The **cacheSweave** Package

Roger D. Peng
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health

June 15, 2007

1 Introduction

The Sweave system of Leisch (2002) is a literate programming tool based on ideas of Knuth (1984) and is currently part of the core R installation. Specifically, Sweave is a system for processing documents that mix LaTeX document formatting with R code. R code can be interspersed within the LaTeX markup by indicating "code chunks". These code chunks are evaluated by the `Sweave` function in R and the code is replaced with the results of the evaluation. For example, the code for fitting a linear model and summarizing the estimated regression coefficients might be replaced by a formatted table of estimated regression coefficients along with standard errors and p-values. Another possibility is for the code to replaced by a plot which shows the data and the fitted regression line. In either case, the author writes the code to generate the output and Sweave runs the code and places the output in the final document.

Given a file written in the Noweb format (Ramsey, 1994), one can generate a LaTeX file by running in R

```
> Sweave("foo.Rnw")
```

where "foo.Rnw" contains both LaTeX markup and R code. Calling `Sweave` in this manner results in a the file "foo.tex" being created, which can subsequently be processed by standard LaTeX tools. In particular, the **tools** package contains the R function `texi2dvi` which calls the system's `texi2dvi` program if it is available.

Sweave has many potential uses, but it is particularly useful for creating statistical documents that are *reproducible*, where the results of computation can be reproduced by executing the original code using the original data. Since the code used for analysis is embedded directly into the relevant document, there is a tighter correspondence between the descriptive text and the computational results and a decreased potential for mismatches between the two. In addition, Sweave's ability to recompute results to reflect changes or updates to the datasets and analytic code is a great benefit to authors who must maintain statistical documents. With Sweave, all of the relevant text and code reside in a master document from which different outputs can be derived by either "weaving" to create a human-readable document or "tangling" to produce a machine-readable code file.

One aspect of Sweave's default mode of operation is that all code chunks are evaluated whenever the document is read/processed by the `Sweave` function in R (except when an authors explicitly indicates that a code chunk should not be evaluated). While this is generally considered a feature, it can be cumbersome during the development of a document if the code chunks contain calculations that are lengthy or resource intensive. In particular, changes to text portions of the document require

that the entire document be re-Sweaved so that the resulting LaTeX file can reflect the changes to the text. In such cases, it might be desirable for the code chunks to either not be evaluated or to be cached in some manner so that subsequent evaluations take less time.

One approximate solution to the problem described above is to indicate that code chunks should not be evaluated (i.e. by setting `eval=false` as an option for each code chunk) so that the `Sweave` function will skip over them and create the LaTeX file. However, such an approach is probably not desirable since then no results can be displayed in the document. Another approach is to separate out the code chunks that contain lengthy computations into a separate file and then include the resulting file via LaTeX's `\input` directive. This way, the file with the expensive code chunks can be Sweaved once while the text can be modified independently in a separate file. This approach has merit and can also benefit greatly from the use of the `make` utility, but it also breaks the principle of including all of the text and code in a single file. The need to manage multiple files has the potential to lead to the same problems that Sweave and other literate programming tools were (in part) designed to solve.

Consider the following code chunk.

```
> set.seed(1)
> x <- local({
+     Sys.sleep(10)
+     rnorm(100)
+ })
> results <- mean(x)
```

Admittedly, this code chunk is not very interesting or realistic but it is useful for demonstrating the basic approach of the **cacheSweave** package. In the code chunk, we (1) set the random number generator seed; (2) generate 100 standard Normal random numbers after sleeping for 10 seconds; and (3) calculate the mean of the Normal random numbers. After executing the code chunk, there are two objects in the user's workspace (i.e. the global environment): `x` and `results` (there is also a hidden object `.Random.seed` that is created by `set.seed`).

On a modern computer executing the code chunk above should take about 10 seconds since the operations other than the call to `Sys.sleep` use a negligible amount of wall clock time. Although the use of `Sys.sleep` here is artificial, one can imagine replacing it with a call to a function that executes a complex or resource intensive statistical calculation. For the purposes of the task at hand, we may only be interested in the mean of the vector `x`, but we have to spend a reasonable amount of time getting there. Repeated evaluation of this code chunk may be neededlessly time consuming if the code and data do not change after the first evaluation.

The **cacheSweave** package allows users to cache the results of evaluating a Sweave code chunk. In the above example, the basic approach would be to cache the objects `x` and `results` in a key-value database with the key being the object name and the value being the R object itself. On future evaluations of this code chunk (assuming the code has not changed otherwise), we could load `x` from the database rather than wait the 10 seconds as we did on the first evaluation. Using the cached value of `x` we could compute various summary statistics. If we were interested in the mean of `x` we could simply load the cached value of `results` from the database (although in this case direct recalculation of the mean would not take much time).

