G. Grothendieck and T. Petzoldt. R help desk: Date and time classes in R. *R News*, 4(1):29–32, 2004. 34

D. Harte. Documentation for the Statistical Seismology Library. Technical Report 98-10, School of Mathematical and Computing Sciences, Victoria University of Wellington, Wellington, New Zealand, 1998. 31

D. Harte. *Multifractals: Theory and Applications*. Chapman and Hall/CRC, Boca Raton, 2001. 34

D. Harte. *Package PtProcess: Time Dependent Point Process Modelling*. Statistics Research Associates, Wellington, New Zealand, 2004. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/pp.pdf`. 32, 33

D. Harte. *Package Fractal: Fractal Analysis*. Statistics Research Associates, Wellington, New Zealand, 2005a. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/fractal.pdf`. 34

D. Harte. *Package ssBase: Base Functions for SSLib*. Statistics Research Associates, Wellington, New Zealand, 2005b. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/base.pdf`. 31, 34

D. Harte. *Package ssEDA: Exploratory Data Analysis for Earthquake Data*. Statistics Research Associates, Wellington, New Zealand, 2005c. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/eda.pdf`. 32

D. Harte. *Package ssM8: M8 Earthquake Forecasting Algorithm*. Statistics Research Associates, Wellington, New Zealand, 2005d. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/m8.pdf`. 34

D. Harte. *Users Guide for the Statistical Seismology Library*. Statistics Research Associates, Wellington, New Zealand, 2005e. URL `http://homepages.paradise.net.nz/david.harte/SSLib/Manuals/guide.pdf`. 31

V. Keilis-Borok and V. Kossobokov. Premonitory activation of earthquake flow: algorithm M8. *Phys. Earth & Planet. Int.*, 61:73–83, 1990. 34

T. Lay and T. Wallace. *Modern Global Seismology*. Academic Press, San Diego, 1995. 31, 32, 33

R. Peng. Multi-dimensional point process models in R. *Journal of Statistical Software*, 8(16):1–24, 2003. ISSN 1548-7660. URL `http://www.jstatsoft.org`. 33

B. Ripley and K. Hornik. Date-time classes. *R News*, 1(2):8–11, 2001. 34

D. F. Swayne, D. Cook, and A. Buja. XGobi: Interactive dynamic data visualization in the X window system. *Journal of Computational and Graphical Statistics*, 7(1):113–130, 1998. ISSN 1061-8600. URL `http://www.research.att.com/areas/stat/xgobi/`. 32

T. Utsu and Y. Ogata. Statistical analysis of seismicity. In J. Healy, V. Keilis-Borok, and W. Lee, editors, *Algorithms for Earthquake Statistics and Prediction*, pages 13–94. IASPEI, Menlo Park CA, 1997. 33

Ray Brownrigg
Victoria University of Wellington
ray@mcs.vuw.ac.nz
David Harte
Statistics Research Associates
david@statsresearch.co.nz

Literate programming for creating and maintaining packages

Jonathan Rougier

Outline

I describe a strategy I have found useful for developing large packages with lots of not-obvious mathematics that needs careful documentation. The basic idea is to combine the 'noweb' literate programming tool with the Unix 'make' utility. The source of the package itself has the usual structure, but with the addition of a `noweb` directory alongside the `R` and `man` directories. For further reference below, the general directory tree structure I adopt is

$$\begin{tikzcd}
 & \text{pkgName} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{R} & \text{man, \ldots} & \text{inst} \arrow[d] & \text{noweb} \arrow[d] \\
 & & \text{doc} & \text{doc}
\end{tikzcd}$$

where '. . . ' denotes other optional directories such as `src` and `data`.

The files in the `noweb` directory are the entry-point for *all* of the documented code in the package, including `*.R` files, `*.Rd` files, low-level source code, datasets, and so on. The `noweb` tool, under the control of the file 'Makefile' and the Unix `make` utility, is used to strip the files in the `noweb` directory into the appropriate places in the other directories. Other targets in 'Makefile' also handle things like building the package, installing it to a specific location, cleaning up and creating various types of documentation.

