the unit interval. All nodes retain their relative positions in the process. In particular, the positions need not correspond to actual values from the plot range; you use a special command (`set cbrange`—see section D.3) to map the entire palette to the desired plot range.

> **TIP** The positions assigned to nodes when using `set palette defined (...)` aren't absolute positions; they're relative positions within the interval of values spanned by the first and last node. Gnuplot internally normalizes the values by shifting and scaling the nodes into the unit interval for further processing.

Gnuplot interpolates linearly between nodes, for each component separately. The interpolation is done *in the currently selected color space*, and only afterward are the interpolated values transformed to RGB for actual presentation. For example, the following command creates a standard rainbow palette:

```
set palette model HSV defined ( 0 0 1 1, 1 1 1 1 )
```

In HSV space, gnuplot interpolates linearly between a hue value of 0 and a hue value of 1, but if you look at the resulting palette in RGB space, the behavior is most definitely not linear.

READING NODES FROM FILE

It's possible to read the nodes from file, using the `file` keyword. The file is treated as a regular data file, which means the `using` directive is available to pick out columns and to apply inline transformations.

Gnuplot expects three or four columns (the position, followed by the three color components). If only three columns are provided, the nodes that are read from file will be spaced uniformly along the palette. Again, the values of the color components that are read from file will be interpreted according to the currently active color model!

Here are two syntax examples, with and without an explicit `using` directive:

```
set palette file "palette.gpf"
set palette file "palette.gpf" using 1:2:3:4
```

The color values read from file (or resulting from an inline transformation) should be between 0 and 1. Values exceeding this interval are *truncated* back into this interval.

> **TIP** Any color values that are read from file using `set palette file ...` and that exceed the unit interval are truncated back into the unit interval.

D.2.3   *Defining palettes with functions*

Rather than provide a set of nodes and let gnuplot interpolate between them, you can also provide a set of three functions (one for each color component). Depending on the color model chosen, the components will be interpreted as red, green, and blue channels (RGB), or as hue, saturation, and value channels (HSV), and so on.

Instead of using `defined`, you use the `functions` keyword in the `set palette` command. The functions are arbitrary, but each one must map the unit interval [0:1] into the unit interval [0:1]. The independent (*dummy*) variable in the function definition *must* be called `gray` (not `x`, not `grey`!).

Defining palettes using functions works best in the relatively intuitive HSV color space. Here are a few simple examples:

```
set palette model HSV functions gray, 1, 1        ◁———— Standard rainbow
set palette model HSV functions 0, 0, gray        ◁———— Linear grayscale
set palette model HSV functions 0, 0, sqrt(gray)  ◁———— Nonlinear grayscale
set palette model HSV functions gray, 1-gray, 1   ◁———— Rainbow that fades to white
```

> **TIP** When defining a palette through functions with `set palette functions ...`, don't forget that the independent (dummy) variable in the function definitions must be called `gray`. If you choose any other variable name (such as `x`), gnuplot will *not* warn you—things just won't work!

OTHER WAYS TO DEFINE PALETTES WITH FUNCTIONS

The `set palette` command offers some further features that I want to mention in passing. Check the gnuplot standard reference documentation for details.

The command `set palette cubehelix` (which is intended to be used with the RGB color model) creates a palette that traces out a rainbow, while simultaneously increasing color intensity. Several sub-options allow you to control various aspects of the resulting palette.

The `set palette rgbformulae` command is intended to reduce the size of Post-Script files. When using this option, you must define your palette in terms of several predefined functions. When exporting your graph to PostScript, these functions are written to the PostScript file so that the palette can be evaluated analytically when the PostScript file is processed. Because the palette need not be saved explicitly, the resulting PostScript file is smaller.

D.2.4 *Displaying and exporting palettes*

The following command is a valuable tool when developing palettes:

```
test palette
```

It produces a figure (using the current settings of `set output` and `set terminal`!) like the one in figure D.3. The bottom panel shows the palette as a spectrum of colors; the top panel shows the intensity levels of the red, green, and blue components. (Regardless of the choice of color model, the `test palette` command *always* displays RGB components.) Also included in the top panel is the value of the *NTSC luminance*, which corresponds to the Y channel in the YIQ model. It indicates the result if the color were displayed on a black-and-white TV screen.
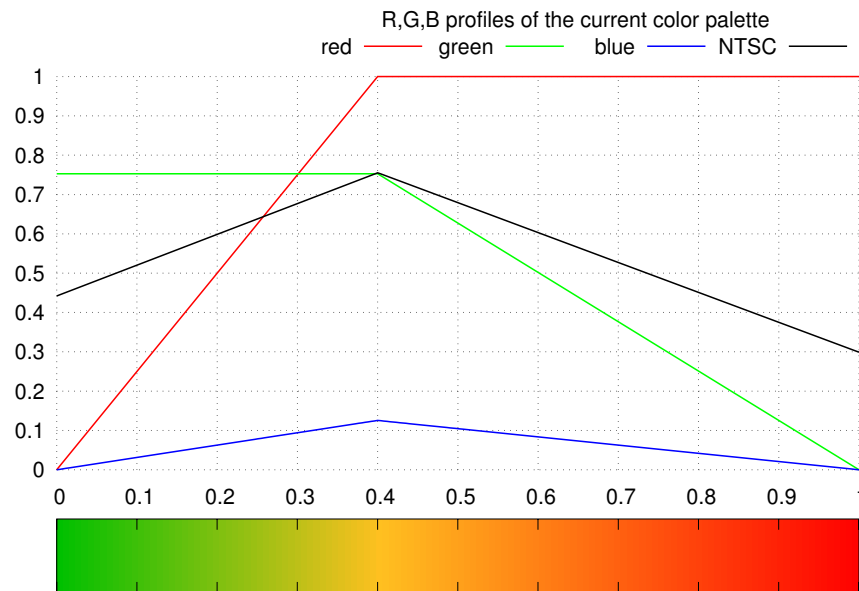
Figure D.3 **The result of the `test palette` command for a palette defined with `set palette` defined (0 'web-green', 0.4 'goldenrod', 1 'red') or, equivalently, `set palette` defined (0 "#00c000", 0.4 "#ffc020", 1 "#ff0000")**

QUERYING THE PALETTE DEFINITION

In addition to this graphical visualization of the palette with `test palette`, you can query the current definition using

```
show palette
```

This prints the current values of all sub-options of `set palette` to the screen, including the three function definitions if the palette was defined using `set palette functions`. If the palette was created from a set of discrete nodes using `set palette defined (...)`, then the printed message won't include explicit information on the nodes. Instead, you must use

```
show palette gradient
```

to list them.

EXPORTING A PALETTE

It may be desirable to *export* a finished palette as a set of RGB triples: for example, to use it in another application. You can do so using the `show palette palette` command:

```
show palette palette {int:n} [ int | float ]
```

This prints a list of RGB triples to the output channel identified by the current value of `set print`. The mandatory integer argument gives the number of intermediate colors in the exported palette. By default, the output is formatted for human readability, but by giving either the `int` or the `float` keyword, you can obtain a listing that's easier to parse from a program. (With `int`, the RGB values are integers in the range 0 to 255; with `float`, they're floating-point values between 0 and 1.)

SAVING A PALETTE

All the details supplied to the `set palette` command are included in the information persisted with `save`; hence an existing graph (and its palette) can be re-created using `load`, as you'd expect. You should be aware, though, that gnuplot saves the color for each node in a palette only in the *third* format, which consists of three numbers between 0 and 1. In particular, gnuplot will *not* write the names of named colors to file when using `save`. The resulting command files are therefore, by construction, much less human-readable. If you want to save a palette to file with the idea of later continuing to edit it manually, then you should edit the respective command file in an editor (see section 12.2), rather than use gnuplot's `save` command.

> **TIP**   Even though you may have defined a palette in terms of named colors, gnuplot will only persist the equivalent numerical values for each of the three color components to file. To retain the named colors, you must edit and save the command file in an external editor.

D.2.5 *Some example palettes*

Let's look at some example palettes. The purpose here is twofold: first, I want to demonstrate how to use the `set palette` command; and I also want to provide you with recommendations for palettes and specific color choices that work well in practice. (For all the following, compare figure D.4.)[2] Defining your own palettes is particularly important because gnuplot's default palette isn't very good. To obtain good results, you need to define your own.

> **TIP** Because gnuplot's default palette isn't very good, it's essential that you learn how to define your own palettes. Moreover, false-color plots generally need palettes that have been adapted to the specific graph in order to be effective, necessitating you to have the required know-how.
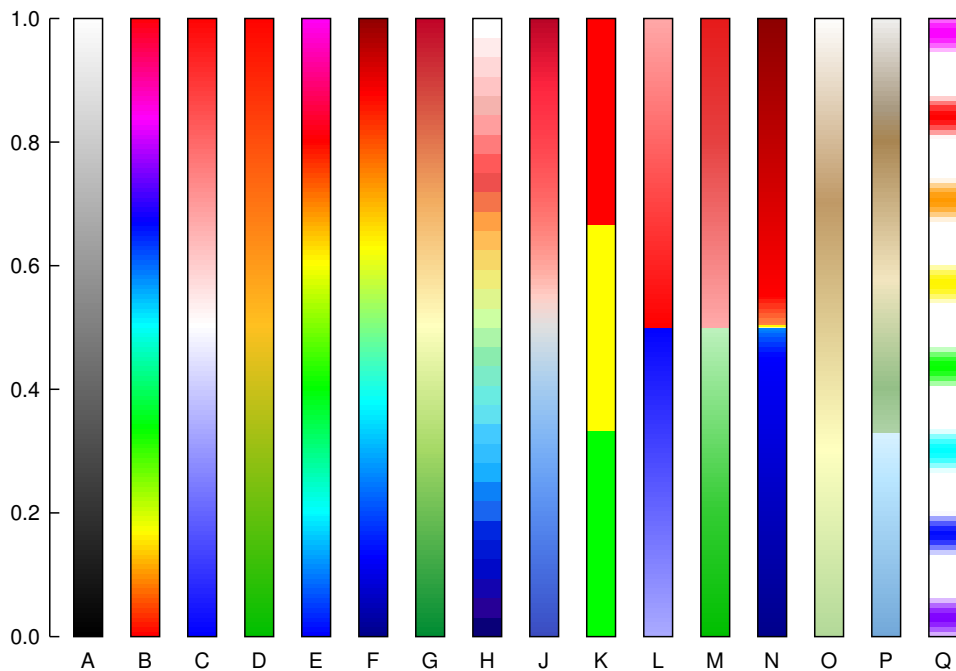
Figure D.4 Several example palettes: each panel in the figure displays a different palette. See the text for details.

GRAYSCALE AND STANDARD RAINBOW

Two palettes are particularly suitable to demonstrate the various features of the `set palette` command: the linear grayscale and the standard rainbow (panels A and B, respectively, in figure D.4). Here are four equivalent ways of creating a linear grayscale

---

2 You can find a very large collection of palettes at http://soliton.vm.bytemark.co.uk/pub/cpt-city. Although many of them are intended for graphic design, not data visualization, the site is a terrific source of inspiration.

palette, using all combinations of the RGB and HSV color model together with either
the `defined` or the `functions` feature:

```
set palette defined ( 0 "black", 1 "white" )
set palette functions gray, gray, gray

set palette model HSV defined ( 0 0 0 0, 1 0 0 1 )
set palette model HSV functions 0, 0, gray
```

The standard rainbow only makes sense to define in HSV space. Here are two ways,
using nodes and functions, respectively:

```
set palette model HSV defined ( 0 0 1 1, 1 1 1 1 )
set palette model HSV functions gray, 1, 1
```

Neither of these two palettes works particularly well in practice. The standard rainbow
doesn't convey an intuitive sense of ordering; it wraps around (so that the highest and
the smallest values are assigned similar colors); and its garish colors don't aid percep-
tion. Graphs drawn using a simple grayscale are dominated by shades of black and
white, which are harder to distinguish than shades of darker and lighter grays. More-
over, a grayscale isn't in color!

THREE SIMPLE WORKHORSES

Here are three palettes that are simple enough to keep in mind, so that you can re-
create them on the spot. The first consists of the colors blue, white, and red (panel C
in figure D.4):

```
set palette defined ( 0 "blue", 0.5 "white", 1 "red" )
```

You should commit this palette to memory! It has several advantages:

- It has an intuitive sense of ordering.
- It provides strong gradients on both sides of the central white band. (The white
  can be replaced with gray if it's hard to distinguish from the background.)
- The white band need not be symmetric in the unit interval. By shifting it rela-
  tive to the other two colors, the palette can be adjusted to asymmetric problems
  (for example, `set palette defined ( 0 "blue", 0.25 "white", 1 "red" )`).
- The basic idea is easy to remember and set up.

   TIP   Despite its extreme simplicity, the blue/white/red palette is amazingly
convenient and versatile. The color white should be replaced by a light gray if
white would be hard to distinguish from the color of the background.

Another combination of colors that conveys a sense of ordering consists of the traffic
light colors: red, yellow, and green (panel D in figure D.4). In addition to the sense of
ordering, this particular trio of colors invariably also conveys a semantic "good/bad"
meaning—which may or may not be appropriate:

```
set palette defined ( 0 "web-green", 0.5 "goldenrod", 1 "red" )
set palette defined ( 0 "#00c000",   0.5 "#ffc020",   1 "#ff0000" )
set palette defined ( 0 0 3/4. 0,    1 1 3/4. 1/8.,   2 1 0 0 )
```

The three definitions given are equivalent—the one using gnuplot's color names may be the easiest to remember, the one using hex strings is the most portable, and the one using numeric fractions makes the specific choice of colors clearest. In this palette, I use a warmer green and a dark shade of yellow: I've found that both colors look better than pure green and yellow at full intensity.

> **TIP** The red/yellow/green color combination conveys not only a high/low but also a semantic good/bad distinction. You need to decide whether this is appropriate for the visualization task at hand. Keep in mind that a pure yellow may be hard to see against a white background; instead, it's better to use a darker shade of yellow or even a light orange. Also keep in mind that this palette will pose difficulties for people with red/green color vision deficiencies.

The last simple workhorse is the "improved rainbow" (panel E in figure D.4):

```
set palette model RGB defined ( 0 'blue', 1 'cyan', 2 'green', 3 'yellow',
➡                                4 'red', 5 'magenta' )
```

It fixes the two biggest problems of the standard rainbow: because the colors run from blue (cold) to red and magenta (hot), it conveys an intuitive sense of ordering. Moreover, because it doesn't wrap around, the color mapping is unambiguous. The improved rainbow isn't a very good palette, but it's much better than the standard rainbow, and—because it only depends on primary colors as nodes—it's still easy to remember and simple to set up.

FOUR INDUSTRIAL-STRENGTH PALETTES

The following palette, which is a further refinement on the improved rainbow palette, is similar to the default color palette in MATLAB (panel F in figure D.4):

```
set palette defined ( 0 'dark-blue', 1/8. 'blue', 3/8. 'cyan',
➡                     5/8. 'yellow', 7/8. 'red', 1 'dark-red' )
```

This is a good, all-purpose palette with a strong intuitive sense of ordering and several distinct visual gradients toward the center of the plot range. Furthermore, the central part of the palette consists of softer, pastel colors, which results in graphs that are easier on the eyes.

In a similar spirit, here's an improved version of the red/yellow/green color combination. Like the most recent palette, this also uses pastel colors in the center of the plot area (panel G in figure D.4):

```
set palette defined ( 0   0    0.55 0.2,  0.3 0.65 0.85 0.4, 0.5 1 1 0.75,
➡                     0.7 0.95 0.68 0.38, 1   0.75 0    0.15 )
```

The next two palettes are very good but also very subtle: each is constructed from a large number of nodes with precisely chosen colors for each node. Instead of giving you an approximation, which would necessarily lose some of the subtlety of either palette, I recommend that you download the original files.

The first of these palettes[3] is based on work by the oceanographer William F. Haxby (panel H in figure D.4). It's similar in spirit to the MATLAB palette you just saw, but it works with even milder colors. Highly recommended!

The other palette[4] was developed by computer scientist Kenneth Moreland and is an improvement on the simple blue/white/red palette (panel J in figure D.4). It uses non-saturated colors and special, nonlinear color ramps and thereby achieves an uncommonly uniform variation of intensity across the entire spectrum. Highly recommended!

It shouldn't be lost on you that all the palettes in this section are *improvements* on simple palettes that were introduced earlier. The difference doesn't come from a conceptual breakthrough, but from a relentless attention to detail.[5]

> **TIP** High-quality palettes can't be improvised but are the result of many incremental improvements in detail. Become familiar with and use high-quality palettes that are available.

PALETTES FOR SEGREGATION TASKS

All the palettes discussed so far consisted of smooth transitions and were suitable for continuously varying data. But sometimes that's not what you want: sometimes it's necessary to indicate sharp boundaries between categories. (Think of the difference between a political compared to a topographical map!)

Using the `set palette` command, you create a sharp color transition by listing one of the positions *twice*, giving a different color each time. The following traffic-light palette consists only of sharp transitions (panel K in figure D.4):

```
set palette defined ( 0 "green", 1 "green", 1 "yellow",
➡                      2 "yellow", 2 "red", 3 "red" )
```

Palettes can feature sharp transitions together with smooth gradients. Here's a variant of the blue/white/red palette that emphasizes the break at the center of the plot range. You might use this palette to indicate the location where a function or data set changes sign—while still indicating total magnitude through the smooth color change on either side. Notice that the palette doesn't fade to white (or gray) away from the center, but always retains some tint (panel L in figure D.4):

```
set palette defined ( -1 '#aaaaff', 0 'blue', 0 'red', 1 '#ffaaaa' )
```

---

[3] "Hx-110-110," *cpt-city*, http://mng.bz/T48T.

[4] "Cool-warm," *cpt-city*, http://mng.bz/m8bp.

[5] You can find another interesting palette at www.gnuplotting.org/matlab-colorbar-parula-with-gnuplot. This palette combines a continuous change in hue with an almost linear increase in lightness over the entire color range.

Here's a red/green palette with a sharp transition. In contrast to the blue/red palette you just saw, this one has the highest color intensity not in the center, but toward the edges. The fully saturated colors were chosen to be relatively mild and to have comparable lightness (panel M in figure D.4):

```
set palette defined (0    0 0.75 0,    0.2 0.2 0.8 0.2,   0.5 0.75 0.95 0.75,
➥                         0.5 1 0.66 0.66, 0.8 0.9 0.25 0.25, 1   0.9  0.1  0.1 )
```

Here's a more complicated take on the same idea (panel N in figure D.4). This palette also exhibits red and blue color gradients away from the middle, but in addition it sports a rapid sequence of color changes right at the center point, thus very effectively highlighting the transition:[6]

```
set palette defined ( 0 "dark-blue", 0.45 "blue", 0.495 "web-blue",
➥                         0.4995 "cyan", 0.5 "white", 0.5005 "yellow",
➥                         0.505 "sienna1", 0.55 "red", 1 "dark-red" )
```

GEO PALETTES

A *geo scale* is yet another type of palette that offers a sense of ordering, using the kind of color sequence familiar from topographic maps (although in this case the sense of ordering is more conventional than inherent). Such palettes seem to work best when all the colors are particularly pale. Here's an example (panel O in figure D.4):

```
set palette defined ( 0   0.7 0.85 0.6, 0.3 1 1 0.75,
➥                         0.7 0.75 0.6 0.4, 1   1 1 1 )
```

Geo palettes may include a sharp transition to indicate the boundary between sea and land. Here's a a rather complex example[7] (panel P in figure D.4):

```
set pal defined (0    0.45 0.66 0.85,0.25 0.725 0.9 1,  0.33 0.85 0.95 1,
➥                     0.33 0.68 0.82 0.65,0.4 0.58 0.75 0.53,0.575 0.95 0.9 0.75,
➥                     0.8  0.66 0.52 0.32,0.85 0.66 0.6 0.5, 1 0.95 0.95 0.95 )
```

TREE RINGS

Yet another idea is to create a palette from well-separated color bands (possibly on a smoothly varying background). Here's an example[8] (panel Q in figure D.4):

```
hue(x) = word( "0.75 0.66 0.5 0.33 0.16 0.1 0.0 0.83", 1+int(8*x) )
sat(x) = cos(2*pi*(x*7.4-0.2))

set palette model HSV functions hue(gray), sat(gray)>0?sat(gray):0, 1
```

If you want to superimpose the colored bands on a smoothly changing grayscale, you can replace the `1` in the last entry (for the value component) with an expression like `sat(gray)>-0.5?1:gray`. Personally, I find that the background clutters up the result too much.

---

[6] For a similar palette, see "Curvature," *cpt-city*, http://mng.bz/QJ7x.

[7] This palette was inspired by the standard palette used by maps on Wikipedia. See "Wikipedia elevation schemes," *cpt-city*, http://mng.bz/6lBR.

[8] This is an improvement on the version given in the first edition. I've since learned that you need to use relatively wide color bands that don't have sharp edges, but that fade away gently.

Thoughts on palette design

It isn't at all obvious what makes a good palette. Here are some considerations and recommendations for you to consider:

- *Be clear about the purpose of the visualization.* Are you looking for a realistic representation of the variations in the data, or are you more interested in segmenting (categorizing) it into different strata? Are you particularly interested in the features within a certain limited range of values? Do you want to identify and highlight data points representing specific values (such as outliers)?

  All of these tasks are valid but lend themselves to different palettes. To represent smooth variation, a palette with a continuous spectrum of colors works best; but for segmentation tasks, a palette consisting of several discrete color bands makes more sense. These attributes can be combined, for instance in a palette varying smoothly in the center but with solid colors once certain thresholds are crossed.

- *Understand the nature of the data.* Does the data form a continuous spectrum, or does it consist of a set of discrete values? In either case, is there a natural ordering in the values?

  If the data is discrete, you want to make sure the color bands in your palette correspond to the values in your data set. If the data comes from a continuous spectrum, you may want either a continuous palette or a discrete one, depending on the purpose of the visualization. Finally, if the data is ordered, you want a palette that has some form of visual ordering (such as a black-to-white grayscale).

- *If you want to express order, make sure your palette has an intuitive sense of ordering.* Changes in lightness are unambiguously perceived as ordered; that's why a grayscale naturally conveys a sense of ordering. In contrast, hues aren't necessarily sorted. Some exceptions (that do convey a sense of ordering) are based on similarity with temperature. Examples include the *heat scale* (black, red, yellow, white—coinciding with a change in lightness) and the combinations of blue and red. Other color combinations are *conventionally* perceived as ordered, such as palettes based on the shades of the traffic-light colors green, yellow, and red, or the geo scales familiar from topographic maps. By contrast, the standard rainbow is *not* ordered: this is one of the primary reasons it's such a poor choice for palettes.

- *Make sure there are strong visual gradients in those regions of the data where significant and relevant change occurs.* There tend to be regions in the data where change is more relevant to the task at hand than in others. What this region is depends on the data (and the task): are you interested in the detailed behavior in the middle of the data set, while caring less about the exact behavior near the edges of the spectrum? Or do you mostly want to identify outliers, so you're less interested in the variation in the middle ground? Either way, you want to make sure the strongest *visual* change in the palette coincides with those regions you care about the most. The consequence is that palettes that distribute change uniformly across the full range of values probably won't give the most insight into the most relevant parts of

your data. (This implies that palettes generated by a simple mathematical formula are often suboptimal.)

Specialized tasks call for more creativity in the creation of palettes. For example, the data may pass through zero (or some other relevant threshold) in the middle of the spectrum. You want to clearly indicate this threshold but also show the smooth variation on both sides of it. In such a situation, you may want to have a sharp transition (such as a hard red/blue changeover) in the spectrum at the location of the threshold, while letting the colors change smoothly as you go away from the threshold on either side.

To summarize: don't distribute strong visual gradients uniformly over the entire plot range if some regions of the plot range contain much less detail than others. Good palettes are often not uniform, and every palette must be tuned for both the current data set and the intended use.

- *Make sure the color-mapping is unambiguous.* One problem with the standard rainbow is that it wraps around: the very highest and the very lowest numeric values are mapped to similar colors. In general, you want to avoid such ambiguous mapping, although there are situations where it may be less of a problem because the data is sufficiently smooth to prevent ambiguity when interpreting a plot.

- *Prefer pastel colors and subtle changes in palettes.* This is possibly the most surprising of all recommendations. Intuitively, you'd expect that bright, fully saturated colors would give the best visual contrast. But this isn't the case. Instead, pastel colors, which are easier on the eyes, make it easier to detect details and structure in the data. High-quality palettes often employ a range of carefully chosen, subtle colors.

- *Use a palette suitable for the intended medium.* Graphs for a print publication may require a different palette than graphs for an overhead presentation. For best results, you may need to choose or adapt a palette for each target medium.

