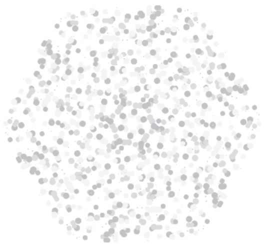
5 GRAPHICS



Graphs are friendly.

—Tukey (1977, p 157)

Graphics is one of the places where the computing world has not yet agreed on a standard, and so instead there are a dozen standards, including JPG, PNG, PDF, GIF, and many other TLAs. You may find yourself in front of a computer that readily handles everything, including quick displays to the screen, or you may find yourself logging in remotely to a command-line at the university's servers, that support only SVG graphics. Some journals insist on all graphics being in EPS format, and others require JPGs and GIFs.

The solution to the graphics portability problem is to use a handful of external programs, easily available via your package manager, that take in plain text and output an image in any of the panoply of graphics formats. The text-to-graphics programs here are as open and freely available as gcc, so you can be confident that your code will be portable to new computers.[1]

But this chapter is not just about a few plotting programs: it is about how you can control any text-driven program from within C. If you prefer to create graphics or do other portions of your analytic pipeline using a separate package (like one of the stats packages listed in the introduction), then you can use the techniques here to do so.

[1]There is some politics about how this is not strictly true: the maintainers of Gnuplot will not allow you to modify their code and then distribute the modified package independently (i.e., to *fork* the code base). The project is entirely unrelated to the GNU project, and the name is simply a compromise between the names that the two main authors preferred: nplot and llamaplot.

Appendix B takes a different approach to converting data to an executable script, via command-line tools to modify text. If your data is not going through any computations or transformations, you may be better off using a shell script as per Appendix B instead of writing a C program as per this chapter.

The plot-producing program with which this chapter will primarily be concerned is Gnuplot. Its language basically has only two verbs—`set` and `plot`—plus a few variants (`unset`, `replot`, et cetera). To give you a feel for the language, here is a typical Gnuplot script; you can see that it consists of a string of `set` commands to eliminate the legend, set the title, et cetera, and a final `plot` command to do the actual work.

```
unset key
set title "US national debt"
set term postscript color
set out 'debt.eps'
plot 'data-debt' with lines
```

In the code supplement, you will find the `plots` file, which provides many of the plots in this chapter in cut-and-paste-able form. You will also find the `data-debt` file plotted in this example. But first, here are a few words on the various means of sending these plot samples to Gnuplot.

PRELIMINARIES As with SQLite or mySQL, there is a command-line interpreter for Gnuplot's commands (`gnuplot`), so you can interactively try different `set` commands to see what options look best. After you have finished shopping for the best settings, you can put them into a script that will always produce the perfect plot, or better still, write a C program to autogenerate a script that will always produce the perfect plot.

There are various ways by which you can get Gnuplot to read commands from a script (e.g., a file named *`plotme`*).

- From the shell command line, run `gnuplot -persist <`*`plotme`*. This runs the script and exits, but the plot persists on the screen. Without the `-persist` option, the plot will disappear after a split second. If you are writing to a file rather than looking at the plot on screen, then the `-persist` option is unnecessary.
- Run `gnuplot` with no options, and from its prompt, type `load` *`'plotme'`*. This leaves you at the Gnuplot prompt to experiment with settings.
- The hybrid: `gnuplot` *`plotme`* `-` from the command line. This executes the instructions in *`plotme`*, but leaves you at the Gnuplot prompt to play with different settings.

`set term ...`	Meaning	Output goal
`x11`	Window system; most POSIX OSes	display on screen
`windows`	Window system; Windows OSes	display on screen
`aqua`	Window system; Mac OS X	display on screen
`png`	Portable network graphics	browser
`gif`	Graphics interchange format	browser, word processor
`svg`	scalable vector graphics	browser, word processor
`ps`	Postscript or encapsulated postscript	PDF
`latex`	LaTeX graphics sub-language	LaTeX docs

Table 5.1 Some of the terminal types that Gnuplot supports.

If you are at a Gnuplot prompt, you can exit via either the `exit` command or <ctrl-d>. On many systems, you can also interact with the plot, spinning 3-D plots with the mouse or zooming in to selected subregions of 2-D plots.

Q5.1 Check your Gnuplot installation. Write a one-line text file named `plotme` whose text reads: `plot sin(x)`. Execute the script using one of the above methods. Once that works, try the national debt example above.
If your system is unable to display plots to the screen, read on for alternative output formats.

`set term` **AND** `set out` Gnuplot defaults to putting plots on the screen, which is useful for looking at data, but is not necessarily useful for communicating with peers. Still worse, some systems are not even capable of screen output. There are many potential solutions to the problem.

The `set terminal` command dictates the language with which to write the output. Table 5.1 presents the more common options, and Gnuplot's help system describes the deatils regarding each of them; e.g., `help set term latex` or `help set term postscript`. The default is typically the on-screen format appropriate for your system.[2]

The `set out` command provides a file name to write to. For example, if a Gnuplot file has

```
set term postscript color
set out 'a_plot.eps'
```

[2]The popular JPG format is not listed because its compression is designed to work well with photographic images, and makes lines and text look fuzzy. Use it for plots and graphs only as a last resort.

at its head, then the script will not display to the screen and will instead write the designated file, in Postscript format.

Comments

Gnuplot follows the commenting standards of many scripting languages: everything on a line after a # is ignored. For example, if a script begins with

```
#set terminal postscript color
#set out 'printme.eps'
```

then these lines are ignored, and Gnuplot will display plots on the screen as usual. If you later decide to print the plot, you can delete the #s and the script will write to `printme.eps`.

You can rest assured that no matter where you are, there is some way to view graphics, but it may take some experimentation to find out how. For example, if you are dialing in to a remote server, you may be able to copy the graphics to a `public_html` directory, make the file publicly readable (`chmod 644` *`plot.png`*) and then view the plot from a web browser. Or, you could produce Postscript output, and then run `ps2pdf` to produe a PDF file, which you can open via your familiar PDF viewer.

Now that you know how to run a Gnuplot script and view its output, we can move on to what to put in the script.

5.1 `plot`

The `plot` command will set the basic shape of the plot.

To plot a basic scatterplot of the first two columns of data in a file, such as the `data-debt` file in the online code supplement, simply use `plot 'data-debt'`.

You will often dump more columns of data to your datafile than are necessary. For example, the first portion of the `data-debt` file includes three columns: the year, the debt, and the deficit. To plot only the first and third columns of data, use `plot 'datafile' using 1:3`. This produces a scatterplot with X values from column one and Y values from column 3. Notice that Gnuplot uses index numbering instead of offset numbering: the first column is one, not zero.

Say that you just want to see a single data series, maybe column three; then `plot 'datafile' using 3`. With only one column given, the plot will assume that the X values are the ordinal series $1, 2, 3, \ldots$ and your data are the Y values.

`replot` You will often want multiple data sets on the same plot, and Gnuplot does this easily using the `replot` command. Just change every use of `plot` after the first to `replot`. Page 170 presents a few more notes on using this function.

```
set xrange [−4:6]
plot sin(x)
replot cos(x)
replot log(x) + 2∗x − 0.5∗x∗∗2
```

- Gnuplot always understands the variable x to refer to the first axis and y to the second. If you are doing a parametric plot (see the example in Figure 11.4, page 360), then you will be using the variables t, u, and v.
- Gnuplot knows all of the functions in the standard C math library. For example, the above sequence will produce a set of pleasing curves. Notice that `x**2` is common math-package notation for x^2.

`splot` The `plot` command prints flat, 2-D plots. To plot 3-D surfaces, use `splot`. All of the above applies directly, but with three dimensions. If your data set has three columns, then you can plot it with `splot` *'datafile'*. If your data set has more than three columns, then specify the three you want with a form like `splot` *'datafile'* `using` *1:5:4*.

