Plots in LaTeX: Gnuplot, Octave, make

Boris Veytsman and Leyla Akhmadeeva

Abstract

Making scientific and engineering documents with complex plots may be difficult and time-consuming. This is especially true if data updates require rebuilding of plots and documents. In this report a workflow based on an integration of (La)TeX, Gnuplot and Octave using Makefiles in a Unix environment is proposed and discussed in detail.

1 Introduction

Some time ago one of us (BV) worked on a report about aircraft navigation accuracy. This report included about forty charts of predicted errors as depending on the aircraft altitude, location of surveillance radios, etc. Then a coworker came to the office to tell me that some parameters of the model had changed and requested replotting all the charts. "How long would it take to do it?", he asked with some trepidation, since the deadline was close. "Well", was the answer, "I have a rather slow computer here. Probably about two to three minutes." Having said this, the author changed several lines in one of the configuration files and typed `make`. In two minutes the machine happily produced an updated report.

This example illustrates a certain point about computers. They can take over the mind-numbing drudgery (like replotting dozens of charts) so we can do interesting things (like analyzing the message behind these charts). Unfortunately, for many people computers are just glorified typewriters/calculators/drawing devices, requiring constant hand holding and manual interventions. These users try to perform all the boring minute steps themselves, redoing them when anything changes. However, humans are not especially good at boring and repetitious work. They make mistakes. This often leads to embarrassing results (see, for example, the discussion of spreadsheet errors in Reinhart and Rogoff's paper by Herndon, Ash, and Pollin, 2013). It is much more rewarding to teach a computer to do such work for you.

In this paper we show how to teach your computer to make high quality plots for your papers and reports, and to remake them as needed. This involves a combination of TeX, Gnuplot or Octave, and Makefiles. The system is highly customizable, and rather easy to use. Of course, TeX is the heart of the system, and it was developed with TeX in mind on each step. Thus we hope it might be of interest to the *TUGboat* readership.

Some points should be made before we discuss this system. First, it was developed "in house", and thus reflects certain tastes and idiosyncrasies. Second, we started to work on it long time ago — before such tools as `latexmk` or Asymptote were available. Thus it uses only classic tools and has a certain "retro" computing spirit. Third, it was developed for a Unix-like environment.[1] One can get it working on Windows using any free implementation of `make`, but that is beyond the scope of this paper.

2 Gnuplot graphics

There are many choices for a plotting program suitable for a TeX user: PSTricks, PGF/Ti*k*Z, METAFONT, METAPOST, and Asymptote come to mind, as well as a plethora of non-free solutions. However for complex graphics, especially three-dimensional ones, Gnuplot is, in our opinion, among the best choices. It has the right combination of sound defaults (axis labeling, tick marks location, etc.) and the option of changing any default if needed. A full discussion of the rich possibilities of this program is also beyond our scope here. We recommend the extensive built-in help (try `help terminal epslatex`, for example) and the book by Janert (2009).

How can we include the graphics produced by Gnuplot into a TeX document? The simplest solution is to make Gnuplot output an EPS or PDF file and use `\includegraphics` to put this plot into the proper place. However, this idea has a number of flaws. First, the text on the graphics will be done in Helvetica and Symbol fonts, which may well clash with your body font. Second, you may want to use TeX for annotations inside the graphics.

Ti*k*Z provides a method for smooth integration of Gnuplot-produced plots in the `\tikzpicture` environment. However, it tends to replot all graphics whenever you change the TeX file, which might be time consuming.

Gnuplot has a number of TeX-compatible output formats ("terminals" in Gnuplot terminology). Probably the best choice for complex graphics is `epslatex` (or `pstex` for plain TeX). These terminals produce a `.tex` file that has all the labels, while the graphics are saved in a PostScript file, which is automatically called by the `.tex` file. An example of the usage of this terminal is shown in Figure 1. Let us discuss it in detail.