  A particular concern is black-and-white publishing. Some palettes attempt to change hue *and* lightness simultaneously, in an attempt to be suitable for both color displays and monochrome printing. (Gnuplot's standard palette is of this kind.) I usually don't find the results very convincing. If in doubt, I'd opt for redrawing a graph using a custom-made grayscale palette for black-and-white displays.

- *Develop a library of high-quality palettes.* Designing good palettes is difficult, involving much trial-and-error. I recommend that you build a library of well-designed palettes, drawing not only on your own experience, but also on palettes developed by others. Adapting a good palette to a specific task is likely to lead to better results than improvising one from scratch.

- *Never publish a false-color plot without showing the mapping of numbers to colors explicitly.* No matter how intuitive you think your chosen color scheme is, keep in mind that there's *no* universally accepted, generally understood mapping from numbers to colors. Without a colorbox, there's a good chance that your graph can't be understood by someone else.

(continued)

What's more, even if the ordering relation in your color scheme is clear (for example, in a simple grayscale), the mapping of numbers to specific colors is impossible to infer without an explicit key. Always provide one!

- *Be aware of color-blindness.* A significant percentage of the population exhibits some form of color-vision deficiency, with men being much more likely to be affected than women. Graphs that rely solely on color to convey information pose special challenges for such individuals.

  The phenomenon of color-vision deficiency is complex, with many different aspects to consider, and it exhibits a great deal of individual variation. It's therefore difficult, if not impossible, to give general advice, but here are two observations to keep in mind. First, be aware that the most common form of color-blindness is a reduced ability (or inability) to distinguish between red and green. Palettes that rely primarily on red and green are therefore particularly challenging. Furthermore, you may want to include secondary clues (such as changes in lightness) in a palette and not rely on color changes exclusively.

Finally, keep in mind that it all depends on the specific task at hand and on the purpose of your visualization. Consider a map: it may either be colored continuously (topographic map) or use discrete colors (political map). In the discrete case, the colors may be unsorted and chosen for contrast (to indicate different countries) or sorted (to indicate a sorted quantity, such as the average income per head for each country). Continuous quantities are always sorted, and therefore smooth palettes should always be constructed to convey a clear sense of ordering.

D.3   *The colorbox*

The colorbox shows the mapping of numeric values to colors. It fulfills a role similar to that of the key (see section 7.4), but for colored plots. You encountered examples in figures D.1 and D.2; further examples can be found in figures D.5 and D.6, in section D.4. In all cases, the colorbox appears to the right of the main figure.

The colorbox is only visible when you use palette-mapped colors. Its appearance is controlled through the `set colorbox` options:

```
set colorbox [ vertical | horizontal ]
             [ noborder | bdefault | border {idx:linestyle} ]
             [ default | user [ origin {pos:orig} ] [ size {pos:size} ] ]
             [ front | back ]
```

The colorbox can be oriented either horizontally or vertically (this is the default). It's usually surrounded by a border, which you can turn off using `noborder`. Alternatively, a predefined line style can be selected as an integer argument to the `border` keyword. You can't use explicit line options with `set colorbox`.

The standard size and position for the colorbox can be chosen using the `default` keyword. Alternatively, the keyword `user` selects customized sizing and placing of the colorbox, indicated through the appropriate optional arguments. For

three-dimensional plots, the only permitted coordinate system is the `screen` system (see section 7.2); but for two-dimensional plots (including `set view map` plots), all coordinate systems can be used.

The colorbox can be suppressed using `unset colorbox`. Keep in mind that hiding the colorbox doesn't allow the viewer to extract any quantitative information from your plot, because there's no obvious mapping from colors to numbers and vice versa. Don't do it unless you're sure the graph is meaningful even without the colorbox.

You can think of the colorbox as just a little plot within a larger one, and so it responds to all commands and options that manipulate plot ranges, axes, and tic marks. For the colorbox, there's only a single axis, running from low values (and the corresponding colors) to high values (and colors). This axis as known as the `cb` (for *colorbox*) axis. Table D.1 lists all relevant options. None of these options change the way colors are distributed in the palette; they merely change the way numeric values are assigned to those colors.

Table D.1 Colorbox options. They control the placement and formatting of tic marks along the colorbox and are therefore similar to the corresponding commands for any of the other axes. (The details given in the indicated sections apply to the colorbox as well.)

Option name	Description	Section
`set cbrange`	Sets the range of numeric values covered by the colorbox, *independently* of the range chosen for the range of z values.	8.2
`set logscale cb`	Distributes numeric values logarithmically across colors.	3.4
`set cblabel`	Assigns a text label to the colorbox. This label is placed next to the colorbox. If the label is invisible, use , because the default placement may put the label outside the canvas area.	7.3.3
`cbtics`	Controls all aspects of tics for the colorbox (major tic marks).	8.3
`mcbtics`	Controls minor tic marks for the colorbox.	8.3.4
`grid cbtics`	Draws grid lines within the colorbox at the major colorbox tic positions.	8.3.6
`grid mcbtics`	Draws a grid within the colorbox at the minor colorbox tic positions.	8.3.6
`cbdata time`	Chooses time-series mode for the values in the colorbox.	8.4
`cbdtics`	Uses weekdays as tic labels for numeric values.	8.4.1
`cbmtics`	Uses months as tic labels for numeric values.	8.4.1

D.3.1   *Mapping the plot range to the palette*

As I pointed out in section D.2.2, the positions assigned to the various nodes making up a palette are merely *relative* positions within the palette. In particular, they don't necessarily correspond to the values that are being plotted. Hence there's a separate step, wherein the range of the palette is mapped to the range of data values.

The relevant command in this context is `set cbrange`. Its syntax is equivalent to the familiar `set xrange` or `set yrange` command (see section 8.2). The `set cbrange` command assigns a specific numeric value to the minimum and the maximum position in the palette; intermediate values are distributed uniformly into the interval. To make this concrete, let's say that you've defined a palette using

```
set palette defined ( 0 "blue", 0.5 "white", 1 "red" )
```

If the data that you want to plot now spans the range from -5 to +5, then you want to set `cbrange` to this interval:

```
set cbrange [-5:5]
```

All of this is extremely straightforward. Confusion can arise through three items in particular:

- It's easy to forget that the positions specified when defining the palette are only relative positions and are in no fixed relationship to the plotted data.
- By default, `cbrange` is set to autoscaling (see section 8.2.1): that is, `set cbrange [*:*]`. In practice, this means gnuplot sets the range of the palette equal to the range of plotted values. Most of the time this is exactly what you want, but it obscures the fact that choosing the `cbrange` is a separate step.
- Values outside the chosen `cbrange` are plotted in the nearest color. That means a data point that exceeds the maximum (or minimum) of an explicitly set `cbrange` is plotted in the color corresponding to the largest (or smallest) position in the palette. In particular, there's no visual indication that a data point is outside the interval defined by `cbrange`.

DEALING WITH OUTLIERS

A common problem when using palettes is the occurrence of outliers in the data. (You'll see a worked example in section D.5.1.) Imagine a data set where all data points take on values between 0 and 1 except for a handful of records that have a value of 10. Gnuplot's default setting of `set cbrange [*:*]` means the palette will be autoscaled to the interval `[0:10]`. Most of the data points (and all the data points that you care about) are therefore mapped to the bottom *tenth* of the palette, with the consequence that they're all drawn in more or less the same color. That's not the intended effect.

The remedy is simple: fix `cbrange` to the relevant range of values using `set cbrange [0:1]`. Now, the "good" data points are mapped to the full range of the palette, and the few outliers are mapped to the color at the top end of the palette.

It may be desirable to indicate data points that exceed the specified `cbrange` using a special color. This can be done—see section D.5.1 for an explanation.

TIP Gnuplot's default of autoscaling `cbrange` means a single outlier can distort the mapping of colors to values. Choose an explicit `cbrange` to make sure the interesting data points span the entire range of the palette. Gnuplot handles outliers that exceed the `cbrange` reasonably.

LOGARITHMIC SCALING OF THE COLORBOX

Finally, you can use logarithmic scaling with the colorbox. Setting `set logscale cb` distributes tic marks (that is, numeric values) logarithmically across the color spectrum. This command doesn't affect the way the palette is constructed from the nodes; it *only* affects the way the colors are mapped to data values.

D.4 *Using palettes*

Palettes can be used with different plot styles. First, wherever gnuplot expects an explicit color specification (such as `rgb "red"` or `rgb "#ff0000"`), you can instead provide a color from a palette using the keyword `frac` followed by the relative position in the palette. See the following listing and figure D.5.

Listing D.1 Commands for figure D.5 (file: airy.gp)

```
set palette defined ( 0 'web-green', 1 'goldenrod', 2 'red' )

set grid lt 1 lw 1.5 lc palette frac 0.5
set label 1 "{/:Bold Origin}" at 1.5,-0.15 center tc palette frac 0.9
set arrow 1 from 1.1,-0.15 to 0.125,-0.02 lw 2 lc palette frac 0.9

plot [-5:3] airy(x) lw 2 lc palette frac 0.1
```
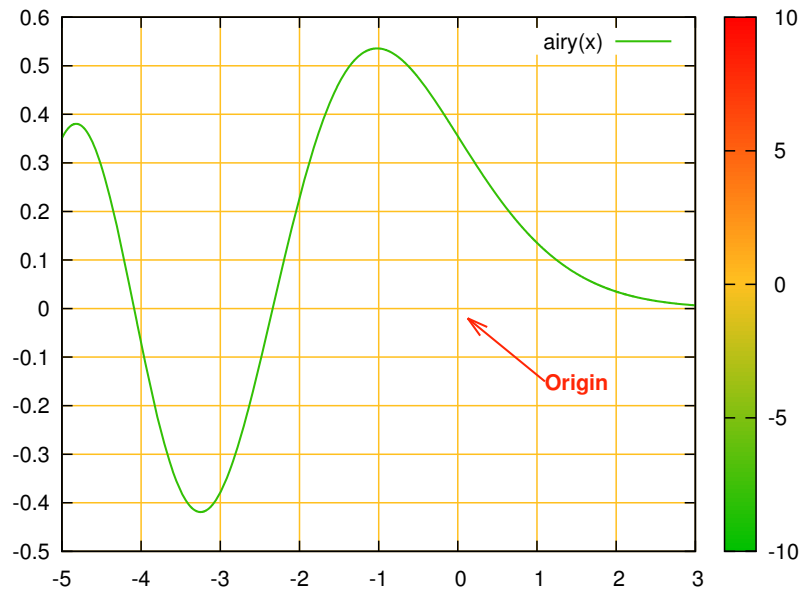

Figure D.5   Selecting colors from a palette (see listing D.1). In this figure, the color position in the palette is fixed, and the colors are therefore *not* data-dependent.

More interestingly, palettes can be used for data-dependent coloring: the color of each point or line segment is found by mapping the values of an additional data column into the palette, using the keyword `palette` in the `plot` command. Figure 9.5 showed how to use this feature to create a curve that changes its color continuously. The next listing shows another application (compare figure D.6).

Listing D.2   Commands for figure D.6 (file: cloud.gp)

```
set palette defined ( 0 'blue', 1 'grey', 2 'red' )
plot "cloud" u 1:2:3 palette pt 7
```

The first two columns determine the location of each point, and the third column is used to find the point's color by mapping the numeric value into the provided palette.
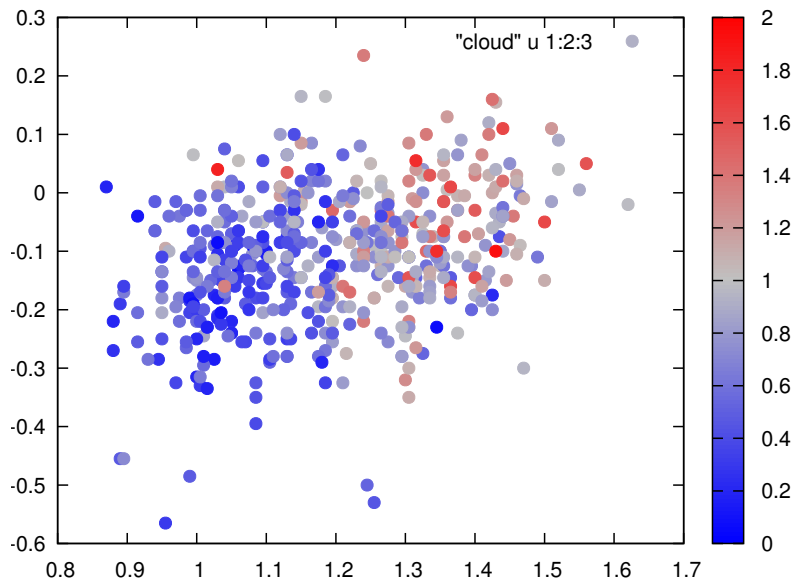
Figure D.6   Selecting colors from a palette (see listing D.2). In this figure, the values of the data points are used to select their colors from the palette; the plot uses *data-dependent* coloring.

Because data-dependent coloring depends on the ability to add columns to the `using` directive, it's only available when plotting *data*. If you want to use it when plotting functions, you must employ the `"+"` pseudofile.

Furthermore, palettes can be used together with `splot` to create colored surface plots—that will be the topic of the following section. But the most exciting application of palettes is to use them in false-color plots that rely entirely on color to convey values in a data set. Because of their importance, they're treated in their own section later in this appendix (section D.5).

Colored surface plots with pm3d

Palettes were originally developed to color surface plots generated by `splot` (see appendix C). You saw an example at the beginning of this appendix (see figure D.1). The `splot` command uses palettes when the `with pm3d` style is given explicitly as part of the `splot` command, or when `pm3d` mode is enabled globally using `set pm3d`.

APPEARANCE OPTIONS FOR COLORED SURFACE PLOTS

The `set pm3d` command controls various aspects of the way color is used:

```
set pm3d [ at [b|s|t] | map ]
        [ implicit | explicit ]
        [ border {lineoptions} | hidden3d {idx:linestyle} | nohidden3d ]
        [ interpolate {int:xsteps},{int:ysteps} ]
        [ corners2color
        [ mean|geomean|harmean|rms|median|min|max|c1|c2|c3|c4 ] ]
        [ scansautomatic | scansforward | scansbackward | depthorder ]
        [ flush [ begin | center | end ] ] [ [no]ftriangles ]
        [ clip1in | clip4in ]
```

In `pm3d` mode, gnuplot constructs a surface from colored, nontransparent polygons. Because the polygons are opaque, no explicit hidden-line removal is required—instead, surface areas closer to the observer hide surface areas further away. The resulting effect therefore depends on the order or direction in which the surface is drawn. Although gnuplot usually chooses a reasonable strategy for drawing surfaces, it helps to keep this point in mind when working in `pm3d` mode. (The gnuplot standard reference documentation contains more information on the algorithm.)

A colored, opaque surface can be drawn at three positions: at the top of the plotting box, on the plotted surface, or at the bottom. The position is specified using the keyword `at` together with a combination of the letters b (bottom), s (surface), and t (top). Each letter can appear twice (for example, `set pm3d at bsb`: this is one instance where the way surfaces are drawn in `pm3d` mode is potentially relevant). You can use the keyword `map` as a shortcut for `set view map; set pm3d at b`.

By default, the command `set pm3d` puts `pm3d` into *implicit* mode, meaning all surfaces drawn with `splot` are drawn using colored, nontransparent polygons. If you want to combine colored surfaces together with transparent, wire-mesh surfaces in a single graph, you need to choose *explicit* mode using `set pm3d explicit`. In explicit mode, you must specify `pm3d` as part of the `splot` command:

```
splot f(x,y) w l, g(x,y) w pm3d
```

This plots the function `f(x,y)` with a transparent wire mesh, but the function `g(x,y)` with a colored, opaque surface.

A colored surface can be drawn together with a wire mesh of the same surface using `set pm3d hidden3d`. This command takes as an additional, mandatory argument the index of a (previously defined) line style, which is used for the wire mesh. When using this plot mode, don't forget to switch off the regular surface and hidden-line

removal using `unset surface; unset hidden3d`. An alternative to `hidden3d` is the `border` keyword, which can be followed by any of the familiar line options (see the tip at the end of section 6.2.3) that should be used for drawing the wire mesh.

As an alternative to `set dgrid3d` (see section C.5), `pm3d` has a similar interpolating capability, triggered by the keyword `interpolate`. It takes two mandatory arguments, giving the number of interpolation steps in both x and y directions.

Two sub-options to `set pm3d` control the way the surface is constructed. The `corners2color` keyword selects how the color of each polygon is determined from the z coordinates of its four corners: as mean, median, and so on, or by choosing the value from one of the corners directly. The `scansforward`, `scansbackward` directives control the direction in which the surface is constructed. The default is `scansautomatic` and usually doesn't need to be changed.

Further sub-options apply to certain edge cases when the input data doesn't fall on a regular grid. See the gnuplot standard reference documentation for details.

ELEVATION AND COLOR

So far, we've assumed that the *color* of each surface element was chosen based on the *elevation* of that surface element, in order to enhance the height perception of the surface. But it's also possible to map a *different* quantity than the height to the color spectrum. This is done with an additional parameter to the `using` directive to the `splot` command:

```
set pm3d at s
splot "data" u 1:2:3:4 w pm3d
```

Here, the first and second columns of the data file are taken as x and y coordinates, the value of the third column is used for the z coordinate (the height) of the surface, and the color is assigned according to the fourth column.
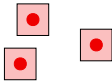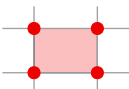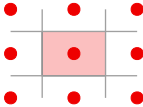
D.5   *False-color plots*

There are at least three different ways to use gnuplot palettes to create false-color plots:

- Using the `with points` style (and other standard styles)
- Using the `with pm3d` style
- Using the `with image` style

All of these approaches have different strengths and weaknesses, which makes them suitable to different tasks. Table D.2 summarizes the most important differences.

Table D.2 Relevant differences between the three plot styles that can be used to create false-color plots

with points	with pm3d	with image
Surface elements can be at arbitrary locations.	Surface elements must form a regular, but possibly distorted, grid.	Surface elements must form a strictly regular grid.
Each surface element is drawn independently from all others.	Each surface element is determined by the data points at its four corners.	Each surface element is centered on a data point and extends half-way to the four neighboring data points.
Surface elements can be at arbitrary locations.	Surface elements can be drawn on an elevated surface.	Surface elements must form a flat surface.
Style is only available for data.	Style works with functions and data.	Style is only available for data.
with points	with pm3d	with image

D.5.1 *Using points*

Imagine that you oversee a fleet of machines or servers, and you want to visualize the number of defects that have occurred for each device and for each day of operation. The situation is complicated by the fact that not all machines operate every day: depending on the daily load, machines are either active on a given day or not.

This problem has a natural grid-like structure: you care about the number of defects for each machine-day. But because not all machines are active on all days, the grid isn't complete and has missing cells. One interesting question is this: do defects cluster in any way? The data format is simple enough: for each day that a machine was active, the data set contains a record with the day number, the machine ID, and the number of observed defects (see listing D.3).

Listing D.3 Incomplete data for figure D.7 and listing D.4 (file: machines)

```
# Day Machine Defects
1    1      2
1    3      0
1    4      1
1    5      2
...
```

Figure D.7 uses gnuplot's `with points` style, together with a color palette, to visualize this data set. The selected point size is relatively large, because each point represents a cell in the data grid. Each record is first plotted as a filled (colored) square; then a

thin, black empty frame is drawn around it purely for aesthetic reasons. (Try it with and without the frame!) Two clusters of sometimes troublesome machines are clearly visible. (The commands are in the following listing.)

> **Listing D.4   Commands for figure D.7 (file: machines.gp)**

```
unset key
set xlab "Day"; set ylab "Machine"; set cblab "Defects" offset 1,0

set palette defined (0 'web-green', 0.5 'goldenrod', 0.999 'red', 1 'black')
set cbrange [0:7]

plot [0:32][0:21] "machines" u 1:2:3 w p pt 5 ps 1.75 pal,
➡                 "" u 1:2 w p pt 4 ps 1.75 lw 0.5 lc black
```

To get a good result with this data set, it's necessary to specify `cbrange` explicitly. Although the number of defects for most of the data points is somewhere between 0 and 7, there is a handful of records that show catastrophic failure, with defect rates up to 50! If you relied on gnuplot's autoscaling of the `cbrange`, then almost all the data points would be drawn in green. Fixing the palette using `set cbrange [0:7]` ensures that the interesting data points are distributed across the entire palette. The palette was designed for this scenario: its uppermost edge is black so that points exceeding the interval defined by `cbrange` stand out.

   One last comment before leaving this section: this approach to creating false-color plots isn't limited to the `with points` style, but works with most of the familiar styles. You
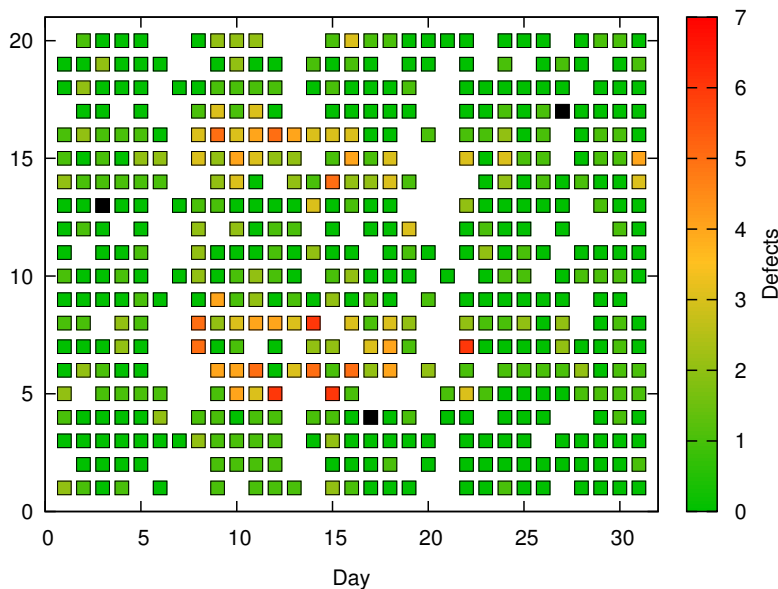
Figure D.7   Showing the number of defects per machine and day in a false-color plot. This plot uses the `with points` style. See listing D.4.

may want to experiment with the `with vectors` and `with circles` styles (section 6.3.5). The `with boxxyerrorbars` style may come in handy to create rectangles of arbitrary size. For example, the following `plot` command is for all practical purposes equivalent to the one from listing D.4:

```
plot [0:32][0:21]
➥    "machines" u 1:2:(.35):(0.35):3 w boxxyerrorbars lc palette fs solid,
➥    "" u 1:2:(.35):(0.35) w boxxyerrorbar lw 0.5 lc black
```

D.5.2 *Using the pm3d style*

Although `pm3d` mode has so far been discussed only in the context of colored surface plots, it can also be used for false-color plots by placing the viewpoint vertically above the surface. This is what the `set pm3d map` option does.

For data sets on a regular grid, the `with image` style is probably more convenient, but there are two instances where you still need `with pm3d` for false-color plots: when plotting functions and when the points fall on a grid that is locally distorted. Figure D.2 demonstrated two examples of plotting functions using `with pm3d`, so I don't need to repeat this here, but I'll show an example of a distorted grid. Listing D.5 is similar to the file in listing C.2 that was graphed in a *surface* plot in figure C.9, but with one difference: the positions of the points down the center line are all slightly shifted in an irregular fashion. The `splot` command can handle this:

```
set palette defined ( 0 'blue', 0.5 'grey', 1 'red' )
set pm3d map

splot "distorted-grid" u 1:2:3
```

Listing D.5   Data for figure D.8: distorted grid (file: distorted-grid)

```
# x       y       z
  0      -1      10
  0       0      10
  0       1      10

  1      -1      10
  1.2    0.2      5
  1       1      10

  2      -1      10
  1.9   -0.2      1
  2       1      10

  3      -1      10
  2.8     0       0
  3       1      10
```

As you look at figure D.8, I'd like to emphasize how the surface elements are spanned by their four corners, and that the color is a combination of the z values at those four corners. You may want to experiment with `set pm3d corners2color` to see what results you get.
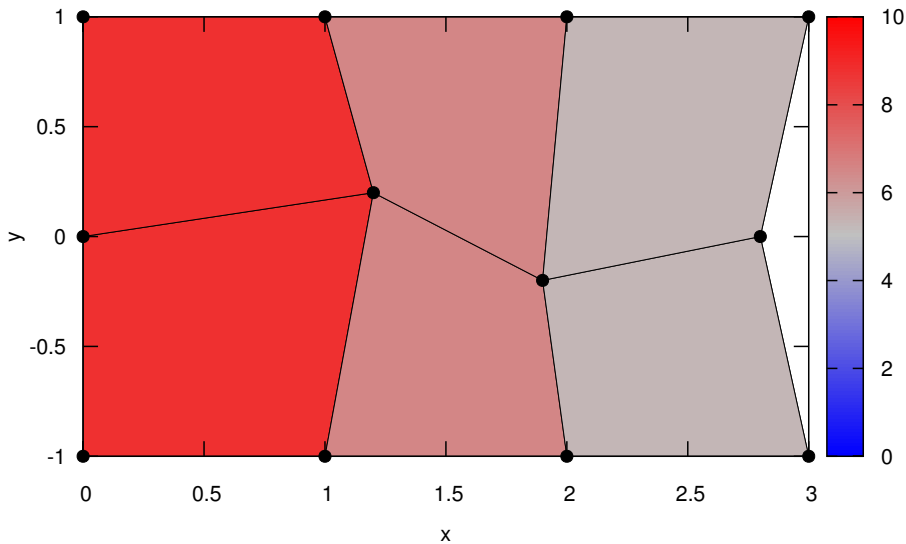
Figure D.8   The `with pm3d` style can create false-color plots even when the grid is distorted. Every tile is spanned by the data points at its four corners. See listing D.5 for the data.