- There is also the crosstab-like case where the data's row and column dimensions represent the X and Y axes directly: the $(1, 1)\mathrm{st}$ element in the data file is the height at the Southwest corner of the plot, and the $(n, n)\mathrm{th}$ element is the height at the Northeast corner. To plot such data, use `splot` *'datafile'* `matrix`.
- Surface plotting goes hand-in-hand with the `pm3d` (palette-mapped 3-D) option, that produces a pleasing color-gradient surface. For example, here is a simple example that produces the sort of plot used in advertisements for math programs. Again, if you run this from the Gnuplot prompt and a system that supports it, you should be able to use the mouse to spin the plot.

```
set pm3d
splot sin(x) ∗ cos(y) with pm3d
```

Here is a more extended script, used to produce Figure 5.2. [This example appears in a slightly-modified form in `agentgrid.gnuplot` in the code supplement. The simulation that produced it is available upon request.]

```
set term postscript color;
set out 'plot.eps';
set pm3d; #for the contour map, use set pm3d map;
unset colorbox
set xlabel 'percent acting'; set ylabel 'value of emulation (n)';
```

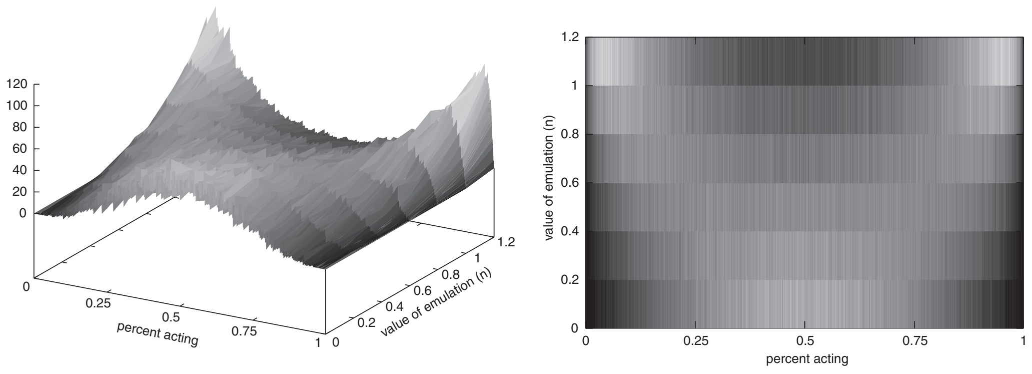


Figure 5.2 Two views of the same density plot from a few thousand simulations of a group of agents. Agents act iff $t_i + nk_i > C$, where t_i is a Normally distributed private preference, k_i is the percentage of other people acting, n is the agent's preference for emulating others, and C is a Uniformly-distributed cutoff. The vertical axis shows the density of simulations with the given percent acting and value of n. When the value of emulation is low, outcomes are unimodally distributed, but outcomes become bimodally distributed as emulation becomes more valuable.

```
6  set palette gray;
7  set xtics ('0' 0,'0.25' 250,'0.5' 500,'0.75' 750, '1' 999);
8  set ytics ('0' 0, '0.2' 1, '0.4' 2, '0.6' 3, '0.8' 4, '1' 5, '1.2' 6)
9  splot 'datafile' matrix with pm3d
```

- Lines one and two send the output to a Postscript file instead of the screen. Given line six, the `color` modifier is optional in this case.
- Line three tells Gnuplot to prepare for a palette-mapped surface, and is necessary before the `splot` command on line nine.
- Line four deletes the legend, which in the pm3d case is known as the colorbox.
- Line five sets the labels, and demonstrates that Gnuplot commands may put each either on a separate line or separated with a semicolon. As you can see from this script, ending a line with a semicolon is optional.
- Line six sets the color scheme to something appropriate for a book printed in black and white. There are many palettes to be had; the default, omitting the `set palette` line entirely, works for most purposes.
- Lines seven and eight fix the axis labels, because Gnuplot defaults to using the index of the column or row as the label. The format requires a text label, followed by the index at which the label will be placed; see below for further notes.
- Changing line three to `set pm3d map` produces an overhead view of the same surface, sometimes known as a *contour plot*, as in the second plot of Figure 5.2.

The short version The sample lines above were all relatively brief, but you can put a huge amount of information onto one line. For example,

```
set style data bars
set style function lines
set linetype 3
set xrange [−10:10]
set yrange [0:2]
plot 'data' using 1:3 title 'data'
replot sin(x) title 'sine'

# can be rewritten as:
plot 'data' using 1:3 [−10:10][0:2] with bars title 'data', sin(x) with lines linetype 3 title 'sine'

# or as
plot 'data' using 1:3 [−10:10][0:2] w bars title 'data', sin(x) w l lt 3 title 'sine'
```

All of these settings will be discussed below. The purpose of this example is to show that style information can be put above the plot command, or it can be mixed in on the line defining the `plot`. The `replot` command is also technically optional, because you can add additional steps on one `plot` line using a comma. Finally, once you have everything on one line, you can abbreviate almost anything, such as replacing `with lines` with `w l`. This is yet another minimalism-versus-clarity tradeoff, and you are encouraged to stick with the clear version at first.

5.2 ※ Some common settings

At this point, you can produce a basic plot of data or a function (or both at once). But you may have in mind a different look from Gnuplot's default, which means you will need to put a few `set` commands before the final `plot`.

This section catalogs the most common settings. For more information on the settings here and many more, see the very comprehensive Gnuplot documentation, which can be accessed from inside the Gnuplot command-line program via `help`, optionally followed by any of the headers below (e.g., `help set style`, `help set pointtype`).

Finally, if you are interactively experimenting with settings—which you are encouraged to do while reading this section—bear in mind that you may have to give a `replot` command (one word with no options) before the settings take effect.

`set style` The basic style of the plot may be a simple line or points, a bar plot, boxes with error bars, or many other possibilities. Gnuplot keeps track of two types of style: that for function plotting (`set style function`) and for data plotting (`set style data`). For example, to plot something that looks like a bar chart, try `set style data boxes`, followed on the next line with `plot` *yourdata*. As above, this is equivalent to the slightly shorter form `plot` *yourdata* `with boxes`, but it is often useful to separate the style-setting from the plot content.

Other favorite data styles include `lines`, `dots`, `impulses` (lines from the x-axis to the data level), `steps` (a continuous line that takes no diagonals), `linespoints` (a line with the actual data point marked), and `errorbars` (to be discussed below).

If you are plotting a function, like `plot sin(x)`, then use `set style function lines` (or `dots` or `impulses`, et cetera). Because there are separate styles for functions and data, you can easily plot data overlaid by a function in a different style.

`set pointtype, set linetype` You can set the width and colors of your lines, and whether your points will display as balls, triangles, boxes, stars, et cetera.[3]

The `pointtype` and `linetype` commands, among a handful of other commands, may differ from on-screen to Postscript to PNG to other formats, depending upon what is easy in the different formats. You can see what each terminal can do via the `test` command. E.g.:

```
set terminal postscript
set out 'testpage.ps'
test
```

Among other things, the test page displays a numbered catalog of points and lines available for the given terminal.

`set title, set xlabel, set ylabel` These simple commands label the X and Y axes and the plot itself. If the plot is going to be a figure in a paper with a paper-side caption, then the title may be optional, but there is rarely an excuse for omitting axis labels. Sample usage:

[3]As of this writing, Gnuplot's default for the first two plotted lines is to use `linetype 1`=red and `linetype 2`=green. Seven percent of males and 0.4% of females are red–green colorblind and therefore won't be able to distinguish one line from the other. Try, e.g., `plot sin(x); replot cos(x) linetype 3`, to bypass `linetype 2`=green, thus producing a red/blue plot.

```
set xlabel 'Time, days'
set ylabel 'Observed density, picograms/liter'
set title 'Density over time'
```

`set key` Gnuplot puts a legend on your plot by default. Most of the time, it is reasonably intelligent about it, but sometimes the legend gets in the way. Your first option in this case is to just turn off the key entirely, via `unset key`.

The more moderate option is to move the key, using some combination of `left`, `right`, or `outside` to set its horizontal position and `top`, `bottom`, or `below` to set its vertical; ask `help set key` for details on the precise meaning of these positions (or just experiment).

- The key also sometimes benefits from a border, via `set key box`.
- For surface plots with `pm3d`, the key is a thermometer displaying the range of colors and their values. To turn this off, use `unset colorbox`. See `help set colorbox` on moving the box or changing its orientation.

`set xrange`, `set yrange` Gnuplot generally does a good job of selecting a default range for the plot, but you can manually override this using `set range[`*min*`:`*max*`]`.

