Includes time-saving Command and Option Reference

Gnuplot IN ACTION

Understanding data with graphs

SAMPLE CHAPTER 2

Philipp K. Janert

FOREWORDS BY COLIN D. KELLEY AND THOMAS WILLIAMS

MANNING

Gnuplot in Action

by Philipp K. Janert

Chapter 2

brief contents

2 Essential gnuplot

This chapter covers

- Invoking gnuplot
- Plotting functions and data
- Saving and exporting

In this chapter, we introduce gnuplot's most important features: generating plots, saving them to a file, and exporting graphs to common graphics file formats. In the next chapter, we'll talk about data transformations and the organization of data sets. By the end of the next chapter, you'll know most of the commands you'll use on a day-to-day basis.

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

2.1 Simple plots

Since 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 we 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 we plot it.

2.1.1 *Invoking gnuplot and first plots*

Gnuplot is a *text-based* plotting program: we interact with it through command-line-like syntax, as opposed to manipulating graphs using the mouse in a WYSIWYG fashion. Gnuplot is also *interactive*: it provides a prompt at which we type our commands. When we 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:

```
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 will be interpreted as gnuplot commands until you issue an `exit` or `quit` command, or type an end-of-file (EOF) character, usually by hitting `Control-D`.

Probably the simplest plotting command we 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 graphical user interface (X11), this command opens a new window with the resulting graph, looking something like figure 2.1.

Please 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.

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

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

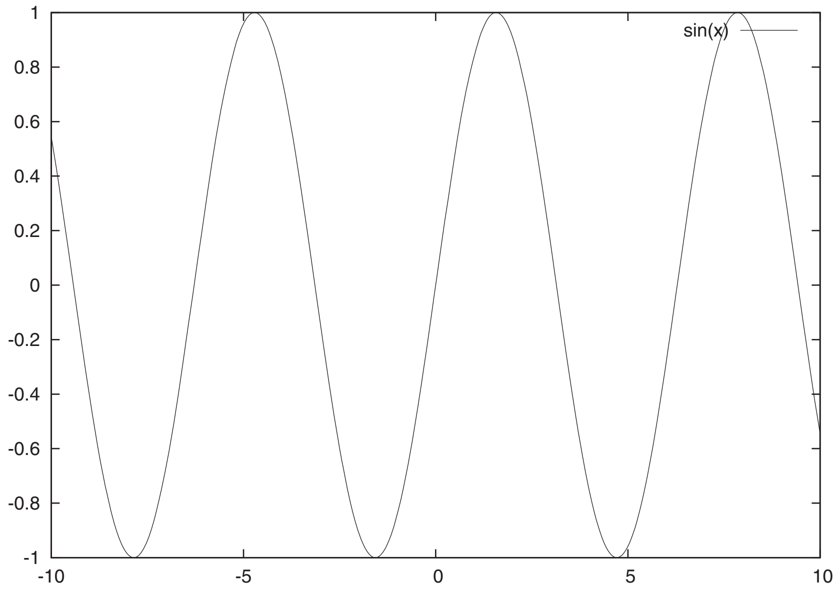

Figure 2.1 Our first plot: `plot sin(x)`

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.[1] (Gnuplot's syntax for mathematical expressions is straightforward and similar to the one found in almost any other programming language. Note the `**` exponentiation operator, familiar from Fortran or Perl. Appendix B has a table of all available operators and their precedences.) The resulting plot (see figure 2.2) is probably *not* what we expected.

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

```
plot [][-2:2] 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 (we leave it empty, since we'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.

[1] A Taylor expansion is a local approximation of an arbitrary, possibly quite complicated, function in terms of powers of *x*. We won't need this concept in the rest of this book. Check your favorite calculus book if you want to know more.

