6-11-2007

DISTRIBUTED REPRODUCIBLE RESEARCH USING CACHED COMPUTATIONS

Roger Peng
Johns Hopkins Bloomberg School of Public Health, Department of Biostatistics, rpeng@jhsph.edu

Sandrah P. Eckel
Johns Hopkins Bloomberg School of Public Health, Department of Biostatistics

Distributed reproducible research

using cached computations

Roger D. Peng   Sandrah P. Eckel

June 11, 2007

Abstract

The ability to make scientific findings reproducible is increasingly important in areas where substantive results are the product of complex statistical computations. Reproducibility can allow others to verify the published findings and conduct alternate analyses of the same data. A question that arises naturally is how can one conduct and distribute reproducible research? This question is relevant from the point of view of both the authors who want to make their research reproducible and readers who want to reproduce relevant findings reported in the scientific literature. We present a framework in which reproducible research can be conducted and distributed via cached computations and describe specific tools for both authors and readers. As a prototype implementation we introduce three software packages written in the R language. The cacheSweave and stashR packages together provide tools for caching computational results in a key-value style database which can be published to a public repository for readers to download. The SRPM package provides tools for generating and interacting with "shared reproducibility packages" (SRPs) which can facilitate the distribution of the data and code. As a case study we demonstrate the use of the toolkit on a national study of air pollution exposure and mortality.

KEY WORDS: Literate programming; R; environmental epidemiology; Sweave

1

2

1 INTRODUCTION

The validity of conclusions from scientific investigations is typically strengthened by the replication of results by independent researchers. Full replication of a study's results using independent methods, data, equipment, and protocols, has long been, and will continue to be, the standard by which scientific claims are evaluated. In many fields of study, there are examples of scientific investigations which cannot be fully replicated, often because of a lack of time or resources. For example, epidemiologic studies which examine large populations and can potentially impact broad policy or regulatory decisions, often cannot be fully replicated in the time frame necessary for making a specific decision. In such situations, there is a need for a minimum standard which can serve as an intermediate step between full replication and nothing. This minimum standard is *reproducible research*, which requires that datasets and computer code be made available to others for verifying published results and conducting alternate analyses.

There are a number of reasons why the need for reproducible research is increasing. Investigators are more frequently examining inherently weak associations and complex interactions for which the data contain a low signal-to-noise ratio. New technologies allow scientists in all areas to compile complex high-dimensional databases and the ubiquity of powerful statistical and computing capabilities allow investigators to explore those databases and identify associations of potential interest. However, with the increase in data and computing power come a greater potential for identifying spurious associations. In addition to these developments, recent reports of fraudulent research being published in the biomedical literature have highlighted the need for reproducibility in biomedical studies and have invited the attention of the major medical journals (Laine et al., 2007).

Interest in reproducible research in the statistical community has been increasing in the past decade with examples such as Buckheit and Donoho (1995), Rossini and Leisch (2003), Sawitzki (2002), and many others. The area of bioinformatics has produced projects such as Bioconductor (Gentleman et al., 2004) which promotes reproducible research as a primary aim (see also Ruschhaupt et al., 2004; Gentleman, 2005).

3

A proposal for making research reproducible in an epidemiologic context was outlined in Peng et al. (2006). The criteria described there include a requirement that analytic data and the analytic computer code be made available for others to examine. The analytic data is defined as the dataset that served as the input to the analytic code to produce the principal results of the article. For example, a rectangular data frame might be analytic data in one case, a regression procedure might constitute analytic code, and regression coefficients with standard errors might be the principal results. Peng et al. (2006) describe the need for reproducible research to be the minimum standard in epidemiologic studies, particularly when full replication of a study is not possible.

The standard of reproducible research requires that the source materials of a scientific investigation be made available to others. This requirement is analogous to the definition of open source software (see e.g. http://www.opensource.org/), which requires that the source code for a computer program be made available. However, by using the phrase "source materials" in the context of reproducible research, we do not mean simply the computer code that was used to analyze the data. Rather, we refer more generally to the preferred form for making modifications to the original analysis or investigation. Typically, this preferred form includes analytic datasets, analytic code, and documentation of the code and datasets.

