D. Bates and D. Sarkar. *lme4: Linear mixed-effects models using S4 classes*, 2006. R package version 0.995-2. 29

D. Böhning, P. Schlattmann, and B. Lindsey. Computer-assisted analysis of mixtures (C.A.MAN): statistical algorithms. *Biometrics*, 48: 283–303, 1992. 26

G. Brush and G. A. Harrison. Components of growth variation in human stature. *Mathematical Medicine and Biology*, 7:77–92, 1990. 28

J. Einbeck and J. Hinde. A note on NPML estimation for exponential family regression models with unspecified dispersion parameter. *Austrian Journal of Statistics*, 35:233–243, 2006. 29

J. Einbeck, R. Darnell, and J. Hinde. *npmlreg: Nonparametric maximum likelihood estimation for random effect models*, 2006. R package version 0.40. 26

N.M. Laird. Nonparametric maximum likelihood estimation of a mixing distribution. *Journal of the American Statistical Association*, 73:805–811, 1978. 26

J. Pinheiro, D. Bates, S. DebRoy, and D. Sarkar. *nlme: Linear and nonlinear mixed effects models*, 2005. R package version 3.1-66. 28

A. Skrondal and S. Rabe-Hesketh. *Generalized Latent Variable Modelling*. Chapman & Hall/CRC, Boca Raton, 2004. 26

N. Sofroniou, J. Einbeck, and J. Hinde. Analyzing Irish suicide rates with mixture models. In *IWSM 2006: Proceedings of the 21th International Workshop on Statistical Modelling, Galway*, pages 474–481, 2006. 28

R. Tsutakawa. Estimation of cancer mortality rates: a Bayesian analysis of small frequencies. *Biometrics*, 41:69–79, 1985. 27

Jochen Einbeck
Durham University, UK
`jochen.einbeck@durham.ac.uk`
John Hinde
National University of Ireland, Galway, Rep. of Ireland
`john.hinde@nuigalway.ie`
Ross Darnell
The University of Queensland, Australia
`r.darnell@uq.edu.au`

Augmenting R with Unix Tools

Andrew Robinson

This article describes a small collection of Unix command-line tools that can be used to augment R. I introduce:

`make` which after judicious preparation will allow one-word execution of large-scale simulations, all the way from compilation of source code to construction of a PDF report;

`screen` which will permit remote monitoring of program execution progress with automatic protection against disconnection; and

`mail` which will allow automatic alerting under error conditions or script completion.

These programs will be part of the default installations of many flavours of Unix-like operating systems. Although these tools have many different applications, each is very useful for running R remotely on a Unix-like operating system, which is the focus of their usage in this article.

Regardless of what kind of computer is on your own desk, you may find these tools useful; they do not require you to be running a BSD or GNU/Linux on your own machine.

R and `screen`

It is sometimes necessary to run R code that executes for long periods of time upon remote machines. This requirement may be because the local computing resources are too slow, too unstable, or have insufficient memory.

For example, I have recently completed some simulations that took more than a month on a reasonably fast cluster of Linux computers. I developed, trialed, and profiled my code on my local, not-particularly-fast, computer. I then copied the scripts and data to a more powerful machine provided by my employer and ran the simulations on R remotely on that machine. To get easy access to the simulations whilst they ran, I used `screen`.

`screen` is a so-called terminal multiplexor, which allows us to create, shuffle, share, and suspend command line sessions within one window. It provides protection against disconnections and the flexibility to retrieve command line sessions remotely. `screen` is particularly useful for R sessions that are running on a remote machine.

We might use the following steps to invoke R within `screen`:

1. log in remotely via secure shell,

2. start `screen`,
3. start R,
4. `source` our script with `echo=TRUE`,
5. detach the screen session, using `Ctrl-a d`, and
6. log out.

The R session continues working in the background, contained within the `screen` session. If we want to revisit the session to check its progress, then we

1. log in remotely via secure shell,
2. start `screen -r`, which recalls the unattached session,
3. examine the saved buffer; scrolling around, copying and pasting as necessary,
4. detach the screen session, using `Ctrl-a d`, and
5. log out.