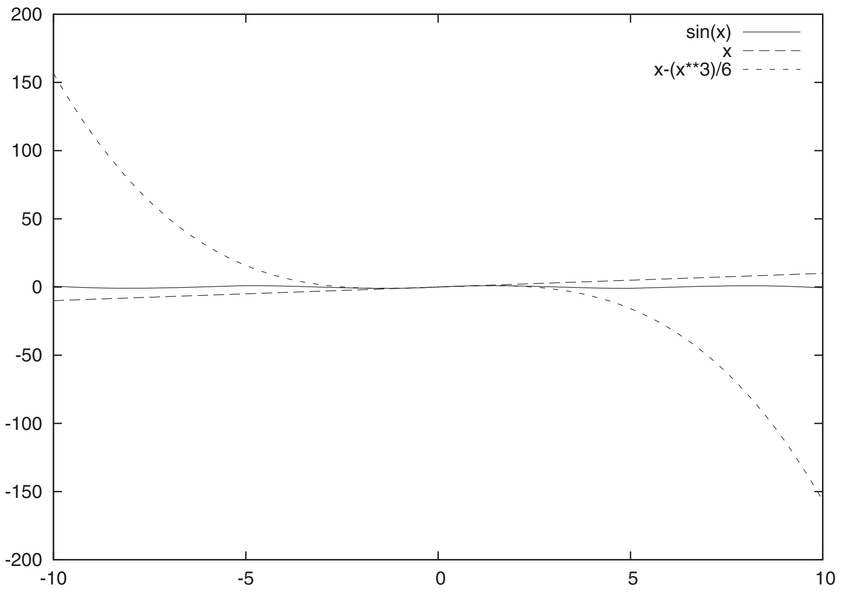

Figure 2.2 An unsuitable default plot range: `plot sin(x), x, x-(x**3)/6`

We can play much longer with function plots, zoning in on different regions of interest and trying out different functions (check the reference section in appendix B for a full list of available functions and operators), but instead let's move on and discuss what gnuplot is most useful for: plotting data from a file.

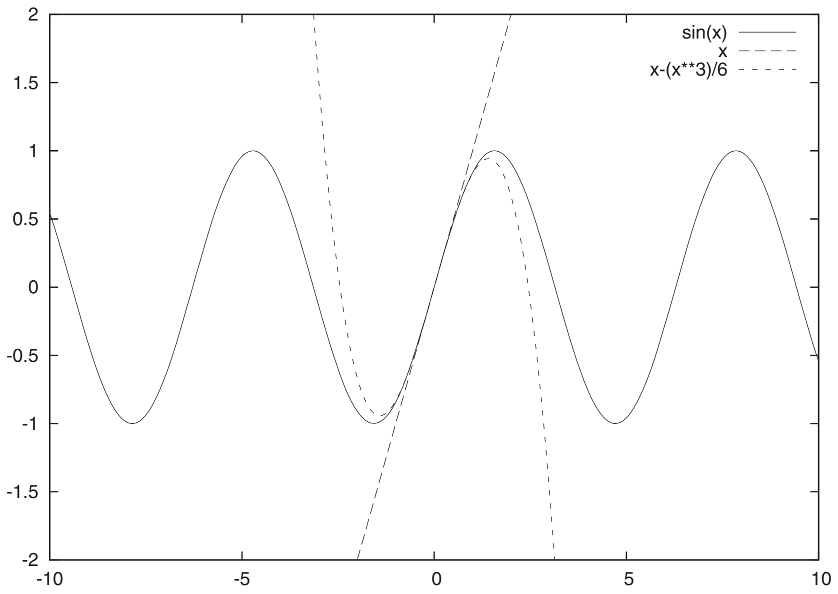

Figure 2.3 Using explicit plot ranges: `plot [][-2:2] sin(x), x, x-(x**3)/6`

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 hashmark (#) are considered to be comment lines and are ignored. Listing 2.1 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

```
# 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 are listed in subsequent columns. (We'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 in this way to begin with. In particular, the ability to keep related data sets in the same file is a big help (so that we don't need to keep PQR's stock price in a separate file from XYZ's, although we could if we wanted to).

While whitespace-separated numerical data is what gnuplot expects natively, recent versions of 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, and chapter 7 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, we can simply type