The task of making research reproducible is not what one might consider a traditional statistical problem. However, statisticians are often confronted with the challenges of reproducing the computational results of others and statisticians are often the ones challenged with making their own results and computations reproducible by others. It is in this sense that we consider reproducible research a statistical problem for which we need to develop models and methods.

In this paper we introduce a framework for conducting and distributing reproducible research using what we call cached computations. Within this framework we describe a model for a reproducible document that divides a document into text and code. We then present a software implementation which makes use of cached computations for distributing

4

and interacting with reproducible documents.

2   MODEL FOR REPRODUCIBLE RESEARCH

The model that we use to describe reproducible research is the "research pipeline" sketched in Figure 1. We model the research pipeline as beginning with "measured data" collected from nature which are then processed into "analytic data" via processing code and an associated software environment. Using a possibly different software environment (most likely a statistical analysis environment), the analytic data are then used to produce "computational results", which might be the output of regression models and various derived quantities. Computational results are then summarized in figures or tables or included as numerical results in the text, and these summarized results are then assembled with the expository text to form an article.

The principle underlying the research pipeline in Figure 1 is *modularity*. Modularity calls for the research process to be separated out into distinct components. We propose that a modular research pipeline lends itself more naturally to reproducible results because the existence of each component of Figure 1 in a semi-persistent state allows for the inspection of that component and the process that led to it. For example, one might be particularly interested in inspecting the analytic code that produces the computational results from the analytic data.

While a modular framework may seem reasonable, not all research is necessarily conducted in this manner. For example, software packages exist which will simultaneously process and analyze data, create a table, and embed the table in text, blurring the distinction between each of these stages. In addition, not all applications allow for the user to easily record the steps which lead from one state to another. Results from applications with graphical user interfaces are notoriously difficult to reproduce.

A modular research pipeline has implications for the way we analyze data, assemble results, and write articles. One important implication is that we must separate content from the presentation of content. This separation can only be achieved in practice if we

have a reproducible means of going from one state to the other. That is why we need software that can take the results of computation and create useful summaries (e.g. figures and tables) or "views" of the data (Gentleman and Temple Lang, 2007).

One feature noted in the pipeline in Figure 1 is that authors and readers operate along opposite directions of the pipeline. Authors of papers start with the data, eventually building up to the analysis and then the paper. Readers start with the paper and, if sufficiently interested, begin to dig deeper into the details of the analysis by obtaining the relevant data and software. Given the data and software for a particular figure or table, the reader can reproduce those results and possibly conduct alternate analyses. In each direction, authors and readers need different sets of tools for conducting reproducible research.

3   DISTRIBUTING REPRODUCIBLE DOCUMENTS

In order to describe how we will make scientific results from an article reproducible, we need to develop a model for the document itself. Using the model, we can connect an article/document with the rest of the system described in Figure 1 and see how results can be distributed to others. A useful starting point comes from the idea of literate programming, a phrase used by Donald Knuth to describe a different way of writing complex computer programs (Knuth, 1984). Knuth wrote,"We understand a complicated system by understanding its simple parts, and by understanding the simple relations between those parts and their immediate neighbors." Knuth's WEB system for literate programming combined a document formatting language and a programming language to produce a single source document for describing a complex program. The source document can subsequently be *weaved* to produce a human readable description and *tangled* to produce a machine executable program.

Knuth's notion of a *web* for describing complex computer programs carries over readily to the world of complex scientific investigations and publications. When we read an article in a journal, we essentially receive the "weaved" version of the research conducted, i.e. in human readable form. However, there are typically other components of the research in

6

which one might be interested that are unavailable to us from the weaved version. The question is how can we make those other components available to those people who wish to reproduce our results?

Leisch (2002) and others have built upon Knuth's literate programming concept and have extended it to the creation of reproducible statistical documents. The specific implementation of Leisch (2002) is called Sweave and combines the LaTeX document formatting language with the R programming language (Ihaka and Gentleman, 1996). In this scheme, documents are divided into "chunks"—either text chunks or code chunks—each of which is processed in a different way. Text chunks are written in a document formatting language and code chunks are written in a computer programming language. Analogous to Knuth's system, the document can be weaved to form a human readable document (e.g. an article or report in PDF) and tangled to produce machine executable code.