The first line sets the output format:

```
set terminal epslatex monochrome
```

The option `monochrome` is chosen here because the printer charges *TUGboat* extra for color pages. In most cases the `color` option is preferable. Note: even

[1] Including GNU/Linux and Mac OS X.

```
set terminal epslatex monochrome
set output "function-fig.tex"
set pm3d                # Colored surface
unset surface           # We do not want to plot the mesh lines
set isosamples 100, 100 # Smooth surface
set ztics 0.2           # Increment for z tick marks
set cbtics 0.2          # Increment for colored box
set format '$%g$'
set xtics offset 0,-.3
set ytics offset 1,0
set ztics offset -1,0
set cbtics offset 1,0
set xrange [-1.5:1.5]
set yrange [-1.5:1.5]
set label 1 \
 '$f(\mathbf{x})=\exp\left(-\lvert\mathbf{x}\rvert^2\right)$' \
 at -1.5,-1,1
set label 2 \
 '$\displaystyle\max_{\mathbf{x}\in \mathbb{R}^2} f(\mathbf{x})$' \
 at 1,1,1.3
set arrow 1 from 1,1,1.3 to 0,0,1 front
splot exp(-x**2-y**2) title ""
set output
```

Figure 1: A Gnuplot script `function.gp` with `epslatex` output

with the `monochrome` option, the package `color` or `xcolor` must be called by your main TEX file.

The next line sets the name of the output `.tex` file; we chose `function-fig.tex`. (We explain this naming convention below.)

The lines `set pm3d` and `unset surface` explain how to plot the three-dimensional graphics: using color (well, shades of gray for our monochrome display) to show the height and not plotting the surface mesh lines.

The line `set isosamples` sets the number of points where the function is calculated. We use $100 \times 100 = 10000$ points for a smooth plot.

The lines `set ztics` and `set cbtics` set the increment for the ticks on the z axis and the color box in the legend. We do not use similar commands for `xtics` and `ytics` since the defaults are good enough.

The line `set format '$%g$'` refers to the format of the tick marks. It makes the numbers to be typeset in the math mode. The default is `%g`, which should be familiar to those knowing C formatting commands. Thus normally Gnuplot typesets tick marks as text, so minus signs become dashes.

The next lines slightly move the tick numbers for the x, y, and z axes and the color box. (Gnuplot is not completely aware of the font metrics for TEX fonts, so its position calculations are sometimes not good enough for the demanding eyes of *TUGboat* editors.)

The lines `set xrange` and `set yrange` set the x and y domains for the plotting: from -1.5 to 1.5. We do not use a similar `set zrange` command since the default (based on the maximal and minimal values of the function plotted) looks good.

The next lines are `label` commands. They have three arguments: label number (`1` and `2` in our case), label text (enclosed within single quotes) and label position (`at` statements). The text in the labels is TEX code interpreted in the context of your main document. Thus you can put arbitrarily complex annotations on your graphics. Gnuplot provides some mechanisms for fine-tuning the label reference point; in most cases, however, you do not need to change the default.

We use an arrow from the label to the top of the plot. It is set by the `set arrow` command. Its arguments include arrow number, arrow start and arrow end. The last keyword, `front`, means that the arrow is plotted on the front layer of the picture, i.e., is not to be obscured by the plot itself.

The actual plot is done by the penultimate line:

```
splot exp(-x**2-y**2) title ""
```

It means: do a surface plot of the function $\exp(-x^2 - y^2)$, with an empty title (by default, Gnuplot typesets the formula as the title).

The last line, `set output`, writes the results to the output files and closes them. See Figure 2.