This approach works best if examining the buffer is informative, which requires that the R script be written to provide readable output or flag its progress every so often. I find that the modest decrease in speed of looping is more than compensated by a cheerful periodic reminder, say every 1000^{th} iteration, that everything is still working and that there are only n iterations left to run. I have also been experimenting with various algorithms for R to use the elapsed time, the elapsed number of simulations, and the number of simulations remaining, to estimate the amount of time left until the run is complete.

`screen` offers other advantages. You can manage several screens in one window, so editing a script remotely using, say, `emacs`, and then sourcing the script in the remote R session in another screen, is quick and easy. If you lose your connection, the session is kept alive in the background, and can be re-attached, using `screen -r` as above. If you have forgotten to detach your session you can do so forcibly from another login session, using `screen -dr`. You can change the default history/scrollback buffer length, and navigate the buffer using intuitive keystrokes. Finally, you can share a `screen` session with other people who are logged in to the same machine. That is, each user can type and see what the other is typing, so a primitive form of online collaboration is possible.

So, running an R session in `screen` provides a simple and robust way to allow repeated access to a simulation or a process.

More information about `screen` can be found from `http://www.gnu.org/software/screen/`, or `man screen`.

R and `mail`

It is very useful to be able to monitor an R session that is running remotely. However, it would also be useful if the session could alert us when the script has completed, or when it has stopped for some reason, including some pre-determined error conditions. We can use `mail` for this purpose, if the computer is running an appropriate mail server, such as `sendmail`.

`mail` is an old email program, small enough to be included by default on most Unix-like systems, and featureless enough to be almost universally ignored by users in favour of other programs, such as `mutt`, or those with shiny graphical user interfaces. However, `mail` does allow us to send email from the command line. And, R can do pretty much anything that can be done from the command line[1]. Of course, a small amount of fiddling is necessary to make it work. A simple function will help[2]. This function will fail if any of the arguments contain single or double quotes. So, craft your message carefully.

```
mail <- function(address, subject, message) {
  system(paste("echo '", message,
  "' | mail -s '", subject,
  "' ", address, sep=""))
}
```

We can now send mail from R via, for example,

```
mail("andrewr", "Test", "Hello world!")
```

You can place calls to this function in strategic locations within your script. I call it at the end of the script to let me know that the simulations are finished, and I can collect the results at my leisure. In order to have R let us know when it has stopped, we can use a function like this:

```
alert <- function() {
      mail("andrewr", "Stopped", "Problem")
      browser()
}
```

then in our script we call

```
options(error = alert)
```

Now in case of an error, R will send an email and drop into the browser, to allow detailed follow-up. If you have more than one machine running, then calling `hostname` via `system`, and pasting that into the subject line, can be helpful.

Alternatively, you can use `mutt` with exactly the same command line structure as `mail`, if `mutt` is installed on your system. An advantage of `mutt` is that it uses the MIME protocol for binary attachments. That would enable you to, say, attach to your email the PDF that your script has just created with `Sweave` and `pdflatex`, or the *cvs* file that your script creates,

[1]In theory, it can do everything, I suppose. I haven't tried.
[2]Caveat: this function is written to work on bash version 3.1.17. Your mileage may vary.

or even the relevant objects saved from the session, neatly packaged and compressed in a `*.RData` object.

Different flavours of Unix `mail` are available. You can find out more about yours by `man mail`.

R and `make`

I have recently been working on bootstrap tests of a model-fitting program that uses maximum likelihood to fit models with multivariate normal and *t* distributions. The program is distributed in FORTRAN, and it has to be compiled every time that it is run with a new model. I wanted to use a studentized bootstrap for interval estimates, gather up all the output, and construct some useful tables and graphics.

So, I need to juggle FORTRAN files, FORTRAN executables, R source files, Sweave files, PDFs, and so on. R does a great job preparing the input files, calling the executable (using `system()`), and scooping up the output files. However, to keep everything in sync, I use `make`.

Although it is commonly associated with building executable programs, `make` can control the conversion of pretty much any file type into pretty much any other kind of file type. For example, `make` can be told how to convert a FORTRAN source file to an executable, and it will do so if and when the source file changes. It can be told how and when to run an R source file, and then how and when to call Sweave upon an existing file, and then how and when to create a PDF from the resulting LaTeX file.

The other advantage that `make` offers is splitting large projects into chunks. I use Sweave to bind my documentation and analysis tightly together. However, maintaining the link between documentation and analysis can be time-consuming. For example, when documenting a large-scale simulation, I would rather not run the simulation every time I correct a spelling mistake.