D.5.3   *Using the image style*

The `with image` style is suitable for flat graphs on a regular, rectangular (or at least rhomboidal) grid. The style is very restrictive (the grid has to be strictly regular, with no missing values) but convenient if the necessary conditions are fulfilled. The style performs few internal consistency checks, and it's relatively easy to get gnuplot (and oneself) confused by trying to use `with image` for anything other than what it was intended for: flat graphs on strictly regular grids!

The `with image` style works with all three data formats familiar from `splot`: grid, matrix, and packed matrix (see section C.4). When using the packed matrix format (which doesn't include coordinate information), then all the formal requirements are automatically fulfilled—this is the sweet spot for the style. When using the general, non-uniform matrix format, be aware that gnuplot does *not* evaluate all the coordinates. It merely examines the coordinates of the bottom-left and top-right corners (the origin and its diagonal counterpart) and then chops the coordinate axes into cells of *equal size*. It isn't possible to have a non-uniform grid with the `with image` style, even when using the non-uniform matrix format!

Even more confusion may result when using the grid format, which includes explicit coordinate information for all cells, because gnuplot ignores most of them and again constructs a regular, equally spaced array of cells. The `with image` style will fail with an error message if any of the cells are missing.

> **TIP**   The `with image` style is strictly for data sets on a regular, rectangular grid of equally spaced cells. If your problem doesn't fit that description, don't use this style.

Figure D.9 shows a false-color plot of the original file from listing C.2 (that is, without distortions). Notice that the individual cells are *centered* on the location of the data points. Because of the large areas of solid color, I used a pastel version of the usual red/white/blue palette. Also, notice that the command is `plot`, *not* `splot`!

```
set palette def ( 0 '#8080ff', 0.5 '#ffffff', 1 '#ff8080' )
plot "grid" u 1:2:3 w image, "" u 1:2 w p pt 7 lc black
```
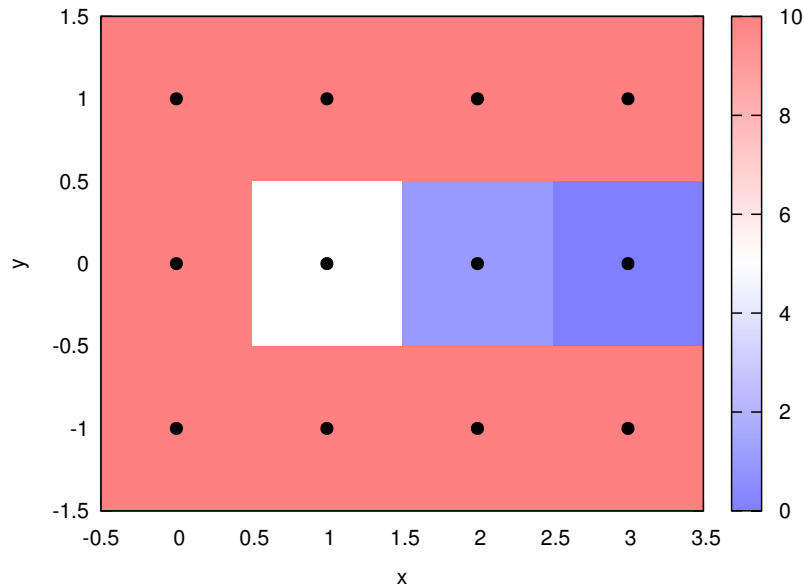

Figure D.9   **A false-color plot for a regular grid, created using the `with image` style. Notice that each tile is `centered` on a data point. The data is shown in listing C.2.**

MISSING VALUES AND UNSORTED DATA FILES

The `with image` style won't tolerate missing values: every cell in the grid must exist and contain a value, even if it's an invalid value (see section 4.3.4). If gnuplot detects an invalid value, `with image` doesn't draw the corresponding surface element; the background remains visible (that's usually what you want).

Furthermore, if your data file is in grid format, you may think it isn't necessary for the file to be sorted: after all, the grid format contains the locations of all the surface elements explicitly, and hence gnuplot should be able to place each surface element individually. But remember that the `with image` style *doesn't process the coordinates*: instead, it places surface elements in the order in which they're found in the data file. The data file therefore must be sorted, by horizontal coordinates first and by vertical coordinates second.

If your file doesn't fulfill these two requirements, then you need to write a short script in the programming language of your choice that reads in the raw data, populates

a suitable grid, and then prints the grid, in order, while supplying `NaN` values for all cells that don't contain data. Or, you can use the `with points` style to begin with.

OTHER WITH IMAGE FORMATS

In addition to the `with image` style described so far, gnuplot also provides the `with rgbimage` and `with rgbalpha` styles. Both are similar to the `with image` style, but they expect each data point to be described not by a single value, but by separate color components. The `with rgbimage` style expects three values per data point (for the red, green, and blue color components), and the `with rgbalpha` style expects four (with the addition of an alpha channel for transparency). All values are expected in the range from 0 to 255. See the standard gnuplot reference documentation for more details.

D.6   *Case study: coloring the Mandelbrot set*

Figure D.10 was created using the `with image` style from a file containing the x and y coordinates for each pixel and a third value that's mapped into a color palette. The figure shows a small section of the complex plane near the edge of the Mandelbrot set. The x and y coordinates correspond to the real and imaginary components of a complex number, and the color-coded quantity is the number of iteration steps before the Mandelbrot iteration diverged.[9] The data file was calculated separately.

This example is interesting not only because the appearance of the Mandelbrot set continues to be fascinating, but also because it poses particular challenges for the
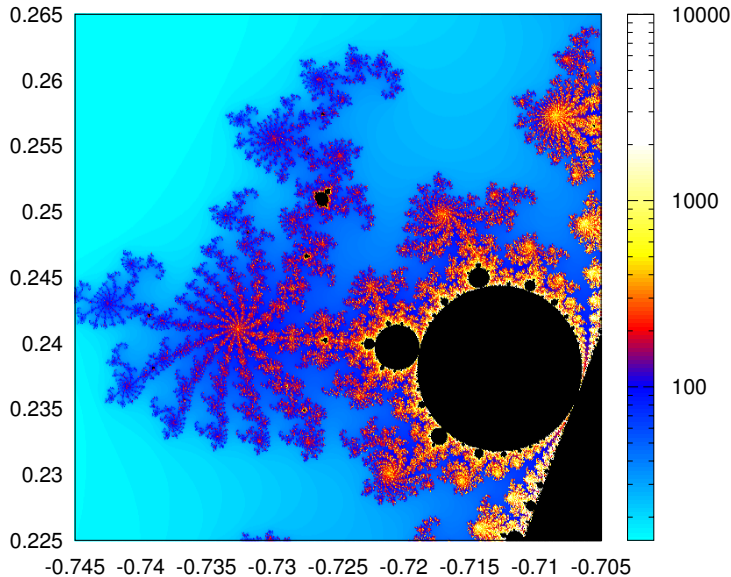
Figure D.10   A section of the complex plane, showing part of the Mandelbrot set and its fractal boundary. See listing D.6.

---

[9] See the Wikipedia page on the Mandelbrot set for details of the calculation.

design of a good palette. The range of values that needs to be represented through color spans three orders of magnitude (from 10 to 10,000). Moreover, the distribution of values is extremely uneven: all larger values are concentrated in a very narrow area near the border of the Mandelbrot set (the black area in figure D.10).

You can attempt to form a palette by distributing nodes non-uniformly, like so:

```
set palette defined ( 0 'cyan', 200 'blue', 300 'red', 600 'yellow',
➡                     2000 'white', 9999 'white', 10000 'black' )
```

The problem is that now the colorbox is almost useless: fully 80% of the area is white (from 2,000 to 10,000). It's better to use logarithmic scales. Remember: set logscale cb doesn't change the way the palette is constructed from the nodes, but only changes the way numerical values are mapped to colors (that is, how tic marks are distributed across the palette). In order to assign nodes to specific numerical values despite the logarithmic scale, the following listing uses the logarithms of the desired values in the definition of the palette and fixes cbrange explicitly.

Listing D.6   Commands for figure D.10 (file: fractal.gp)

```
set size ratio -1
unset key

set palette defined ( log(15) 'cyan', log(100) 'blue', log(200) 'red',
➡                     log(500) 'yellow', log(2000) 'white',
➡                     log(9995) 'white', log(10000) 'black')

set logscale cb
set cbrange [15:10000]

plot "fractal" u 1:2:3 w p pt 5 ps .05 lc pal z
```

One last question concerns the style to use with the plot command. This example may look like an ideal application of the with image style, because it involves a regular grid without missing cells. But the implementation of with image contains certain optimizations that lead to a loss of accuracy for a figure containing as much fine-grained detail as the Mandelbrot set. I therefore used the with points style, but with a very small point size.

D.7   *Case study: an interactive palette explorer*

For palettes that are defined through nodes, two things need to be fixed:

- The precise *color* of each node
- The absolute *position* of each node within the palette

In section D.2.5, I gave many examples of colors and color combinations. In this section, I'll demonstrate a technique you can use to experiment with the position of each node.[10]

---

[10] I'd like to thank Ethan Merritt for making me aware of this technique.

In section D.2.5, the nodes were distributed more or less uniformly across the spectrum. The need to place the nodes more carefully arises when there's more than one strong visual gradient in the palette and you want these gradients to coincide with regions of significant change in the data set. Depending on the placement of the nodes, a false-color plot of the same data set can appear quite different—even when the colors for all the nodes remain unchanged. Wouldn't it be lovely if you could move the nodes back and forth in the palette and have gnuplot simultaneously redraw the graph you're working on? The following listing shows a tool that does just that.

> **Listing D.7   Command file for an interactive palette explorer (file: palette-explorer.gp)**

```
# Invoke as: call 'palette-explorer.gp' 'blue' 'white' 'red'

c1 = ARG1
c2 = ARG2                          Stores the
c3 = ARG3                          supplied colors
                                                    Initializes the location
s = 0.5                                             of the central node
ds = 0.025                                  Selects the step size

p = sprintf( "set palette defined (0 '%s', s '%s', 1 '%s'); ", c1, c2, c3 )

bind 'F1' "s=(s-ds>0?s-ds:0); " . p . "replot;"    Creates
bind 'F2' "s=(s+ds<1?s+ds:1); " . p . "replot;"    hotkey bindings

eval(p)                         Evaluates the string when a hotkey is pressed
```

Constructs a string containing the palette definition → `p = sprintf(...)` line

This script is intentionally simple: it's intended for palettes of only three nodes. But the principles carry over directly to more complicated situations. You invoke the script using the `call` command and giving the three colors that you would like to use in your palette:

```
call "palette-explorer.gp" "blue" "white" "red"
```

The script creates a palette using the supplied colors and also defines a parameter `s` that controls the relative position of the central node within the palette. Furthermore, the script creates two *key bindings* to the two function keys F1 and F2 (the choice of keys is obviously arbitrary; see section 12.4.3 for more information on the `bind` command). If the plot window has keyboard focus, then with each key press, the parameter `s` and therefore the position of the middle color are moved by the amount `ds`, either up or down. Then the palette is re-created, and the plot is redrawn—using the new palette—via `replot`. The effect needs to be seen to be believed. Try it out with a graph like figure D.6 or figure D.7.

As I said, this script is intentionally simple and is meant to demonstrate the basic principle most clearly. Three generalizations suggest themselves:

- Extension to more than three colors
- Different choice for the size of the step width `ds`
- Displaying or exporting the final palette

Have fun!

D.8 Further reading

It's difficult to find information on color selection that is based on empirical evidence; many recommendations aren't much more than personal opinion. At the same time, it's interesting to observe that there has been real (albeit slow) progress over time. Here are some suggestions:

- A relatively recent and very careful paper has been prepared by Kenneth Moreland at Sandia National Lab. This is a good place to start, and it also reviews previous work. (He's also the inventor of the palette in panel J in figure D.4.) Check out his website: http://kennethmoreland.com/color-maps.
- Two early papers by Bernice Rogowitz and Lloyd Treinish at IBM suggest using changes in lightness for data exhibiting lots of fine detail and using changes in hue and saturation for data with smooth changes. The titles are "Why Should Engineers and Scientists Be Worried About Color?" and "How Not to Lie with Visualization," and you can find copies on the web.
- Another interesting paper is "Escaping RGBland: Selecting Colors for Statistical Graphics" by Achim Zeileis, Kurt Hornik, and Paul Murrell. These authors emphasize the importance of taking human visual perception into account when designing palettes and recommend palettes involving very pastel colors. This paper also can be found on the web.

Several websites make palettes available for download:

- J. J. Green maintains a very large selection of color gradients at his cpt-city website (http://soliton.vm.bytemark.co.uk/pub/cpt-city). Many of the palettes found here are intended for graphic design, not data visualization, and the sheer number of examples can be overwhelming at first. Palettes can be downloaded in a format that is suitable for gnuplot.
- Hagen Wierstorf (who also maintains the www.gnuplotting.org website) provides a selection of suitable palettes at https://github.com/Gnuplotting/gnuplot-palettes. Many of these palettes are based on the work of Cynthia Brewer (see the next entry).
- Cynthia Brewer has done extensive research into the development of color schemes, primarily for use in the preparation of maps. Be aware that not all of these findings may carry over to scientific visualization! There's an online tool that you can use to create your own color palettes, incorporating her ideas, at http://colorbrewer2.org.

appendix E
Special plots

In this appendix, we'll discuss several "special" plotting options. All of them produce graphs that are visually and semantically more complicated than the line-and-point plots we've been dealing with so far.

First, you'll learn about gnuplot's multiplot feature. This is a way to combine different plots into a single graph. The individual components may either be arranged on a regular array (that's the more common case) or be placed arbitrarily (for example, to create *insets* within a larger graph).

In the remainder of the appendix, we'll introduce three types of plots that summarize entire data sets in some way. First, we'll discuss *box-and-whisker* plots: a popular method to visualize the distribution of points in a simple and robust fashion. Then we'll turn to *parallel coordinates*. This is a relatively new type of plot that's intended to reveal structure in multidimensional data sets, but gnuplot makes them easy to create. The final section deals with gnuplot's support for drawing histograms: pretty and popular, they're most suitable for presentations and information graphics.

Except for the first topic (multiplot), the material in this appendix is rather specialized, and not everyone may have a need for it. Make sure you catch the multiplot features, but feel free to skip the rest if it's not relevant to you right now.

E.1   Multiplot

Gnuplot's multiplot feature allows you to combine several plots into a single graph. This can be useful for a number of purposes:

- To make arrays of related graphs
- To create *insets*: graphs within a graph (for example, to show some details at greater magnification)
- For special effects, such as multiple, clearly separate plots aligned on a common axis

E.1.1   Using multiplot mode

Multiplot mode is enabled like an option through a `set multiplot` command and remains active until an `unset multiplot` command has been issued. When multiplot mode is active, the usual `gnuplot>` prompt is replaced by a `multiplot>` prompt. All `plot` (or `splot` or `replot` or `test`, but not `test palette`) commands issued at this prompt are directed to the *same* graph, and you can control their relative location by

<center>E1</center>

giving additional directives to the `set multiplot` command. Any other option modified while in multiplot mode is applied to *all subsequent* plots. This is the normal gnuplot behavior; multiplot mode doesn't change that. Any option or decoration you want to apply to only one of the plots in a multiplot combination has to be explicitly set before and unset after the respective `plot` command. (This doesn't apply to decorations positioned with `screen` coordinates: because there's only a single `screen` coordinate system even for a multiplot graph, they're *global* objects.)

Different terminals may exhibit different behavior regarding the time when plots become visible. Some show each plot right when the corresponding `plot` has been issued; others may delay generation of the entire array of graphs until multiplot mode is switched off again.

There's one very important limitation to keep in mind: you can't change the terminal while in multiplot mode! This implies that you may have to change the way you use gnuplot: it's not possible to build up a graph iteratively using an interactive (screen) terminal and then, finally, export it to a graphics file format, because this would require a change of terminal before the last step. Instead, the file-based terminal has to be selected *first*, with all other commands following. Personally, I write all my commands to a command file (using a text editor) and then run it using `load`, while using an interactive terminal. When I'm satisfied with the resulting graph, I insert a terminal and output setting at the top of the command file, and run it one final time to export the graph to file. Because multiplot mode is mostly a tool for generating final presentation graphics (as opposed to doing exploratory data analysis, which must be interactive to be useful), this works quite well. (Also see section 12.2 on using external editors in general.)

There's one additional gotcha to be aware of when using multiplot mode: if you want to capture the commands for the plot using `save`, you must issue the `save` command *before* leaving multiplot mode. (Otherwise, only the commands for the last plot in the multiplot set will be saved to file!) And remember to exit multiplot mode using `unset multiplot`, not using Ctrl-D (which terminates gnuplot).

> **TIP** When using multiplot mode, it makes the most sense to prepare a command file with all plot and appearance commands and to load it into gnuplot using `load`. Using multiplot mode interactively is just too fickle and frustrating.

E.1.2   *Layout options and the set multiplot command*

The multiplot feature can be used in two different ways. You can specify the layout of a tabular array of graphs as part of the `set multiplot` command and let gnuplot figure out sizing and positioning of all components in the overall graph automatically. Or you can take control of all aspects of the layout yourself and arrange the individual graphs using `set size` and `set origin`. All of this will become much clearer once we look at some examples.

The set multiplot option has the following sub-options:

```
set multiplot [ title "{str:title}" ] [ font "str:font [,int:size]" ]
             [ layout {int:rows},{int:cols}
               [ rowsfirst | columnsfirst ] [ downwards | upwards ]
               [ scale {flt:xfactor}[,{flt:yfactor}] ]
               [ offset {flt:xoff}[,{flt:yoff}] ]
               [ margins {flt:left},{flt:right},{flt:bottom},{flt:top}
                 [ spacing {flt:horizontal} [, {flt:vertical} ] ] ]
             ]

set multiplot [ prev | next ]
```

The title directive is equivalent to set title (see section 7.3.3) for regular plots: it can be used to give an overall title to the entire assembly of graphs. (The set title command can still be used to assign a title to each of the individual component graphs.) All other directives (besides title) are only meaningful in combination with the layout keyword, which is a convenience feature to create arrays of graphs easily. In the following sections, I'll first guide you through some examples using the layout facility before explaining how to place graph components explicitly.

> **TIP** The layout feature of gnuplot's multiplot mode may seem daunting at first, but—once mastered—it takes the pain out of multipanel graphs. I highly recommend that you learn how to use it and prefer it over explicit placement of individual graphs whenever possible.

The layout directive takes two mandatory integer arguments, which describe the number of rows and columns in the resulting array of graphs. This array will be filled with subsequent plot commands. The order in which subsequent panels are filled can be controlled through the rowsfirst, columnsfirst, downwards, and upwards keywords (see figure E.1). The default is rowsfirst downwards.

Each new plot command moves to and fills the next array location (when using layout). You can skip one location by using set multiplot next *instead of* a plot
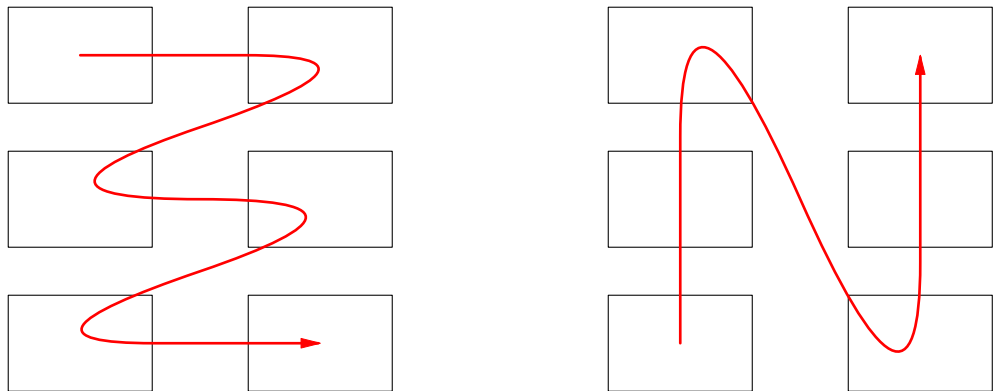
Figure E.1 Choosing the layout direction in multiplot mode: `rowsfirst downwards` (left; this is the default) and `columnsfirst upwards` (right)

command: gnuplot will leave the next location in the sequence blank. You can back up using `set multiplot prev`.

Individual plots can be scaled and shifted from their (automatically assigned) size and position using the `scale` and `offset` keywords. The arguments to `scale` are multiplicative scale factors for the x and y size of the individual plots (if only one is given, it's applied to both axes). Using the `offset` directive, you can shift all graphs by the same amount; the arguments are interpreted as coordinates in the `screen` system (see section 7.2), so that `offset 0.25, 0.1` shifts everything by a quarter of the overall width of the graph to the right and by a tenth of the height up.

The `margins` and `spacing` directives must be used together (and together with the `layout` option). They create a regular array of equally sized graphs that are properly aligned, even if only some of these graphs have marginal labels. (This will become much clearer when we look at an example in section E.1.4.) One important limitation: `set multiplot layout` doesn't nest—you can't have a small array of graphs as a panel in a larger array.

E.1.3   *Regular arrays of graphs with layout*

The `layout` feature makes it truly easy to generate multipanel graphs (as in figure E.2): just see how short and clear the following listing is! (It's important to use a complete `do for` loop here, and not just an inline loop: if I had used an inline loop, all six curves would have been drawn to a *single* plot, instead of six plots with a single curve in each.)

Listing E.1   Commands for figure E.2. (file: chebyshev.gp)

```
t(x,n) = cos(n*acos(x))                                    ⟵——— Defines the function to plot

set xrange [-1:1]                                          Fixes the range for all plots
set yrange [-1:1]
                                                                       Turns on
set multiplot title "Chebyshev Polynomials" layout 3,2  ⟵——  multiplot mode

do for[k=1:6] { plot t(x,k) t sprintf( "k=%d", k ) }    ⟵——  Creates six
                                                              individual plots
unset multiplot      ⟵—— Don't forget to turn
                         off multiplot mode!
```

If you want to have more control over the way individual plots are assembled to form a plot, you can't use `layout`; instead you have to handle everything yourself using `set size` and `set origin`. You'll see an example in a moment (in section E.1.5), but first, let's turn to another problem: tic marks and axes labels in a multiplot array.

E.1.4   *Accommodating marginal labels with margins and spacing*

Figure E.2 is pretty and was easy to do. Yet it isn't fully satisfactory: it feels cluttered. Can't we remove some of the decorations, so that we have more room for the graphs, but do so without losing any information? An obvious candidate are some of the tic labels: all the individual graphs show exactly the same plot range (both horizontally and vertically), and hence there's no need to repeat the tic labels on all the graphs. Just a single set of tic labels around the outer edges of the entire array would be sufficient. In fact, that's exactly what you see in figure E.3.
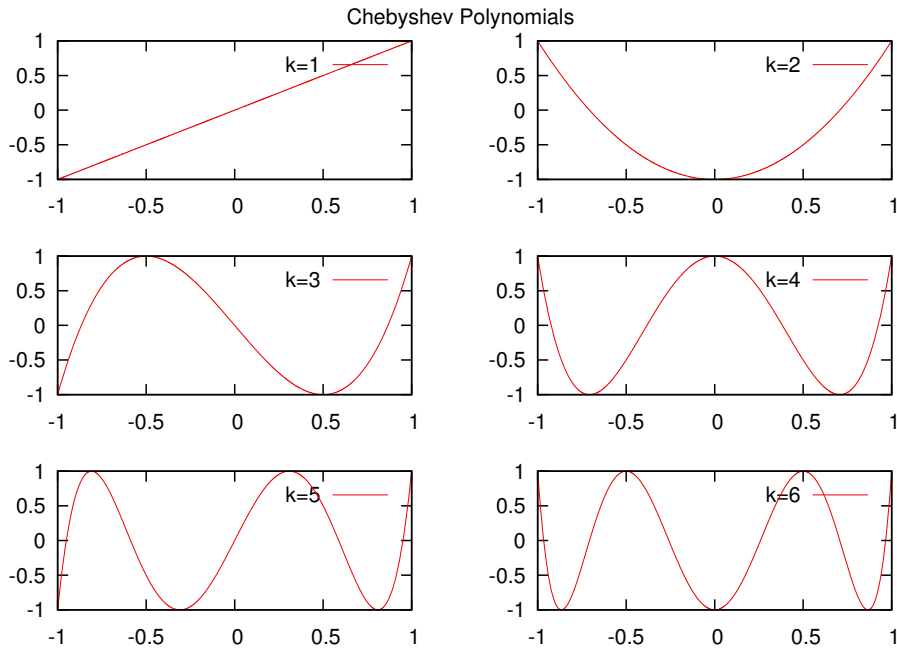
Figure E.2   A regular array of small plots created in multiplot mode. See listing E.1.