With this model of a document as a stream of text and code chunks, we can formulate a strategy for making the code and data available in a manner that is both convenient for the author and the reader. The basic idea is to use the Sweave approach, which evaluates each code chunk in a document and replaces the code chunk with the results of the evaluation. Upon evaluation of a code chunk, we can simultaneously cache the results of the evaluation in a database. This database of *cached computations* can then be made publicly available to readers for the purposes of reproducing specific results. This database of cached computations can be thought of as a middle ground between the author and the reader which contains intermediate results. These results are not quite measured or raw data but the are also not finished products (i.e. figures, tables).

This caching of results from code evaluations serves the purpose of providing an intermediate step for readers who want to explore a certain complex computation but are not willing to reproduce it in its entirety. For computations that are very lengthy and resource intensive, a reader may not be able to reproduce the results from start to finish but may be willing to explore some of the outputs. In addition, an author writing a document may not wish to repeatedly run a lengthy computation every time the document is weaved to

produce the human readable version. In such a case it may be much more efficient to run the computation once, cache the results, and then subsequently load the results from the cache. Hence, the caching of computational results helps both the author and the reader.

4   REPRODUCIBLE RESEARCH TOOLKIT

In this section we illustrate our ideas by describing an approach to caching computations using R and LaTeX. Our implementation builds on the work of Leisch (2002) and the basic approach of Sweave as well as the concept of a compendium described by Gentleman and Temple Lang (2007). Sweave reads documents written in the Noweb format (Ramsey, 1994), where a standard code chunk might appear as follows:

```
<<Code Chunk 1>>=
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 to the data. The beginning of the chunk is indicated by the `<<>>=` symbol and a label for the chunk is provided between the `<<` and `>>` symbols. The end of the code chunk is indicated by the `@` symbol. Any text appearing outside of these symbols are considered part of text chunks and are processed separately using the document formatting language processor (in this case, LaTeX). One exception is the `\Sexpr` directive, which allows the author to place code inline with the text rather than in a separate code chunk.

Briefly, when the Sweave function reads the document, the code chunks (and the `\Sexpr` directives) are evaluated in the R environment and are replaced with the results of the evaluations. The result is a LaTeX file which can be handled with the standard LaTeX tools. By default, the code in the code chunks is echoed in the resulting LaTeX file along with the

8

results so one can see the input followed by the corresponding output.

4.1   Caching computations

The `cacheSweave` package is the first package in our toolkit and it allows an author to cache the results of a computation in a code chunk by setting the `cache=true` option in the code chunk declaration.

```
<<Code Chunk 1,cache=true>>=
library(datasets)
library(stats)
data(airquality)
fit <- lm(Ozone ~ Temp + Solar.R + Wind, data = airquality)
@
```

In the above code chunk, there are two objects that are created in the R workspace. First, the `airquality` dataset is loaded and second, the `fit` object is created to store the output of the call to the linear modeling function `lm`. The effect of setting `cache=true` is to store these two objects in a key-value database on the disk. This database of objects is what we refer to as a cached computation. For a given document there may be a collection of databases storing results for all of the code chunks in the document.

The `cacheSweave` package provides a separate driver function for Sweave which takes the place of the default Sweave driver `RweaveLatex`. The `cacheSweaveDriver` function is a modification of the standard driver which handles the caching of code chunk computations. If the option `cache=true` is set for a given code chunk, the driver function first creates a key-value database for storing the results of that chunk. The key-value database implementation used by `cacheSweave` is provided by the second package in our toolkit, the `stashR` package (Eckel and Peng, 2006), which derives some functionality from the `filehash` package (Peng, 2006). The driver function then evaluates each expression in the code chunk and stores the results of the evaluation in the key-value database.

9

When evaluating a given expression, a check is made to see if the expression has already been evaluated. This is achieved by computing an MD5 digest (Rivest, 1992) of the expression itself on first evaluation. Subsequently, if the expression has not changed, then the driver function will see that the MD5 digest for the current expression matches the previous digest and will not evaluate the expression again. This approach is useful for preventing repeated computation but has some limitations which we discuss in Section 6.

If an expression has already been evaluated, then the objects associated with that expression are lazy-loaded into the R workspace and the expression itself is not evaluated (see e.g. Ripley, 2004). That is, instead of evaluating the expression a second time and storing the result, we retain a *promise* to evaluate the expression when it is needed. For example, the expression