2.1 Plots from data points

In the above example we plotted a mathematical

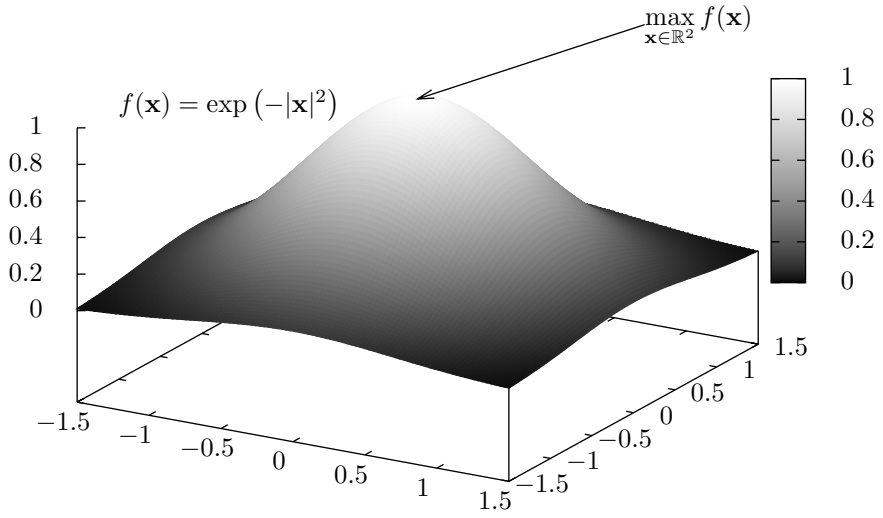

Figure 2: The plot generated by the script in Figure 1

expression. Gnuplot can also plot data from a file, obtained from an experiment or other calculations. As an example we show a script in Figure 3. Here we plot the stopping distances of cars moving with different speeds as measured in 1920s. The data is from the R distribution (R Core Team, 2013). We put them in a space-separated file `cars.dat`, which looks like this:

```
# "speed" "dist"
4 2
4 10
7 4
...
```

The first line shows the column names. It starts with the comment symbol `#` to tell Gnuplot not to try to plot it.

Most of the commands in Figure 3 are similar to those in Figure 1. Let us look at the ones that are different.

The line `set logscale xy` tells Gnuplot to create a log–log plot. The lines `set xlab` and `set ylab` set the labels for x and y axes correspondingly. The command `set label 1` contains a TEX expression that includes a mathematical formula and rotation to typeset the formula along the line it describes. All rotation and typesetting is done by TEX rather than by Gnuplot.

Since we make a two-dimensional plot rather than a three-dimensional one, we use the `plot` command rather than `splot`:

```
plot "cars.dat" with points pt 4 title "", \
   exp(-0.73+1.6*log(x)) title ""
```

This command has two arguments separated by a comma and corresponding to two objects we want to plot: a data file, plotted with points of

```
set terminal epslatex monochrome
set output "cars-fig.tex"
set logscale xy
set xrange [1:100]
set yrange [1:500]
set xlab 'Speed, mph'
set ylab 'Stopping distance, feet'
set format '$%g$'
set label 1 \
  '\rotatebox{41}{$\ln y=-0.73+1.6\ln x$}' \
  at 1.8, 4
plot "cars.dat" with points pt 4 title "", \
     exp(-0.73+1.6*log(x)) title ""
set output
```

Figure 3: Another Gnuplot script, `cars.gp`

type 4 (these happen to be unfilled squares), and a mathematical expression corresponding to a straight line on the log–log scale. The result is shown in Figure 4.

A good way to debug and tune the graphics is to comment out the first two lines of the script and run it through Gnuplot, thus seeing the results online, changing the script until you get a satisfactory result.

3 Octave graphics

Gnuplot's built-in features cover most mathematical needs. However, sometimes they are not enough. What can we do then? As discussed in the previous section, we can calculate the data points in an external program and feed the result to Gnuplot as a (space separated) text file. Another possibility is to use software that can talk to Gnuplot directly.

A good choice for this is Octave. Octave is a high level language and program for numerical calculations. It is similar to and mostly compatible with