You now need to switch on tic and axis labels for some, but not all, of the graphs. As a result, the commands in the next listing aren't nearly as nice as the previous version (in listing E.1).

Listing E.2   Commands for figure E.3 (file: marginals.gp)

```
t(x,n) = cos(n*acos(x))

set xrange [-1:1]                                          Now includes margins
set yrange [-1:1]                                          and spacing options!

set multiplot layout 3,2 margins 0.15,0.95,0.15,0.95 spacing 0.03,0.05

set xtics format "";   set ytics format "%g"; plot t(x,1) t "k=1"    Top
set xtics format "";   set ytics format "";   plot t(x,2) t "k=2"    row

set ylabel "Chebyshev\nPolynomials"
set xtics format "";   set ytics format "%g"; plot t(x,3) t "k=3"    Middle
unset ylabel                                                         row
set xtics format "";   set ytics format "";   plot t(x,4) t "k=4"

set xlabel "k odd";
set xtics format "%g"; set ytics format "%g"; plot t(x,5) t "k=5"
set xlabel "k even";                                                 Bottom
set xtics format "%g"; set ytics format "";   plot t(x,6) t "k=6"    row
unset xlabel

unset multiplot
```

Defines the overall vertical axis label

Turns off the vertical label again

Turns off the horizontal label

Don't forget to turn off multiplot mode again!

Defines the horizontal axis label on the right

Defines the horizontal axis label on the left
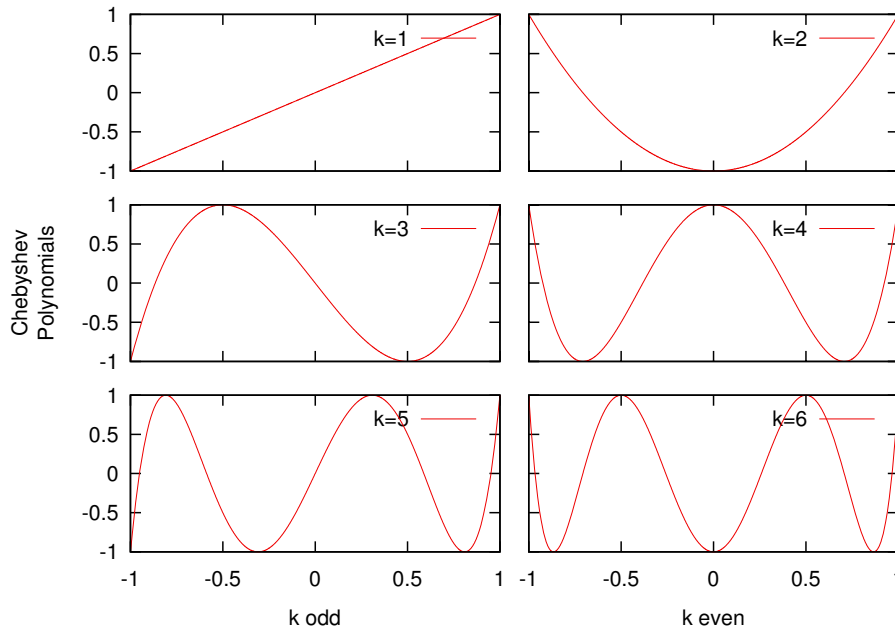
Figure E.3   A regular array of small plots. All individual plots are of the same size and therefore line up, although some of them have marginal notes whereas others don't. See listing E.2.

Before each `plot` command, the tic labels for both axes are turned either on or off. The title is dropped from the top of the plot (and hence from the `set multiplot` command), but axis labels are attached to some of the plots.

An *essential* change has occurred in the `set multiplot` command: it now includes the `margins` and `spacing` keywords. When these keywords (which *must* be used together) are included in the `set multiplot` command, then the individual graphs are drawn so that their *graph area* (that is, the area within their borders) is exactly the same size for all of them—regardless of whether an individual graph has marginal labels (such as tic or axis labels). In contrast, if the `margins` and `spacing` keywords aren't included, then gnuplot draws the individual plots such that their *screen area* (that is, their total extent, including their margins) is the same. (See section 7.2 for the definition of *graph* and *screen* coordinates, specifically figure 7.2.)

To put it differently: when `margins` and `spacing` are included in the `set multiplot` command, then gnuplot *ignores the margins of individual plots* and instead arranges plots with equally sized graph areas on a regular array. In this case, it's the user's responsibility to make sure there's enough room for any marginal notes attached to individual graphs. By contrast, when using `set multiplot layout` *without* the `margins` and `spacing` keywords, then gnuplot shrinks the graph area of individual plots to make room for that plot's margins. In this case, you can be sure the margins fit into

the multiplot array, but the individual plots may not all be the same size (and, in consequence, their borders may not line up).

The numerical arguments to `margins` and `spacing` control the extent of the margins around the outer edge of the array of plots and the internal spacing between individual plots, respectively. Two different systems of measurement are available: character widths and heights, and fractions of the total screen size. Character widths must be indicated through the `char` keyword, and fractions of the screen width through the `screen` keyword. If neither is found, `screen` is assumed.[1]

The `margins` sub-option behaves similarly to the `set margins` command (see section 9.5.3): when the `char` system is used, then the arguments are interpreted as the width of the margins around the array of subplots; if the `screen` system is used, then the arguments are interpreted as the absolute position of the outer boundaries of the array of subplots. In contrast to `set margins`, the keyword doesn't need to be repeated for each argument, but applies to all subsequent numbers (until the next appearance of either `screen` or `char`).

GRAPHS SHARING A COMMON AXIS

Here's another example that demonstrates the features just introduced. Occasionally, you may want to show two or more graphs together, aligned on a common axis, to facilitate comparison between curves in both graphs. Of course, it's possible to put all the curves into a single plot (as in section 8.1), but sometimes doing so would lead to an overly cluttered graph—for example, if you wanted to compare not just two curves, but two entire *sets* of curves.

To make this concrete, let's assume I want to compare the log-normal probability density function and its cumulative distribution function for an entire set of parameters. I could place all these curves into a single graph, but the graph would appear crowded, and it would be difficult to distinguish the curves properly from each other.

Instead, let's put them into two different graphs, aligned on a common axis, as shown in figure E.4. By now, you know how to create a multipanel plot using multiplot mode (see the following listing), so I only need to point out a few details.

Listing E.3   Commands for figure E.4 (file: sidebyside.gp)

❶ The log-normal probability density

❷ Its cumulative distribution function

```
lgn( x, s )  = exp(-0.5*(log(x)/s)**2)/(x*s*sqrt(2*pi))
clgn( x, s ) = norm(log(x)/s)
```

❸ Defines parameters as a whitespace-separated string

```
vals = "0.05 0.1 0.25 0.5"

set multiplot title "" layout 2,1 margins 0.1,0.9,0.1,0.95 spacing 0,0
```

---

[1] For gnuplot 5.0, only `screen` is available; `char` is available for gnuplot versions beginning with 5.1.

```
set key bottom right Left title "Cumulative"
set xtics 0.2 format ""
set ytics 0.2,0.2
set grid

plot [0:2][0:1] for[v in vals] clgn(x,v) t "s=" . v
```

Top plot

No tic labels for the x axis, but tic marks are drawn with the given frequency

Suppresses the tic label for y=0 (tics start at 0.2 ❹ with step width 0.2)

```
set key top right Left title "Density"
set format x "%g"
set ytics auto
unset grid

plot [0:2][0:8.5] for[v in vals] lgn(x,v) t "s=" . v

unset multiplot
```

Bottom plot

Restores the tic labels for the x-axis

First come the definitions of the two functions to plot ❶ ❷. You also define the set of parameters, as a whitespace-separated string (because gnuplot doesn't have an array type) ❸.

The spacing value in set multiplot is 0, so that the two graphs will touch each other. So that the axis labels for the vertical axis don't clobber each other where the two graphs meet, the first label (for y=0) in the top graph is suppressed ❹ (see section 8.3.3 for details of the set ytics option).
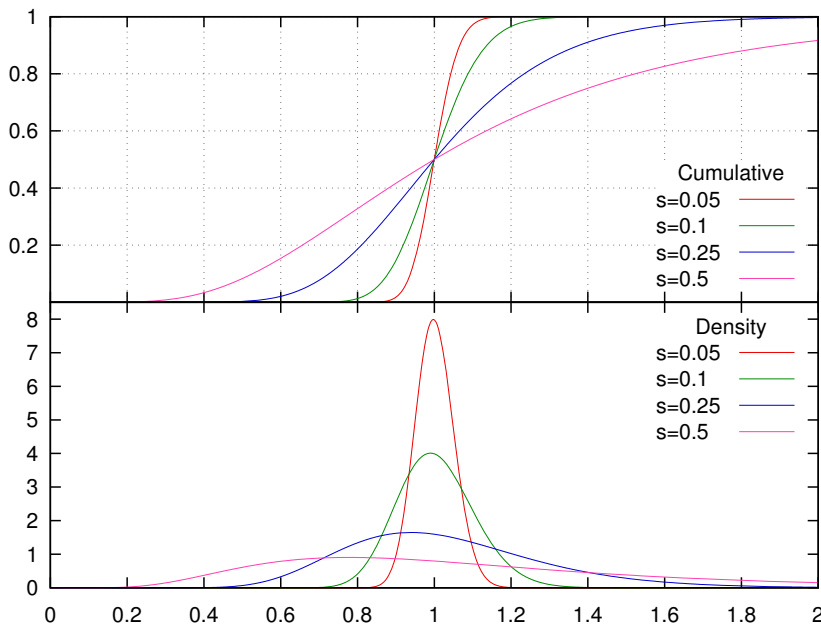
Figure E.4   Showing two plots side by side using multiplot mode (see listing E.3)

E.1.5   *Graphs within a graph*

Sometimes it's useful to place small graphs inside a larger one: for example, to show a detail of the overall graph at greater magnification, or to provide some form of ancillary information (as in figure E.5). This can't be accomplished using `layout`, so you have to roll your own.
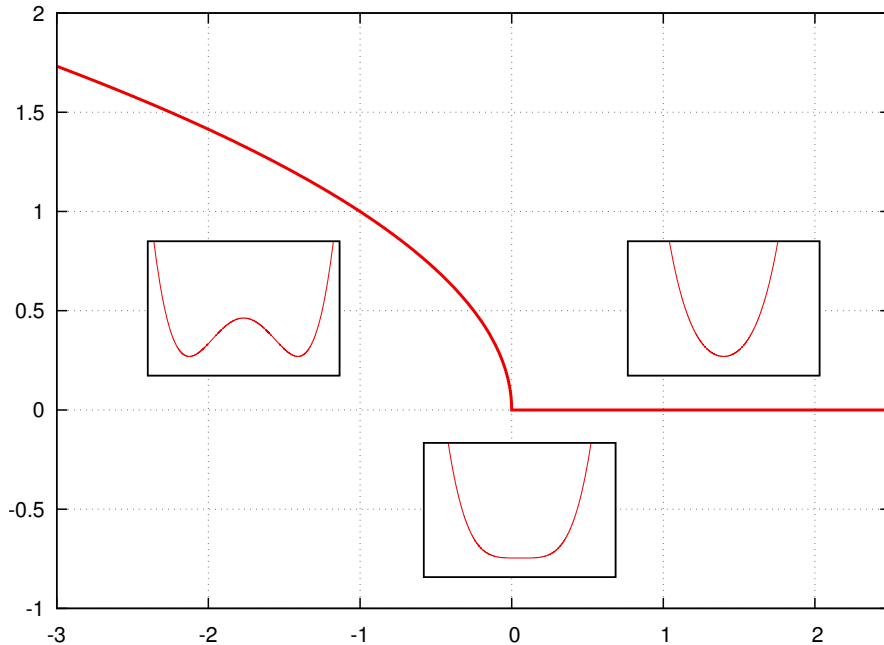
Figure E.5   A larger graph with insets showing ancillary information. See listing E.4.

Listing E.4 shows how it's done.[2] Let's step through this example carefully and point out some details and potential pitfalls.

In this example, the main curve is drawn first, and then the small insets are added to it. Options (such as their size) that apply to all insets are defined first. Because the insets are small, all decorations and margins are removed, to reduce clutter.

The insets are placed on the existing graph individually by specifying the location of their bottom-left corner using the `set origin` command. By default, gnuplot doesn't clear the area where the insets will be placed, and therefore it's necessary to

---

[2] This example is taken from the thermodynamics of phase transitions: if you heat a magnet beyond its *critical temperature*, it loses all magnetization. In the graph, the thick curve shows the magnetization as a function of temperature, and the three insets show the typical form of the *free energy* as a function of the magnetization for three different temperatures: below, at, and above the critical value. Below the critical temperature, the free energy develops two minima at nonzero values of the magnetization, indicating that a magnetized phase is stable. Right at the critical temperature, these two minima coalesce, yielding a curve that's flat at the origin. Above the transition temperature, there's only a single minimum at zero: the magnetization of the sample is now zero.

do so explicitly using the clear command. The subsequent plot command then draws one of the insets on the cleared area.

Listing E.4   Commands for figure E.5 (file: insets.gp)

```
f(x,a) = x<a ? sqrt(a-x) : 0               Defines the functions to plot
g(x,b) = 0.25*x**4 + b*x**2

set multiplot            ◁─────  Switches on multiplot—this time without the layout option

set grid                                  ◁─────  Turns on the background grid
plot [-3:2.5][-1:2] f(x,0) notitle w l lw 3   ◁─────  Plots the main curve as a thick line
unset grid                      ◁─────  Switches off the background grid for subsequent plots

unset tics                      Removes all decorations, such as tic marks
unset key                       and text labels, from the inset plots
set margins 0, 0, 0, 0          ◁─────  Sets all margins for the insets to zero

set size 0.2                            ◁─────  Sets the size of the insets, as a
                                                fraction of the overall plot size
```

Left inset	`set origin 0.175,0.425` ◁─ **Sets the inset location by specifying its origin in screen coordinates** `clear` ◁───── **Clears the background for the inset** `plot [-2.5:2.5][-1.5:2] g(x,-1)` ◁───── **Generates the inset plot**
Middle inset	`set origin 0.4625,0.125` ◁─ **Sets the inset location by specifying its origin in screen coordinates** `clear` ◁───── **Clears the background for the inset** `plot [-2.5:2.5][-0.5:3] g(x,0)` ◁───── **Generates the inset plot**
Right inset	`set origin 0.675,0.425` ◁─ **Sets the inset location by specifying its origin in screen coordinates** `clear` ◁───── **Clears the background for the inset** `plot [-2.5:2.5][-0.5:3] g(x,1)` ◁───── **Generates the inset plot**

```
unset multiplot          ◁─────  Switches off multiplot mode
```

E.2   Box-and-whisker plots

Box-and-whisker plots (or *boxplots* for short) are a simple and robust way to visualize univariate point distributions. They serve the same purpose as kernel-density estimates (see section 3.5.2). Boxplots are somewhat easier to use, because they don't depend on an adjustable parameter (kernel-density estimates require a choice for the bandwidth). This convenience comes at a price: boxplots lose some detail of the represented data set (in particular, boxplots are unable to identify data sets having multiple maxima—an important limitation to keep in mind!) and aren't very intuitive.

E.2.1   Individual box-and-whisker plots

Boxplots are created using the familiar plot command together with the with boxplot style. (The boxplot keyword can't be abbreviated!) At least two parameters must be supplied in the using specifier. The first is the horizontal position at which the boxplot
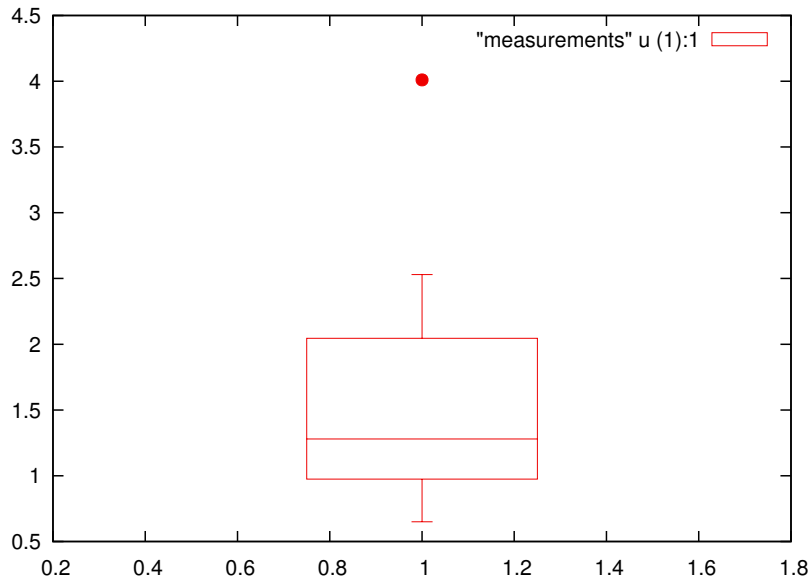
Figure E.6   A box-and-whisker plot, when drawn using only default appearance options

will be drawn (usually this is a numeric constant, but it can also be a column number so that the horizontal position will be read from file). The second entry in the `using` specifier indicates which column from the data file to represent via box and whiskers. (This means the first column in the `using` specifier gives the horizontal position and the second column the vertical "data"—just as for any other plot style.) To create a boxplot of the data from listing 3.4, you'd use this command (see figure E.6):

```
plot "measurements" u (1):1 w boxplot
```

It's not a problem to have multiple boxplots in the same graph (for easy comparison). You can also combine the boxplot style with other styles, such as lines or points.

SEMANTICS OF BOX-AND-WHISKER PLOTS

The various elements of box-and-whisker plots represent different properties of the underlying data set. Boxplots usually represent various *percentiles*: for skewed point distributions, these quantities are more informative than the (more familiar) mean and standard deviation. (The `boxplot` style uses gnuplot's `stats` command to find the median and percentiles. Check section 5.6 for details on the calculation.)

- The box indicates the *interquartile range*: the top and bottom boundaries of the box are chosen such that one quarter of points falls outside and below the box, and one quarter outside and above the box.
- The location of the *median* is indicated by a marker inside the box. The median divides the data set in two equal parts: half of all data points have a value less than the median, and half of the points have a value greater than the median.

- The construction of the whiskers is more complicated (and more ambiguous). Traditionally, whiskers extend to the most extreme data point that lies within 1.5 times the interquartile range from either edge of the box. The process is as follows: draw a line that's 1.5 times the height of the box outward from either end of the box, and then trim the line back to the last data point within its range. This location marks the end point of the whisker. (See figure E.7.)
- Points that fall outside the range of the whiskers are considered *outliers* and are indicated with individual symbols.

The complicated prescription for constructing the whiskers requires an explanation. The idea is to indicate the actual extent of the "tails" of the distribution, while distinguishing them clearly from true outliers.

What are outliers? A point is considered an outlier only if it's *exceptionally* far away from the other points. It isn't enough for a point to be the smallest or largest value in a data set: a data set that's very compact (so that all points cluster close together) doesn't have any outliers—but it still has a smallest and a largest value! All this means true outliers are points that are far from the center of the distribution, when measured *in units of the width of the distribution.* The method for constructing whiskers reflects this: only points far from the center *in comparison to the interquartile range* are considered outliers. (Look at figure E.7: it has an outlier at its upper range, but its lower part is sufficiently compact that no point is considered an outlier.)
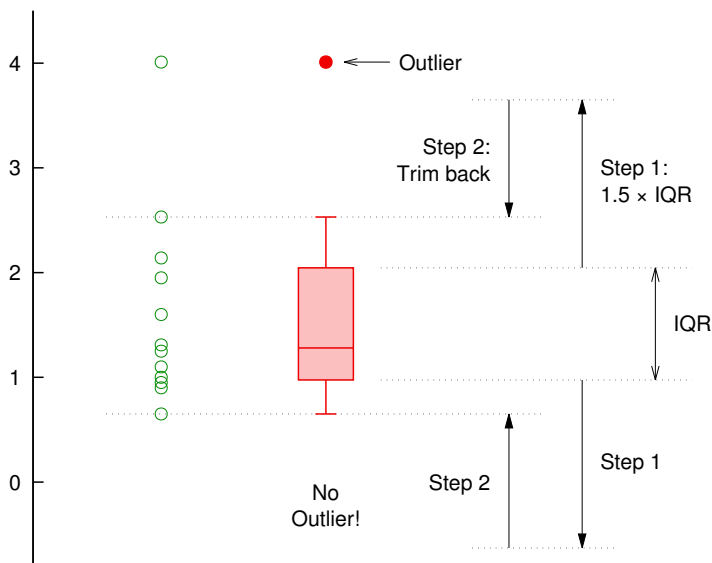
Figure E.7   Construction of a box-and-whisker plot from the data set. (The individual data points are shown on the left.)

Once outliers have been identified, each whisker is trimmed back toward the most extreme data point still within the range of the respective whisker. The length of each whisker therefore indicates the true extent of the tail of the distribution on either side of the distribution.

You can customize the precise determination of the whiskers using `set style boxplot`:

```
set style boxplot [range {flt:range} | fraction {flt:frac}]
```

If invoked with the `range` keyword, followed by a floating-point number, the whiskers are constructed in the way just explained, and the numerical argument is used as the factor by which the interquartile range is multiplied to determine the original extent of the whisker. This is the default behavior, and the default value of the multiplier is 1.5.

But it's also possible to designate a fixed *fraction* of points as outliers. This is done with the `fraction` keyword. It must be followed by a numeric argument between 0.5 and 1.0, which indicates the fraction of *regular* (that is, non-outlier) points. For example, `set style boxplot fraction 0.9` treats 10% of points as outliers. As before, the whiskers will extend to the data point furthest from the center that isn't considered an outlier.

APPEARANCE OPTIONS

In addition to the options just discussed, which control the *semantics* of the boxplot, there are also options that merely modify the visual appearance of the plot. Boxplots are similar to the `with candlesticks` style (see section 6.3.2), and many of the options mentioned there apply to the present situation as well. In particular, you can use `set bars` (see section 6.3.2) to customize the length of the bars at the end of the whiskers; `set boxwidth` controls the width of the central box, and `set style fill` its fill style. Instead of using global options, you can also use inline directives. For example, to have the central box lightly shaded, you can use

```
plot "measurements" u (1):1 w boxplot fs solid 0.2
```

Similarly, the box width can be specified inline, as a third entry to the `using` specifier. This makes it possible to have the box *width* be data-dependent. (Compare figure 14.14 and listing 14.9 for an example.)

Finally, there are some additional sub-options to `set style boxplot`:

```
set style boxplot [ [no]outliers ] [pointtype | pt {int:pointtype}]
                  [ candlesticks | financebars ]
```

By default, outliers are indicated with filled circles (point type 7). The point type can be overridden in the `plot` command or changed globally using `set style boxplot`. If there are several outliers with the same value, then they're plotted as individual points in a horizontal row. The keyword `nooutliers` suppresses the visual representation for individual outliers. It isn't possible to draw *horizontal* box-and-whisker plots with

gnuplot at this time, but you can use the `financebars` appearance in conjunction with the `boxplot` style instead of the `candlesticks` appearance (see section 6.3.2).

By default, gnuplot draws a box around the plotted data. I've argued elsewhere (see the sidebar in section 8.3.2) why this makes sense. For box-and-whisker plots, this additional boundary is less important, and you may want to switch it off (using the `set border` command—see section 9.5.2). But if you choose to do so, you should retain the left border and its tic marks: otherwise, there's no way for an observer to infer *quantitative* information from the plot! (See figure E.8 and listing E.5 for an example.)