- Sometimes, you will want to leave one end of the range to be set by Gnuplot, but fix the other end, in which case you can use a `*` to indicate the automatically-set bound. For example, `set yrange [*:10]` fixes the top of the plot at ten, but lets the lower end of the plot fall where it may.
- You may want the axes to go backward. Say that your data represents rankings, so 1 is best; then `set yrange [*:1] reverse` will put first place at the top of the plot, and autoset the bottom of the plot to just past the largest ranking in the data set.

`set xtics` **AND** `set ytics` Gnuplot has reasonably sensible defaults for how the axes will be labeled, but you may also set the tick marks directly. To do so, provide a full list in parens for the text and position of every last label, such as on lines seven and eight of the code on page 161.

Producing this by hand is annoying, but as a first indication of producing Gnuplot code from C, here is a simple routine to write a `set ytics` line:

```
static void deal_with_y_tics(FILE *f, double min, double max, double step){
  int j = 0;
     fprintf(f, "set ytics (");
     for (double i=n_min; i< n_max; i+=n_step){
         fprintf(f, "'%g' %i", i, j++);
         if (i+n_step <n_max−1)
            fprintf(f, ", ");
     }
     fprintf(f, ")\n");
}
```

ASSORTED Here are a few more settings that you may find handy.

```
unset border          #Delete the border of the plot.
unset grid            #Make the plot even more minimalist.
set size square       #Set all axes to have equal length on screen or paper.
set format y "%.3g"   #You can use printf strings to format axis labels.
set format y ""       #Or just turn off printing on the Y axis entirely.
set zero 1e−20        #Set limit at which a point is rounded to zero (default: 1e−8).
```

5.3 FROM ARRAYS TO PLOTS

The scripts above gave a file name from which to read the data (`plot 'data-debt'`). Alternatively, `plot '-'` tells Gnuplot to plot data to be placed immediately after the `plot` command. With the `'-'` trick, the process of turning a matrix into a basic plot is trivial. In fact, the principle is so simple that there are several ways of implementing it.

Write to a file Let `data` be an `apop_data` set whose first and fifth columns we would like to plot against each other. Then we need to create a file, put a `plot '-'` command in the first line (perhaps preceded by a series of static `set` commands), and then fill the remainder with the data to be plotted. Below is the basic code to create a Gnuplot file. Since virtually anything you do with Gnuplot will be a variant of this code, it will be dissected in detail.

```
FILE *f = fopen("plot_me", "w");
if (!f) exit(0);
fprintf(f, "set key off; set ylabel 'picograms/liter'\n set xrange [−10:10]\n");
fprintf(f, "plot '−' using 1:5 title 'columns one and five'\n");
fclose(f);
apop_matrix_print(data−>matrix, "plot_me");
```

- The first argument to fopen is the file to be written to, and the second option should be either "a" for append or "w" for write anew; in this case, we want to start with a clean file so we use "w".
- The fopen function can easily fail, for reasons including a mistyped directory name, no permissions, or a full disk. Thus, you are encouraged to check that fopen worked after every use. If the file handle is NULL, then something went wrong and you will need to investigate.
- As you can see from lines three and four, the syntax for fprintf is similar to that of the rest of the printf family: the first argument indicates to what you are writing, and the rest is a standard printf line, which may include the usual insertions via %g, %i, and so on.
- You can separate Gnuplot commands with a semicolon or newline, and can put them in one fprintf statement or many, as is convenient. However, you will need to end each line of data (and the plot line itself) with a newline.
- Since Gnuplot lets us select columns via the using 1:5 clause, there is no need to pare down the data set in memory. Notice again that the first column in Gnuplot is 1, not 0.
- The title clause shows that Gnuplot accepts both “double quotes” and ‘single quotes’ around text such as file names or labels. Single quotes are nothing special to C, so this makes it much easier to enter such text.
- Line five closes the file, so there is no confusion when line six writes the data to the file. It prints the matrix instead of the full apop_data structure so that no names or labels are written. Since apop_matrix_print defaults to appending, the matrix appears after the plot header that lines two and three wrote to the file. You would need to set apop_opts.output_append=0 to overwrite.
- At the end of this, plot_me will be executable by Gnuplot, using the forms like gnuplot -persist < plot_me, as above.

Instant gratification The above method involved writing your commands and data to a file and then running Gnuplot, but you may want to produce plots as your program runs. This is often useful for simulations, to give you a hint that all is OK while the program runs, and to impress your friends and funders. This is easy to do using a pipe, so named because of UNIX’s running data-as-water metaphor; see Appendix B for a full exposition.

The command popen does two things: it runs the specified program, and it produces a data pipe that sends a stream of data produced by your program to the now-running child program. Any commands you write to the pipe are read by the child as if someone had typed those commands into the program directly.

Listing 5.3 presents a sample function to open and write to a pipe.

```
 1 #include <apop.h>
 2
 3 void plot_matrix_now(gsl_matrix *data){
 4   static FILE *gp = NULL;
 5     if (!gp)
 6         gp = popen("gnuplot -persist", "w");
 7     if (!gp){
 8         printf("Couldn't open Gnuplot.\n");
 9         return;
10     }
11     fprintf(gp,"reset; plot '-' \n");
12     apop_opts.output_type = 'p';
13     apop_opts.output_pipe = gp;
14     apop_matrix_print(data, NULL);
15     fflush(gp);
16 }
17
18 int main(){
19     apop_db_open("data-climate.db");
20     plot_matrix_now(apop_query_to_matrix("select (year*12+month)/12., temp from temp"));
21 }
```

Listing 5.3 A function to open a pipe to Gnuplot and plot a vector. Online source: `pipeplot.c`.

- The `popen` function takes in the location of the Gnuplot executable, and a `w` to indicate that you will be writing to Gnuplot rather than reading from it. Most systems will accept the simple program name, `gnuplot`, and will search the program path for its location (see Appendix A on paths). If `gnuplot` is not on the path, then you will need to give an explicit location like `/usr/local/bin/gnuplot`. In this case, you can find where Gnuplot lives on your machine using the command `which gnuplot`. The `popen` function then returns a `FILE*`, here assigned to `gp`.
- Since `gp` was declared to be a static variable, and `popen` is called only when `gp==NULL`, it will persist through multiple calls of this function, and you can repeatedly call the function to produce new plots in the same window.
  If `gp` is `NULL` after the call to `popen`, then something went wrong. This is worth checking for every time a pipe is created.
- But if the pipe was created properly, then the function continues with the now-familiar process of writing `plot '-'` and a matrix to a file. The `reset` command to Gnuplot (line 11) ensures that next time you call the function, the new plot will not have any strange interactions with the last plot.
- Lines 12 and 13 set the output type to `'p'` and the output pipe to `gp`; `apop_matrix_print` uses these global variables to know that it should write to that pipe instead of a file or `stdout`.
- Notice the resemblance between the form here and the form used to write to a file

above. A `FILE` pointer can point to either a program expecting input or a file, and the program writes to either in exactly the same manner. They are even both closed using `fclose`. The only difference is that a file opens via `fopen` and a program opens with `popen`. Thus, although the option is named `apop_opts.output_pipe`, it could just as easily point to a file; e.g., `FILE *f=fopen(plotme, "w"); apop_opts.output_pipe = f;`.

- One final detail: piped output is often kept in a *buffer*, a space in memory that the system promises to eventually write to the file or the other end of the pipe. This improves performance, but you want your plot now, not when the system deems the buffer worth writing. The function on line 15, `fflush`, tells the system to send all elements of the `gp` buffer down the pipeline. The function also works when you are expecting standard output to the screen, by the way, via `fflush(stdout)`.[4]

The main drawback to producing real-time plots is that they can take over your computer, as another plot pops up and grabs focus every half second, and can significantly slow down the program. Thus, you may want to settle for occasional redisplays, such as every fifty periods of your simulation, via a form like

```
for (int period =0; period< 1000; period++){
    gsl_vector *output = run_simulation();
    if (!(period % 50))
        plot_vector_now(output);
}
```

Q5.2

Now would be a good time to plot some data series.

- Query a two-column table from `data-tattoo.db` giving the birth year in the first column and the mean number of tattoos for respondents in that birth year in the second.
- Dump the data to a Gnuplottable file using the above `fprintf` techniques.
- Plot the file, then add `set` commands that you `fprintf` to file to produce a nicer-looking plot. E.g., try boxes and impulses.
- Modify your program to display the plot immediately using pipes. How could you have written the program initially to minimize the effort required to switch between writing to a file and a pipe?
- How does the graph look when you restrict your query to include only those with a nonzero number of tattoos?