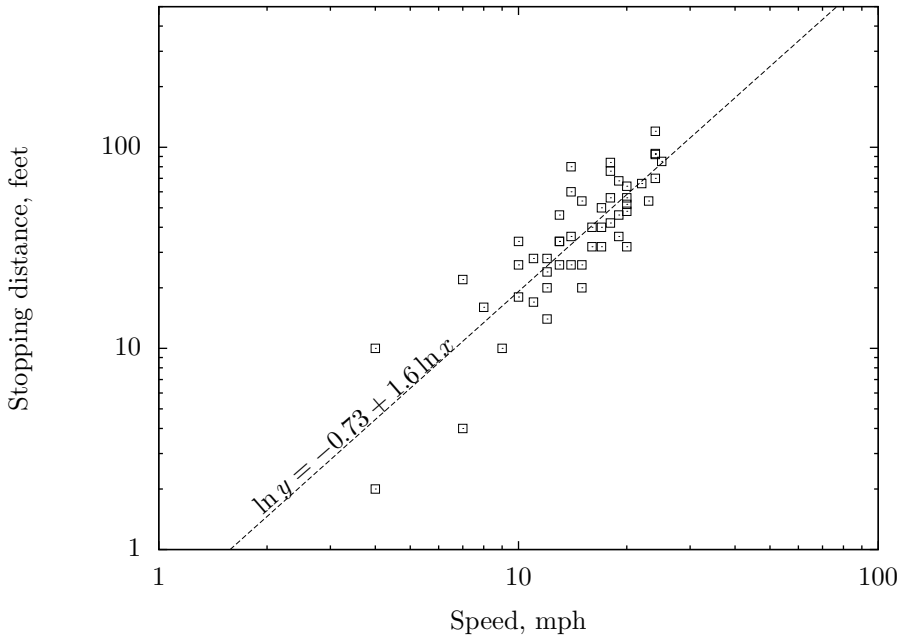

Figure 4: The plot generated by the `cars.gp` script in Figure 3

the commercial program MATLAB. As often happens with free software, some of Octave's capabilities surpass those of its commercial sibling. In particular, the technique of generating TEX-compatible graphics described in this article *does not* work in MATLAB. The latter can produce plots in the EPS format (Octave can do this too), but text annotations on these plots are done in its own fonts, which may clash with the body text.

The current version of Octave uses Gnuplot for graphics, so most of the features discussed in the previous section are available in Octave. However, the syntax is slightly different. The most important difference is the order of commands. In Gnuplot we first "set" the annotations: legend, labels, etc., and then plot the data. In Octave we plot the data first, and only then add annotations to the existing figure.

In Figures 5 and 6 we show a plot used in one of our reports. In this report we discussed the behavior of a certain system. It depended on a dimensionless parameter ρ. The report showed that this parameter ought to satisfy the following condition:

$$\mathrm{ber}_1 \rho = \mathrm{bei}_1 \rho$$

where ber_1 and bei_1 are so-called Kelvin functions, related to the Bessel function of complex argument (Olver, Lozier, Boisvert, and Clark, 2010, § 10.61). Thus we wanted a plot of the expression

$$\delta(\rho) = \mathrm{ber}_1 \rho - \mathrm{bei}_1 \rho$$

and the point ρ_0 for which $\delta(\rho_0) = 0$.

Unfortunately, Gnuplot does not know anything about Kelvin functions. An attempt to calculate them using Bessel functions of complex argument leads (at least in version 4.4) to the following truthful, but not especially helpful message: *can only do bessel functions of reals*. To tell the truth, Octave also knows nothing about Kelvin functions. However, unlike Gnuplot, it is not afraid of Bessel functions of complex argument.

In Figure 5 we show an Octave script `kelvin.m` that generates the required plot. Let us discuss the script in detail.

The first three lines define functions ber_1, bei_1 and δ using Octave's built-in `besselj` command. The next line, `rho0 = fsolve(delta, 4)` calculates the root of equation $\delta(x) = 0$ starting from the point $x = 4$, and assigns the result to the variable `rho0`. By the way, this root is $\rho_0 = 3.7727$.

The reason for the next command is the way Octave executes plot commands. In Gnuplot we first set up the "terminal", and only then draw the plot. Thus, at the time of plotting our computer already knows we want to save the result to a file and does not make on-screen plots. In Octave we first draw the plot on the screen and only then "save" the result. This slows down the execution. The command `figure('visible', 'off')` switches off this on-screen drawing.