A complete example R package is available at http://maths.dur.ac.uk/stats/people/jcr/myPkg.tar.gz containing the code below, and a bit more to make 'myPkg' into a proper R package.

Literate programming and `noweb`

The basic idea of *literate programming* is that when writing complicated programs, it makes a lot of sense to keep the code and the documentation of the code together, in one file. This requires a mechanism for treating the file one way to strip out and assemble the code, and another way to create the documentation. With such a mechanism in place, the functionality of the code can be made much more transparent—and the documentation much more informative—because the code itself can be written in a modular or 'bottom-up' fashion, and then assembled in the correct order automatically.

I use the `noweb` literate programming tool. A `noweb` file looks like a LaTeX file with additional environments delimited by `<<name of environment>>=` and `@`. These define code chunks that appear specially-formatted in the documentation, and which get stripped out and assembled into the actual code. Here is an example of a minimal `noweb` file, called, say, 'myDoc1.nw'.

```
\documentclass[a4paper]{article}
\usepackage{noweb}\noweboptions{smallcode}
\pagestyle{noweb}

\begin{document}

Here is some \LaTeX\ documentation.  Followed
by the top-level source code.
<<R>>=
# The source of this file is noweb/myDoc1.nw
<<foo function>>
<<bar function>>
@

Now I can describe the [[foo]] function in more
detail, and then give the code.
<<foo function>>=
"foo" <- function(x) 2 * x^2 - 1
@

And here is the [[bar]] function.
<<bar function>>=
"bar" <- function(y) sqrt((1 + y) / 2)
@

% stuff for man page not shown
\end{document}
```

The `[[name]]` syntax is used by `noweb` to highlight variable names. There are many additional features of `noweb` that are useful to programmers. More information can be found at the `noweb` homepage, http://www.eecs.harvard.edu/~nr/noweb/.

Tangle

`noweb` strips out the code chunks and assembles them using the 'notangle' command. One useful feature is that it can strip out chunks with specific identifiers using the `-R` flag. Thus the command `notangle -RR myDoc1.nw` will strip all the code chunks identified directly or indirectly by `<<R>>=`, assembling them in the correct order. The result of this command is the file

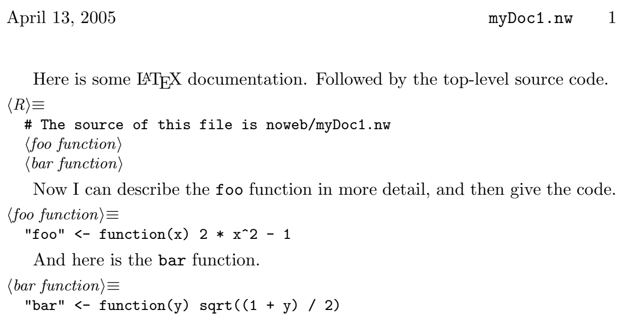
```
# The source of this file is noweb/myDoc1.nw
"foo" <- function(x) 2 * x^2 - 1
"bar" <- function(y) sqrt((1 + y) / 2)
```

which is written to standard output. By redirecting the standard output we can arrange for this file to go in the appropriate place in the package directory tree. So the full command in this case would be `notangle -RR myDoc1.nw > ../R/myDoc1.R`. So as long as we are consistent in always labelling the 'top' chunk of the R code with an `<<R>>=` identifier, we can automate the process of getting the R code out.

If we want, we can do the same thing using `<<man>>=` for the content of `man` pages, `<<src>>=` for low-level code, `<<data>>=` for datasets, and so on. Each of these can be stripped out and saved as a file into the appropriate place in the package directory tree.

Weave

To build the documentation, `noweb` uses the 'noweave' command, where it is useful to set the `-delay` option. Again, this is written to the standard output, and can be redirected to place the resulting LaTeX file at the appropriate place in the package tree. I put the LaTeX versions of the `*.nw` files in the directory `noweb/doc`. Thus the appropriate command is `noweave -delay myDoc1.nw > doc/myDoc1.tex`. This file contains quite a lot of LaTeX commands inserted by `noweb` to help with cross-referencing and indexing, and so is not particularly easy to read. But after using `latex`, the result is a `dvi` file which integrates the LaTeX documentation and the actual code, in a very readable format.