One option is to tweak the number of runs. `make` provides a flexible infrastructure for testing code, as we can pass parameters, such as the number of runs to perform, to R. For example, with an appropriate Makefile, typing `make test` at the operating system prompt will run the entire project with only 20 simulations, whereas typing `make` will run the project with 2000 simulations.

Another option is to split the project into two parts: the simulation, and the analysis, and run only the second, unless important elements change in the first. Again, this sort of arrangement can be constructed quite easily using `make`.

For example, I have an R source file called `sim.r` that controls the simulations and produces a .RData object called `output.RData`. The content is:

```
randoms <- runif(reps)
save.image("output.RData")
```

I also have a Sweave file called `report.rnw` which provides summary statistics (and graphics, not included here) for the runs. The content is:

```
\documentclass{article}
\begin{document}
<<>>=
load("output.RData")
@
We performed \Sexpr{reps} simulations.
\end{document}
```

The latter file often requires tweaking, depending on the output. So, I want to separate the simulations and the report writing, and only run the simulations if I absolutely have to. Figure 1 is an example Makefile that takes advantage of both options noted above. All the files referenced here are assumed to be held in the same directory as the Makefile, but of course they could be contained in subdirectories, which is generally a neater strategy.

I do not actually need `make` to do these tasks, but it does simplify the operation a great deal. One advantage is that `make` provides conditional execution without me needing to fiddle around to see what has and has not changed. If I now edit `report.rnw`, and type `make`, it won't rerun the simulation, it will just recompile `report.rnw`. On the other hand, if I make a change to `sim.r`, or even run `sim.r` again (thus updating the `.RData` object), `make` will rerun everything. That doesn't seem very significant here, but it can be when you're working on a report with numerous different branches of analysis.

The other advantage (which my Makefile doesn't currently use) is that it provides wildcard matching. So, if I have a bunch of Sweave files (one for each chapter, say) and I change only one of them, then `make` will identify which one has changed and which ones depend on that change, and recompile only those that are needed. But I don't have to produce a separate line in the Makefile for each file.

Also, the Makefile provides implicit documentation for the flow of operations, so if I need to pass the project on to someone else, all they need to do is call `make` to get going.

More information on `make` can be found from `http://www.gnu.org/software/make/`.

Andrew Robinson
Department of Mathematics and Statistics
University of Melbourne
Australia
`A.Robinson@ms.unimelb.edu.au`

```
## Hashes indicate comments.  Use these for documentation.
## Makefiles are most easily read from the bottom up, the first time!

# 'make' is automatically interpreted as 'make report.pdf'.

# Update report.pdf whenever report.tex changes, by running this code.

report.pdf : report.tex
	( \
	\pdflatex report.tex; \
	while \grep -q "Rerun to get cross-references right." report.log;\
	do \
		\pdflatex report.tex; \
	done \
	)

# Update report.tex whenever report.rnw *or* output.RData changes,
# by running this code.

report.tex : report.rnw output.RData
	echo "Sweave(\"report.rnw\")" | R --no-save --no-restore

# Update output.RData whenever sim.r changes, by running this code.

output.RData : sim.r
	echo "reps <- 2000; source(\"sim.r\")" | R --no-save --no-restore

########################################################################

## The following section tells make how to respond to specific keywords.

.PHONY: test full neat clean

# 'make test' cleans up, runs a small number of simulations,
# and then constructs a report

test:
	make clean;
	echo "reps <- 10; source(\"sim.r\")" | R --no-save --no-restore;
	make

# 'make full' cleans up and runs the whole project from scratch

full :
	make clean;
	make

# 'make neat' cleans up temporary files - useful for archiving

neat :
	rm -fr *.txt *.core fort.3 *~ *.aux *.log *.ps *.out

# 'make clean' cleans up all created files.

clean :
	rm -fr *.core fort.3 *~ *.exe *.tex *.txt *.lof *.lot *.tex \
		*.RData *.ps *.pdf *.aux *.log *.out *.toc *.eps
```

Figure 1: The contents of a Makefile. Note that all the indenting is done by means of tab characters, not spaces. This Makefile has been tested for both GNU make and BSD make.