The next few lines perform the actual plotting. Octave has two ways to make a plot of a function. One is the `fplot` command: `fplot(delta, [0,4])`

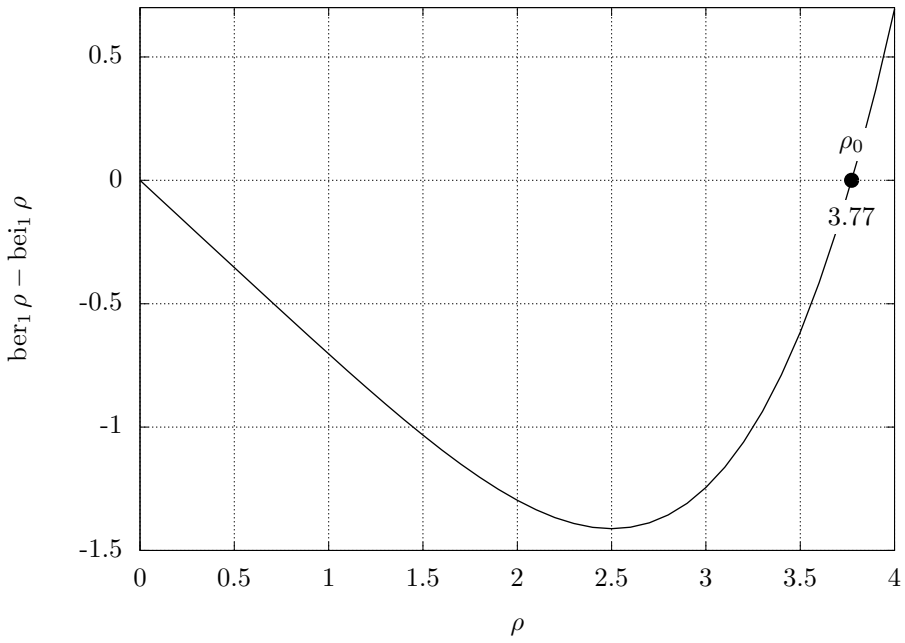

Figure 6: The plot generated by the `kelvin.m` script in Figure 5

```
ber1 = @(x) -real(besselj(1,x*exp(pi*1i/4)));
bei1 = @(x)  imag(besselj(1,x*exp(1i*pi/4)));
delta = @(x) ber1(x)-bei1(x);
rho0 = fsolve(delta,4);
figure('visible','off');
x = 0:0.1:4;
plot(x, delta(x), 'linewidth', 2);
hold on;
plot([rho0], [0], 'o', 'linewidth', 10);
text(rho0, 0.15, \
  '\colorbox{white}{$\rho_0$}', \
  'horizontalalignment', 'center');
text(rho0, -0.15, \
  sprintf("\\colorbox{white}{$%.2f$}", rho0), \
  'horizontalalignment', 'center');
title("");
legend("off");
grid();
xlabel('$\rho$');
ylabel('$\ber_1\rho-\bei_1\rho$');
print -depslatex -mono "-S800,600" \
      "kelvin-fig.tex"
```

Figure 5: An Octave script, `kelvin.m`

would make the graph shown in Figure 6. Unfortunately, in the current version of Octave the command `fplot` has rather limited options for fine control of the plot; in particular, it does not allow to change the default line width. The default lines look weak in our monochrome version. So here we use the generic `plot` command instead of `fplot`. We generate an array of abscissas with the command `x = 0:0.1:4`, and then plot `delta(x)` versus `x`. The command `plot(x, delta(x), 'linewidth', 2)` generates the plot with a line width of 2, meaning twice the default width.

Normally a `plot` or `fplot` command erases previous graphics and starts afresh. The line `hold on` preserves the graphic and combines it with the next plot. This next plot consists of only one point, a large dot at the zero of the function $\delta(x)$: `plot([rho0], [0], 'o', 'linewidth', 10);` The parameter `'o'` sets the shape of the marker, while setting the line width to 10 makes it large.

The next two commands, using `text(...)`, are similar to Gnuplot `label` commands. We set up the coordinates of the text and the text itself. Additional parameters help to tune the output.