```
plot "prices"
```

Since data files typically contain many different data sets, we'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 will plot 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, while the second argument specifies the column for the vertical (y) axis. If we want to plot the price of XYZ shares in the same plot, we can do so easily (as in figure 2.4):

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

By default, data points from a file are plotted using unconnected symbols. Often this isn't what we want, so we need to tell gnuplot what *style* to use for the data. This is done using the `with` directive. Many different styles are available. Among the most useful ones 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
```

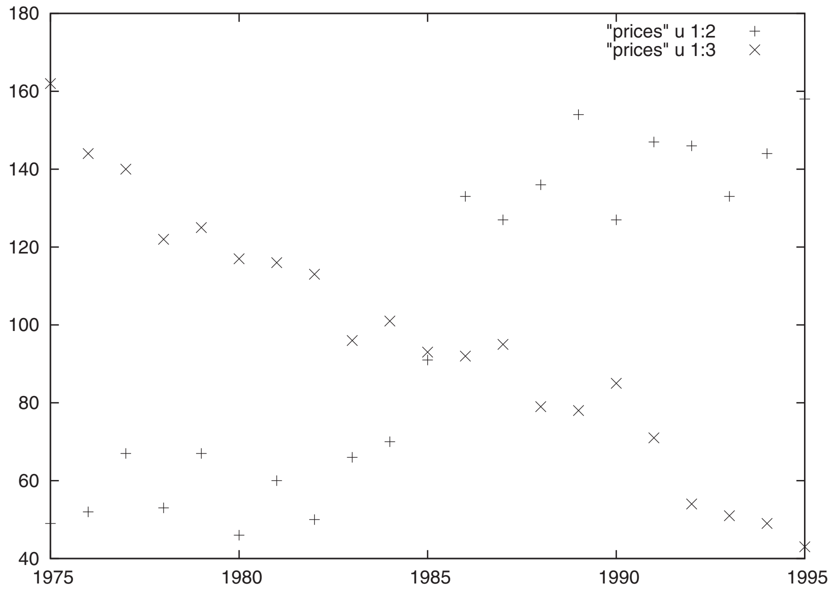

Figure 2.4 Plotting from a file: `plot "prices" using 1:2, "prices" using 1:3`

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 our case. We 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 (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 we must specify the data set to plot *first* and provide the description *second*: define it first, then describe what you defined.