April 13, 2005 myDoc1.nw 1

Here is some LaTeX documentation. Followed by the top-level source code.

⟨*R*⟩≡
```
# The source of this file is noweb/myDoc1.nw
⟨foo function⟩
⟨bar function⟩
```
Now I can describe the `foo` function in more detail, and then give the code.

⟨*foo function*⟩≡
```
"foo" <- function(x) 2 * x^2 - 1
```
And here is the `bar` function.

⟨*bar function*⟩≡
```
"bar" <- function(y) sqrt((1 + y) / 2)
```

Similarity to 'Sweave'

Many users of R will be familiar with Friedrich Leisch's 'Sweave' function, available in the **tools** package. `Sweave` provides a means of embedding R code into a report, such as a LaTeX report, and then automatically replacing this code with its output. It uses the `notangle` functionality of a literate programming tool like `noweb`, because the R code must be stripped out, prior to evaluating it. The difference is that when `Sweave` builds the resulting document it does more than `noweave` would do, because it must actually evaluate R on each code chunk and put the *output* back into the document. In the literate programming described in this article, all that `notangle` does is wrap the code chunk in a LaTeX command to format it differently. `Sweave` can duplicate the simpler behaviour of `notangle` with the arguments `echo=TRUE,eval=FALSE` in each code chunk: see the help file for the function 'RWeaveLatex' for more details.

Using `make`

Our starting point is a collection of `*.nw` files in the `noweb` subdirectory. We could, by hand, issue a series of `notangle` and `noweave` commands, each time we update one or more of these files. Happily, the Unix tool `make` can be used instead. `make` requires a file of specifiers, often called 'Makefile', which describe the kinds of things that can be made, and the commands necessary to make them. Obvious candidates for things we would like to make are the `*.R` and `*.Rd` files that go into the `R` and `man` subdirectories, and the documentation that goes into the `inst/doc` subdirectory.

The `make` utility is very powerful, and complicated. I am not an expert at writing a 'Makefile', but the following approach seems to work well. My `Makefile` lives in the `noweb` directory, and all of the path specifiers take this as their starting point.

Simple `make` targets

Suppose that we have compiled a list of files that ought to be in the `R` subdirectory (say, 'RFILES'), and we want to use the command 'make R' to update these files according to changes that have occurred in the `noweb/*.nw` files; the argument `R` is known as a 'target' of the `make` command. We need the following lines in `Makefile`:

```
# make R files in ../R

R:      $(RFILES)

$(RFILES):      %.R : %.nw
        @echo 'Building R file for $<'
        @notangle -RR $< > ../R/$@
        @if test `egrep '^<<man>>=' $<` ; then \
          notangle -Rman $< > ../man/$*.Rd ; fi
```

After the comment, prefixed by `#`, the first line states that the command 'make R' depends on the files in the list 'RFILES', which we have already compiled. The next set of lines is executed for each component of 'RFILES' in turn, *conditionally on the* `*.R` *file being older than the corresponding* `*.nw` *file*. Where the `*.R` file is out-of-date in this way, a message is printed, `notangle` is called, and the `*.R` file is rebuilt and placed in the `R` subdirectory; if there is a `<<man>>=` entry, the `*.Rd` file is also rebuilt and placed in the `man` subdirectory.

A 'Makefile' has its own syntax. Within a target specification, the tags '$<', '$@' and '$*' indicate the origin file, the result file, and the file stem. Under each initial line the collection of subsequent commands is indented by tabbing, and the backslash at the end of the line indicates that the next line is a continuation. The `@` at the start of each command line suppresses the printing of the command to the standard output.

The attractive feature of `make` is that it only rebuilds files that are out-of-date, and it does this all automatically using information in the file date-stamps. This saves a lot of time with a large package in which there are many `*.nw` files in the `noweb` directory. In my experience, however, one feels a certain loss of control with so much automation. The useful command 'make --dry-run R' will print out the commands that will be performed when the command 'make R' is given, but not actually do them, which is a useful sanity check. The command 'touch myDoc1.nw' will change the date-stamp on 'myDoc1.nw', so that all derived files such as 'myDoc1.R' will automatically be out-of-date, which will force a rebuild even if 'myDoc1.nw' has not been modified: this can be another useful sanity-check.