[4]Alternatively, `fflush(NULL)` will flush all buffers at once. For programs like the ones in this book, where there are only a handful of streams open, it doesn't hurt to just use `fflush(NULL)` in all cases.

※*Self-executing files* This chapter is about the many possible paths from a data set to a script that can be executed by an external program; here's one more.

Those experienced with POSIX systems know that a script can be made executable by itself. The first line of the script must begin with the special marker #! followed by the interpreter that will read the file, and the script must be given execute permissions. Listing 5.4 shows a program that produces a self-executing Gnuplot script to plot $\sin(x)$. You could even use `system("./plotme")` to have the script execute at the end of this program.

```
#include <apop.h>
#include <sys/stat.h> //chmod

int main(){
    char filename[] = "plot_me";
    FILE *f = fopen(filename, "w");
    fprintf(f, "#!/usr/bin/gnuplot −persist\n\
                plot sin(x)");
    fclose(f);
    chmod(filename, 0755);
}
```

Listing 5.4 Write to a file that becomes a self-executing Gnuplot script. Run the script using `./plot_me` from the command line. Online source: `selfexecute.c`.

※ `replot` **REVISITED** `replot` and the `plot '-'` mechanism are incompatible, because Gnuplot needs to re-read the data upon replotting, but can not back up in the stream to reread it. Instead, when replotting data sets, write the data to a separate file and then refer to that file in the file to be read by Gnuplot. To clarify, here is a C snippet that writes to a file and then asks Gnuplot to read the file:

```
apop_data_print(data, "datafile");
FILE *f = fopen("gnuplot", "w");
fprintf(f, "plot 'datafile' using 1 title 'data column 1';\n \
            replot 'datafile' using 5 title 'data column 5';\n");
fclose(f);
```

Also, when outputting to a paper device, replotting tends to make a mess. Set the output terminal just before the final replot:

```
plot 'datafile' using 1
replot 'datafile' using 2
replot 'datafile' using 3
set term postscript color
set out 'four_lines.eps'
replot 'datafile' using 4
```

∑

- ➤ A Gnuplot command file basically consists of a group of `set` commands to specify how the plot will look, and then a single `plot` (2-D version) or `splot` (3-D version) command.
- ➤ You can run external text-driven programs from within C using `fopen` and `popen`.
- ➤ Using the `plot '-'` form, you can put the data to be plotted in the command file immediately after the command. You can use this and `apop_data_print` to produce plot files of your data.

5.4 A SAMPLING OF SPECIAL PLOTS

For a system that basically only has a `set` and a `plot` command, Gnuplot is surprisingly versatile. Here are some specialized visualizations that go well beyond the basic 2-D plot.

LATTICES Perhaps plotting two pairs of columns at a time is not sufficient—you want bulk, displaying every variable plotted against every other. For this, use the `apop_plot_lattice` function.

```
#include "eigenbox.h"

int main(){
    apop_plot_lattice(query_data(), "out");
}
```

Listing 5.5 The code to produce Figure 5.6. Link with the code in Listing 8.2, p 267. Online source: `lattice.c`.

Listing 5.5 produces a file that (via `gnuplot -persist <out`) produces the plot in Figure 5.6. Yes, it is one line of code, but you will need to link it with the data-querying code in Listing 8.2, p 267. Each variable is plotted against every other; e.g., the upper-middle plot shows males per 100 females versus state population, and the middle-left plot shows a mirror image (along the diagonal line) of the same plot.

What is a lattice plot good for? Some call it *getting a lay of the land*, while others call it *data snooping*. Given ten perfectly random variables, there is a good chance that at least one pair of lattice plots will look to you as if it demonstrates a nice correlation. A formal regression on the chosen pair of variables will likely verify your initial visual impression. For more on this conflict, see page 316.

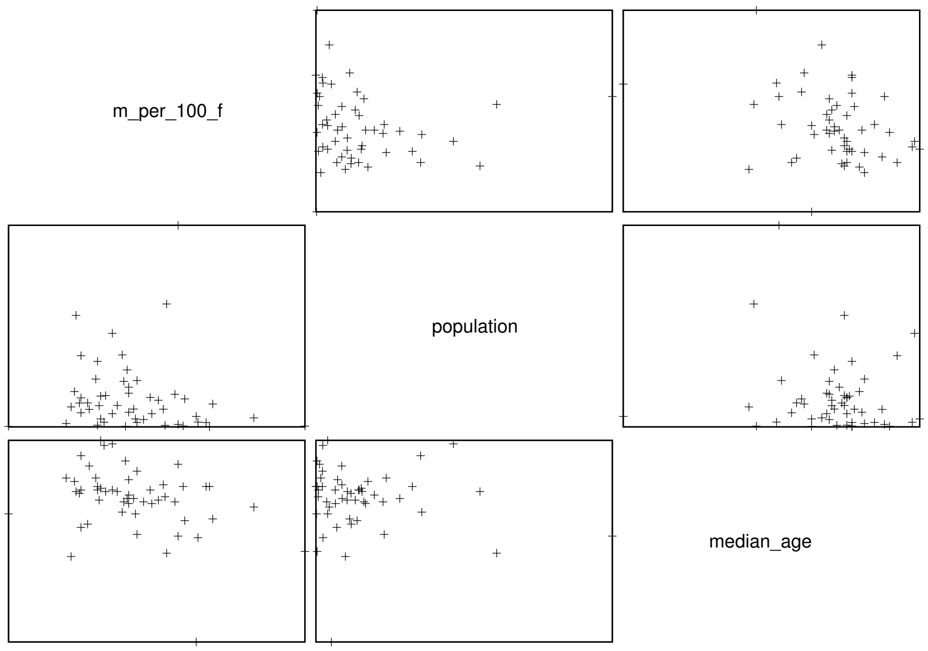


Figure 5.6 The Census data, as queried on page 267.

ERROR BARS The typical error bar has three parts: a center, a top limit, and a bottom limit. Gnuplot supports this type of data directly, via `set style data errorbars`. You can provide the necessary information in a variety of formats; the most common are (x, y center, y top, y bottom) and (x, y center, y range), where the range is typically a standard deviation. Listing 5.7, which produces Figure 5.8, takes the second approach, querying out a month, the mean temperature for the month, and the standard deviation of temperature for the month. Plotting the data shows both the typical annual cycle of temperatures and the regular fluctuation of variances of temperature.

HISTOGRAMS A histogram is a set of X- and Y-values like any other, so plotting it requires no special tools. However, Gnuplot will not take a list of data and form a histogram for you—you have to do this on the C-side and then send the final histogram to Gnuplot. Fortunately, `apop_plot_histogram` does the binning for you. Have a look at Listing 11.2, page 359, for an example of turning a list of data items into a histogram (shown in Figure 11.3).

```
#include <apop.h>

int main(){
    apop_db_open("data−climate.db");
  apop_data *d = apop_query_to_data("select \
            (yearmonth/100. − round(yearmonth/100.))*100 as month, \
            avg(tmp), stddev(tmp) \
            from precip group by month");
    printf("set xrange[0:13]; plot '−' with errorbars\n");
    apop_matrix_show(d−>matrix);
}
```

Listing 5.7 Query out the month, average, and variance, and plot the data using errorbars. Prints to stdout, so pipe the output through Gnuplot: `./errorbars | gnuplot -persist`. Online source: `errorbars.c`.