> **TIP**   Many of gnuplot's default appearance options aren't particularly suitable for box-and-whisker plots. You can improve the quality of your graphs by customizing borders, tic marks, and tic labels. But it's essential to retain enough information that an observer can still infer numeric values from the graph!

E.2.2   *Serial box-and-whisker plots*

So far, we've only considered individual box-and-whisker plots for single data sets. But the technique becomes significantly more useful when you're dealing with an entire *series* of data sets, distinguished by a discrete label. For example, imagine that some experiment has been repeated several times, each time subject to a different *treatment level.* It would be nice to see a graph of all point distributions at once, showing a separate boxplot for each different treatment level. Fortunately, gnuplot let's you do just that.

All you have to do is to include a fourth column in the `using` specifier. The value of this fourth column is interpreted as the treatment level, and a separate boxplot is constructed for each new value of the level. (Of course, the values in the fourth column should form a discrete set—frequently it will be a string.) All the resulting boxplots are then shown together and on the same scale, facilitating comparison across treatment levels.

We'll demonstrate this feature using one of the most famous data sets in the history of statistics: Fisher's Iris data.[3] The data set consists of size measurements of different parts of Iris flowers. For each flower, four quantities were measured (sepal length and width, and petal length and width, all in centimeters). Three different kinds of Iris are considered (*I. Setosa, I. Versicolor,* and *I. Virginica*), and for each kind, the data set contains values for 50 instances. The question is whether it's possible to determine the type of plant from the measurements in the data set.

Figure E.8 shows the sepal length as box-and-whisker plots separately for each type of flower. The commands for this graph are in the next listing. (The file format of the data set available at the UCI repository is comma-separated, not whitespace-separated, requiring a call to `set datafile separator`!)

---

[3] A quick internet search will turn up multiple sources; I use the version found in the UCI Machine Learning Repository at http://archive.ics.uci.edu/ml/datasets/Iris.

Listing E.5   Commands for figure E.8. (file: iris1.gp)

```
set datafile separator ","
unset key
set border 2                    ◁─────── Retains only the border on the left
set xtics scale 0               ◁─────── Removes the tics, but retains the tic labels
set ytics nomirror              ◁─────── Avoids tics on the right
set ylabel "Sepal length [cm]"
plot "iris.data" u (1):1:(0.5):5 w boxplot
```

This graph follows the suggestions made at the end of the previous section: it does away with most decorations, but retains the numerical scale along the left edge. You'll also notice that gnuplot automatically labels each boxplot with the corresponding level (that is, the entry in the fourth column in the `using` expression).
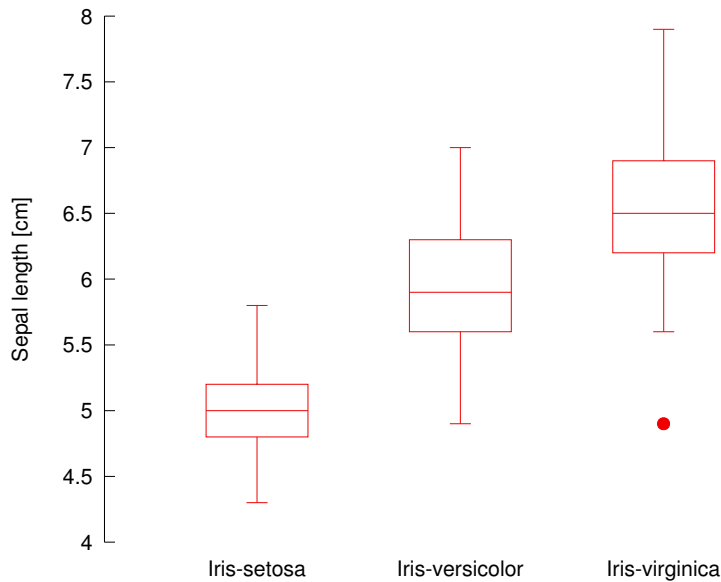
Figure E.8   A serial box-and-whisker plot. The distribution of sepal lengths is shown separately for each type of flower.

APPEARANCE OPTIONS FOR SERIAL BOX-AND-WHISKER PLOTS

Most of what you learned for a single box-and-whisker plot carries over to serial box-and-whisker plots. The first entry in the `using` specification (which previously gave the horizontal location of the boxplot) now gives the horizontal position of the *first* (left-most) boxplot in the series. The spacing between successive boxplots is controlled using `set style boxplot`, which also allows you to customize some other appearance aspects:

```
set style boxplot [ separation {flt:s} ]
                  [ labels [ off | auto | x | x2 ] ]
                  [ [un]sorted ]
```

In brief:

- Using the `separation` keyword, you can control the distance between boxes. The argument is the desired separation in x axis units; its default is 1.
- By default, the value of the fourth column (the level) is printed as a tic label on the horizontal axis. The axis can be selected using `x` (below) or `x2` (above). Choosing `auto` places the labels on whichever axis is indicated by the `axes` keyword inside the `plot` command (see section 8.1.2). The keyword `off` suppresses printing of labels.
- Boxplots are sorted by the *first* appearance of their level in the data file. (It isn't necessary for the data file to be grouped by level, hence the first appearance of a level determines the sort order.) This can be changed using the `sorted` option. Levels are assumed to be strings, so the sort order is alphabetical, not numeric. An arbitrary sort order isn't possible at this time.

Here are three further things to keep in mind:

- As you saw in figure E.8, gnuplot automatically places a string label for the level in the margin. These labels are equivalent to tic marks and tic labels; you can customize their appearance using `set xtic`.
- Although I don't show an example here, it's absolutely possible to combine multiple data sets in a single graph. For example, if you want to plot both sepal length and width in a single plot, you can do so as follows:

```
plot "iris.data" u (1):1:(.5):5 w boxplot, "" u (5):2:(.5):5 w boxplot
```

This creates boxplots for the sepal lengths of the three types of flower starting at x=1, together with corresponding boxplots of the sepal widths, starting at x=5.
- It's unfortunately not possible to draw each level in a separate color or style at this time.

WORKED EXAMPLE: AN ARRAY OF SERIAL BOXPLOTS

In figure E.8, you can see the distribution of sepal lengths for all three types of flowers at once—but wouldn't it be great if you could compare all *four* measured quantities for all types of flowers in a single graph? The `multiplot` mode that was explained earlier in this appendix makes this easy.

Figure E.9 shows all four measurements in a 2×2 matrix of serial boxplots. Looking over these graphs, it immediately stands out that *I. setosa* is distinguished by petals that are smaller, in both length and width, than those of the other two variants. Because the boxplots display outliers explicitly, you know for sure that even the largest instance of *I. setosa* in this data set was smaller than the smallest instance of either of the other two types!
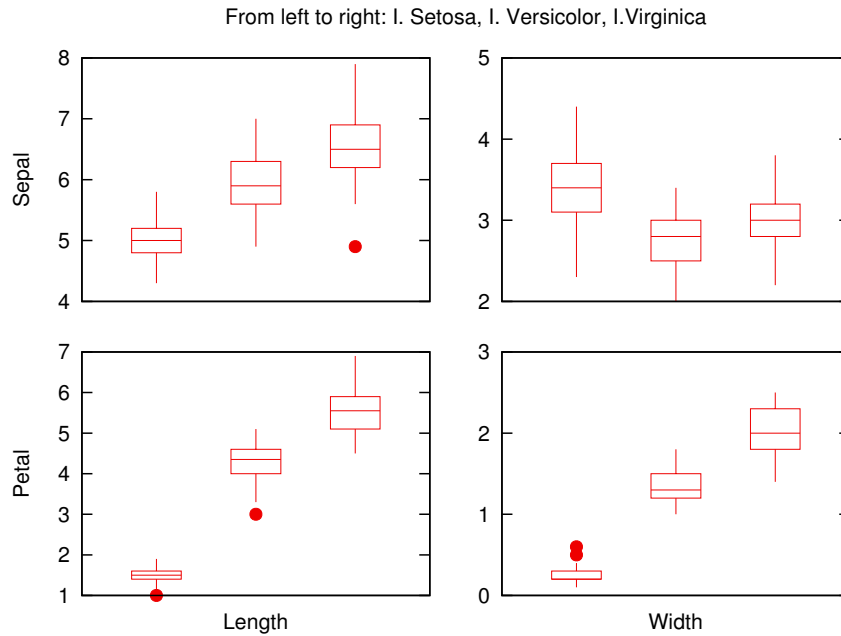
Figure E.9 An array of serial box-and-whisker plots. The distribution of all four observed quantities is displayed for all flower types separately. (See listing E.6.)

You can see the commands for figure E.9 in the next listing. Notice that I retained the borders around the individual plots in this case: this helps to distinguish them from one another in the array of graphs.

Listing E.6   Commands for figure E.9 (file: iris2.gp)

```
set datafile separator ","
unset xtics; set ytics 1
set bars 0; unset key;

set multiplot
➡    title "From left to right: I. Setosa, I. Versicolor, I.Virginica"
➡    layout 2,2 margins 0.1,0.9,0.1,0.9 spacing 0.075

set xlabel ""; set ylabel "Sepal"
plot "iris.data" u (1):1:(0.5):5 w boxplot
set xlabel ""; set ylabel ""
plot "iris.data" u (1):2:(0.5):5 w boxplot

set xlabel "Length"; set ylabel "Petal"
plot "iris.data" u (1):3:(0.5):5 w boxplot
set xlabel "Width"; set ylabel ""
plot "iris.data" u (1):4:(0.5):5 w boxplot

unset multi
```

E.3   *Parallel coordinates*

Graphs using *parallel coordinates* are a relatively new idea. Their purpose is to reveal structure in *multivariate* data sets: that is, data sets that have more than two or three attributes per record. In contrast to some other multivariate visualization techniques, parallel coordinates aren't limited to data sets containing only a few records. The parallel-coordinates technique therefore scales well, both in the number of attributes and in the number of records. The downside is that plots of parallel coordinates are neither pretty nor particularly intuitive. You aren't likely to exhibit them in a presentation, but they may serve as an intermediate discovery tool.

To understand parallel coordinates, let's go back to the Iris data set introduced in section E.2.2. For the moment, consider just the first record in that data set. It has the following values in its five columns:

```
Sepal length      5.1 cm
Sepal width       3.5 cm
Petal length      1.4 cm
Petal width       0.2 cm
Name              Iris setosa
```

To construct a parallel-coordinates plot, you draw a coordinate axis for each attribute; but in contrast to other types of graphs, all axes are arranged in parallel. The values for every record are then marked on each coordinate axis and connected with lines. For the first record from the Iris data set, the result is shown in figure E.10.
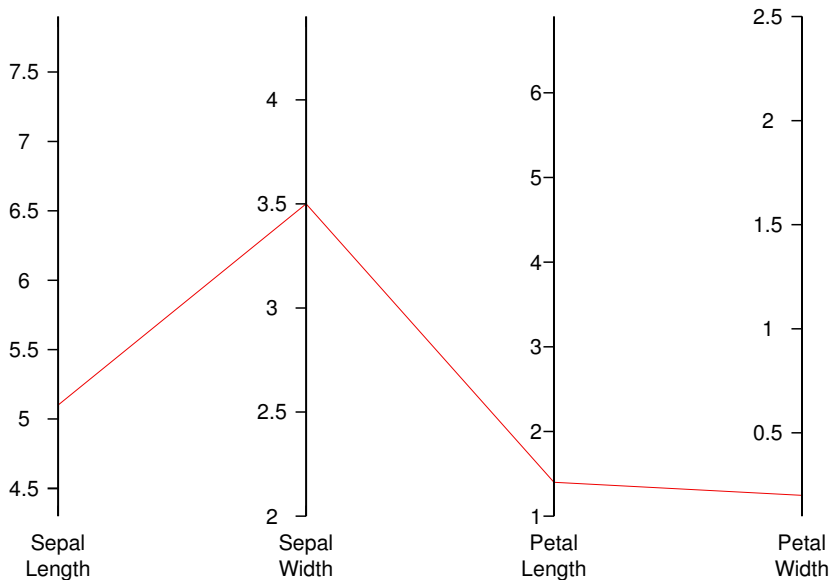
Figure E.10   Parallel-coordinates plot for a *single* record

This procedure is repeated for all records in the data set (see figure E.11). Now this plot can be examined for possible structure in the data set. Look for clusters or gaps along one of the axes: those can be used to classify data sets into groups. You can also look for outliers and for correlations among data sets: positively correlated quantities show up as parallel (or nearly parallel) lines, whereas negative correlation is indicated by lines crossing each other. From figure E.11, for instance, a cluster of records with relatively small values for petal length and width stands out: those belong to *I. setosa*—something you already knew from figure E.9. But from the parallel-coordinates plot, it becomes also obvious that short sepal length implies long sepal width (and vice versa)—a fact that isn't so easy to discern from figure E.9.

Having seen what graphs of parallel coordinates are and how they're used, let's understand how to create them with gnuplot. (Support for this feature is still new. The basic functionality works well, but you'll frequently find that you need to change the default appearance yourself.)

E.3.1 *Creating parallel-coordinates graphs*

You create graphs of parallel coordinates in gnuplot using the `plot` command together with the `with parallelaxes` style. The individual columns that are used for the axes are listed as entries in the `using` specifier. The basic command to create figure E.11 is therefore

```
set datafile separator ","
plot "iris.data" u 1:2:3:4 w parallelaxes
```
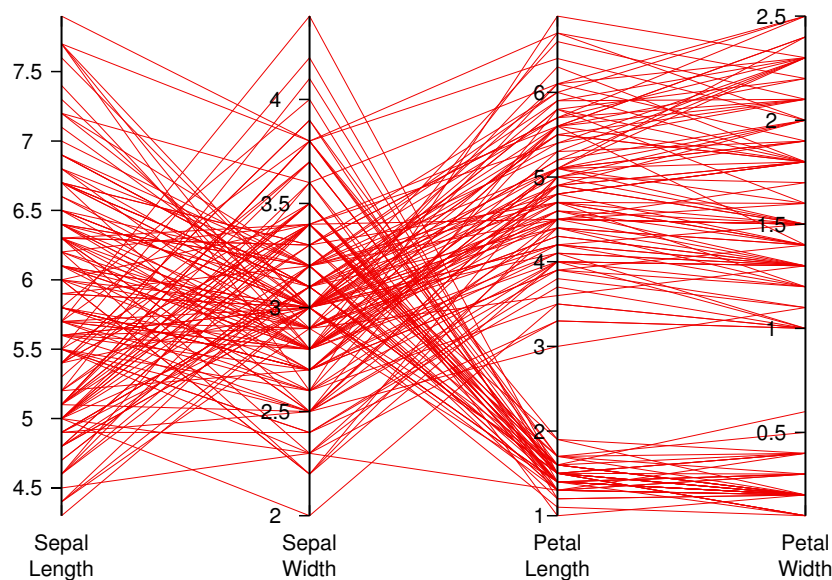

Figure E.11   Parallel-coordinates plot for the entire Iris data set

A word of warning: the current implementation arbitrarily limits the number of columns to seven.[4] If you want to use more columns, you need to employ multiplot mode—see section 14.2.5 for a worked example.

APPEARANCE OPTIONS FOR PARALLEL COORDINATES

You can use the `set paxis` command to control the appearance of individual axes in a parallel-coordinates plot:

```
set paxis {int:n} [ range {range-options} | tics {tic-options} ]
```

This command combines the functionality of the `set _range` and the `set _tics` commands familiar from chapter 8. The mandatory first argument to `set paxis` is an integer between 1 and 7 that identifies which of the parallel axes the options should be applied to. (The axis with index 1 is always the left-most one.) Only a single axis can be identified with each invocation to `set paxis`; if you want to make adjustments to several axes, you need to do so explicitly or use an inline loop (`set for[k=1:7] paxis k ...`).

   The `range` keyword must be followed by a legal range specification (like so: `set paxis 1 range [0:10]`). The `tics` keyword acts like the familiar `set xtics` or `set ytics`; all options available to those commands apply in the current case as well.

   The appearance of the parallel *axes* can be customized using[5]

```
set style parallel [ front | back ] [ lineoptions ]
```

By default, the axes and the associated tic marks are drawn in front of the data. The axes are drawn as thick black lines; you may find that you want to reduce the thickness. (See the tip at the end of section 6.2.3 for all the appearance aspects covered by `lineoptions`.)

E.3.2   *Worked example: Iris data, again*

I hope the preceding discussion has convinced you that gnuplot's parallel-coordinates feature is remarkably easy to use, despite the intimidating appearance of the resulting graphs. In order to get the most value out of these graphs, it's nonetheless necessary to augment gnuplot's functionality. In this section, you'll first see ways to improve the appearance of the plot and then learn how to *highlight a subset of records.* The latter is an essential capability when using parallel-coordinates plots. (Chapter 14 shows some additional tricks applicable in particular to data sets containing a large number of records.)

---

[4]   The details of the implementation may have changed by the time you read this. Check the release notes for the most up-to-date information.

[5]   This feature wasn't documented in the standard gnuplot reference documentation at the time of this writing.

IMPROVING APPEARANCE

Similar to the situation you encountered for serial box-and-whisker plots (section E.2.1), gnuplot's defaults aren't the most suitable for parallel-coordinates plots, either. (As you may have found out already, the graphs resulting from the simple command given earlier look quite different than figure E.11.) You can improve the appearance of the plot by adapting the defaults appropriately. Here's how it's done.

> **Listing E.7 Commands for figure E.11 (file: iris3.gp)**

```
set datafile separator ','

unset key
unset border                                    Switches off unnecessary decorations
unset ytics
set xtics ("Sepal\nLength" 1, "Sepal\nWidth" 2,     Replaces the tic labels along
          "Petal\nLength" 3, "Petal\nWidth" 4 )     the horizontal axis with the
                                                     names of the quantities plotted
set lmargin 6                                   Adjusts the left and bottom margins
set bmargin 3                                   to make room for the labels

set for[k=1:4] paxis k tics          ◁──── Turns on tic marks for all four parallel axes

plot [1:4] "iris.data" u 1:2:3:4 w parallel
```

Because there are exactly four parallel axes (located at positions 1 through 4), the plot range is restricted in the plot command to that interval. This gets rid of the margins that gnuplot leaves by default on both sides of a parallel-coordinates plot and maximizes the area available for the plot. Whether you want to switch off the borders around the entire graph is a personal choice. I prefer the appearance of the graph without them—in particular the horizontal lines at top and bottom feel visually constraining.

HIGHLIGHTING A SUBSET OF RECORDS

The purpose of parallel-coordinates plots is to identify structure in a data set. The visual presentation makes it easy to see where lines cluster (for example, in figure E.11, there is an obvious cluster of lines with small petal length). This leads to the next question: how are lines that cluster for some attribute distributed elsewhere? If you could just select the lines in the cluster so that they're *highlighted* everywhere, then this question would be answered. In this section, I'll show you how to do this, but there's more: you'll also learn how to take the type of flower into account and to include it in the highlighting (see figure E.12).[6]

The name of the flower is given in the data set as a string. To plot it, you must map the names to numbers. The most convenient way to do such transformations is

---

[6] After this section was written, I noticed that the Wikipedia page for parallel coordinates also uses Fisher's Iris data to demonstrate this technique and includes a figure very similar to figure E.12. As I said, it's one of *the* best-known and most-frequently used data sets in all of statistics!
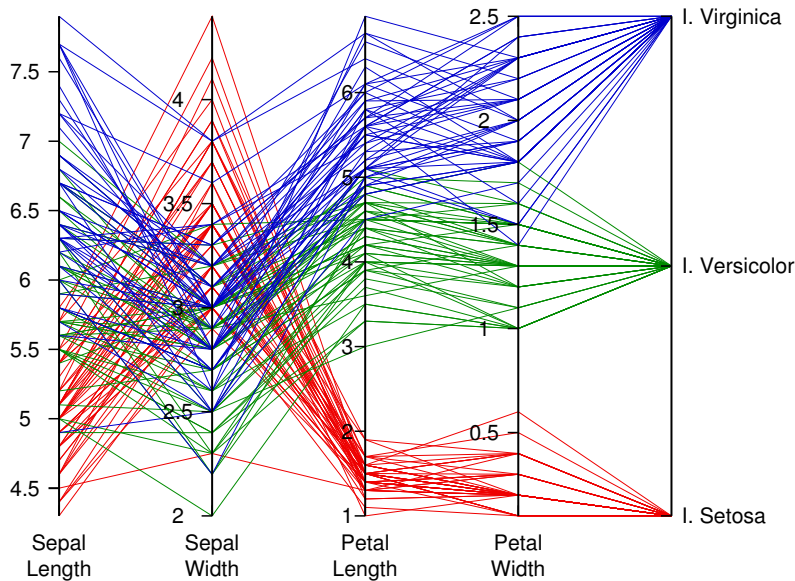
Figure E.12 A parallel-coordinates plot for the entire Iris data set. Records are color-coded based on the flower type.

probably using an external scripting language, but it can be done entirely in gnuplot, too. The following function (`cls` is short for *class*)

```
cls(s) = s eq 'Iris-setosa' ? 1:(s eq 'Iris-versicolor' ? 2:3)
```

consists of two nested conditionals and maps the names to the value 1, 2, or 3. To include the names in the plot, remember that all of gnuplot's commands to control tic labels are available for each axis in a parallel-coordinates plot. With

```
set paxis 5 tics ( "I. Setosa" 1, "I. Versicolor" 2, "I. Virginica" 3 )
➠   left offset 4,0
```

you can place informative labels on the tics of the new, fifth axis.

The last trick is to use data-dependent coloring for the lines themselves (see section 9.1.5)! You remember: if you use `linecolor variable` (or `lc var` for short), then the right-most entry in the `using` specifier is interpreted as the *index of a color* from gnuplot's sequence of linetypes (and their associated colors). Because the function `cls(s)` already maps the flower names to the values 1, 2, and 3, you can use these values directly as a color index: each flower type is drawn with a different line type (and therefore a different color). You can find all the commands in the next listing. (One last comment: instead of adjusting the left and right margins explicitly, here I decided to manage the horizontal extent of the plot entirely through the plot range. That's sheerly a matter of convenience—you can do it either way.)

Listing E.8 Commands for figure E.12 (file: iris4.gp)

```
set datafile separator ','

unset key
unset border
unset ytics
set xtics ( "Sepal\nLength" 1, "Sepal\nWidth" 2,
➥          "Petal\nLength" 3, "Petal\nWidth" 4 )
set bmargin 3

set for[k=1:4] paxis k tics
set paxis 5 tics ( "I. Setosa" 1, "I. Versicolor" 2, "I. Virginica" 3 )
➥   left offset 4,0

cls(s) = s eq 'Iris-setosa' ? 1:(s eq 'Iris-versicolor' ? 2:3)

plot [0.6:5.8] "iris.data" u 1:2:3:4:(cls(strcol(5))):(cls(strcol(5)))
➥             w parallel lc var
```

E.4   *Histograms*

The histogram styles are a bit of a departure from gnuplot's usual processing model, in that they have the concept of a *data set*. The overall appearance of the plot depends on both row and column information simultaneously.

Histograms are the result of counting statistics. For the sake of discussion, let's assume that there are three political parties (Red, Blue, and Green), and you want to show the number of votes for each party. The outcome of a *single* election can be shown easily using, for instance, the histeps style.

But what to do if elections are held annually, and you want to show the results for a number of years in one plot? To make matters concrete, let's consider a specific data file, shown next.

Listing E.9   Data for figures E.13 and E.14 (file: elections)

```
# Year   Red   Green Blue   Spending
1990     33    45    18     21
1991     35    42    19     18
1992     34    44    14     25
1993     37    43    25     27
1994     47    15    30     28
1995     41    14    32     33
1996     42    20    35     35
1997     39    21    31     32
```

One possible solution would be to plot the data as a regular time series (see figure E.13—ignoring the last column for the time being):

```
set style data lp            ◄——— Uses line points              Uses a filled circle
                                                                as a symbol
set for [k=1:3] lt k pt 7                                    ◄——┘
plot "elections" u 1:2 t "Red", "" u 1:3 t "Green", "" u 1:4 t "Blue"
```
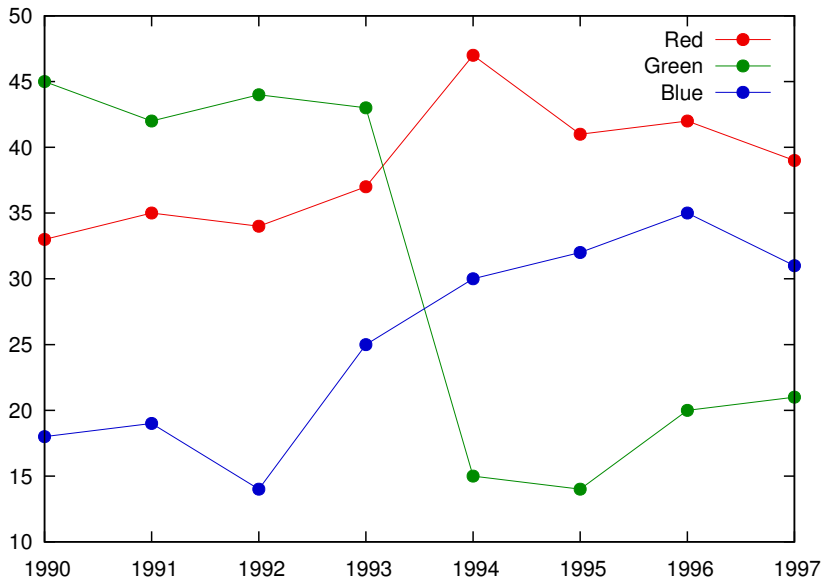
Figure E.13   Election results as a time series. The data file is shown in listing E.9.