2 Expression caching mechanism

A simple code chunk in a Sweave document might appear as follows.

```
<<FitLinearModel>>=
library(datasets)
library(stats)
data(airquality)
fit <- lm(Ozone ~ Temp + Solar.R + Wind, data = airquality)
@
```

This code chunk loads the `airquality` dataset from the **datasets** package and fits a linear model using the `lm` function from the **stats** package. In this case, two objects are created in the workspace: the `airquality` data frame and the `fit` object containing the output from the `lm` call.

To make use of the caching mechanism provided in **cacheSweave**, the user must set the option `cache=true` in the code chunk declaration. The modified code chunk would be

```
<<FitLinearModel,cache=true>>=
library(datasets)
library(stats)
data(airquality)
fit <- lm(Ozone ~ Temp + Solar.R + Wind, data = airquality)
@
```

The user must also modify the standard invocation of `Sweave` by using the `cacheSweaveDriver` function instead of the default `RweaveLatex` driver function. If the above code chunk were contained in the file "foo.Rnw", then one would call

```
> library(cacheSweave)
> Sweave("foo.Rnw", driver = cacheSweaveDriver)
```

to process the file with the caching mechanism.

On the first evaluation the `cacheSweaveDriver` function does a number of computations in addition to the standard Sweave processing:

1. For each code chunk, a key-value database is created, by default, in a directory called `cache` for storing data objects. The database implementation is a simple one implemented internally in the package. The name of the database is derived from the name of the code chunk and an MD5 digest (Rivest, 1992) of the entire code chunk. Users can change the location of the key-value database by calling the `setCacheDir` function and providing a path.

2. Within each code chunk, there may be multiple expressions and the each expression is handled separately. For each expression:

   (a) The MD5 digest of the expression is taken and looked up in the key-value database. If the digest does not exist, then the expression is evaluated in a temporary environment that has the global environment as a parent.

   (b) After evaluation, the names of the objects created as a result of the evaluation are stored in the key-value database as a character vector with the digest expression as the key.

   (c) The objects created as a result of the evaluation are then stored separately in the database using their own names as keys.

(d) The objects are then lazy-loaded (see e.g. Ripley, 2004) into the global environment via the internal `lazyLoad` function.

3. A "map file" is created which is a text file that contains metadata about the code chunks and any resulting databases or figures produced.

The result of running `Sweave` with the `cacheSweaveDriver` function is a LaTeX file, a collection of databases in a directory specified by `setCacheDir`, and a map file which contains information about each of the code chunks.

On a subsequent evaluation, the processing is slightly different. Namely, for each expression in a code chunk:

1. The MD5 digest of the expression is taken and looked up in the key-value database. If the digest exists (indicating that the same expression has been evaluated previously), the names of the objects associated with this expression are retrieved.

2. Given the names of the objects associated with this expression, the objects are then lazy-loaded into the global environment.

In this situation, the evaluation of a cached expression is replaced by the lazy-loading of the objects associated with that expression into the global environment.

If a future expression (either within the same code chunk or in a subsequent code chunk) requires an object created in a previous code chunk, then that object will be automatically loaded into the global environment via the lazy-loading mechanism.

3 Lazy-loading of objects

The lazy-loading of objects into the global environment once they have been cached is a useful feature of the **cacheSweave** package when large objects are used in a code chunk. For example, one code chunk might read in a large dataset and calculate a summary statistic based on that dataset, e.g.

```
<<loadLargeDataset,cache=true>>=
data <- readLargeDataset("datafile")
x <- computeSummaryStatistic(data)
@
```

With caching turned on for this code chunk, the objects `data` and `x` are stored in the cached computation database for this code chunk. A future code chunk then might simply print the summary statistic `x`, for example,

```
<<printX>>=
print(x)
@
```

If the primary interest is in the summary statistic `x`, then on future evaluations of both of these code chunks, the object `data` is never needed. It is only needed on the first evaluation so that the summary statistic can be calculated and stored in the object `x`. When `data` is lazy-loaded in future Sweave runs, it is never accessed and hence never actually loaded from the database. Therefore, code can be written in the manner shown above and there is no need to worry about the `data` object being loaded repeatedly into R when it is not actually needed.

4 Construction of `cacheSweaveDriver`

The construction of the `cacheSweaveDriver` function is modeled on the `RweaveLatex` function from the **utils** package. The `cacheSweaveDriver` function returns a list of five functions:

1. `setup`, creates a list of available options. We add an extra option `cache` for indicating whether a code chunk should be cached. We also add the name of the map file so that it can be updated after evaluating each code chunk.