```
x <- 1
```

creates an object named `x` with a value of 1. The first time the expression is evaluated, the object `x` is created and its name/value pair is stored in the database for that code chunk. Subsequently, if the expression remains unmodified, the value of `x` is lazy-loaded into the global environment. The first time `x` is needed, its value (e.g. 1) is automatically loaded from the database and stored in the global environment. The lazy-loading mechanism is made available via the `dbLazyLoad` function from the `filehash` package and, in particular, allows large objects to be available "on demand" without having to take up any space or time if they are not needed.

Simple expressions such as the one above will typically result in a single object being created in the R workspace so that most expressions will only have one object associated with them in the database. However, expressions such as

```
source("myfunctions.R")
```

can potentially have many objects associated with them. The `cacheSweave` package takes care to keep track of all of the objects associated with an expression and has special handling for functions like `source` which create objects in an environment different from the calling

10

environment.

For authors of Sweave documents, the caching of computations is useful because it allows one to avoid reevaluating long or costly computations when only changes to a text chunk have been made. In situations where the computations are trivial or quick, there is no need for caching. However, when long computations are involved, typically loading objects from a database will be faster than recomputing the results. To users who are already familiar with the Sweave system, the only change that is needed to take advantage of cached computations is adding the `cache=true` statement to each code chunk and using the `cacheSweaveDriver` function instead of the default `RweaveLatex` driver function.

There is one caveat with using the caching mechanism which is that expressions that have side effects which do not result in the creation of objects in the R workspace cannot be cached. In particular, plotting code that either creates plots on a graphics device or annotates existing plots always has to be evaluated in order to guarantee that the associated plot file is generated since the plot file itself is not cached in the database. An exception to this rule is that if the plot file is retained and its file name does not change, then there may be no need to evaluate the plotting code after the first evaluation. Code that interacts with the system (i.e. copying or renaming files) will generally not be cache-able since those expressions do not generate anything to be stored in the database.

When writing code to be used with the `cacheSweave` system, it is useful to employ a style of coding which divides the code into chunks that are cache-able and chunks that are not. For example, when creating a plot, it is usually possible to divide the code into a section which sets the data up for plotting and a section which creates the plot itself, either with the `plot` function or a related function. In this situation, the setup code could be placed in a chunk with caching turned on and the plotting code could go in a separate non-caching chunk. The case study in Section 5 has examples of this type of division of code.

11

4.2 Packaging reproducible documents

The end results of using the `cacheSweave` package to compile a document are the weaved document itself (e.g. a PDF file), a collection of `stashR` databases containing cached computations from each of the code chunks, files containing figures/plots (if any), and a *map file* which contains metadata for each of the code chunks. The metadata in the map file indicates the filename of any figure produced by a code chunk and the location of any `stashR` databases containing cached computations.

The final package in the reproducible research toolkit is the `SRPM` package which handles the creation of "shared reproducibility packages" (SRPs) for the purposes of distributing the code, text, and cache databases that make up a reproducible document. The `SRPM` package contains tools for both authors and readers of reproducible documents. For authors, the `makeSRP` function reads the original Sweave file and the map file and creates a directory into which the code chunks, cache databases, and figure files (in PDF format) are copied. This newly created directory is the shared reproducibility package which can be archived and distributed.

For the reader, there are a number of simple tools for interacting with an SRP created by an author. They include functions for examining individual code chunks, loading cache databases into the reader's workspace, and executing the code in code chunks. These tools provide the reader with the ability to reproduce results and re-run code provided by the author. In addition, the reader can explore individual data objects and make changes to the code in order to explore aspects of the original analysis that were not published. The usage of these functions will be demonstrated in greater detail in Section 5.

5 APPLICATION AND CASE STUDY

Estimation of the health risks of ambient air pollution is controversial for many of the reasons cited earlier. The risks are inherently small (although the exposed population is large), there is a need for sophisticated computational and statistical tools, and substantive findings can

play a significant role in the development of policy and regulation. These elements all conspire to make reproducibility a necessity in air pollution and health research.

The basis of our case study is the National Morbidity, Mortality, and Air Pollution Study (NMMAPS), which is a large observational study of the health effects of outdoor air pollution (Samet et al., 2000a). The purpose of NMMAPS is to investigate the short-term health effects of air pollution by 1) integrating national databases of population health, air pollution monitoring, weather, and socioeconomic variables; 2) developing statistical methods and computational tools for analyzing these databases; and 3) estimating the short-term associations between air pollution levels and mortality and their uncertainties in the largest U.S. metropolitan areas (Samet et al., 2000b,c). The database and statistical methodology developed for NMMAPS are available from the Internet Health and Air Pollution Surveillance System [iHAPSS] (Zeger et al., 2006) website at http://www.ihapss.jhsph.edu/.

Our case study deals specifically with an update to NMMAPS originally published in Peng et al. (2005), which examined seasonal and regional variation in the health effects of particulate matter. The study estimated the short-term association between PM_{10} and all-cause nonaccidental mortality in 100 cities in the United States by integrating mortality data from the National Center for Health Statistics, air pollution data form the Environmental Protection Agency, and weather data from the National Weather Service. The data used in this paper have been packaged separately as the `NMMAPSdata` R package (Peng and Welty, 2004) and can be downloaded separately from the iHAPSS website. The shared reproducibility package associated with this article can be downloaded from