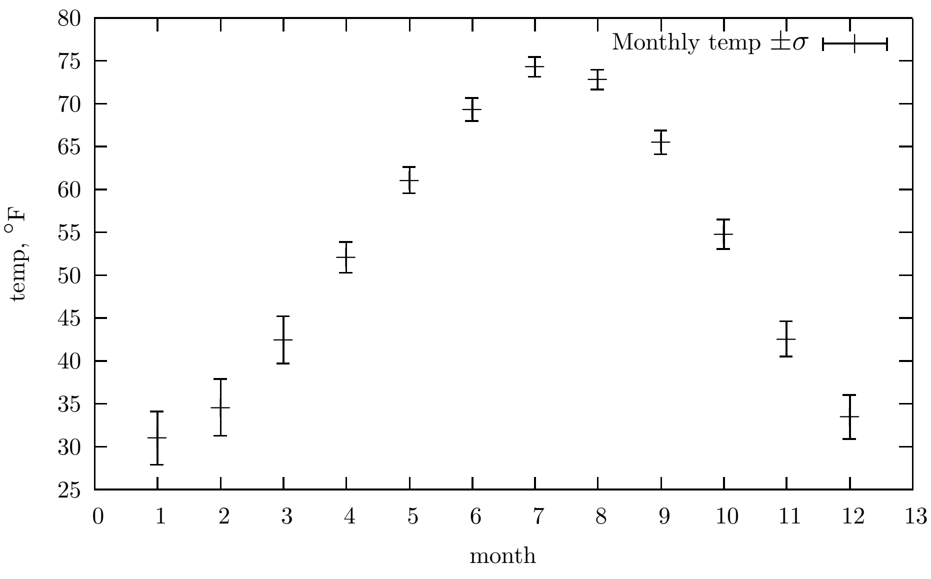


Figure 5.8 Monthly temperature, $\pm\sigma$.

The *leading digit* of a number is simply its most significant digit: the leading digit of 3247.8 is 3, and the leading digit of 0.098 is 9. *Benford's law* (Benford, 1938) states that the digit $d \in \{0, 1, \ldots, 9\}$ will be a leading digit with frequency

$$F_d \propto \ln\left((d+1)/d\right). \tag{5.4.1}$$

$\mathbb{Q}_{5.3}$ Check this against a data set of your choosing, such as the `population` column in the World Bank data, or the `Total_area` column from the US Census data. The formal test is the exercise on page 322; since this is the chapter on plotting, just produce a histogram of the chosen data set and verify that it slopes sharply downward. (*Hint*: if $d = 3.2\text{e}7$, then $10^{\texttt{(int)log10}(d)} = 1\text{e}7$.)

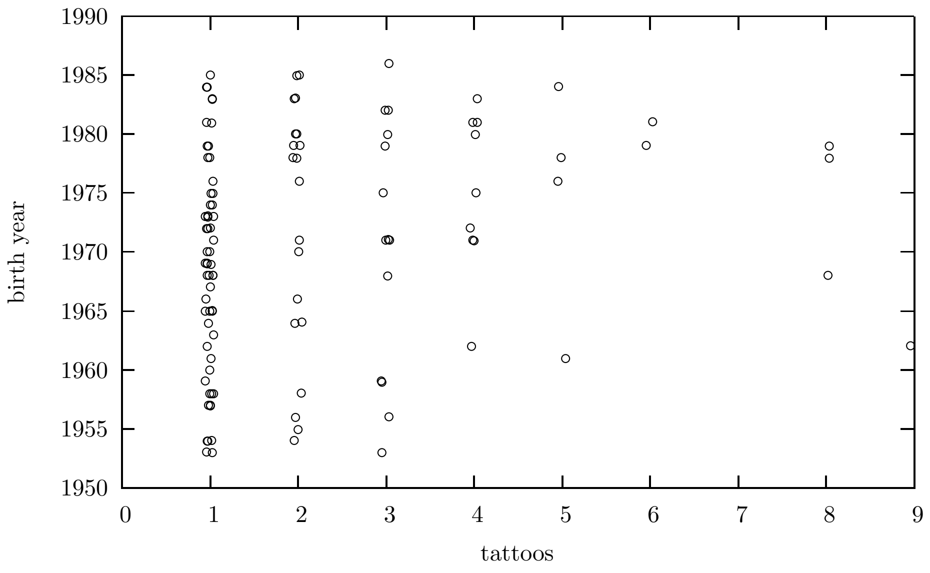


Figure 5.9 Each point represents a person.

Log plots You can plot your data on a log scale by either transforming it before it gets to Gnuplot or by transforming the plot.

Log plots work best on a log-base-10 scale, rather than the typical natural logarithm, because readers can immediately convert a 2 on the scale to 1e2, a -4 to 1e-4, et cetera. From C, you can use the `log10(x)` function to calculate $\log_{10}$`x`, and if your data is in a `gsl_vector`, you can use `apop_vector_log10` to transform the entire vector at once.

In Gnuplot, simply `set logscale y` to use a log-scaled Y axis, `set logscale x` to use a log-scaled X axis, or `set logscale xy` for both.

$\mathbb{Q}_{5.4}$ Redo the `data-debt` plot from the beginning of this chapter using a log scale.

Pruning and jittering Plotting the entire data set may be detrimental for a few reasons. One is the range problem: there is always that one data point at $Y = 1e20$ throws off the whole darn plot. If you are using an interactive on-screen plot, you can select a smaller region, but it would be better to just not plot that point to begin with.

The second reason for pruning is that the data set may be too large for a single page. The black blob you get from plotting ten million data points on a single piece of paper is not very informative. In this case, you want to use only a random subset of the data.[5]

Both of these are problems of selecting data, and so they are easy to handle via SQL. A simple `select * from plotme where value < 1e7` will eliminate values greater than a million, and page 84 showed how to select a random subset of the data.

In Gnuplot, you can add the `every` keyword to a plot, such as `plot 'data' every 5` to plot every fifth data point. This is quick and easy, but take care that there are no every-five patterns in the data.

Now consider graphing the number of tattoos a person has against her year of birth. Because both of these are discrete values, we can expect that many people will share the same year of birth and the same tattoo count, meaning that the plot of those people would be exactly one point. A point at (1965, 1 tattoo) could represent one person or fifty.

```
#include <apop.h>
gsl_rng *r;

void jitter(double *in){
    *in += (gsl_rng_uniform(r) − 0.5)/10;
}

int main(){
    apop_db_open("data−tattoo.db");
    gsl_matrix *m = apop_query_to_matrix("select \
                    tattoos.'ct tattoos ever had' ct, \
                    tattoos.'year of birth'+1900 yr \
                    from tattoos \
                    where yr < 1997 and ct+0.0 < 10");
    r = apop_rng_alloc(0);
    apop_matrix_apply_all(m, jitter);
    printf(set key off; set xlabel 'tattoos'; \n\
            set ylabel 'birth year'; \n\
            plot '−' pointtype 6\n");
    apop_matrix_show(m);
}
```

Listing 5.10 By adding a bit of noise to each data point, the plot reveals more data. Online source: `jitter.c`.

[5]Another strategy for getting less ink on the page is to change the point type from the default cross to a dot. For the typical terminal, do this with `plot ... pointtype 0`.

One solution is to add a small amount of noise to every observation, so that two points will fall exactly on top of each other with probability near zero. Figure 5.9 shows such a plot. Without jittering, there would be exactly one point on the one-tattoo column for every year from about 1955 to 1985; with jittering, it is evident that there are generally more people with one tattoo born from 1965–1975 than in earlier or later years, and that more than half of the sample with two tattoos was born after 1975.

Further, the process of jittering is rather simple. Listing 5.10 shows the code used to produce the plot. Lines 10–14 are a simple query; lines 17–20 produce a Gnuplot header and file that get printed to `stdout` (so run the program as `./jitter | gnuplot`). In between these blocks, line 16 applies the `jitter` function to every element of the matrix. That function is on lines 4–6, and simply adds a random number $\in [-0.05, 0.05]$ to its input. Comment out line 16 to see the plot without jitter.

$\mathbb{Q}_{5.5}$ Modify the code in Listing 5.10 to do the jittering of the data points in the SQL query instead of in the `gsl_matrix`.

$\sum$

- ➤ Apophenia will construct a grid of plots, plotting every variable against every other, via `apop_plot_lattice`.
- ➤ With Gnuplot's `set style errorbars` command, you can plot a range or a one-σ spread for each data point.
- ➤ You need to aggregate data into histograms outside of Gnuplot, but once that is done, plotting them is as easy as with any other data set.
- ➤ If the data set has an exceptional range, you can take the log of the data in C via `log10`, plot the log in Gnuplot via `set logscale y`, or trim the data in SQL via, e.g., `select` *`cols`* `from` *`table`* `where` *`var<1e20`*.
- ➤ If data falls on a grid (e.g., integer-valued rows and columns), then you can add *jitter* to the plot to reveal the density at each point.