2. `runcode`, based on the `RweaveLatexRuncode` function in the **utils** package, this function executes code in each code chunk and saves objects to key-value databases. While much of the original code is retained, we replace the call to `RweaveEvalWithOpt` with our own `cacheSweaveEvalWithOpt` function, which handles the evaluation of the expression, creation of the key-value database, and the saving of objects to the database. We also add a call to the function `writeChunkMetadata` which writes out information to the map file.

3. `writedoc`, handles writing of output LaTeX file; we import the `RweaveLatexWritedoc` function from **utils**.

4. `finish`, closes the output connection and prints some final messages; we import the `RweaveLatexFinish` function from **utils**.

5. `checkopts`, checks that code chunk options are valid; we import the `RweaveLatexOptions` function from **utils**.

The bulk of the work is done in the `runcode` function, which handles the evaluation of the expressions in each code chunk. The code in that function is based on the code from R version 2.5.0 (R Development Core Team, 2007).

5 Expressions with side effects

Simple expressions, such as assignments, will typically result in a single object being created in the global environment. For example, the expression

```
> x <- 1:100
```

results in an object named `x` being created in the global environment whose value is an integer sequence from 1 to 100.

However, there are other types of expressions which can result in either multiple objects being created in the user's workspace or no objects being created. For example, the `source` function is often used to load objects from an R code file. Unless the `local` argument is set to `TRUE`, these objects will by default be created in the global environment. When the `cacheSweaveDriver` function evaluates an expression that contains a call to `source`, there will be objects created outside of the temporary environment in which the expression is evaluated (again, unless the argument `local = TRUE` is specified in the call to `source`). The `set.seed` function behaves in a similar way by modifying (or creating) the `.Random.seed` object in the global environment.

In order to handle the effects of functions like `source` the function `evalAndDumpToDB`, which evalutes an expression and saves the results to the database, first obtains a character vector of the names of all the objects in the global environment. After evaluating the expression in a temporary environment, a check is made to see if any new objects have been created or modified in the global

environment. If so, those objects are saved to the database as well as any objects that were created in the temporary environment. Note that we currently make a special case of the global environment. If the code being evaluated creates objects in some other environment, then **cacheSweave** will not be able to cache those objects.

Another example of a function with side effects is the `plot` function (and related functions) from the **graphics** package. Since `plot` does not create any objects in the global environment, but rather creates a plot on a graphics device, there is nothing for `cacheSweaveDriver` to cache. Currently, it is not clear what is the best way to handle this behavior and so calls to plotting functions cannot be cached using the **cacheSweave** package. In the future, we may attempt to detect the creation of a graphics file (e.g. a PDF or EPS file) and store that file along with the cached computations.

There are many other types of expressions that have side effects and do not result in the creation of objects in the global environment. Expressions such as calls to `system` or functions which write out files (e.g. `save`, `save.image`, `write.table`, `dput`, etc.) all result in objects being created outside of R. In general, these expressions cannot yet take advantage of the caching mechanism in **cacheSweave** and must be executed every time `Sweave` is run.

When caching is used, it is useful to divide the code into chunks which setup the data and results (and can use caching) and chunks that present or display the results (and cannot use caching). For example, with the linear model example from the previous section, one might have one code chunk for loading the data and fitting the model

```
<<FitLinearModel,cache=true>>=
library(datasets)
library(stats)
data(airquality)
fit <- lm(Ozone ~ Temp + Solar.R + Wind, data = airquality)
@
```

and another code chunk for summarizing the results in a standard table of regression coefficients.

```
<<LinearModelTable,results=tex>>=
library(xtable)
print(xtable(fit))
@
```

Here, we use the **xtable** package to create a formatted LaTeX table of the regression output. A similar approach could be used for plots by separating out the code that generates the plot, e.g.

```
\begin{figure}
  \centering
<<LinearModelDiagnosticPlot,fig=true>>=
par(mfcol = c(2, 2))
plot(fit)
@
\caption{Linear model diagnostic plots}
\end{figure}
```

References

Knuth DE (1984). "Literate Programming." *Computer Journal*, **27**(2), 97–111.

Leisch F (2002). "Sweave: Dynamic generation of statistical reports using literate data analysis." In W Härdle, B Rönz (eds.), "Compstat 2002 — Proceedings in Computational Statistics," pp. 575–580. Physika Verlag, Heidelberg, Germany. ISBN 3-7908-1517-9.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL `http://www.R-project.org`.

Ramsey N (1994). "Literate Programming Simplified." *IEEE Software*, **11**(5), 97–105.

Ripley BD (2004). "Lazy Loading and Packages in R 2.0.0." *R News*, **4**(2), 2–4. URL `http://CRAN.R-project.org/doc/Rnews/`.

Rivest RL (1992). *The MD5 Message-Digest Algorithm*. RFC 1321. http://tools.ietf.org/html/rfc1321.