```
http://www.biostat.jhsph.edu/~rpeng/RR/srp_seasonal_R.zip
```

The zip file contains the article and separate figures in PDF format and the code for each of the code chunks in the article. The cached computation databases are stored on the Johns Hopkins University Department of Biostatistics website and can be accessed remotely via the `SRPM` package.

13

5.1 Exploring a shared reproducibility package

After loading the SRPM package one must register a shared reproducibility package (SRP) with the system via the setPackage function so that the reader tools in the SRPM package know where to look for code, figures, and cache databases.

```
> library(SRPM)

Loading required package: stashR

Loading required package: filehash

Simple key-value database (1.0 2007-04-27)

A Set of Tools for Administering SHared Repositories (0.3 2007-04-27)

Shared Reproducibility Package Management (0.1 2007-04-30)

> setPackage("srp_seasonal_R")

remote cache: http://www.biostat.jhsph.edu/~rpeng/RR/seasonal

local storage: srp_seasonal_R

initializing cache databases...
```

In this example, the srp_seasonal_R package has cache databases which are stored on a remote webserver and the directory "srp_seasonal_R" will be used for storing local copies of the remote cache databases (the default is to store local copies under a subdirectory named "cacheDB"). The URL of the remote cache databases can be retrieved with the getRemoteURL function and the location of the locally stored cache databases can be retrieved with the getLocalDir function.

```
> getRemoteURL()

[1] "http://www.biostat.jhsph.edu/~rpeng/RR/seasonal"

> getLocalDir()

[1] "srp_seasonal_R"
```

The mechanism by which local copies of the cache databases are stored and synchronized with the remote is described in greater detail in Eckel and Peng (2006).

14

Once an SRP has been registered with the `setPackage` function, the first action is to call the `code` function with no arguments to print a list of the code chunks available in this package.