Often this is exactly what you want, but this format can be clumsy, in particular if there are many competing parties or if there is a lot of variation year over year. The `histogram` style offers an alternative.

Using `set style histogram clustered` generates a sequence of histograms (see figure E.14). Each histogram corresponds to *one row* in the input file (in this example, this corresponds to one year):

```
set style fill solid 0.5
set style histogram clustered
plot "elections" u 2 t "Red" w histograms,
➥    "" u 3 t "Green" w histograms, "" u 4 t "Blue" w histograms
```

Instead of repeating the `with histograms` style directive multiple times in the `plot` command, you can select the global style to use histograms for all data sets:

```
set style fill solid 0.5
set style histogram clustered
set style data histograms
plot "elections" u 2 t "Red", "" u 3 t "Green", "" u 4 t "Blue"
```

For all the histogram styles, it's usually a good idea to have the boxes filled to make them more easily distinguishable, and so this option is enabled here. You can control the spacing between consecutive histograms using the optional `gap` parameter to the `set style` command: `set style histogram clustered gap 2`. The size of the gap is measured in multiples of individual boxes in the histograms. (To create gaps within each histogram, so that neighboring boxes don't touch each other, use `set boxwidth`.)
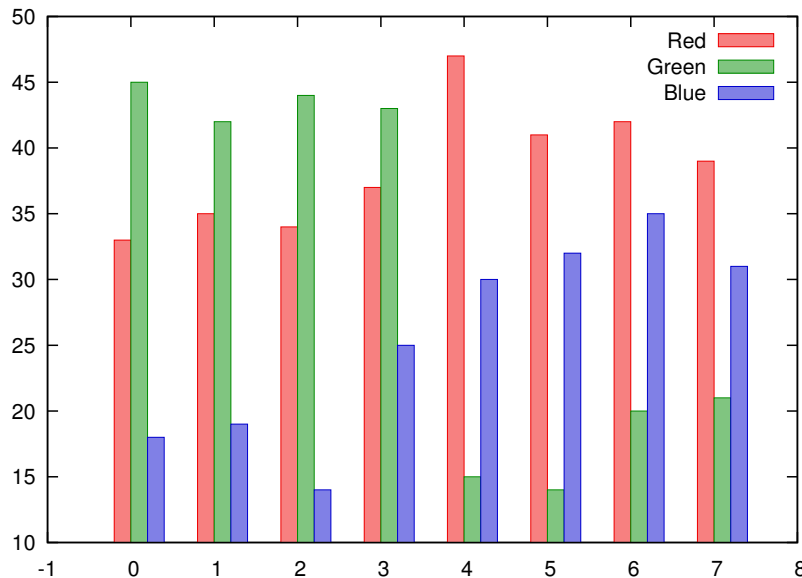
Figure E.14 **Election results using `set style histogram clustered`. This is the same data set as in figure E.13.**

Keep in mind that the way gnuplot reads the data file for histograms is a bit unusual: each new row generates a new histogram *cluster*, but the `histogram` style accepts only a *single* column in the `using` directive. You therefore have to list the file repeatedly in the same `plot` command to generate meaningful histograms, as shown previously.

Finally, the labels gnuplot places along the x axis aren't very meaningful. You can either set explicit labels for each histogram using the `set xtics add` command or read a textual label from the data file using the function `xticlabels()` (`xtic()` for short). The effect of the latter is demonstrated in figure E.15. (For more information on tic labels, see chapter 8.)

In addition to the `clustered` histogram style we looked at so far, there is also a *stacked* style. Using this style, the individual boxes aren't placed next to one another, but stacked on top of each other, so that each vertical box forms an entire histogram. By default, adjacent boxes touch each other, but as usual, you can use `set boxwidth` to control the width of individual boxes (see figure E.15):

```
set style fill solid 0.5
set style histogram rowstacked
set style data histograms
plot "elections" u 2:xtic(1) t "Red", "" u 3 t "Green", "" u 4 t "Blue"
```

There are two additional histogram styles, which I won't describe here in detail, because they're similar to the ones we discussed already. The `set style histogram errorbars` style is similar to the `clustered` style, except that it reads *two* values for
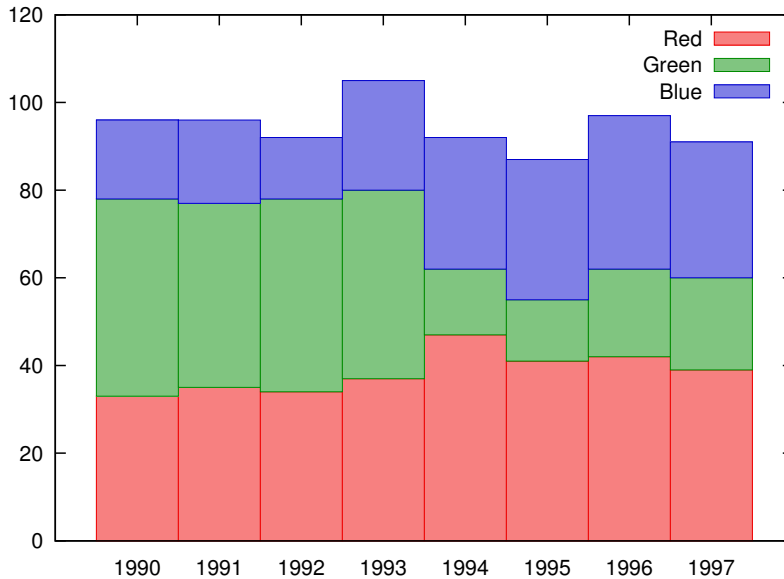
Figure E.15   Election results using `set style histogram rowstacked`. This is the same data set yet again. Note the effect of the `xtic()` function, which is used to read x axis labels directly from the data file.

each box, the second being the uncertainty in the data, represented with a standard error bar. The `set style histogram columnstacked` style is equivalent to the `rowstacked` style, except that each vertical box is built from a single *column* (not row) in the input file.

One last directive related to histograms is `newhistogram` (see figure E.16). It can be used to have several *sets* of clustered histograms on the same plot. An example will suffice:

```
set style fill solid 0.5
set style histogram clustered
set style data histogram

set bmargin 5                            ⊲————— Makes room for xtics
set xtic rotate by 90 offset 0,-2        ⊲————— Rotates xtics so they don't overlap

plot newhistogram "\n\nElection Results",
➥        "elections" u 2:xtic(1) t "Red", "" u 3 t "Green", "" u 4 t "Blue",
➥    newhistogram "\n\nCampaign Spending",
➥        "" u 4:xtic(1) t "Spending"
```

The syntax for the `newhistogram` command is a bit unintuitive (I *still* tend to get it wrong), so let me point out the salient features: the `newhistogram` keyword is followed by a *mandatory* string label (an empty string is permitted) and a *mandatory* comma, and then the rest of the `plot` command follows, starting with the filename.
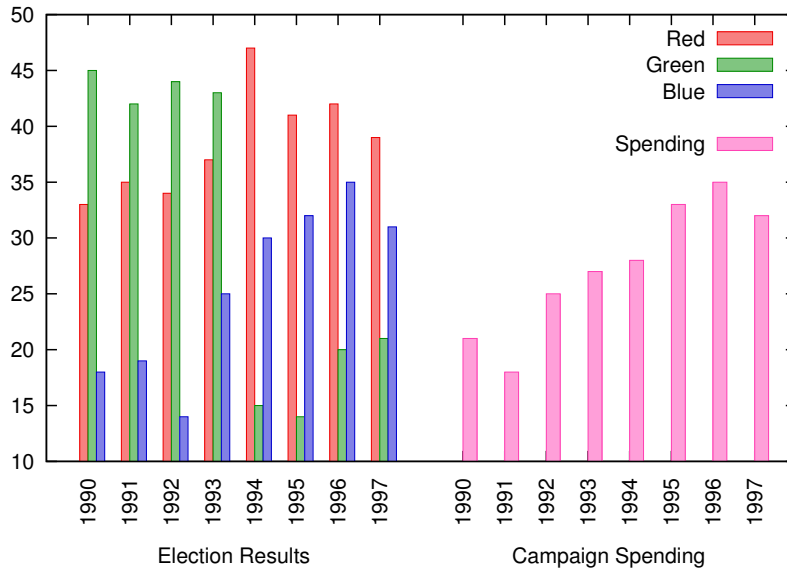
Figure E.16 **Several histograms can be combined in a single graph using `newhistogram`.**

In addition to the commands that are directly related to the histogram feature, this example includes some appearance options: tic labels are rotated so that they don't overlap each other, and the bottom margin has been increased to make room for both the rotated tic labels as well as the histogram label supplied to the `newhistogram` keyword. Finally, I began each of the two histogram labels with some newlines: otherwise, they interfere with the tic labels.

Histograms such as those discussed in this section look good and are frequently used in business presentations or in the media. But they make it difficult to see trends in the data, and in particular quantitative comparison of data can be difficult. (Gnuplot offers some additional features that I didn't mention; check the standard reference information if you want to know more.)

appendix F
Higher math

This appendix discusses several topics that are mathematically a bit more advanced than the rest of the book. We'll explore gnuplot's support for parametrized curves and non-Cartesian coordinates (such as polar, cylindrical, and spherical coordinates). In section 6.3.5, you learned about the `with vectors` style; here, we'll demonstrate some additional uses for it. Gnuplot has built-in support for complex numbers, which we haven't mentioned so far—we'll remedy this now. Next I'll show you how to create probability plots with gnuplot; and finally, there's a lengthy section on fitting nonlinear functions to data.

Because this isn't a math book but a book on gnuplot, I won't provide much of the background for these concepts; I'll just show you how gnuplot handles them. This appendix is best skipped unless you have a specific need for its contents. Rest assured: if you don't know what I'm talking about, you aren't going to need it.

F.1   Parametric plots

Parametric plots are only relevant when you're plotting functions with gnuplot: the concept has no meaning when plotting data from files. So far, whenever you've plotted a function, the function provided a y value (the *dependent* variable) as a function of an x value (the *independent* variable): `plot sqrt(x)`. Alternatively, you can give *two* functions, one for the x and one for the y value separately, but both depending on a common *parameter*, which by convention is called t. For certain kinds of curves, such a parametrization is simpler than the explicit form. The following `plot` command draws a circle (after adjusting the aspect ratio appropriately):

```
set size square

set parametric
plot [0:2*pi] cos(t), sin(t)
```

This parametric representation is significantly simpler than the equivalent explicit form:

```
set size square

unset parametric
plot [-1:1] sqrt(1-x**2), -sqrt(1-x**2)
```

F1

In general, the following two commands give the same result:

```
unset parametric
plot f(x)

set parametric
plot t, f(t)
```

For parametric plots, there are now *three* relevant plot ranges: `trange`, `xrange`, and `yrange`. The first controls the range of values assumed for the parameter (the dummy variable) `t`, whereas the other two are used (as before) to control the range of points visible in the plot. All three ranges can be adjusted either explicitly using `set trange` or the familiar `set xrange` and `set yrange` commands (see section 8.2), or inline as part of the `plot` command. If used inline, the setting for the `trange` precedes the other two. This is demonstrated in the following listing. The resulting plot is shown in figure F.1.

> **Listing F.1   Parametric mode: commands for figure F.1 (file: lissajous.gp)**

```
set parametric
set size square

plot [0:2*pi][-1.1:1.1][-1.1:1.1] sin(2*t), cos(5*t)
```
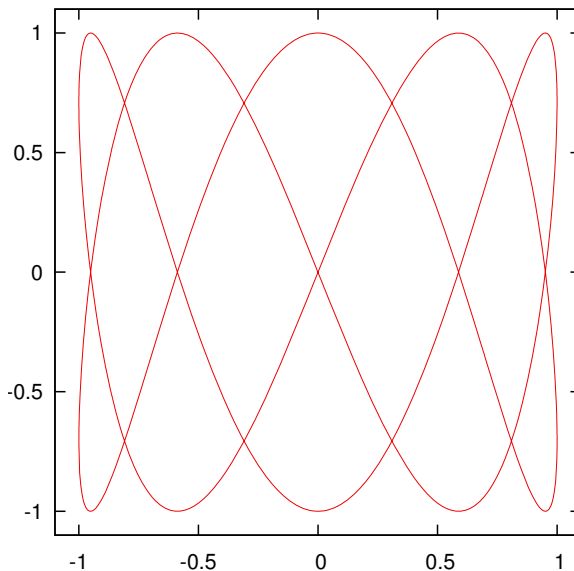

Figure F.1   A simple Lissajous figure, drawn using parametric mode. See listing F.1.

Parametric graphs of trigonometric functions (such as figure F.1) are known as *Lissajous figures*, and gnuplot's parametric mode makes drawing them very easy. Simple

expressions, as in listing F.1, lead to simple graphs, but more complicated expressions can also be fun. Figure F.2, for instance, was generated as follows:[1]

```
set parametric
set size square
unset key; unset border; unset tics

set samples 100000
plot [0:100*pi] sin(0.99*t)-0.7*cos(3.01*t), cos(1.01*t)+0.1*sin(15.03*t)
```
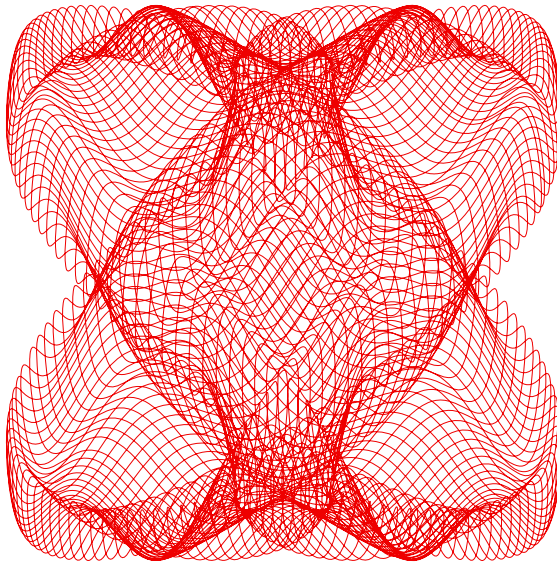

Figure F.2   A more complicated figure drawn with parametric mode. Plotted are the functions `sin(0.99*t)-0.7*cos(3.01*t)` for the horizontal coordinate and `cos(1.01*t)+0.1*sin(15.03*t)` for the vertical coordinate.

For three-dimensional plots (using the `splot` command), you obviously need to provide a third function, with all three functions depending on two variables called `u` and `v` and possessing corresponding options `set urange` and `set vrange`.

F.2   Non-Cartesian coordinates

Not all coordinate systems are made out of straight lines at right angles to each other. To plot points on a circle, for instance, it makes more sense to use polar coordinates, or to use spherical or cylindrical coordinates for systems possessing those kinds of symmetry. Gnuplot has built-in support for these three symmetries through the `set polar` and `set mapping` facilities. To generate plots using other coordinate systems, you have to resort to inline data transformations or external filters.

---

[1] This example was taken from A. K. Dewdney's "Computer Recreation" column in *Scientific American*, May 1988.

F.2.1 Polar coordinates

Polar mode is enabled using

```
set polar
```

which doesn't take any further options. In polar mode, the independent variable is interpreted as an *angle* and the dependent variable as a *radius*. When plotting *data*, this means the first column specified in the using clause is interpreted as the angle and the second column as the radius. When plotting *functions* in polar mode, the independent (dummy) variable is denoted with t (not x), similar to the convention used for parametric plots (see section F.1).

The angle may be given either in radians (multiples of π—this is the default) or in degrees. You can switch between both representations using

```
set angles [ degrees | radians ]
```

The current value of this option affects not only polar plots, but also how the arguments of trigonometric functions are interpreted.

Let's look at an example (see figure F.3). The process to generate attractive plots in polar mode requires several steps, because some artifacts from regular plots must be removed explicitly. It's also necessary to introduce some additional options controlling the appearance of polar plots.[2] The individual steps are shown in the following listing.

> **Listing F.2 Polar mode: commands for figure F.3 (file: polar.gp)**

Switches on polar mode. Angles are interpreted as radians (the default).

```
 ⊳  set polar
    set size square                       ◁────── Ensures a square aspect ratio

    unset border
    unset tics                              │ Switches off border and tic marks
    set tmargin 2; set bmargin 2          ◁────── Makes room for labels at the top and bottom
    set key left at graph 1,0.95          ◁────── Gets the key out of the way

    set rrange [0:1]                      ◁────── Fixes the radial range
    set raxis                             ◁────── Switches on the axis at zero-angle
    set rtics 0.25 nomirror scale 0.5     ◁────── Defines the spacing for radial tic marks and grid lines
    set grid polar pi/8.0                 ◁─┐ Turns on the grid, and defines the angular
                                             spacing tic marks and grid lines
```

Removes the usual decorations

Adds polar plot decorations

---

2 The way polar plots are handled has changed from previous versions of gnuplot; in particular, the commands set raxis and set rtics are relatively new.

```
set label "0π" at 1,0 offset char 1,0
set label "+0.5π" at 0,1 center offset char 0,0.75
set label "-0.5π" at 0,-1 center offset char 0,-0.75
set label "+/-π" at -1,0 offset char -5,0
```

Defines angular tic labels

```
plot [-0.875*pi:0.875*pi] "polardata" u 1:2 t "Data" w l,
➥                         0.27*(2-cos(3*t)+0.75*cos(5*t)) t "Model"
```

Plots the data together with a function. The plot range controls the dummy variable t (effectively, the angle variable).
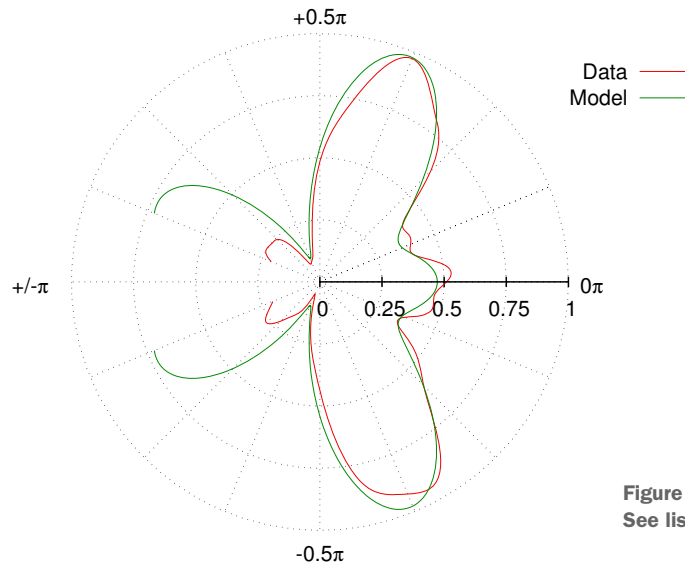
Figure F.3   A graph in polar mode. See listing F.2.

APPEARANCE OPTIONS FOR POLAR DIAGRAMS

Gnuplot draws polar diagrams with the zero angle located on the right (rather than at the top, which is possibly more common). By default, the half-axis at zero-angle is drawn explicitly; you can turn it off using unset raxis.

With polar plots, it's helpful to have a grid (as is the case with figure F.3). The spacing of radial lines in the polar grid is controlled through the argument to set grid polar, and the concentric circular grid lines are controlled through set rtics. The set rtics command behaves just like the familiar set xtics and set ytics commands and takes the same sub-options (see section 8.3). One difference is the keyword mirror: if it's used in conjunction with set raxis, then radial tic marks are drawn on the left half-axis as well.

The radial plot range is controlled through `set rrange`. The radial range need not include 0. It's possible to specify a range of values such as `set rrange [0.2:0.8]`, but the diagram always appears like a circular disk, not as an annulus. Only a small circle near the origin is left blank to indicate that 0 isn't included. (Gnuplot will neither warn you if you specify negative values in `set rrange` nor suppress such data points if you try to plot them. You have been warned.)

Finally, if you study the commands in listing F.2 carefully, you'll notice that gnuplot doesn't really have a polar coordinate system! The positions for the labels, for instance, are given in the `first` coordinate system (see section 7.2), despite the fact that this coordinate system isn't visible in the graph. To identify arbitrary locations, you'll probably want to use explicit trigonometric expressions to perform the necessary calculations on the fly.

F.2.2   *Cylindrical and spherical coordinates*

Polar mode only makes sense for graphs generated using `plot`—the equivalent for graphs created with `splot` is the `set mapping` option:

```
set mapping [ cartesian | cylindrical | spherical ]
```

I'll demonstrate this facility here by means of a familiar object (see figure F.4). The coordinates (at a scale of 1 to 10 million) are contained in the file world_110m.txt.[3] This data is now plotted twice: once using `plot` as a projection into the plane, and once with `set mapping spherical`.

The projection into the plane holds no surprises, but the spherical representation warrants a few comments. As you can see in listing F.3, you first switch on spherical
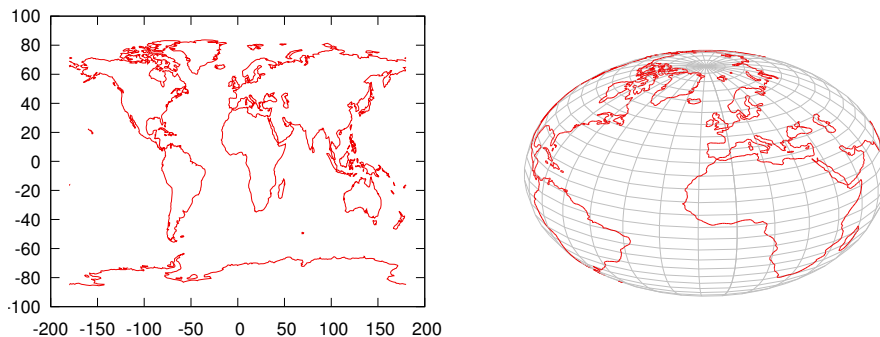
Figure F.4   Using a spherical coordinate system together with `splot`. On the left, the data is plotted as a regular two-dimensional plot using `plot`; on the right, it's plotted (together with a grid) using `set mapping spherical` and `splot`. (See listing F.3.)

---

3   This data set has a much greater resolution than the world.dat file that is part of the gnuplot distribution. You can find the data set used here at www.gnuplotting.org/plotting-the-world-revisited/. It's based on original data, which you can find at www.naturalearthdata.com/downloads.

coordinates and turn off most standard decorations—just as for polar plots (in section F.2.1 a moment ago). But in this case, you also turn on *parametric mode* (see section F.1). Remember that parametric mode has no bearing on the plotting of data; it only affects functions. Here, it's used to draw a grid on the surface of the globe, using the parametrized version of the surface of a sphere that makes up the last entry in the `plot` command.

This should be enough to demonstrate non-Cartesian coordinates. If you need to know more, check the standard gnuplot reference documentation for further details.

Listing F.3 Commands for figure F.4 (file: world.gp)

```
set mapping spherical          ◁——————  Switches on spherical coordinates
set angles degrees             ◁——————  Uses degrees, not radians

unset border
unset tics                     Cleans up
unset key                      unnecessary
set hidden3d                   decorations

set parametric
set urange [-90:90]            Used by the functions
set vrange [0:360]             that draw the grid lines

set view 55,80,1.5,1.25                 ◁——————  Chooses the view point

splot "world_110m.txt" w l,
➥     cos(u)*cos(v),-cos(u)*sin(v),sin(u) w l lc rgb "grey"
```

F.3 *Vector fields*

Vector-valued quantities have both a magnitude and a direction, in contrast to scalars, which only have a magnitude. Simply put, vectors are arrows, whereas scalars are numbers. Think of a climber in a mountainscape: the local height is a scalar, but the direction in which the climber would tumble down (if he let go) is a vector.

You've already encountered gnuplot's `with vectors` style (in section 6.3.5), which can be used to plot vector-valued quantities. So far, it's only been used in conjunction with the `plot` command, but it's also available for `splot`.