Suppose instead we want to rebuild the LaTeX documentation in the `noweb/doc` subdirectory, using the command 'make latex'. We need the following lines in 'Makefile':

```
# make latex files in doc

latex:  $(TEXFILES)

$(TEXFILES):    %.tex : %.nw
        @echo 'Building tex file for $<'
        @noweave -delay $< > doc/$@
        @cd doc; \
          latex $@ > /dev/null
```

where 'TEXFILES' is a list of files that ought to be in the noweb/doc subdirectory. As before, each *.tex file is only rebuilt if it is out-of-date relative to the corresponding *.nw file. The resulting *.tex file is put into noweb/doc, and then latex is called to create the corresponding *.dvi file (rather lazily, the output from latex is diverted to /dev/null and lost).

Slightly more complicated, suppose we want to put pdf versions of the *.tex files into the subdirectory inst/doc, using the command 'make pdf'. We need the following lines in 'Makefile':

```
# make pdf files in ../inst/doc

pdf:    $(TEXFILES) $(PDFFILES)

$(PDFFILES):    %.pdf : %.tex
        @echo 'Building pdf file for $<'
        @cd doc; \
          pdflatex $< > /dev/null;
        @mv doc/$@ ../inst/doc/
```

where 'PDFFILES' is a list of files that ought to be in the inst/doc subdirectory. The new feature here is that the command 'make pdf' depends on both 'TEXFILES' and 'PDFFILES'. This means that each component of 'TEXFILES' is updated, if necessary, *before* the *.pdf file is rebuilt from the *.tex file. To rebuild the *.pdf file, the pdflatex command is called on the *.tex file in noweb/doc, and then the result is moved to inst/doc.

Additional lines in the 'Makefile'

The main role of additional lines in 'Makefile' is to supply the lists 'RFILES', 'TEXFILES' and so on. The following commands achieve this, and a little bit else besides, and go at the start of 'Makefile':

```
## minimal Makefile

# look in other directories for files
vpath %.R ../R
vpath %.tex doc
vpath %.pdf ../inst/doc

# get lists of files
NWFILES = $(wildcard *.nw)
RFILES = $(NWFILES:.nw=.R)
TEXFILES = $(NWFILES:.nw=.tex)
PDFFILES = $(NWFILES:.nw=.pdf)

# here are the various targets for make

.PHONY: R latex pdf install

# Now insert the lines from above ...
```

The three 'vpath' lines are necessary to help make to look in other directories than noweb for certain types of file. Thus the *.R files are to be found in ../R, the *.tex files in doc, and so on (all relative to the noweb directory). The next four lines compile the lists of various types of file: 'NWFILES' are found in the noweb directory, 'RFILES' are 'NWFILES' files with the 'nw' suffix replaced with a 'R' suffix, and so on. For the final line, the .PHONY command is a bit of defensive programming to identify the targets of the make command.

Example

The file http://maths.dur.ac.uk/stats/people/jcr/myPkg.tar.gz contains a small example of an R package created using noweb and make, where there is only one *.nw file in the noweb directory, namely myDoc1.nw. The following commands illustrate the steps outlined above ('debreu' is my computer):

```
debreu% make --dry-run R
make: Nothing to be done for `R'.
debreu% touch myDoc1.nw
debreu% make --dry-run R
echo 'Building R file for myDoc1.nw'
notangle -RR myDoc1.nw > ../R/myDoc1.R
if test `egrep '^<<man>>=' myDoc1.nw` ; then \
  notangle -Rman myDoc1.nw > ../man/myDoc1.Rd ; fi
debreu% make R
Building R file for myDoc1.nw
debreu% make --dry-run pdf
echo 'Building tex file for myDoc1.nw'
noweave -delay myDoc1.nw > doc/myDoc1.tex
cd doc; \
  latex myDoc1.tex > /dev/null
echo 'Building pdf file for myDoc1.tex'
cd doc; \
  pdflatex myDoc1.tex > /dev/null;
mv doc/myDoc1.pdf ../inst/doc/
debreu% make pdf
Building tex file for myDoc1.nw
Building pdf file for myDoc1.tex
```

Initially the source file myDoc1.nw is up-to-date. I touch it to make it more recent than the derived files myDoc1.{R,Rd,tex,pdf}. The dry run shows the commands that will be executed when I type make R: in this case the file myDoc1.R is built and moved to ../R. After issuing the command make R the only output to screen is the comment 'Building R file for myDoc1.nw'. The dry run for make pdf shows a more complicated set of operations, because before myDoc1.pdf can be built, myDoc1.tex has to be built.