Want to see how PQR's price correlates with XYZ's? No problem; just 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
```

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

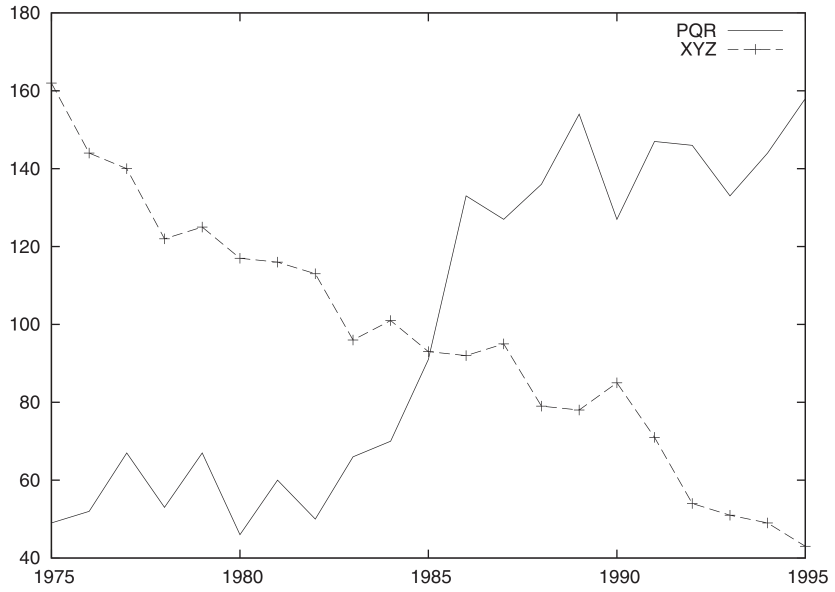

Figure 2.5 Introducing styles and the `title` keyword: `plot "prices" using 1:2 title "PQR" with lines, "prices" using 1:3 title "XYZ" with linespoints`

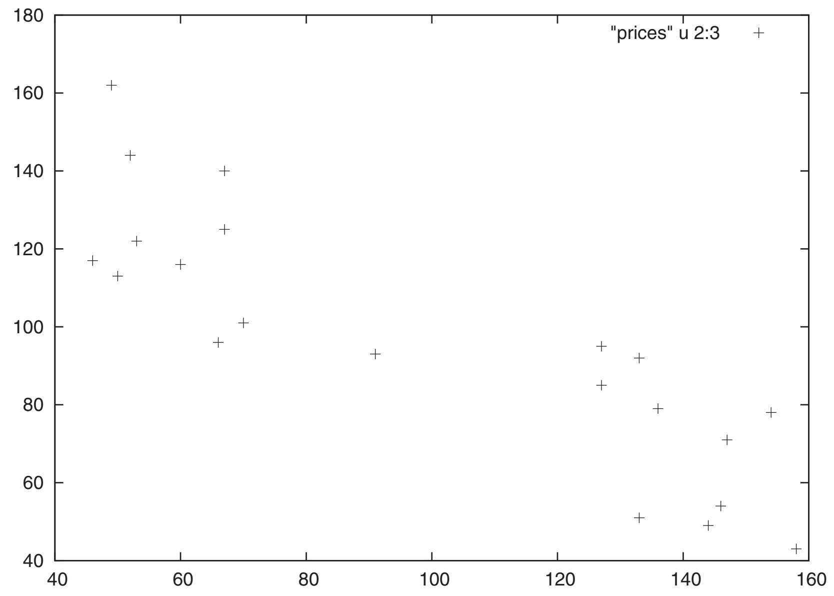

Figure 2.6 Any column can be used for either x or y axis: `plot "prices" using 2:3 with points`

A graph like figure 2.6 is known as a *scatter plot* and can show correlations between two data sets. In this graph, we can see a clear negative correlation: the better PQR is doing, the worse XYZ's stock price develops. We'll revisit scatter plots and their uses later in chapter 13.

Now that we'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 very good at encouraging iterative, exploratory data analysis. Whenever we 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.

There are two more features which gnuplot offers to the more proficient user: *abbreviations* and *sensible defaults*.

Any command and subcommand or option can be abbreviated to the shortest, nonambiguous form. So the command

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

would much more likely have been issued as

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

This compact style is very useful when doing interactive work and should be mastered. From here on, I'll increasingly start using it. (A list of the most frequently used abbreviations can be found in table 2 in the section on conventions in the front of the book.)

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

Whenever a filename is missing, the most recent filename is interpolated. We can use this to abbreviate the last 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) will remain unaffected until explicitly overridden by subsequent input. The way the filename is interpolated in the preceding example is a good example for this behavior. In later chapters, we'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 will use this column for y values and provide 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.

Let's close this section with a general comment regarding the syntax of gnuplot commands. Gnuplot syntax is mostly positional, not keyword oriented. This makes for compact commands, since the meaning of an abbreviation can be inferred from the position within the command. The price to pay is that occasionally subcommands that are expected earlier in the command need to be specified, even if we do not want to change their default settings. In this case, they are left blank. We've encountered this in the way empty brackets for the x range have to be supplied, even if we only want to change the y range, or in the way empty quotes indicate that the previous filename should be used again.

2.2 Saving and exporting

There are two ways to save our work in gnuplot: we can save the gnuplot commands used to generate a plot, so that we can regenerate the plot at a later time. Or we can export the actual graph to a file in one of a variety of supported graphics file formats, so that we can print it or include it in web pages, text documents, or presentations.

2.2.1 *Saving and loading commands*

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

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

This will save 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 actual `plot` command) at the gnuplot prompt.

An alternative to `load` is the `call` command, which is similar to `load`, but also takes up to 10 additional parameters after the filename to load. The parameters are available inside the loaded file in the variables `$0` through `$9`. Strings are passed without their enclosing quotes, and the special variable `$#` holds the number of parameters to `call`. We can use `call` to write some 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 (`;`). The hashmark (`#`) is interpreted as a comment character: the rest of the line following a hashmark is ignored. The hashmark 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.

Since 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, for instance to set up preferences or to imitate a limited macro capability (we'll give an example later in the chapter).

We'll discuss command files in more detail in chapter 12 on batch operations and user configurations.

2.2.2 *Exporting graphs*

As we've just seen, saving a set of plotting commands to a file is very simple. Unfortunately, exporting a graph in a file format suitable for printing is more complicated. It's not actually difficult, but unnecessarily cumbersome and prone to errors of omission. In this section, we'll first look at the steps required to export a printable graph from gnuplot; then we'll discuss the ways this process can go wrong. Finally, I'll show you a simple script that takes most of the pain out of the experience.

For any graph we want to generate (using gnuplot or anything else), we need to specify two things: the format of the graph (GIF, JPG, PNG, and so on) and the output device (either a file or the screen). In gnuplot, we do this using the `set` command:

```
set terminal png            # choose the file format
set output "mygraph.png"    # choose the output device
```

We'll discuss the `set` command in much more detail in chapter 4. For now, it's enough to understand that it sets a parameter (such as `terminal`) to a value. However, and this is often forgotten, it does *not* generate a plot! The only commands to do so are `plot`, `splot` (which is used for three-dimensional graphs, which we'll discuss in chapter 8), and `replot` (which simply repeats the most recent `plot` or `splot` command).

So, with this in mind, the complete sequence to export a graph from gnuplot and to resume working is shown in listing 2.2.

Listing 2.2 The complete workflow to generate a PNG file from gnuplot

```
plot exp(-x**2)             # some plot command
set terminal png            # select the file format
set output "graph.png"      # specify the output filename
replot                      # repeat the most recent plot command,
                            #   with the output now going to the
                            #   specified file.
set terminal x11            # restore the terminal settings
set output                  # send output to the screen again,
                            #   by using an empty filename.
```

This example demonstrates an important point: after exporting to a file, gnuplot does *not* automatically revert back to interactive mode—instead, all further output will be directed to the specified file. Therefore, we need to explicitly restore the interactive terminal (`x11` in this example) *and* the output device. (The command `set output` without an argument sends all output to the interactive device, usually the screen.) This should come as no surprise. As we've seen before, gnuplot remembers any previous settings, and so neither the `terminal` nor the `output` setting change until we explicitly assign them a different value.

Nevertheless, this behavior is rather different than what we've come to expect from user interfaces in most programs: we usually do not have to restore the interactive session explicitly after exporting to a file. It's also unexpected that three separate commands are required to generate a file (`set terminal`, `set output`, and `replot`), making it easy to forget one.

It's helpful to understand the technical and historical background for this particular design. Gnuplot was designed to be portable across many platforms, at a time (late 1980s!) when graphic capabilities were much less dependable than today. In fact, it wasn't even safe to assume that the computer had an interactive graphics terminal at all (only an attached hardware plotter, for example). So all graphics generation was encapsulated into the `terminal` abstraction. And since it wasn't safe to assume that every installation would have a graphics-capable interactive terminal as well as a plotter or a file-based output device, the same terminal abstraction was used for both the interactive session as well as the printable export, requiring you to switch between different modes in a way that seems so cumbersome today.

Nevertheless, what we really want most of the time is a simple export routine, which takes the name of a file to export to, as well as the desired file format, and does all the required steps in one fell swoop. In the next section, I show you how to build one yourself.

2.2.3 One-step export script

The multistep process we just described to generate printable graphics from gnuplot is clearly a nuisance. Luckily, we can use the `call` command introduced earlier to bundle all required steps into one handy *macro*.

The `call` command executes all commands in a single file. Therefore, we can put all commands required to generate (for example) a PNG file *and* to restore the gnuplot session back to its original state into a command file, which we can then invoke through a single `call` command. And because `call` can take arguments, we can even pass the name of the desired output file as part of the same command.

If the commands shown in listing 2.3 are placed into a file, this file can be executed using `call` and will write the most recent plot to a PNG file and restore the initial session to its original state.

Listing 2.3 A useful script to export the current plot to file