5.5 Animation

Perhaps three dimensions is not quite enough, and you need one more. Gnuplot easily supports animation: just stack matrices one after the next and call `plot` in between. However, many media (such as paper) do not yet support animation, meaning that your output will generally be dependent on your display method. The GIF format provides animation and is supported by all major web browsers, so you can also put your movies online.[6]

There are two details that will help you with plotting multiple data sets. First, Gnuplot reads an `e` alone on a line to indicate the end of a data set. Second, Gnuplot allows you to define constants via a simple equals sign; e.g., the command `p = 0.6` creates the variable `p` and sets it to 0.6. Third, Gnuplot has a `pause` *p* command that will wait p seconds before drawing the next plot.

Tying it all together, we want a Gnuplottable file that looks something like this:

```
set pm3d;
  p = 1;
splot '−' with pm3d
{data[0] here}
e
 pause p;
splot '−' with pm3d
{data[1] here}
e
 pause p;
splot '−' with pm3d
{data[2] here}
e
```

You can run the resulting output file through the Gnuplot command line as usual. If a one-second pause is too long or too short, you only need to change the single value of `p` at the head of the file to change the delay throughout.

You can write the above plots to a paper-oriented output format, in which case each plot will be on a separate page. You could then page through them with a screen viewer, or perhaps print them into a flipbook. When writing a file, set `p=0` in the above, since there is no use delaying between outputs.

In addition to the example below, Listings 7.12 (page 253) and 9.1 (page 298) also demonstrate the production of animations.

[6]For GIFs, you will need to request animation in the `set term` line, e.g., `set term gif animate delay 100`. The number at the end is the pause between frames in hundredths of a second.

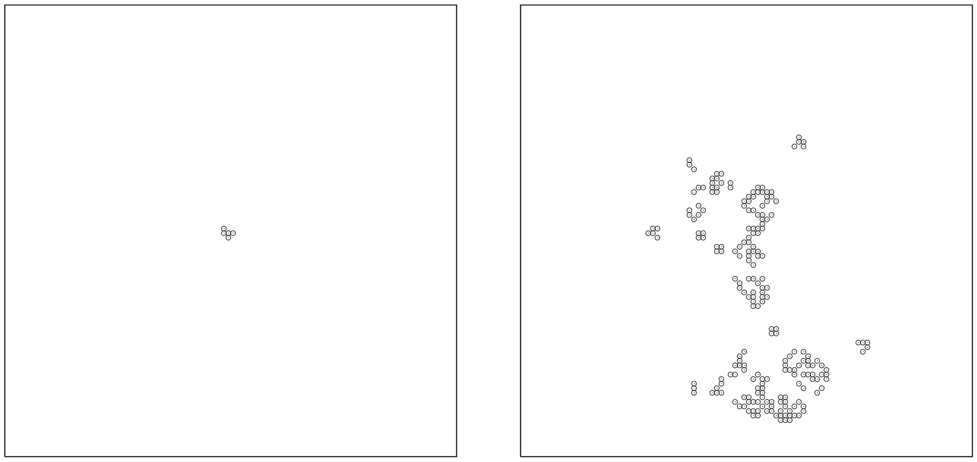


Figure 5.11 The Game of Life: at left is the original colony; at right is the colony after 150 periods. The vaguely V-shaped figures (such as the groups farthest left, right, and top) are known as *gliders*, because they travel one step forward every four periods.

An example: the game of life There is a tradition of agent-based modeling built around plotting agents on a grid, pioneered by Epstein & Axtell (1996). A simple predecessor is Conway's Game of Life, a *cellular automaton* discussed at length in Gardner (1983). The game is played on a grid, where each point on the grid can host a single blob. These blobs can not move, and are somewhat delicate: if the blob has only zero or one neighbors, it dies of loneliness, and if it has four or more neighbors, it dies of overcrowding. If an empty cell is surrounded by exactly three blobs, then a new blob is born on that cell.

These simple rules produce complex and interesting patterns. At left in Figure 5.11 is the so-called *r pentomino*, a simple configuration of five blobs. At right is the outcome after 150 periods of life. Listing 5.12 presents the code to run the game. Because the program prints Gnuplot commands to `stdout`, run it using `./life | gnuplot`.

- The game uses two grids: the completed grid from the last period, named `active` in the code, and the incomplete grid that will soon represent the state of life in the next period, named `inactive`.
- Both grids are initialized to zero (lines 27–28), and then lines 29–31 define the r pentomino.
- Line 32 is the Gnuplot header, and line 34 tells Gnuplot that data points are coming.
- The main work each period is preparing the inactive grid, which is what the `calc_grid` function does. It sets everything to zero, and then the two loops (`i`-indexed for rows and `j`-indexed for columns) checks every point in the grid except the borders.
- The `area_pop` function calculates the population in a 3×3 space; line 16 needs

```
#include <apop.h>
int area_pop(gsl_matrix *a, int row, int col){
  int i, j, out = 0;
    for (i=row−1; i<= row+1; i++)
        for (j=col−1; j<= col+1; j++)
            out += gsl_matrix_get(a, i, j);
    return out;
}

void calc_grid(gsl_matrix* active, gsl_matrix* inactive, int size){
  int i, j, s, live;
    gsl_matrix_set_all(inactive, 0);
    for(i=1; i< size−1; i++)
        for(j=1; j< size−1; j++){
            live = gsl_matrix_get(active, i, j);
            s = area_pop(active, i, j) − live;
            if ((live && (s == 2 || s == 3))
                || (!live && s == 3)){
                    gsl_matrix_set(inactive, i, j, 1);
                    printf("%i %i\n", i, j);
            }
        }
}

int main(){
  int i, gridsize=100, periods = 550;
  gsl_matrix *t, *active = gsl_matrix_calloc(gridsize,gridsize);
  gsl_matrix *inactive = gsl_matrix_calloc(gridsize,gridsize);
    gsl_matrix_set(active, 50, 50, 1); gsl_matrix_set(active, 49, 51, 1);
    gsl_matrix_set(active, 49, 50, 1); gsl_matrix_set(active, 51, 50, 1);
    gsl_matrix_set(active, 50, 49, 1);
    printf("set xrange [1:%i]\n set yrange [1:%i]\n", gridsize, gridsize);
    for (i=0; i < periods; i++){
        printf("plot '−' with points pointtype 6\n");
        calc_grid(active, inactive, gridsize);
        t = inactive;
        inactive = active;
        active = t;
        printf("e\n pause .02\n");
    }
}
```

Listing 5.12 Conway's game of life. Online source: `life.c`.

to subtract the population (if any) at the central point to get the population in the point's neighborhood.

- Notice that the loops in both `area_pop` and `calc_grid` never consider the edges of the grids. That means that `area_pop` does not need to concern itself with edge conditions like `if (i!=0)....`

- The rules of the Game of Life are summarized in the `if` statement in lines 17–18. There will be a blob at this point if either there is currently a blob and it has two or three neighbors, or there is no living blob there but there are three neighbors.
- If the `if` statement finds that there will be a blob in this space next period, it prints a point to Gnuplot, and marks it in the currently inactive grid.
- Lines 36–38 is the classic swap of the active and inactive grids, using a temp location to help make the exchange. Since we are only shunting the addresses of data, the operation takes zero time.

Q5.6 Other rules for life or death also produce interesting results. For example, rewrite the rules so that a living blob stays alive only if there are 2, 3, 4, or 5 neighbors, and an empty space has a birth only if there is one neighbor. Or try staying-alive rules of 2 and 6 with birth rules of 1 and 3.