```
> code()
1 SetupCacheSweave
2 mortalityTop10setup  [C]
3 mortalityTop10plot [Figure 1]
4 seasonRegionPM10setup  [C]
5 seasonRegionPM10plot [Figure 2]
[... OMITTED ...]
28 ComputeCopollutantModelTable  [C]
29 copollutantModelTable
30 ComputePeriodicLags012  [C]
31 periodicLags012 [Figure 6]
```

This document has 31 code chunks associated with it, each of which may have a cache database or figure associated with them. In the code chunk listing above, a `[C]` next to the code chunk name indicates that a cache database is associated with that chunk and a `[Figure X]` label indicates that Figure X is created by that code chunk (this document has six figures in it).

Calling the `code` function with either a numeric or character argument prints the code associated with a code chunk to the console. For example, calling `code(3)` and `code("mortalityTop10plot")` both print the R code for the code chunk that produces Figure 1 in the original document. For code chunks with longer labels it may be more convenient to use the numeric sequence numbers instead of the full labels.

In order execute code in a code chunk, one can use the `runcode` function, which takes numeric and character arguments like the `code` function. Calling `runcode(3)`, for example, will execute on the reader's machine the code in code chunk 3. However, in this example, running the code in code chunk 3 alone will result in an error.

```
> runcode(3)
```

running code in code chunk 3

ERROR: unable to run code chunk 3

object "dSeas" not found

This error occurs because the code in code chunk 3 depends on code in code chunks 1 and 2 being executed first. One can execute the code in a series of code chunks by passing a sequence of numbers to the `runcode` function.

```
> runcode(1:3)
```

running code in code chunk 1

ERROR: unable to run code chunk 1

could not find function "setCacheDir"

loading cache for code chunk 2

running code in code chunk 3

Here, there was an error in executing code chunk 1 but the error did not affect the execution of future chunks. Also notice that code chunk 2 was not in fact executed. Rather, objects from the cache database associated with code chunk 2 were lazy-loaded into the reader's workspace. The reason for this is that the author of the document set the "`cache=true`" argument when using `cacheSweave` and cached the computations in code chunk 2. Therefore, the reader does not need to re-run the computations but can load the results directly from the cache database. In this example, the cache database is located on a remote webserver but the lazy-loading mechanism prevents objects from being downloaded to the reader's machine until the are needed. Because of the synchronization mechanism provided by the `stashR` package, objects that have been downloaded once will not be downloaded again, so that future executions of the same code chunks will not repeat potentially lengthy downloads.

One can explore the objects in a cache database directly by calling the `loadcache` function. The `loadcache` function lazy-loads objects in a cache database into the reader's workspace. While the objects appear in the workspace, their values are not immediately

16

downloaded from the remote server. The value associated with a given object will only be downloaded upon first access.

Multiple cache databases can be loaded by passing a vector to `loadcache`. In the cases where a code chunk does not have a cache database associated with it, `loadcache` simply skips over that code chunk. For example, in order to execute code chunk 12, we need to load some of the objects created previously in chunks 1 through 11. We do not, however, need to execute the code in those previous chunks. We can load the cache databases from the previous chunks and execute the code in code chunk 12 alone.

```
> loadcache(1:11)

loading cache for code chunk 2

loading cache for code chunk 4

loading cache for code chunk 6

loading cache for code chunk 7

loading cache for code chunk 8

loading cache for code chunk 10

loading cache for code chunk 11

> runcode(12)

running code in code chunk 12
```

In this case, executing code chunk 12 reproduces Figure 3 in the original article.

5.2   Alternate analyses

The original article of Peng et al. (2005) focused on computing national estimates of air pollution risk and did not publish city-specific relative risk estimates. However, it is clear from the code and the description of the analysis that individual city-specific risk estimates were computed and subsequently combined via an hierarchical model. In order to examine those estimates we need to identify the code chunk in which they are computed.

The non-seasonal city-specific estimates are computed in the code chunk named "ComputeNonSeasonalEstimates" and they are stored in a list named `results.nonseasonal`.

17

This list is of length 3 where each element stores city-specific results corresponding to a specific time lag of exposure to PM_{10} (in this case, 0, 1, and 2 day lags). We can plot the city-specific estimates with approximate 95% confidence intervals by loading the cache database associated with code chunk 6 and extracting the relevant parts.