The first command places the "text" string `\colorbox{white}{$\rho_0$}` at the point $(\rho_0, 0.15)$, i.e. above the root of our equation. The text is centered on the reference point (the other options for `horizontalalignment` are `left` and `right`). This "text" is a LaTeX command that creates the symbol ρ_0 inside a rectangle with white background (`colorbox`). We use this rectangle because otherwise the plot overlaps the symbol.

The second command is more complicated. Here the "text" is dynamically constructed by Octave itself. The function `sprintf` is similar to the function

of the same name in C, awk, perl, and many other languages. Its first argument is the "format", and the remaining arguments are interpreted according to this format. In our case we output a string which includes `rho0` typeset with two decimal figures per the specification `%.2f` in the format. The function returns the string[2] `\colorbox{white}{$3.77$}`, which is typeset centered at the point $(\rho_0, -0.15)$, i.e. under the root of the equation. We again use `\colorbox` to create the white background for the label.

The next commands are analogous to those in Gnuplot: we switch off the main title and legend, switch on the grid and set up the axes labels. Since `\ber` and `\bei` are *not* standard LaTeX operators, our `.tex` file has the following definitions based on the macro `\DeclareMathOperator` provided by the amsmath package:

```
\DeclareMathOperator{\ber}{ber}
\DeclareMathOperator{\bei}{bei}
```

The last line,

```
print -depslatex -mono "-S800,600" \
  "kelvin-fig.tex"
```

saves the graphs into the file `kelvin-fig.m`. It uses `epslatex` format (similar to Gnuplot's `epslatex` terminal), and `mono`-chrome rendering. The magic string `"-S800,600"` sets the size of the figure in points (the reason why it must be in quotes is better known to the authors of Octave).

Like Gnuplot, Octave can create complex three-dimensional graphics, which we leave as an exercise to the reader.

Unfortunately, we failed to find the analog of the Gnuplot `set format` command, so tick marks on Figure 6 are typeset in text mode: compare '-1' (wrong!) and '-1' (right). This problem can be corrected by a simple `sed` script acting on the file `kelvin-fig.m`, which we leave as another exercise to the reader.

4 Insertion of graphics in the `.tex` file

The methods described in the previous sections produce two files for each graphics: a `.tex` file with the textual material, and a PostScript file (either `.eps` or `.ps`) with the graphics material. For example, in the directory with this paper are the following files:

```
cars-fig.eps       cars-fig.tex
function-fig.eps   function-fig.tex
kelvin-fig.eps     kelvin-fig.tex
```

To use these graphics in the text, we "read in" the `.tex` file using the command `\input`, for example, `\input{function-fig}`. The associated PostScript file is automatically inserted by the `.tex` file with the corresponding `\includegraphics` command.

This works fine with the traditional `latex` with `dvips` route. What happens if you use `pdflatex`? Fortunately, recent distributions are smart enough to automatically convert `.eps` files to `.pdf` (using `epstopdf`), so after a run of `pdflatex` you can find in the working directory the files

```
cars-fig-eps-converted-to.pdf
function-fig-eps-converted-to.pdf
kelvin-fig-eps-converted-to.pdf
```

This conversion is done transparently to the user.[3] Of course, the PostScript files must have a correct bounding box for correct results.

5 Putting everything together: Makefiles

The process described in the previous sections may seem rather complex. We run Gnuplot and/or Octave, `latex`, `dvips`, `ps2pdf`, `pdflatex`, ... If we change some of the files, we need to rerun the necessary portions of the process. A human should not do this manually (and probably cannot do it without introducing errors).

The famous utility `make` can do this for you.

Let us recall the basics. The utility reads a *Makefile* which sets up rules and dependencies. Rules tell it how to "make" a certain file: you run `latex` on a `.tex` file to generate a `.dvi` file, you run `dvips` on a `.dvi` file to generate a `.ps` file, etc. Dependencies record that a file A *depends* on the file B: whenever B is changed, A must be regenerated. You can find more information, for example, in the classic book by Mecklenburg (2004).

In this section we set up a typical Makefile for TeX and Gnuplot or Octave.