5.6 On producing good plots

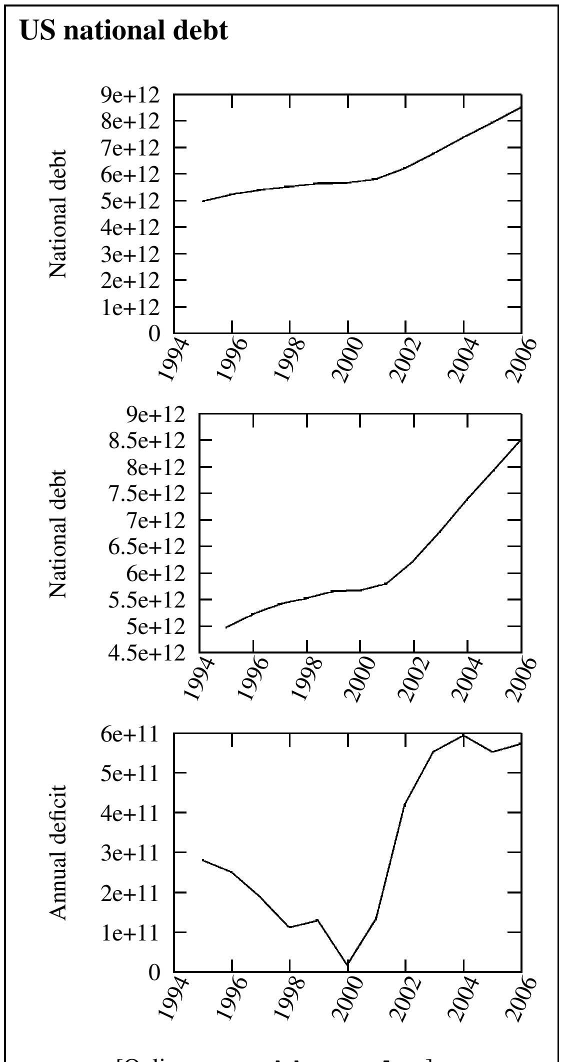
Cleveland & McGill (1985) offer some suggestions for producing plots for the purpose of perceiving the patterns among the static. Their experiments were aimed at how well people could compare data presented in various formats, and arrived at an ordering of graphical elements from those that were most likely to allow accurate perception to layouts that inhibited accurate perception:

i) Position along a common scale (e.g., the height of the means in the temperature plot on page 173)
ii) Position on identical but nonaligned scales (such as comparing points on two separate graphs)
iii) Length (e.g., the height of the error bars in the temperature plot on page 173)
iv) Angle
v) Slope (when not too close to vertical or horizontal)
vi) Area
vii) Volume, Density, Color saturation (e.g., a continuous scale from light blue to dark blue)
viii) Color hue (e.g., a continuous scale from red to blue)

Data should be presented using techniques as high up on the scale as possible. Pie charts (representing data via angle and area) are a bad idea because there are many better ways to present the same data. Gnuplot does provide a means of setting data-dependent color, but given that color is at the bottom of Cleveland and

McGill's list, it should be used as a last resort. They did not run experiments with animation, but our eyes have cells exclusively dedicated to sensing motion, so it seems sensible that if movement were included on the list, it would rank highly.

As for angles and slopes, consider these plots of the US national debt since 1995. The top plot has a y axis starting at $0, while the second has a y axis starting at $4.5 trillion. Is the rate of change from 1995–1996 larger or smaller than the rate of change from 2005–2006? Was growth constant or decelerating from 1995–2000? It is difficult to answer these questions from the top graph, because everything is somewhat flat—the change in angles is too small to be perceived, and we are bad at discerning slopes. Differences in slope on the second scale are more visible. The lesson is that plots show their patterns most clearly when the axes are set such that the slope is around 45°.



[Online source: `debt.gnuplot`.]

The bottom plot is the US national deficit. This is the amount the government spends above its income, and is thus the rate of change of the debt. The rate of change in the first two graphs (a slope in those graphs) is the height of this plot for each year, and position along a common scale is number one on Cleveland and McGill's list. The answers to the above questions are now obvious: the rate of change for the debt slowed until 2000 and then quickly rose to about double 1995 rates.

Darrell Huff, in the classic *How to Lie With Statistics* (Huff & Geis, 1954, Chapter 5), has a different goal and so makes different recommendations. He points out that the top two graphs tell a different narrative. The second graph tells a story that the national debt is rapidly increasing, because the height of the point at 2006 is about eight times the height at 1995. The top graph shows that the debt rose, but not at a fast-multiplying rate. Huff concludes that the full story is best told when the zero of the y axis always in the picture, even if this means blank space on the page—advice that directly contradicts the advice above. So the choice for any given plot depends on the context and intent, although some rules—like avoiding pie charts and continuous color scales—are valid for almost all situations.

5.7 ※ GRAPHS—NODES AND FLOWCHARTS

In common conversation, we typically mean the word *graph* to be a plot of data like every diagram to this point in the chapter. The mathematician's definition of *graph*, however, is a set of nodes connected by edges, as in Figure 5.13. Gnuplot can only plot; if you have network data that you would like to visualize, Graphviz is the package to use. Like all of the tools in this book, Graphviz installs itself gracefully via your package manager.

The package includes various executables, the most notable of which are `dot` and `neato`. Both take the same input files, but `dot` produces a flowchart where there is a definite beginning and end, while `neato` produces more amorphous plots that aim only to group nodes via the links connecting them.

The programs are similar to Gnuplot in that they take in a plain text description of the nodes and edges, and produce an output file in any of a plethora of graphics formats. The syntax for the input files is entirely different from Gnuplot's, but the concept is familiar: there are elements to describe settings interspersed with data elements.

For example, flip back to page 3 and have a look at Figure 1.3. Produce the first graph in the figure using the following input to `dot`:

```
digraph {
    rankdir = LR;
    node [shape=box];
    "Data" −> "Estimation" −> "Parameters";
}
```

The lines with `=` in them set parameters, stating that the graph should read left-to-right instead of the top-to-bottom default, and that the nodes should be boxes instead of the default ellipses. The line with the `->`s defines how the nodes should link, and already looks like a text version of Figure 1.3. The command line

```
dot −Tpng <graphdata.dot > output.png
```

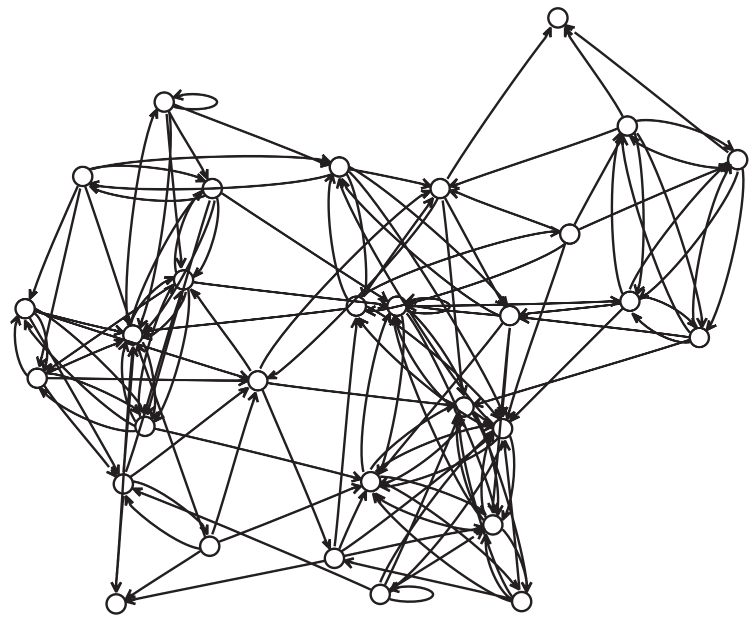


Figure 5.13 The social network of a Junior High classroom.

produces a graph much like that in Figure 1.3, in the PNG format.[7]

At this point, you have all the tools you need to autogenerate a graph. For example, say that you have an $n \times n$ grid where a one in position (i, j) indicates a link between agents i and j and a zero indicates no link. Then a simple `for` loop would convert this data into a `neato`-plottable file, with a series of rows with a form like, e.g., `node32 -> node12`.

```
void produce_network_graph(apop_data *link_data, char *outfile){
  FILE *g = fopen(outfile, "w");
    fprintf(g, "digraph{\n");
    for (int i=0; i< link_data->matrix->size1; i++)
        for (int j=i+1; j< link_data->matrix->size2; j++)
            if (apop_data_get(link_data, i,j))
                fprintf(g, "node%i -> node%i;\n", i,j);
    fprintf(g, "}\n");
    fclose(g);
}
```

In the code supplement, you will find a file named `data-classroom`, which lists the survey results from a Junior High classroom in LA, in which students listed

[7]The second half of Figure 1.3 was produced using exactly the same graph, plus the `psfrag` package to replace text like *Data* with the OLS-specific math shown in the figure.

their five best friends. The *ego* column is the student writing down the name of his or her best friend, and the *nominee* column gives the number of the best friend. Figure 5.13 graphs the classroom, using `neato` and the following options for the graph:

```
digraph{
      node [label="",shape=circle,height=0.12,width=0.12];
      edge [arrowhead=open,arrowsize=.4];
...
      }
```

A few patterns are immediately evident in the graph: at the right is a group of four students who form a complete clique, but do not seem very interested in the rest of the class. The student above the clique was absent the day of the survey and thus is nominated as a friend but has no nominations him or herself; similarly for one student at the lower left. Most of the graph is made from two large clumps of students who are closely linked, at the left and in the center, probably representing the boys and the girls. There are two students who nominated themselves as best friends (one at top right and one at the bottom), and those two students are not very popular.