F.3.1 *Plane vector fields with plot*

Additional information is required when plotting vectors, to indicate the direction. For two-dimensional plots using `plot`, gnuplot requires four columns: the first two give the position of the start of each arrow, and the last two give the offset (relative to the start) of the arrow's end. In other words, the columns must represent x, y, dx, and dy, and gnuplot shows the data as arrows from (x, y) to (x+dx, y+dy). All quantities are interpreted as locations in the diagram's coordinates.

Figure F.5 shows an example. The data can be interpreted as the force on a test charge in the presence of three other charges (indicated by dots). It was created using the following:

```
set size square
unset key

set style arrow 100 size screen 0.01,10 lt 1

plot [-5.3:5.3][-5.3:5.3] "arrows" i 0 u 1:2:3:4 w vec as 100,
➡                         "" i 1 u 1:2 w p pt 7 ps 0.5 lc rgb 'black'
```

All the options to change the appearance of an arrow (see section 7.3.2) are available to you. You can provide them inline as part of the `plot` command or define your own custom arrow style, as I did here (see section 9.4.3).
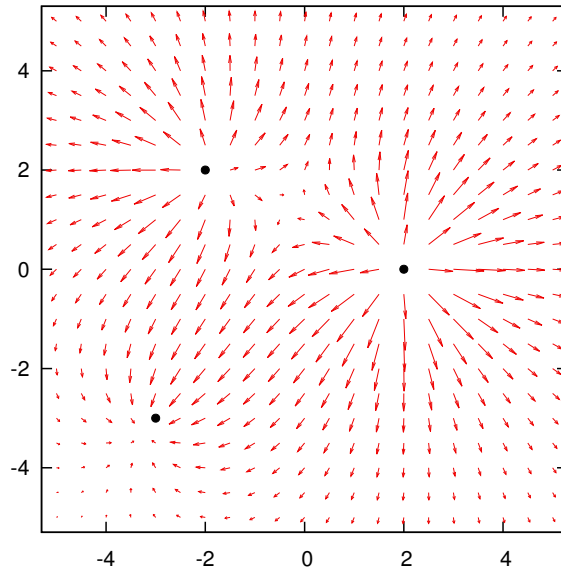
Figure F.5   A vector field, plotted using the `with vectors` style

Another possibility is to use data-dependent coloring (see section 9.1.5). Figure F.6 shows the same data set as figure F.5 but uses a fifth column (giving the relative field strength) to color the arrows. To map the latter value to a color, it's convenient to use a palette:

```
set size square
unset key
set palette defined ( -1 "blue", 0 "grey", 1 "red" )

set style arrow 100 size screen 0.02,10,90 filled fixed lc palette z

plot [-5.3:5.3][-5.3:5.3] "arrows" i 0 u 1:2:3:4:5 w vec as 100,
➡                         "" i 1 u 1:2 w p pt 7 ps 0.5 lc rgb 'black'
```
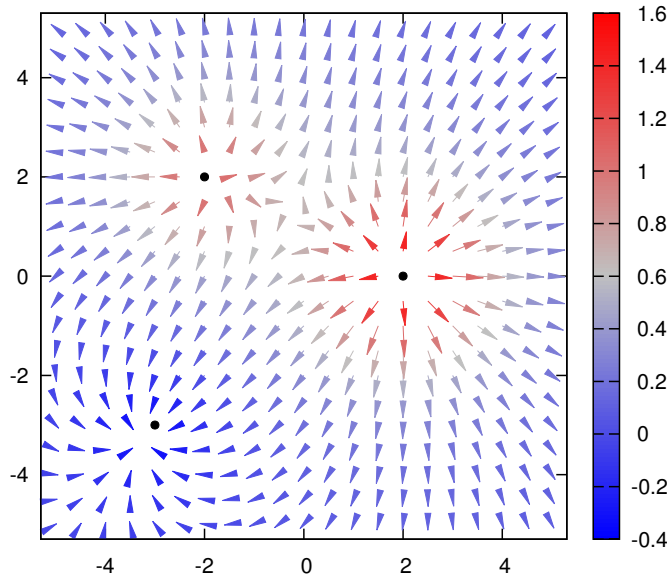
Figure F.6 **Same as figure F.5, but using data-dependent coloring to represent additional information in the graph**

F.3.2 *Three-dimensional vectors with splot*

Similar considerations apply to vectors in three dimensions, only now you need to supply *six* columns in total. In three dimensions (when using `splot`), the `with vectors` style can be used in creative ways to visualize three-dimensional objects. In particular, together with the mousing capabilities of interactive terminals (see section 10.6), this can be useful—a lot of visualization power that's usually only achieved with much more complicated programs.

A quick example will help to clarify. Listing F.4 shows a data file. The top part of the data file gives the coordinates of a pyramid-shaped body (a tetrahedron, to be precise). The bottom part of the file shows the edges of the pyramid. The labels A, B, C, and D at the corners help to identify the edges of the body.

Listing F.4   Data for figure F.7 (file: pyramid)

# x	y	z	color	label	
1	0	-0.7	1	"A"	
-1	0	-0.7	1	"B"	**End points**
0	1	0.7	2	"C"	
0	-1	0.7	2	"D"	

	# x1	y1	z1	x2	y2	z2		
	1	0	-0.7	-1	0	-0.7	⟵———	**A → B**
	-1	0	-0.7	0	1	0.7	⟵———	**B → C**
Edges	0	1	0.7	0	-1	0.7	⟵———	**C → D**
	0	-1	0.7	1	0	-0.7	⟵———	**D → A**
	1	0	-0.7	0	1	0.7	⟵———	**A → C**
	-1	0	-0.7	0	-1	0.7	⟵———	**B → D**

When attempting to plot this data, you need to remember that the `with vectors` style requires the starting points and the relative offsets of the end points, not the coordinates of the start and end points directly. The commands in listing F.5 use inline data transformations to convert the coordinates into offsets on the fly. Furthermore, this example uses some of the tricks you learned in sections 6.3.5 and 9.1.5 to attach a text label to each end point and to give them different colors. The resulting graph is shown in figure F.7. Try it yourself and then grab the figure with the mouse to rotate it. Have fun!

> **Listing F.5   Commands for figure F.7 using data from listing F.4 (file: pyramid.gp)**

```
unset border
unset tics
unset key
set view 200,80

splot [-1:1][-1:1][-1:1] "pyramid" i 0 u 1:2:3:4 w p pt 7 ps 3 lc var,
➡      "" i 1 u 1:2:3:($4-$1):($5-$2):($6-$3) with vectors nohead lt -1,
➡      "" i 0 u 1:2:3:5 w labels offset 1,1
```
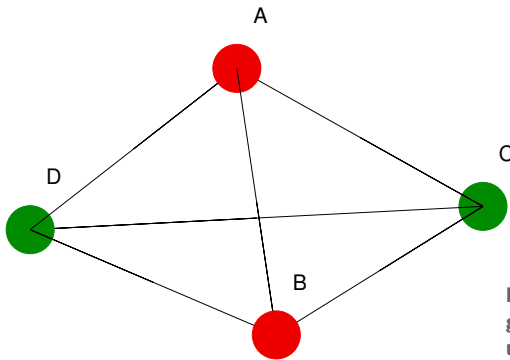

Figure F.7   The body whose coordinates were given in listing F.4. Load listing F.5 into gnuplot, using an interactive terminal, and then grab this figure with the mouse and try to rotate it!

F.4   *Built-in mathematical functions*

As mentioned in chapter 3, gnuplot provides a large number of standard mathematical functions as well as quite a few that are much less standard. Table F.1 gives the full list.[4]

Several of the more common functions can handle complex arguments (see section F.5) and return complex-valued return values if necessary. Other functions accept complex-valued arguments but *silently ignore* the imaginary part. Yet other functions require real (or integer) arguments. See table F.1 for the behavior of each function. Be aware that the behavior of the trigonometric functions depends on the current value of the `set angles` option (see section F.2.2 earlier in this appendix).

---

[4]   For non-elementary functions with ambiguous notations, I've given a pointer to the defining relationship in *Handbook of Mathematical Functions* by M. Abramowitz and I. Stegun (Dover Publications, 1965), quoted as A&S. (The full content of the book can be found on the web, for instance at www.nr.com/aands/ and www.convertit.com.)

Additional functions may be available if gnuplot has been linked against the appropriate libraries. Check the release notes for your installation. (To find the list of available mathematical functions in the gnuplot standard reference documentation, use `help expressions functions`. The command `help functions` pulls up information on user-defined functions.)

Table F.1 All built-in mathematical functions

Function	Arguments	Description
`sqrt(x)`	Any	Square root function
`exp(x)`	Any	Exponential function
`log(x)`, `log10(x)`	Any	Logarithm to base *e* and to base 10
`sin(x)`, `cos(x)`, `tan(x)`	Any	Trigonometric functions. Arguments interpreted according to `set angles`.
`asin(x)`, `acos(x)`, `atan(x)`	Any	Inverse trigonometric functions. Return values depend on the value of `set angles`.
`atan2(y, x)`	Real	Evaluates to `atan(y/x)`, but uses the signs of both arguments to determine the resulting quadrant. Return value depends on the value of `set angles`.
`sinh(x)`, `cosh(x)`, `tanh(x)`	Any	Hyperbolic functions
`asinh(x)`, `acosh(x)`, `atanh(x)`	Any	Inverse hyperbolic functions
`erf(x)`	Any[a]	Error function (A&S 7.1.1)
`erfc(x)`	Any[a]	Complementary error function (A&S 7.1.2): `erfc(x) = 1 - erf(x)`
`inverf(x)`	Any[a]	Inverse error function
`invnorm(x)`	Any[a]	Inverse normal distribution function
`norm(x)`	Any[a]	Normal (Gaussian) distribution function (A&S 26.2.2): `norm(x) = 0.5*(1+erf(x/sqrt(2)))`
`gamma(x)`	Any[a]	Gamma function (A&S 6.1.1).
`ibeta(p, q, x)`	Any[a]	Incomplete beta function (A&S 6.6.1)
`igamma(a, x)`	Any[a]	Incomplete gamma function (A&S 6.5.1)
`lgamma(x)`	Any[a]	Natural logarithm of the gamma function: `lgamma(x) = log( gamma(x) )`

[a] For complex-valued arguments, the function ignores the imaginary part of the argument.

Table F.1   All built-in mathematical functions *(continued)*

Function	Arguments	Description
`besj0(x)`, `besj1(x)`	Real	Bessel function of the first kind of order 0 and 1. Argument is always interpreted as radians.
`besy0(x)`, `besy1(x)`	Real	Bessel function of the second kind of order 0 and 1. Argument is always interpreted as radians.
`airy(x)`	Any[a]	Airy function (A&S 10.4.2)
`EllipticK(x)`, `EllipticE(x)`	Real	Complete elliptic integral of the first and second kind (A&S 17.3.1 and 17.3.3)
`EllipticPi(n,x)`	Real	Complete elliptic integral of the third kind (A&S 17.7.2)
`expint(n, x)`	*n* integer, *x* real	Exponential integral (A&S 5.1.4)
`lambertw(x)`	Real	Lambert-W function (the inverse of $f(w) = w \exp(w)$
`abs(x)`	Any	Absolute value for real or complex *x*
`floor(x)`, `ceil(x)`	Any[a]	Smallest integer not less than *x*, and largest integer not greater than *x* (both as floating-point values)
`int(x)`	Any[a]	Integer part of *x*, truncated to zero
`rand(x)`	Any[a]	Random number generator (see table 3.3)
`sgn(x)`	Any[a]	Sign function (-1 if $x < 0$; 0 if $x = 0$; 1 if $x > 0$)

[a] For complex-valued arguments, the function ignores the imaginary part of the argument.

F.5   *Complex numbers*

Gnuplot has limited support for complex numbers, so let's take a quick look. (If you don't know what complex numbers are, you probably want to skip this section.)

Complex numbers are pairs of numbers, the so-called *real* and *imaginary* parts. In gnuplot, a complex number is indicated using curly braces (`{}`). The following expression would be interpreted as the complex number $z = 1 + i$, where i is the imaginary unit:

```
z = { 1, 1 }
```

Gnuplot can perform simple arithmetic on complex numbers, such as `{ 1, 1 } + { -1, 0 }`. Furthermore, several of the built-in mathematical functions (such as `sin(x)`, `exp(x)`, and so forth) can accept complex arguments and return complex numbers as results.

You can use the special functions `real(x)` and `imag(x)` to pick out the real and imaginary parts, respectively (see table F.2).

One important limitation of gnuplot's complex numbers is that both parts must be numeric *constants*—not variables, not expressions! You can always work around this limitation, though, by using a complex constant as part of a more general expression. For example, the following command will plot real and imaginary parts of the exponential function, evaluated for a purely imaginary argument:

```
plot real( exp(x*{0,1}) ), imag( exp(x*{0,1}) )
```

Complex numbers are of fundamental importance in mathematics and theoretical physics, and they have important applications in signal processing and control theory. Gnuplot's ability to handle them makes it particularly suitable for such applications.

Table F.2 Functions to access parts of complex numbers

Function	Description
`abs(x)`	Absolute value of *x*: `sqrt( real(x)**2 + imag(x)**2 )`
`arg(x)`	Phase angle of *x*: `atan( imag(x)/real(x) )`
`imag(x)`	Imaginary part of *x*
`real(x)`	Real part of *x*

F.5.1 *Application: Mandelbrot set (pure gnuplot)*

Gnuplot's support for complex numbers makes it possible to write a program that calculates and visualizes the Mandelbrot set *entirely in gnuplot*.[5] Listing F.6 is short and sweet. The core of the script is the function `f(z,c,k)` that implements the iteration that defines the Mandelbrot set. The function is *recursive* (that is, it calls itself): because the body of a gnuplot function must be an expression, that's the only way to define an iteration inside a function. The function takes three arguments: the recursion variable, the starting point for the iteration (which is kept constant during the recursion), and the depth of the recursion. The function returns the number of iterations before the absolute value of the recursion variable (the first argument) exceeds 2, unless the maximum number of iterations `MAXITR` is reached first. This function is evaluated on a regular grid using the pseudofile ++ (see section 4.5). Remember to use the same granularity for both `set samples` and `set isosamples` when using ++ (see section C.1.2). The results are shown in figure F.8. (Depending on the speed of your computer, the figure may take a few seconds to render.)

---

[5] The Wikipedia page on the Mandelbrot set is quite detailed and informative.

Listing F.6   Commands for figure F.8 (file: mandelbrot.gp)

```
set size sq; set view map

set samples 500; set isosamples 500

MAXITR = 100
f(z,c,k) = abs(z)>2 || k>=MAXITR ? k : f(z**2+c,c,k+1)

set palette defined
➥( 0 'white', 5 'blue', 15 'red', 50 'yellow', 99 'yellow', 100 'black' )

splot [-1.5:0.5][-1:1] "++" u 1:2:( f({0,0}, $1+$2*{0,1}, 0) ) w pm3d
```
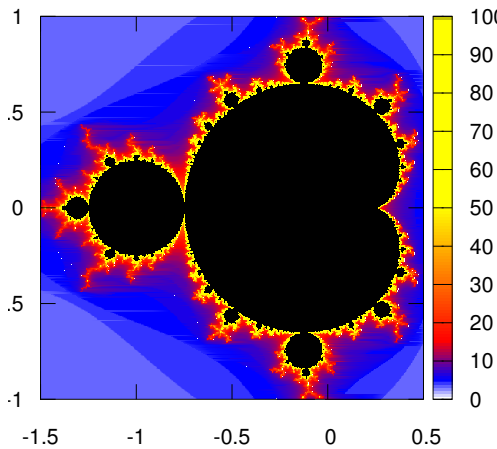

Figure F.8   **The Mandelbrot set, rendered *and calculated* in gnuplot. See listing F.6 for the complete implementation.**

F.6   *Probability plots*

A *probability plot* (or, more precisely, a "normal" probability plot) is a graphical technique to determine whether a bunch of values are distributed according to a normal (Gaussian) distribution. From such a plot, it is also possible to obtain the two parameters of the Gaussian distribution (mean and standard deviation) by purely graphical means. Gnuplot doesn't have probability plots built in, but it does provide all the ingredients necessary for you to roll your own.

To understand what a normal probability plot does, consider the cumulative distribution function of a set of points. If the data points were drawn from a Gaussian distribution, then their empirical cumulative distribution function would have the shape of the Gaussian cumulative distribution function. Applying the *inverse* Gaussian distribution function to the empirical distribution function of the set of measurements should therefore result in a straight line. Because deviations from a straight line are easy to spot, this technique provides an easy, graphical way to test a data set for Gaussian behavior.

Gnuplot can calculate the cumulative distribution function of a set of points using smooth cumul in the plot command. The inverse Gaussian distribution function is

available as the built-in function `invnorm(x)`. To create a probability plot in gnuplot, you need to employ these facilities in sequence: first, obtain the empirical distribution function using `plot ... smooth cumul`. The results aren't plotted right away; instead, they're stored in a heredoc via the `set table` (see section 5.3.2) facility. Then apply the inverse Gaussian distribution function to the data in the heredoc and plot the results.

Figure F.9 shows the probability plots for two data sets: one that's Gaussian and hence forms a straight line, and one that's clearly not Gaussian. (It's truncated on the left and somewhat heavy-tailed on the right.) Listing F.7 summarizes the basic sequence of steps to create a probability plot for a single data set. This code attempts to be generic, so you must configure the name of the file and the column to use in the first lines of the listing.
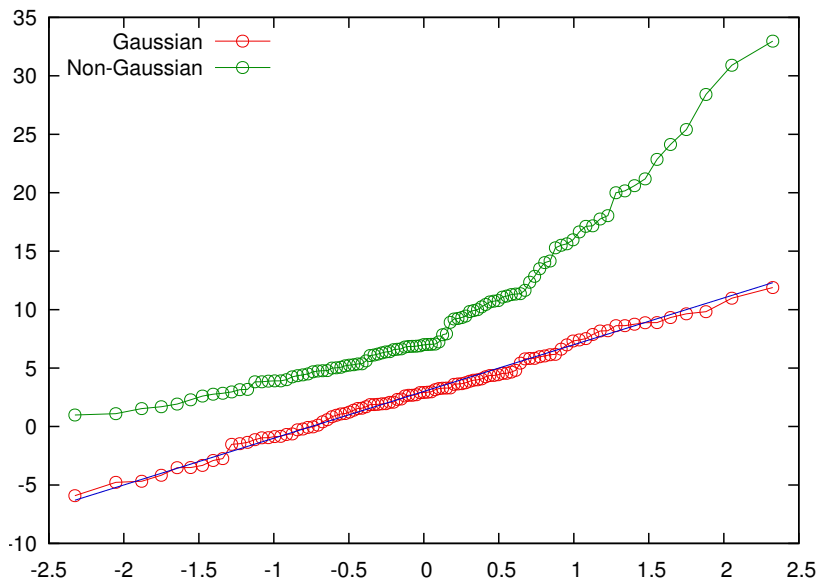
Figure F.9   Normal probability plots for two data sets: one that follows a Gaussian distribution, the other showing non-Gaussian behavior

Listing F.7   Commands to generate a probability plot (file: probplot1.gp)

```
FILE = "probplot"                          ◁──────  Use this to configure the data file …
COL  = 1                                   ◁──────  … and which column to use in the file.

stats FILE u COL nooutput                       ◁──────  Populates STATS_records

set table $tmp
plot FILE u COL:(1./(STATS_records+1)) s cumul       Stores the cumulative distribution
unset table                                          function in a heredoc

plot $tmp u (invnorm($2)):1 w l            ◁──────  Creates the probability plot
```

F.6.1   *Adding a probability scale*

You can improve the graph by adding a second axis, with divisions in *probability units* (probits). This way, you also learn more about the ability to link two axes using the set link command introduced in section 8.1.3.

In a probability plot, the observed values are plotted along the vertical axis, whereas the distribution of points is plotted along the horizontal axis, measured in standard deviations. Instead of measuring the distribution of points along the horizontal axis in standard deviations, you can also express the distribution in fractions of the entire data set (see figure F.10). The meaning of this scale is possibly more intuitive, but it comes at the cost of a non-uniform division of the axis.
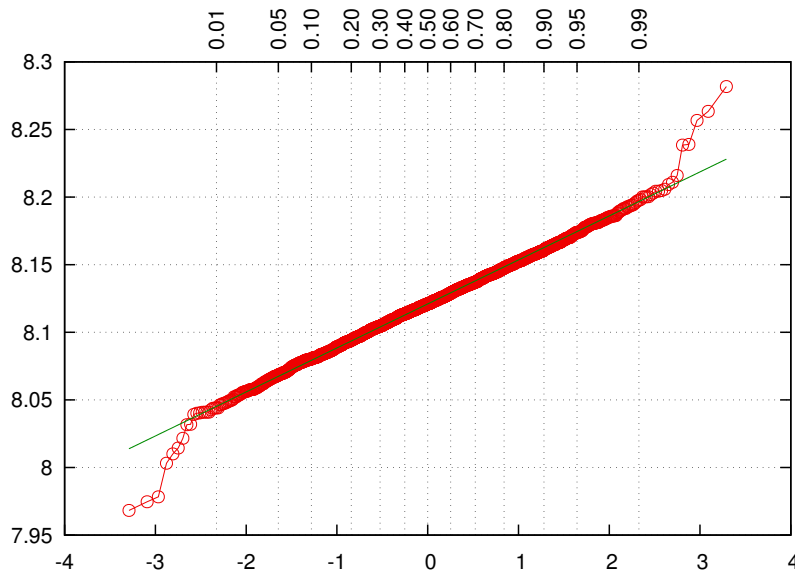
Figure F.10   A normal probability plot, with secondary tic marks and grid lines added. (See listing F.8.)

The complete code is shown in listing F.8. Two things are new, compared to the previous example. First, the secondary axis (x2) is linked to the primary one using

```
set xrange [-1:1]
set link x2 via norm(x) inverse invnorm(x)
set xrange [*:*]
```

I've explained the link command before (section 8.1.3). The two calls to set xrange are necessary to suppress some gnuplot warnings: to validate its input, the set link command tests that the two functions supplied are actually inverses of each other, by

evaluating them at the ends of the current plot range. The problem is that the functions used here encounter numerical precision loss at the extreme values of the default plot range (which is `[-10:10]`). Restricting the plot range first to a safe interval, and then turning on autoscaling again afterward, suppresses the warnings.

The `link` command by itself doesn't switch on the tic marks on the secondary axis; you have to do so yourself. Because the tic marks are distributed in such a non-uniform manner, it's necessary to add some tic marks for extreme values explicitly, using `set x2tic add ( ... )`.

The other new feature in listing F.8 is the second call to `stats`, this time not on the data set, but on the heredoc. The linear regression results from the `stats` command are used as a model for the data and combined in the probability plot together with the data (see figure F.10).

Listing F.8   Commands for figure F.10 (file: probplot2.gp)

```
FILE = "probplot"                          ⟵——— Configures the data file …
COL  = 1                                   ⟵——— … and column

stats FILE u COL nooutput                  ⟵——— Populates STATS_records

set table $tmp                             ┃ Stores the cumulative distribution
plot FILE u COL:(1./(STATS_records+1)) s cumul   ┃ function in a heredoc
unset table

set xrange [-1:1]                          ┃
set link x2 via norm(x) inverse invnorm(x) ┃ Links the two axes
set xrange [*:*]                           ┃

set xtics nomirror                                    Fixes the appearance
set x2tics format "%.2f" rotate                         of the tic labels
set x2tics add ( "0.01" 0.01, "0.05" 0.05, "0.95" 0.95, "0.99" 0.99 )

set grid x2tics ytics                                 Performs a linear
                                                      regression, and …
stats $tmp u (invnorm($2)):1 noout         ⟵
print "mu=", STATS_intercept, " sigma=", STATS_slope  ⟵— … prints the results

plot $tmp u (invnorm($2)):1 w l, STATS_slope*x + STATS_intercept
```

The data set shown in figure F.10 is much larger than the one in figure F.9 (2,000 data points now versus 100 data points before). It's mostly Gaussian, except for a handful of outliers. Using the newly added axis, you can conclude that more than $0.99-0.01=0.98$ or 98% of the points are normally distributed. The number of true outliers isn't large (about 12). Notice that their values aren't *massively* different from the rest of the data points, but sufficiently different that in a data set of 2,000 points, you wouldn't expect to find such values—*if* all the values were drawn from a Gaussian distribution, that is. The probability plot helps to identify the outliers and their range of values.

F.7   Curve fitting

Gnuplot includes a facility to perform *nonlinear least-square fits*. This subject is a little outside gnuplot's primary area of applicability (which is to generate graphs), but it's a popular part of gnuplot's feature set.