Let us start with Gnuplot. We will use the extension `.gp` for our Gnuplot scripts and the following naming convention: a file `file.gp` generates the files `file-fig.tex` and `file-fig.eps`. The part `-fig` is used to set up the `clean` task: to clean the directory we delete all generated files.

So, we can define the following simple rule: each `file-fig.tex` file depends on the `file.gp`, and `gnuplot` is used to generate it:

```
%-fig.tex: %.gp
        gnuplot $<
```

Here, according to Makefile syntax, `%` is a "wildcard", and `$<` means the "prerequisite" (the `.gp` file).

Here is a similar rule for Octave-generated files:

```
%-fig.tex: %.m
        octave $<
```

[2] Exercise: why does the format use double backslash?

[3] Unfortunately, Gnuplot's `pstex` terminal for plain TeX uses PostScript `special`s instead of `\includegraphics`. This makes the technique described here inapplicable.

So now we have two rules for generation of ⟨*file*⟩`-fig.tex` files: either from Gnuplot or from Octave. Happily, `make` is smart enough to choose the right one: if it finds an appropriate file ending in `.gp`, it uses the first rule, and if it finds an appropriate file ending in `.m`, it uses the second.[4]

Let us now discuss the generation of PDF files from `.tex` sources. In this article we discuss the `pdflatex` route; the rules for the "traditional" `latex` → `dvips` route are left as another exercise.

The basic idea is relatively simple: run `pdflatex` until the labels converge. The code below has an additional quirk of running `bibtex` several times because citations may use `crossref` fields:

```
%.pdf: %.tex
        pdflatex $*
        - bibtex $*
        pdflatex $*
        - while ( grep -q \
'^LaTeX Warning: Label(s) may have changed' \
$*.log ) do (bibtex $*; pdflatex $*;) done
```

Of course, this is not the full story. We need to tell `make` that whenever a plot is changed, all PDF files must be regenerated. This can be done by adding to the Makefile lines like

```
document.pdf: plot-fig.tex
```

for each `\input{plot-fig}` line.

This line tells `make` to regenerate the main PDF (using the rule above) when `plot-fig.tex` changes; the companion `plot-fig.eps` could be added as another dependency, but since the two `plot-fig.*` files are always created simultaneously, it's not necessary.

What about the conversion `.eps`→`.pdf`? Will that be done as well? The answer is yes. TEX uses a simple but sufficient algorithm for this conversion: whenever `.eps` file is newer than the generated `.pdf` file, the latter is regenerated. Thus, after you change `plot.gp`, one line in the Makefile triggers a number of events:

1. The program `make` finds the new `plot.gp` and calls Gnuplot to regenerate `plot-fig.tex`.
2. As a side effect the file `plot-fig.eps` is recreated by Gnuplot.
3. TEX finds the new `plot-fig.eps` and generates a new `plot-fig-eps-converted-to.pdf`.
4. The new version of the main PDF file is created.

If you have many plots, you might find it cumbersome to add a dependency for each. Fortunately, Makefiles can include subfiles, allowing us to automatically generate the dependencies. Each line will have the form `A: B`, showing that file A depends on file B. The way to do this is the following:

1. Add to the Makefile the line `-include depend`, which instructs `make` to read the file *depend* if it exists. The dash at the beginning tells `make` not to worry if this file is not found (e.g. at the start of the run).
2. Add to the Makefile the rules to generate the file `depend` from the sources.

For the second task we need to write a program to generate the dependency file. A simple Perl script `makefigdepend.pl` to do this is shown in Figure 7. (The choice of Perl makes our solution less "retro" than it could be: `sed` and `awk` could do the job.) Then the rule

```
depend: ${TEXFILES}
        perl makefigdepend.pl \
        ${TEXFILES} >depend
```

generates the required file. For example, the file `depend` for this article is the following:[5]

```
gnuplotmk.pdf: function-fig.tex
gnuplotmk.pdf: cars-fig.tex
gnuplotmk.pdf: kelvin-fig.tex
gnuplotmk.pdf: function-fig.tex
```