Q5.7

Write a program to replicate Figure 5.13.

- Read the `data-classroom` file into an `apop_data` set, using either `apop_text_to_db` and a query, or `apop_text_to_data`.
- Open your output file, and copy in the header from above.
- Write a single `for` loop to write one line to the output file for each line in the data set. Base your printing function on the one from page 183.
- Close the file.

Running the program will produce a `neato`-formatted file.

- From the command line, run `neato -Tps <my_output > out.eps`. The output graph should look just like Figure 5.13.
- In the `dot` manual (`man dot` from the command line), you will see that there are many variant programs to produce different types of graph. What does the classroom look like via `dot`, `circo`, and `twopi`?

The exercise on page 414 will show you another way to produce the Graphviz-readable output file.

Internal use If you have been sticking to the philosophy of coding via small, simple functions that call each other, your code will look like a set of elements linked via function calls—exactly the sort of network for which Graphviz was written. If you feel that your code files are getting a bit too complex, you can use Graphviz to get the big picture.

For example, say that your database is growing involved, with queries that merge tables into new tables, other queries to split the tables back into still more tables, et cetera. For each query that creates a table, it is easy to write down a line (or lines) like `base_tab -> child_tab`. Then, `dot` can sort all those individual links into a relatively coherent flow from raw data to final output.[8]

You could also graph the calling relationships among the functions in your C code—but before you start manually scanning your code, you should know that there is a program to do this for you. Ask your package manager for `doxygen`, which generates documentation via specially-formatted comments in the source file. If configured correctly, it will use Graphviz to include call graphs in the documentation.

The online code supplement includes a few more examples of Graphviz at work, including the code used to create Figures 1.1, 6.5, and 6.7.

$\sum$

- ➤ The Graphviz package produces graphs from a list of nodes and edges. Such lists are easy to autogenerate from C.
- ➤ You can also use Graphviz to keep track of relationships among functions in your code or tables in your database.

5.8 ※ Printing and LaTeX

This book focuses on tools to write replicable, portable analyses, where every step is described in a handful of human-legible text files that are sent to programs that behave in the same manner on almost any computer. The TeX document preparation system (and the set of macros built on top of it, LaTeX) extend the pipeline to the final writeup. For example, you can easily write a script to run an analysis and then regenerate the final document using updated tables and plots.[9]

[8]SQL does not have the sort of metadata other systems have for describing a table's contents in detail (e.g., the `apop_data` structure's `title` element). But you can set up a metadata table, with a column for the table name, its description, and the tables that generated that table. Such a table is reasonably easy to maintain, because you need only add an `insert into metadata ...` query above any query that generates a table. $\mathbb{Q}$: Write a function to take in such a table and output a flowchart demonstrating the flow of data through the database tables.

[9]To answer some questions you are probably wondering: yes, this book is a LaTeX document. Most of the plots were produced via `set term latex` in Gnuplot, to minimize complications with sending Postscript to the press. Save for the pointer-and-box diagrams in the C chapter, Student's hand-drawn menagerie, and the snowflake at

A complete tutorial on LaTeX would be an entire book—which has already been written dozens of times. But this chapter's discussion of the pipeline from raw data to output graphs is incomplete without mention of a few unpleasant details regarding plots in LaTeX.

You have two options when putting a plot in a TeXed paper: native LaTeX and Postscript.

Native format Just as you can set the output device to a screen or a Postscript printer, you can also send it to a file written using LaTeX's graphics sub-language. One the plus side, the fonts will be identical to those in your document, and the resolution is that of TeX itself (100 times finer than the wavelength of visible light). On the minus side, some features, such as color, are currently not available.

Producing a plot in LaTeX format requires setting the same `term/out` settings as with any other type of output: `set term latex; set out 'plot.tex'`.[10]

Just as you can dump one C file into another via `#include`, you can include the Gnuplot output via the `\input` command:

```
\documentclass{article}
\usepackage{latexsym}
\begin{document}
...
\begin{figure}
\input outfile.tex
\caption{This figure was autogenerated by a C program.}
\end{figure}
...
\end{document}
```

Another common complaint: the Y-axis label isn't rotated properly. The solution provides a good example of how you can insert arbitrary LaTeX code into your Gnuplot code. First, in the Gnuplot file, you can set the label to any set of instructions that LaTeX can understand. Let λ be an arbitrary label; then the following command will write the label and tell LaTeX to rotate it appropriately:

the head of every chapter, I made a point of producing the entire book using only the tools it discusses.

The snowflake was generated by covering a triangle with a uniform-by-area distribution of dots, each with a randomly selected color and size, and then rotating the triangle to form the figure. Therefore, any patterns you see beyond the six-sided rotational symmetry are purely apophenia.

And in my experience helping others build data-to-publication pipelines, the detail discussed in this section about rotating the Y-axis label really *is* a common complaint.

[10]By the way, for a single plot, the `set out` command is optional, since you could also use a pipe: `gnuplot < plotme > outfile.tex`.

```
set ylabel '\rotatebox{90}{Your $\lambda$ here.}'
```

Two final notes to complete the example: `\rotatebox` is in the `graphicx` package, so it needs to be called in the document preamble:

```
\usepackage{latexsym, graphicx}
```

Second, many dvi viewers do not support rotation, so if you are viewing via TEX's native dvi format, the rotation won't appear. Use either `pdflatex` or `dvips` to view the output as it will print.

The Postscript route Which brings us to the second option for including a graphic: Postscript.

Use the *graphicx* package to incorporate the plot. E.g.:

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
...
\begin{figure}
\rotatebox{90}{\scalebox{.35}{\includegraphics{outfile.eps}}}
\caption{This figure was autogenerated by a C program.}
\end{figure}
...
\end{document}
```

Notice that you will frequently need to rotate the plot 90° and scale the figure down to a reasonable size.

The first option for generating PDFs is to use `epstopdf`. First, convert all of your `eps` files to `pdf` files on the command line. In `bash`, try

```
for i in *.eps; do
    epstopdf $i;
done
```

Then, in your LATEX header, add

```
\usepackage[pdftex]{epsfig}
```

The benefit to this method is that you can now run `pdflatex my_document` without incident; the drawback is that you now have two versions of every figure cluttering up your directory, and must regenerate the PDF version of the graphic every time you regenerate the Postscript version.

The alternative is to go through Postscript in generating the document:

```
latex a_report
dvips < a_report.dvi > a_report.ps
ps2pdf a_report.ps
```

Either method is a lot of typing, but there is a way to automate the process: the `make` program, which is discussed in detail in Appendix A. Listing 5.14 is a sample makefile for producing a PDF document via the Postscript route. As with your C programs, once the makefile is in place, you can generate final PDF documents by just typing `make` at the command line. The creative reader could readily combine this makefile with the sample C makefile from page 387 to regenerate the final report every time the analysis or data are updated. (*Hint*: add a `gen_all` target that depends on the other targets. That target's actions may be blank.)

```
DOCNAME = a_report

pdf: $(DOCNAME).pdf

$(DOCNAME).dvi: $(DOCNAME).tex
        latex $(DOCNAME); latex $(DOCNAME)

$(DOCNAME).ps: $(DOCNAME).dvi
        dvips −f < $(DOCNAME).dvi > $(DOCNAME).ps

$(DOCNAME).pdf: $(DOCNAME).ps
        ps2pdf $(DOCNAME).ps $(DOCNAME).pdf

clean:
        rm −f $(DOCNAME).blg $(DOCNAME).log $(DOCNAME).ps
```

Listing 5.14 A makefile for producing PDFs from LaTeX documents. Online source: `Makefile.tex`.

$\sum$

- ➤ Once a plot looks good on screen, you can send it to an output file using the `set term` and `set out` commands.
- ➤ For printing, you will probably want to use `set term postscript`. For online presentation, use `set term png` or `set term gif`. For inserting into a LaTeX document, you can either use Postscript or `set term latex`.