```
> loadcache("ComputeNonSeasonalEstimates")  ## or 'loadcache(6)'
> lag1 <- results.nonseasonal[["lag1"]]
> coefs <- lapply(lag1, "[[", "coefficients")
> betase <- sapply(coefs, function(x) x["l1pm10tmean", 1:2])
> b <- betase[1,]
> se <- betase[2,]
> plot(b,ylim=range(b-1.96*se, b+1.96*se),xlab="city")
> segments(1:102, b-1.96*se, 1:102, b+1.96*se)
> abline(h=0,lty=3)
```

The resulting figure is shown in Figure 2.

One may also be interested in editing the author's original code to produce an alternate analysis that is different from the original. For example, a reader might be interested in understanding how sensitive a result is to a specific tuning parameter or prior assumption. Or one might want to modify a plot slightly to zoom in to certain areas or highlight interesting areas. In such situations it may be desirable to edit a code chunk and re-run the edited code.

The SRPM package provides a method for editing code chunks via the edit function, which accepts a code chunk object as input and loads the code for that chunk in an external text editor. For example, executing the code

```
newcode <- edit(code(12))
runcode(newcode)
```

loads code chunk 12 in an external editor where the reader is then free to make modifications to the original code. Once the editing is completed, the edit function returns the modified

18

code chunk object on which the reader can then apply the `runcode` function. Note that the original code which came with the SRP does not get modified—`edit` only modifies a copy of the original code.

5.3   Browsing a shared reproducibility package via the web

Readers who wish to explore the contents of a reproducible document may prefer the convenience of using a web browser rather than having to use the R system itself to browse an SRP. The `SRPM` package provides a function `makeWebpage` which produces a webpage from the contents of an SRP. An example of the webpage produced by this function corresponding to Peng et al. (2005) can be found at

`http://www.biostat.jhsph.edu/~rpeng/RR/seasonal/html/`

The webpage interface is limited because code cannot be executed and the web interface to the cache databases is rudimentary. For small objects, functions, and data frames one can browse the complete object, but for larger or more complex objects, only a summary is provided.

6   DISCUSSION

In this article we have described a method by which reproducible research can be distributed using cached computations. Cached computations are created during the execution of a statistical analysis and are stored in a collection of databases. These databases can subsequently be published in public repositories for others to download and explore. These databases of cached computations provide a middle ground where the authors and readers can virtually interact with each other for the purposes of reproducing scientific results. The method we describe enables authors to easily give access to their data and code to a large number of potential readers and allows readers to obtain immediate access to the materials needed for reproducing a specific result.

As an implementation of our cached computation framework, we have developed the

suite of R packages `cacheSweave`, `stashR`, and `SRPM` which together form a toolkit for distributing reproducible research. The `cacheSweave` and `stashR` packages allow an author to use LaTeX, R, and Sweave while caching computations to key-value databases. The `SRPM` package provides tools for authors to create reproducible resarch packages and for readers to interact with the code and data in these packages. Cached computation databases can be stored on public webservers with the data being downloaded as needed by the reader using functionality in the `stashR` package.

In general, there is still a need for developing tools that authors can use to create reproducible documents. For authors using R, a variety of tools are currently being developed by others towards this aim. The Emacs Speaks Statistics package is a powerful tool for writing documents using the Emacs text editor and a number of different statistical programming languages (Rossini et al., 2004). Also, the `DynDoc` package from Bioconductor defines data structures for working with vignettes created by Sweave. The StatDocs project (http://www.stat.berkeley.edu/users/statdocs/) is an interesting effort to provide a number of tools via Omegahat (http://www.omegahat.org/) for creating dynamic statistical documents with a focus on reproducibility.

The `weaver` package (Falcon, 2007) of the Bioconductor Project implements functionality similar to that provided in `cacheSweave`, with a different underlying implementation. The driver function in the `weaver` package caches R objects in standard R saved workspace files. Then, on subsequent evaluation, the cached results are loaded (as opposed to lazy-loaded) into the R workspace. One very useful feature implemented in `weaver` is dependency checking on previous code chunks using the `codetools` package developed by Luke Tierney. Hence, if code chunk 2 depends on an object created in code chunk 1 and code chunk 1 has changed, `weaver` will attempt to detect this and subsequently reevaluate code chunk 2 rather than load it from the cache. No such code checking is done by the `cacheSweave` package, which can only detect whether the code for a given chunk (rather than one of its dependencies) has been modified. The `weaver` package does not provide support for remotely accessing cache databases, a feature of the `stashR` package which can be useful