More complicated targets

The make command can also be used to perform more complicated operations. Two that I have found useful are building the package, and installing the package onto my $R_LIBS path. In both cases, these operations must first make sure that the files are all up-to-date. For 'make install', for example, we might have:

```
PKGNAME = myPkg
RLIBRARY = /tmp/
R = R

# install

install:        $(RFILES) $(PDFFILES)
     @echo 'Installing $(PKGNAME) at $(RLIBRARY)'
     @cd ../..; \
       $(R) CMD INSTALL -l $(RLIBRARY) $(PKGNAME)
```

where the variables 'PKGNAME', 'RLIBRARY' and 'R' are specified separately, so that it is easy to change them for different packages, different locations on `$R_LIBS` or different versions of R. These are best put at the top of the 'Makefile'. The target `install` should be added to the `.PHONY` line.

Issuing the `make install` command when everything is up-to-date gives:

```
debreu% make --dry-run install
echo 'Installing myPkg at /tmp/'
cd ../..; R CMD INSTALL -l /tmp/ myPkg
debreu% make install
Installing myPkg at /tmp/
* Installing *source* package 'myPkg' ...
** R
** inst
** help
>>> Building/Updating help pgs for package 'myPkg'
     Formats: text html latex example
* DONE (myPkg)
```

If some of the files were not up-to-date then they would have been rebuilt from the original `*.nw` files first.

Conclusion

The great thing about literate programming is that there is no arguing with the documentation, since the documentation actually includes the code itself, presented in a format that is easy to check. For those of us who use LaTeX, `noweb` is a very simple literate programming tool. I have found using `noweb` to be a good discipline when developing code that is moderately complicated. I have also found that it saves a lot of time when providing code for other people, because the programmer's notes and the documentation become one and the same thing: I shudder to think how I used to write the code first, and then document it afterwards.

For large projects, for which the development of an R package is appropriate, it is often possible to break the tasks down into chunks, and assign each chunk to a separate file. At this point the Unix `make` utility is useful both for automating the processing of the individual files, and also for speeding up this process by not bothering with up-to-date files. The 'Makefile' that controls this process is almost completely generic, so that the one described above can be used in any package which conforms to the outline given in the introduction, and which uses the tags 'R', 'man' and so on to identify the type of code chunk in each `*.nw` file.

Jonathan Rougier
Department of Mathematical Sciences, University of Durham, UK
J.C.Rougier@durham.ac.uk

CRAN Task Views

by Achim Zeileis

With the fast-growing list of packages on CRAN (currently about 500), the following two problems became more apparent over the last years:

1. When a new user comes to CRAN and is looking for packages that are useful for a certain task (e.g., econometrics, say), which of all the packages should he/she look at as they might contain relevant functionality?

2. If it is clear that a collection of packages is useful for a certain task, it would be nice if the full collection could be installed easily in one go.

The package **ctv** tries to address both problems by providing infrastructure for maintained task views on CRAN-style repositories. The idea is the following: a (group of) maintainer(s) should provide: (a) a list of packages that are relevant for a specific task (which can be used for automatic installation) along with (b) meta-information (from which HTML pages can be generated) giving an overview of what each package is doing. Both aspects of the task views are equally important as is the fact that the views are maintained. This should provide some quality control and also provide the meta-information in the jargon used in the community that the task view addresses.

Using CRAN task views is very simple: the HTML overviews are available at http://CRAN.R-project.org/src/contrib/Views/ and the task view installation tools are very similar to the package installation tools. The list of views can be queried by `CRAN.views()` that returns a list of `"ctv"` objects:

```
R> library(ctv)
R> x <- CRAN.views()
R> x


CRAN Task Views
```