```
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
```

```
---------------
Name: Econometrics
Topic: Computational Econometrics
Maintainer: Achim Zeileis
Repository: http://cran.r-project.org
---------------
Name: Finance
Topic: Empirical Finance
Maintainer: Dirk Eddelbuettel
Repository: http://cran.r-project.org
---------------
Name: MachineLearning
Topic: Machine Learning&Statistical Learning
Maintainer: Torsten Hothorn
Repository: http://cran.r-project.org
---------------
Name: gR
Topic: gRaphical models in R
Maintainer: Claus Dethlefsen
Repository: http://cran.r-project.org
```

```
R> x[[1]]
```

```
CRAN Task View
--------------
Name:       Econometrics
Topic:      Computational Econometrics
Maintainer: Achim Zeileis
Repository: http://cran.r-project.org
Packages:   bayesm, betareg, car*, Design,
            dse, dynlm, Ecdat, fCalendar,
            Hmisc, ineq, its, lmtest*, Matrix,
            micEcon, MNP, nlme, quantreg,
            sandwich*, segmented, sem,
            SparseM, strucchange, systemfit,
            tseries*, urca*, uroot, VR,
            zicounts, zoo*
            (* = core package)
```

Note that currently each CRAN task view is associated with a single CRAN-style repository (i.e., a repository which has in particular a `src/contrib` structure), future versions of **ctv** should relax this and make it possible to include packages from various repositories into a view, but this is not implemented, yet.

A particular view can be installed subsequently by either passing its name or the corresponding `"ctv"` object to `install.views()`:

```
R> install.views("Econometrics",
+                lib = "/path/to/foo")
R> install.views(x[[1]],
+                lib = "/path/to/foo")
```

An overview of these client-side tools is given on the manual page of these functions.

Writing a CRAN task is also very easy: all information can be provided in a single XML-based format called `.ctv`. The `.ctv` file specifies the name, topic and maintainer of the view, has an information section (essentially in almost plain HTML), a list of the associated packages and further links. For examples see the currently available views in **ctv** and also the vignette contained in the package. All it takes for a maintainer to write a new task view is to write this `.ctv` file, the rest is generated automatically when the view is submitted to us. Currently, there are task views available for econometrics, finance, machine learning and graphical models in R—furthermore, task views for spatial statistic and statistics in the social sciences are under development. But to make these tools more useful, task views for other topics are needed: suggestions for new task views are more than welcome and should be e-mailed to me. Of course, other general comments about the package **ctv** are also appreciated.

Achim Zeileis
Wirtschaftsuniversität Wien, Austria
Achim.Zeileis@R-project.org

Using Control Structures with Sweave

Damian Betebenner

Sweave is a tool loaded by default with the `utils` package that permits the integration of R/S with LaTeX. In one of its more prominant applications, Sweave enables literate statistical practice—where R/S source code is interwoven with corresponding LaTeX formatted documentation (R Development Core Team, 2005; Leisch, 2002a,b). A particularly elegant implementation of this is the `vignette()` function (Leisch, 2003). Another, more pedestrian, use of Sweave, which is the focus of this article, is the batch processing of reports whose contents, including figures and variable values, are dynamic in nature. Dynamic forms are common on the web where a base `.html` template is populated with user specific data drawn, most often, from a database. The incorporation of repetitive and conditional control structures into the processed files allows for almost limitless possibilities to weave together output from R/S within the confines of a LaTeX document and produce professional quality dynamic output.