Gnuplot's fitting subsystem consists of two commands: the `fit` command and the `update` command that's used to export fitted parameter values. Many aspects of the fitting algorithm can be modified using `set fit`.[6]

F.7.1   Background

Before describing how to use the `fit` command and the associated options, let me explain the context and define some specific terms. Assume that you have a set of N data points (x_i, y_i) and a function $f(x; a_1, …, a_m)$ of *known functional form*. Note that the function doesn't just depend on the independent variable x, but also on m parameters a_1 through a_m. Fitting this function to the data means finding those values for $(a_1, …, a_m)$ that optimize a certain *merit function*—a function measuring how well the data is represented by the function. The most common merit function is the sum of the squares of the deviations (or *residuals*):

$$E_0^2 \;=\; \sum_i^N (y_i - f(x_i; a_1, ..., a_m))^2$$

I'll refer to it as the *squared error* E_0^2; the gnuplot reference documentation calls it *SSR*, for "sum of squared residuals." You can assign a weight to each term in this sum, such that data points that are known more accurately contribute more strongly. Let's write the weight for the ith term as $1/\delta_i$, where δ_i is a measure for the uncertainty (such as the standard deviation) of that term. The result is the *weighted least-squares* merit function (which I denote by E^2, and which is called *WSSR* for "weighted sum of squared residuals" in the gnuplot reference documentation):

$$E^2 \;=\; \sum_i^N \left(\frac{y_i - f(x_i; a_1, ..., a_m)}{\delta_i} \right)^2$$

The optimal choice of parameters (in a least-squares sense) minimizes this value.[7]

If the function $f(x; a_1, …, a_m)$ is *linear* in the independent variable—that is, if $f(x; a, b) = ax + b$, then the minimization of the error with respect to the parameters a and b can be performed analytically, so that expressions for the optimal values for

---

[6] In previous versions of gnuplot, the aspects now bundled in `set fit` were controlled through a variety of internal variables (and two environment variables) that are now deprecated.

[7] Although the sum of squared residuals is by far the most commonly used merit function, it's not the only possible choice. It can be justified on statistical grounds when the noise in the data has certain properties. It has the technical advantage of being simple and analytic everywhere. A disadvantage is that it tends to overemphasize outliers.

these parameters can be obtained in closed form (*linear regression*). If the function isn't linear in x but is linear in the parameters—for example, $f(x; a, b) = a \sin(x) + b \cos(x)$—a similarly simple relation holds. But if the function is *not* linear in the parameters—for example, $f(x; a, b) = \sin(a\,x)/\text{sqrt}(x+b)$—there's generally no solution to the minimization problem in closed form, and you have to resort to an *iterative* approximation scheme.

In this case, the merit function is minimized using an iterative algorithm, in which different values for the parameters are tried repeatedly until the merit function reaches its optimal value. The standard numerical algorithm for this task, which takes into account the specific analytic form of the merit function as a sum of squares, is known as the *Marquardt-Levenberg* method. This is what gnuplot's `fit` command uses.

F.7.2 *A worked example*

Before describing the `fit` command and all its options in detail, let's see how it works in practice. Consider the data in figure F.11, showing the results of a scattering experiment. The horizontal axis (corresponding to the scattering angle and thus the energy) has been divided into 140 equal-sized bins, and the vertical axis gives the number of events registered in each bin. You can observe different peaks in this spectrum and would like to know their position, strength, and width.
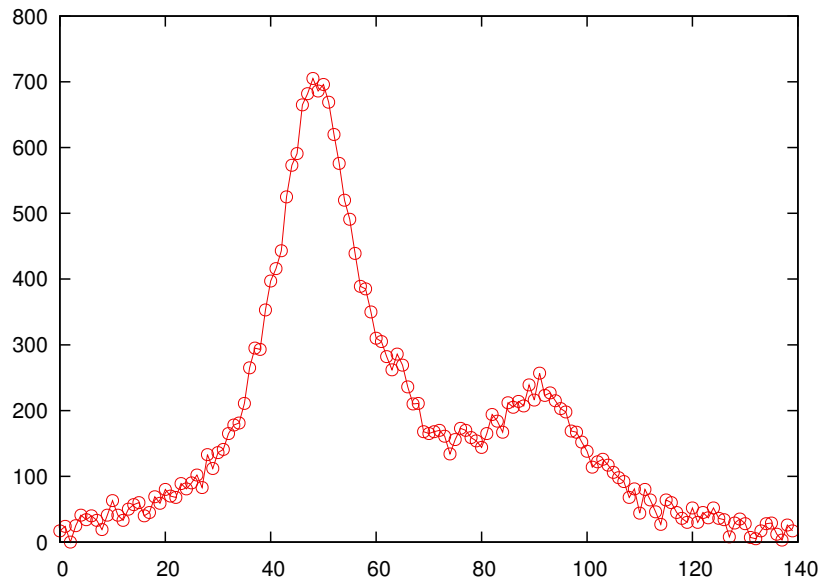

Figure F.11   Data from a scattering experiment. How many peaks are there, where are they located, and what's their width?

Because this data stems from a scattering experiment, you have *good reason to believe* that the shape of each peak is well described by a function of this form

$$c(x; m, s) = \frac{1}{1 + \left(\dfrac{x - m}{s}\right)^2}$$

and that the entire spectrum is a mixture of such functions, for proper values of the parameters. It's these parameters that you want to determine using `fit`.

Simply eyeballing figure F.11, two peaks can be identified at once: one near $x=50$ with a strength of about 700, and another near $x=90$ with a strength of about 250. (Experienced practitioners will point to the little shoulder on the right flank of the taller peak near $x=65$ and suggest the existence of a third peak, but we'll ignore this for now.) These initial guesses for the position and strength of the peaks must be provided as starting values for the iteration. The following listing shows the commands required to fit the two primary peaks, and figure F.12 shows a plot of the data together with the fitted function.

> ### Listing F.9   Results from `fit` for the data in figure F.11
>
> ```
> c(x,m,s) = 1/(1.0 + ((x-m)/s)**2)          ◁——— Defines the function
>
> set fit results                  ◁——— Suppresses most output
>
> a1=700; m1=50; s1=10             ◁——— Guess for the first peak
> a2=250; m2=90; s2=15             ◁——— Guess for the second peak
>
> fit a1*c(x,m1,s1) + a2*c(x,m2,s2) "bumps" u 1:2 via a1,m1,s1,a2,m2,s2
>
> After 4 iterations the fit converged.
> final sum of squares of residuals : 33965.5
> rel. change during last iteration : -7.50711e-06
>
> degrees of freedom    (FIT_NDF)                      : 134
> rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)  : 15.9209
> variance of residuals (reduced chisquare) = WSSR/ndf : 253.474
>
> Final set of parameters           Asymptotic Standard Error
> =======================           ==========================
> a1              = 682.455         +/- 5.853          (0.8577%)
> m1              = 48.8178         +/- 0.08384        (0.1717%)
> s1              = 9.5442          +/- 0.1334         (1.397%)
> a2              = 193.285         +/- 5.162          (2.671%)
> m2              = 90.5625         +/- 0.3414         (0.3769%)
> s2              = 12.3524         +/- 0.5457         (4.417%)
>
> ...                                        ◁——— Further output omitted
> plot "bumps" u 1:2 w lp pt 6, a1*c(x,m1,s1) + a2*c(x,m2,s2)
> ```

Overall, the function fits the data quite nicely, and you can read the position, strength, and width of the peaks from the fitted values. Comparing the data to the curve, the deviation near $x=65$ seems too systematic to be noise—you may want to repeat the process but include a third peak. What's its relative strength, compared to the two primary ones?
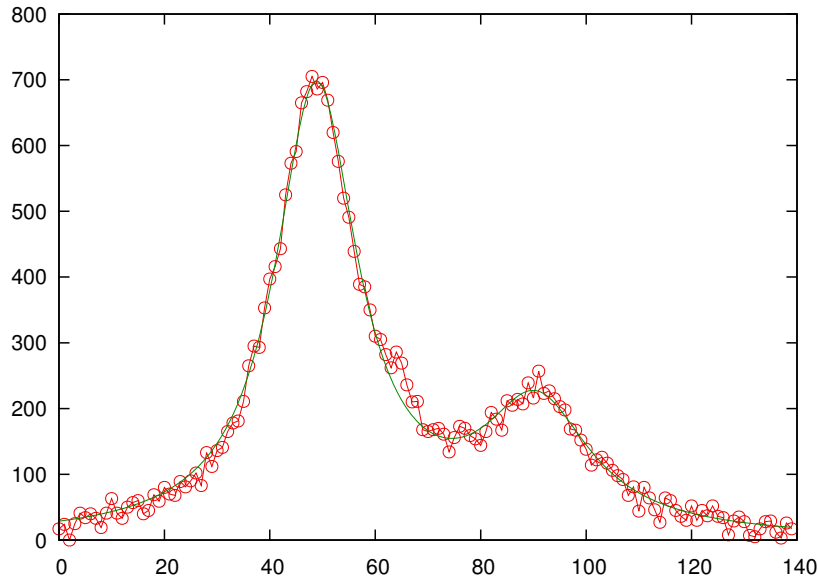
Figure F.12 The data from figure F.11 when fitting two peaks using the `fit` command. The agreement is quite good, but is there a third peak, centered near x=65?

Listing F.9 shows the basic steps of fitting data with gnuplot:

1 Define a function to fit. This isn't strictly required; you can use an explicit expression in the `fit` command, but in order to keep things neat, it's almost always worth the effort to define a function depending on the fitting parameters explicitly.

2 Choose starting values for all parameters. Again, this isn't *strictly* required, because gnuplot will initialize all parameters listed in the `via` clause to 1.0 before starting the iteration. But most realistic problems will need reasonable guesses for the starting values for the iteration to converge well.

3 Perform the fit.

4 Plot the data together with the fitted function. By the end of the fit, the resulting parameters are stored in the variables named in the `via` clause, thus overriding the starting values. This makes it possible and convenient to express the fitted function (in the `plot` command) in terms of these variables, rather than using the numerical values instead.

This concludes the process of "fitting" a function to the data, but the work doesn't stop here. The next step would be to subtract the fit from the data to obtain the residual. You might want to plot it and inspect it for trend and correlation (there shouldn't be either) and Gaussian distribution (validating assumptions required for interpretation of the results—a probability plot may come in handy).

F.7.3　Using the fit command

The syntax of the `fit` command is superficially similar to the `plot` command. The main difference is that *before* the name of the data file, you must specify the function to fit, and *at the end* of the command, you must name the fitting parameters using the keyword `via`:

```
fit [ {ranges} ]
    {function} "{str:datafile}" [ index ... ] [ every ... ]
                using {columnspec}
                via [ {var1}, {var2}, ... | "{str:parameterfile}" ]
```

The requirements for the format of the data file are the same as for the `plot` command. The `index` and `every` directives to select parts from the data file are available for `fit` as well, and so are on-the-fly transformations through the `using` option. The `using` clause specifies columns, as with `plot`. Plot ranges can be given inline, using the syntax familiar from the `plot` command. Any data point outside the plot range is ignored in the fitting procedure.

The function must depend on the dummy variable explicitly but otherwise can be any valid gnuplot expression. Usually, it's a user-defined function. The names of the parameters with respect to which the fit is performed are specified using the `via` directive. There are two variants: the names of the parameters can be given explicitly as part of the `fit` command or can be read from a file (more on this in a moment). At the end of the iteration, the variables are assigned their final values and therefore can be used directly in subsequent plots. (We made use of this in listing F.9.)

During the iteration, the `fit` command prints out progress and diagnostic messages to screen and to a log file (by default called fit.log). The file is always *appended* to, so you may want to delete it every once in a while. You can control the verbosity of these messages as well as the filename using `set fit` (see section F.7.5).

Specifying columns for data and weighting factors

When used with `fit`, the `using` directive can specify one to three columns: if only a single column is given, it's assumed to be the y value, and the line number is used as the x value. Two columns are interpreted as (x, y) pairs. A third column is interpreted as the *weight* for the corresponding value and is used to form a weighted least squares merit function: if the value in the third column is δ, then each term in the sum of squares is multiplied by $1/\delta^2$).

In recent versions of gnuplot, it's also possible to fit functions of more than one independent variable: in other words, rather than fit a *curve* to a set of points, you can fit a surface (or hyper-surface). Obviously, this requires additional columns to be included in the `using` clause. Check the gnuplot standard reference for details of the syntax.

MANAGING FITTING PARAMETERS

Nonlinear least squares require an iterative algorithm, and therefore it's necessary to initialize all parameters with good starting values. With badly chosen starting values, the procedure may require many steps to converge, or it may diverge altogether.

If the variable names are listed explicitly in the `via` clause, then you should initialize the corresponding session variables with suitable starting values. Any parameter that hasn't been defined is set equal to 1.

It's also possible to specify the parameters and their starting values in a file. This method is intended for batch operations, when you want to fit several data sets and need to give different starting values to each. The parameter file may contain starting values for all parameters, using the following syntax:

```
varname = value
varname = value    # FIXED
```

The comment on the second line forces gnuplot to keep this value fixed during the iteration; that is, the corresponding variable won't be adjusted when attempting to optimize the merit function.

If the list of parameters (and their starting values) has been read from a file, then the resulting values can be written out to a file using the `update` command. This command takes the name of the parameter file as mandatory first argument. If no second argument is given, the input file is renamed by appending .old, and the resulting values are written to a file with the original name. Alternatively, if a second filename is supplied, the resulting values are written to this file, and the original file is left untouched.

VARIABLES SET BY THE FIT COMMAND

After the iteration has converged (or has been interrupted), gnuplot sets several variables that can be read out to obtain the results of the fitting procedure:

- `FIT_CONVERGED`—A flag that equals 1 if the iteration converged or 0 if it was interrupted or exceeded the allowed number of steps.
- `FIT_NITER`—The number of iteration steps taken.
- `FIT_NDF`—The number of degrees of freedom for this fit. It equals the number of data points less the number of adjustable parameters and is usually denoted as ν: $\nu = N - m$.
- `FIT_WSSR`—The weighted sum of squares residual, evaluated at the point to which the iteration converged: E^2 (as defined earlier).
- `FIT_STDFIT`—sqrt(E^2/ν).
- `FIT_P`—The p-value, calculated from the chi-square distribution, given the number of degrees of freedom and the final weighted sum of squares.

Depending on the settings chosen by `set fit`, gnuplot may also create session variables holding the uncertainties in the estimated parameter values and for the entries in the covariance matrix. (See section F.7.5.)

F.7.4   *Practical advice*

Like any other nondeterministic, iterative calculation, nonlinear least-squares fitting isn't guaranteed to converge (and even less guaranteed to converge to the "desired" solution.) The following considerations may be helpful if you encounter problems.

Most problems with convergence can be solved with a better choice of starting values. Spend some time trying to fit the curve "by eye" before calling fit, manually trying out different values for the fitting parameters. (This is *much* more useful than fiddling with the iteration limits and lambda parameters accessible through set fit.)

If you need to fit a function with many parameters, try to reduce the complexity by getting the dominant behavior right before dealing with smaller effects. (In section F.7.2, we fitted only the two obvious peaks first, before asking whether there might be secondary peaks present as well.)

All fitting parameters should be of roughly equal scale. If some of the parameters differ wildly (by many orders of magnitude) from one another, the fitting function should be modified to take these factors into account explicitly. (Say you want to fit $f(x; a, b) = a \cos(b\, x)$, and you know that a is close to 1,000, whereas b is close to 1. You can either divide the data by 1,000 or write the function as $f(x; a, b) = 1000\, a \cos(b\, x)$ instead.) You can use the set fit prescale option to let gnuplot do some of the work for you.

Finally, the iteration problem becomes mathematically badly conditioned if the fitting function contains several parameters that have similar effects on the behavior of the function. A blatant example is $f(x; a, b) = a \exp(x+b)$, which despite its appearance has only *one* adjustable parameter, because $a \exp(x+b) = (a \exp(b)) \exp(x) = c \exp(x)$. For more complicated models, this kind of cross-correlation may be hard to detect ahead of time.

F.7.5   *Options for the fit command*

You can control many aspects of the fitting operation through the set fit command.[8] Here's a summary:

```
set fit [ verbose | brief | results | quiet ]
        [ logfile "{str:filename}" | default ]

        [ maxiter {int:val} | default ]
        [ limit {flt:eps} | default ] [ limit_abs {flt:eps }
        [ start_lambda {flt:val} | default ]
        [ lambda_factor {flt:val} | default ]

        [ [no]prescale ]

        [ [no]errorvariables ] [ [no]covariancevariables ]
        [ [no]errorscaling ]

        [ script [ "{str:command} | default ] ]
```

---

[8]   The internal variables that were used for this purpose in previous versions of gnuplot have been deprecated.

Let's step through them:

- Gnuplot generates a diagnostic for each step in the iteration. Four levels of verbosity are available:
  - `verbose` prints the entire message.
  - `brief` prints a one-line message for each iteration step.
  - `results` suppresses messages for individual steps and only prints the results of the fit.
  - `quiet` switches off all diagnostic output.
- By default, diagnostic messages are written (appended) to a file called fit.log. The filename can be overridden using `set fit logfile`. If the path name configured identifies a *directory* (rather than a file), then a file named fit.log is created in that directory.[9]
- The maximum number of iterations can be controlled using `set fit maxiter`. A value of 0 or `default` allows for an infinite number of steps.
- The main convergence criterion is the size of the most recent reduction in the squared residual. If the relative change in the sum of squared residuals between successive iteration steps is less than the value specified with `set fit limit`, then the iteration stops. The default is 10^{-5}.

  It's also possible to define a limit for the absolute size of the correction step using `set fit limit_abs`. It defaults to 0.
- The Marquardt-Levenberg algorithm always attempts to take the largest possible step that will reduce E^2. A numerical factor (usually called λ or *lambda*) controls the size of the step taken and is itself modified by the algorithm as the iteration proceeds. The value defined using `set fit start_lambda` is the initial value of lambda at the start of each iteration step; the value of `set fit lambda_factor` is the factor by which lambda is varied when searching for the optimal step size. Controlling these variables can help to achieve convergence in difficult situations.[10]
- For best convergence, all the fitting parameters should be roughly of the same scale. If they aren't, you can either rewrite the problem (to take the differing scales into account explicitly) or try to use `set fit prescale`. If it's switched on, then every variable is scaled by its starting value before each iteration step.
- When `set fit errorvariables` is selected, gnuplot generates session variables that hold the estimates of the uncertainty in the fitted parameters. The names of these session variables are obtained by appending `_err` to the actual variable

---

[9] The current gnuplot development version also provides a `set fit nolog` option that suppresses the creation of a log file entirely.

[10] Keep in mind that the simplest and most effective way to influence the behavior of the fitting routines is to provide better starting values for the fit parameters. A reasonable selection of starting values will usually obviate the need to fiddle with the lambda values.

name. That is, the uncertainty in the estimate for a parameter called `a` is stored in a variable called `a_err`, and so on.

If `set fit covariancevariables` is switched on, then the entries in the covariance matrix are similarly made available. The resulting variable names begin with `FIT_COV_`, followed by the names of the two variables in question, separated by an underscore. For example, the covariance of two variables named `a` and `b` is in a variable called `FIT_COV_a_b`.

- The `set fit errorscaling` parameter concerns the way the accuracies of the parameter estimates are calculated. (Check the gnuplot standard reference for details.)
- Using `set fit script`, you can provide a gnuplot command to run when the iteration is interrupted using Ctrl-C. (The default is to run the `replot` command.)

For the sake of completeness, I'll mention that two of these options can be prepopulated through environment variables: `FIT_LOG` can give the name of a file used for diagnostic output, and `FIT_SCRIPT` can contain the name of a command file that's to be executed when the iteration is interrupted. Configurations made with `set fit` take precedence over those made using environment variables.

Should you do it?

To be honest, I'm not a great fan of least-squares fitting. Despite (or because of) its appeal and popularity, in my experience it's easily overused and frequently *mis*used. It's wrong to regard it as a "versatile general-purpose tool." In particular, *non-linear* fitting as described here has only some very specific application areas that it's suited for; for other purposes, better methods exist. Personally, I think it should be regarded more as a "last resort" than as a "first thing to try." Instead consider the following options:

What are the alternatives?

Depending on what you're trying to do, there may be easier or otherwise better methods to use.

If you're trying to find a smooth curve as a purely *graphical visualization* of a noisy set of points, then you should use a non-parametric approximation such as weighted splines (see section 3.5.1), rather than attempt to fit a function. By choosing the function to fit, you're making a *very* strong statement about the shape of the curve. You shouldn't make such a statement unless you have a reasonable justification for making this particular choice.

Also, keep in mind that your mathematical model may describe the dominant behavior over *some region* of the data set well, but may nevertheless be incorrect elsewhere. The fitting process, by contrast, attempts to minimize the merit function globally. If the model isn't applicable over some part of the data set, the values obtained from the fit may be less accurate than if the fit had been constrained to only that part of the original data that can be described accurately by the model. You should therefore be careful to perform fits only on those parts of the data set for which the model is good. Proceed with caution!

In a similar spirit, *outliers* can wreak havoc on results obtained using standard statistical methods. Both linear and nonlinear regression are affected.

If you're trying to *estimate parameter values*, you'll frequently find that better, more direct methods exist. Instead of "fitting" a Gaussian to a histogram of values, you can simply calculate the mean and standard deviation of the data set! Similar closed-form, maximum-likelihood estimators exist for many commonly used distributions.

Often, a *transformation* of the data will remove the need for a nonlinear fit. Single and double logarithmic transformations turn exponential and power-law behavior into linear problems, for example. If only the dominant or asymptotic behavior is of interest, you may want to concentrate on it alone and seek a transformation to linear form for it, rather than fitting a complicated model to the entire problem.

When does curve fitting make sense?

It can be appropriate to fit a phenomenological model to a data set, in order to obtain an analytical (yet still only phenomenological) description of the data. Although this looks innocuous, it still can lead to problems because the justification and origin of the model are so easily ignored. Typically, it won't be long before it'll be conveniently forgotten that the simple analytical model that was assumed for the fitting procedure has no theoretical justification, and all kinds of unwarranted conclusions will be drawn from it. Furthermore, the fit is only as good as the function chosen in the first place. Maybe a different choice would have given different (and better) results. If more or newer or better data becomes available, will somebody really remember to reevaluate the choice of model?

Nonlinear fitting can make sense if you want to find good estimates for parameter values and the nature of the problem is such that no simpler method will do. The difficulty here lies in the interpretation of the results—not so much for the parameter values themselves, but for the estimated uncertainty of these results. The interpretation of the reported "errors" isn't obvious and depends on conditions that aren't necessarily fulfilled in practice. This is even more true when you're trying to decide between different models based on some measure for the goodness of fit. The entire topic is complex and difficult, resting on subtle assumptions and producing results that aren't easily interpreted correctly. Beware of a false sense of security!

OPEN SOURCE/DATA VISUALIZATION

gnuplot IN ACTION Second Edition
Philipp K. Janert

Gnuplot is an open-source graphics program that helps you analyze, interpret, and present numerical data. Available for Unix, Mac, and Windows, it is well-maintained, mature, and totally free.

gnuplot in Action, Second Edition is a major revision of this authoritative guide for developers, engineers, and scientists. The book starts with a tutorial introduction, followed by a systematic overview of gnuplot's core features and full coverage of gnuplot's advanced capabilities. Experienced readers will appreciate the discussion of gnuplot 5's features, including new plot types, improved text and color handling, and support for interactive, web-based display formats. The book concludes with chapters on graphical effects and general techniques for understanding data with graphs. It includes four pages of color illustrations. 3D graphics, false-color plots, heatmaps, and multivariate visualizations are covered in chapter-length appendixes available in the eBook.

What's Inside
- Creating different types of graphs in detail
- Animations, scripting, batch operations
- Extensive discussion of terminals
- Updated to cover gnuplot version 5

No prior experience with gnuplot is required. This book concentrates on practical applications of gnuplot relevant to users of all levels.

Philipp K. Janert, PhD, is a programmer and scientist. He is the author of several books on data analysis and applied math and has been a gnuplot power user and developer for over 20 years.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/gnuplot-in-action-second-edition

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]

Free eBook
SEE INSERT

"The highly anticipated, updated version of my go-to-for-everything book on gnuplot."
—Ryan Balfanz, Shift Medical, Inc.

"The essential guide for newcomers and the definitive handbook for advanced users."
—Zoltán Vörös
University of Innsbruck

"Learn how to use gnuplot to convert meaningful data into attention-grabbing visualizations that communicate your message quickly and accurately."
—David Kerns
Rincon Research Corporation

"An accessible guide to gnuplot and best practices of everyday data visualization."
—Wesley R. Elsberry, PhD
RealPage, Inc.

54499

9 781633 430181

www.it-ebooks.info