When we plot a data file, we want `make` to redo the plot not only when the `.gp` file is new, but also if the original data change. To this end, we add a line to the Makefile:

```
cars-fig.tex: cars.dat
```

An astute reader can see that we wrote this dependency manually. It is of course possible to write a

```
#!/usr/bin/env perl
# Extract information from input statements
# in TeX files. Usage:
# makefigdepend FILE FILE FILE ... > depend

foreach my $file (@ARGV) {
 open (FILE, $file) || die "open($file): $!";
 $file =~ s/\.tex$/.pdf/;
 while (<FILE>) {
  while (/\\input(?:\[[^\]]+\])*\{([^\}]+)\}/g) {
   print "$file: $1.tex\n";
  }
 }
 close FILE;
}
exit 0;
```

Figure 7: A Perl script for generation of dependencies

[4] If both Gnuplot and Octave files are present, `make` chooses the rule that appears first in the Makefile.

[5] An exercise: why is an identical line about `function-fig.tex` repeated in the generated file?

```
TEXFILES = gnuplotmk.tex
PDFS = ${TEXFILES:%.tex=%.pdf}

all: ${PDFS}

%.pdf:  %.tex
        pdflatex $*
        - bibtex $*
        pdflatex $*
        - while ( grep -q \
'^LaTeX Warning: Label(s) may have changed' \
                    $*.log ) \
        do (bibtex $*; pdflatex $*;) done

%-fig.tex: %.gp
        gnuplot $<
%-fig.tex: %.m
        octave $<

cars-fig.tex: cars.dat

clean:
        $(RM) *.aux *.bbl *.dvi *.log \
        *.out *.toc *.blg *.lof *.lot \
        *.eps *-fig* depend
distclean: clean
        $(RM) ${PDFS}

depend:  ${TEXFILES}
        perl makefigdepend.pl \
        ${TEXFILES} > depend
-include depend
```

Figure 8: Makefile for this paper

script to parse the Gnuplot files and put such lines in the file `depend`. Again, this is left as an exercise.

If the data files themselves are generated by another program, as would be typical, we can tell `make` to run this program if necessary by adding the dependencies of data files upon the necessary input parameters. This leads to incredibly smart behavior: as soon as any of the configuration or data files change, `make` regenerates all pieces that could be influenced by this change.

The last task is *cleaning*. A common convention is to provide two targets: `clean` removes all generated files except the principal (PDF) output, while `distclean` or `veryclean` deletes everything but the original sources:

```
clean:
        $(RM) *.aux *.bbl *.dvi *.log \
        *.out *.toc *.blg *.lof *.lot \
        *.eps *-pics.* *-fig* depend
distclean: clean
        $(RM) ${PDFS}
```

In Figure 8 we show the Makefile for this paper.

Acknowledgements

We are grateful to Marcel Richter who urged us to put together the notes in a readable form, and to Michael Kotelyanskii who made many useful comments about the manuscript.

References

Herndon, Thomas, M. Ash, and R. Pollin. "Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff". Working Paper 322, University of Massachusetts, Amherst. Political Economy Research Institute, 2013. `http://www.peri.umass.edu/fileadmin/pdf/working_papers/working_papers_301-350/WP322.pdf`.

Janert, Philipp K. *Gnuplot in Action. Understanding Data with Graphs*. Manning Publications Co., 2009.

Mecklenburg, Robert. *Managing Projects with GNU Make*. O'Reilly Media Inc., Sebastopol, CA, third edition, 2004. Available at `http://oreilly.com/catalog/make3/book/index.csp`.

Olver, F. W. J., D. W. Lozier, R. F. Boisvert, and C. W. Clark, editors. *NIST Handbook of Mathematical Functions*. Cambridge University Press, New York, NY, 2010.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

◇ Boris Veytsman
School of Systems Biology &
Computational Materials
Science Center, MS 6A2
George Mason University
Fairfax, VA 22030
`borisv (at) lk dot net`

◇ Leyla Akhmadeeva
Bashkir State Medical University
3 Lenina Str., Ufa, 450000, Russia
`la (at) ufaneuro dot org`
`http://www.ufaneuro.org`