```
set terminal push   # save the current terminal settings
set terminal png    # change terminal to PNG
set output "$0"     # set the output filename to the first option
replot              # repeat the most recent plot command
set output          # restore output to interactive mode
set terminal pop    # restore the terminal
```

Here we've used the two pseudoterminals `push` and `pop` to help with the back-and-forth between the interactive and file terminals. The former (`push`) saves the current terminal settings onto a stack; the latter (`pop`) restores the terminal to the state saved on the stack. Neither makes any assumptions about the choice of interactive terminal, and therefore both can safely be used in scripts that must be portable across architectures.

Assuming the file shown in listing 2.3 was named export.gp in the current directory, we would call it like this, to write the current plot to a file called graph.png:

```
call "export.gp" "graph.png"
```

Here, both arguments are quoted. Quoting the second argument isn't strictly necessary, but highly recommended to avoid unexpected parsing of the output filename. The quotes are stripped before the argument is made available to the command file in the `$0` variable.

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 (such as fixing the typo in a label, or adding one more data set, or adjusting the plot range slightly) at a later time. This can only be done from the commands saved to file using `save`, not from plots exported to a graphics file. In chapter 11, section 11.1, I'll give an improved version of the export script which does both at the same time—that's how I generate all of my graphs.

2.3 Summary

In this chapter, we learned how to do the most important things with gnuplot: plotting, saving, and exporting. In detail, we discussed

- How to plot functions or data with the `plot` command: `plot sin(x)`
- How to restrict the plot range using bracket notation: `plot [0:5] sin(x)`
- How to select which columns from a data file to plot through `using`: `plot "data" using 1:2`
- How to save our work to file with the `save` command and how to load it again using `load`
- How to export a graph to a printable file format using `set output`, `set terminal`, and `replot`
- How to write simple scripts and use them through the `call` command

This means that we can do the three most important things for day-to-day work already: generate a plot, save it to file, and export it. In the next chapter, we'll learn about further things we can do with data in gnuplot: smoothing and filtering.

OPEN SOURCE/DATA VISUALIZATION

Gnuplot IN ACTION

Philipp K. Janert

FOREWORDS BY COLIN D. KELLEY AND THOMAS WILLIAMS

Free ebook
SEE INSERT

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, very mature, and ... totally free.

Gnuplot in Action is a comprehensive tutorial written for all gnuplot users: data analysts, computer professionals, scientists, researchers, and others. It shows how to apply gnuplot to data analysis problems. It gets into tricky and poorly documented areas. You'll quickly move from basic charts to advanced graphics, mastering powerful techniques like multi-dimensional and false-color plots. You'll also learn scripting techniques for unattended batch jobs or how to use gnuplot to generate web graphics on demand.

This book does not require programming skills, nor previous knowledge of gnuplot.

What's Inside

- Generate simple and complex graphics
- Graphic methods to understand data
- Scripting and advanced visualization

A programmer and data analyst, **Philipp K. Janert** has been a gnuplot power user for over 15 years, in business and academic environments. He holds a Ph.D. in theoretical physics.

For online access to the author and a free ebook for owners of this book, go to manning.com/GnuplotinAction

"Knee-deep in data? This is your guidebook to exploring it with gnuplot."
—Austin King
Senior Web Developer, Mozilla

"Sparkles with insight about visualization, image perception, and data exploration."
—Richard B. Kreckel, Hacker and Physicist, GiNaC.de

"Incredibly useful for beginners—indispensible for advanced users."
—Mark Pruett, Systems Architect Dominion

"Bridges the gap between gnuplot's reference manual and real-world problems."
—Mitchell Johnson
Software Developer, Border Stylo

"A Swiss Army knife for plotting data."
—Nishanth Sastry, Computer Laboratory, University of Cambridge/IBM

ISBN 13: 978-1-933988-39-9
ISBN 10: 1-933988-39-8
53499
9 781933 988399

MANNING

$34.99 / Can $43.99 [INCLUDING eBOOK]