20

for very large cache databases that are difficult to distribute in their entirety.

The idea of using a single file that contains both statistical programming code and human readable text is not limited to Sweave and R. Another common statistical software package, Stata, is used mainly for interactive data analysis by way of hand-entered commands or commands run through a .do file. Stata can save command outputs except for graphs in a .log file, but this .log file is not readily readable by a larger audience. Recently, Gini and Pasquini (2006) have demonstrated that it is possible to generate reports, presentations or webpages in an automated fashion using Stata .do files. Their work builds on that of Newson (2003) and others in translating Stata output into markup languages for inclusion in LaTeX or HTML documents. Instead of interweaving code and text chunks, the Gini and Pasquini (2006) approach consists entirely of code. Human readable text is included by writing each line of the text to an output file using the Stata command 'file write' in the master .do file. Hence human readable text is not as readily incorporated in the final document as it is in the Sweave approach. MATLAB offers a structured, Windows interface Report Generator that allows for push-button generation of analysis documentation (MathWorks, 2007). The MATLAB Report Generator can be used to automatically generate reports in a wide variety of formats including a navigatable set of HTML webpages, PDF, and Microsoft Word. The MATLAB Report Generator can produce reports whose content is conditional on analysis results.

The software described in this article consists primarily of tools for assisting authors in conducting reproducible research and producing reproducible documents. For readers, the `SRPM` package provides tools for viewing code, exploring data, and reproducing results in a scientific document via shared reproducibility packages. However, substantial improvements could be made to allow the reader to interact more closely with the document and to lessen the dependence on knowledge of R programming. One possible step in this direction would be to allow the reader to interact with a reproducibility package using a graphical user interface or a web browser. The webpage generation facilities in the `SRPM` package currently provide a rudimentary interface for readers and are in need of further development. Another

21

example is the `vExplorer` package of Zhang and Gentleman (2004) which is written with the tcl/tk language and toolkit and provides a graphical user interface for exploring R package vignettes.

Finally, our implementation of a cached computation framework in R is not meant to imply that it is not possible to develop a similar implementation using another software system. The reproducibility of research necessarily depends on the specific software system used in an analysis (and its availability to others) and at this point it does not seem possible to develop a more general system to encompass all possible analyses. We propose our three R packages as a prototype system for R users and as a model around which similar systems could be developed.

The potential benefits of reproducible research to scientists and statisticians are substantial. By expediting the dissemination of ideas and publishing the "research behind the research", investigators can more easily build on existing knowledge. However, reproducible research cannot be conducted without the proper tools available to authors and readers and the development of a reproducibility infrastructure should be a major focus of future work.

References

Buckheit, J. and Donoho, D. L. (1995), "Wavelab and reproducible research," in *Wavelets and Statistics*, ed. Antoniadis, A., Springer-Verlag, New York.

Eckel, S. P. and Peng, R. D. (2006), "Interacting with local and remote data repositories using the stashR package for R," Tech. Rep. 127, Johns Hopkins University Department of Biostatistics, http://www.bepress.com/jhubiostat/paper127.

Falcon, S. (2007), *weaver: Tools and extensions for processing Sweave documents*, R package version 1.2.0.

Gentleman, R. (2005), "Reproducible Research: A Bioinformatics Case Study," *Statistical Applications in Genetics and Molecular Biology*, 4, Article 2.

Gentleman, R. and Temple Lang, D. (2007), "Statistical Analyses and Reproducible Research," *Journal of Computational and Graphical Statistics*, 16, 1–23.

Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., Ellis, B., Gautier, L., Ge, Y., Gentry, J., Hornik, K., Hothorn, T., Huber, W., Iacus, S., Irizarry, R., Leisch, F., Li, C., Maechler, M., Rossini, A. J., Sawitzki, G., Smith, C., Smyth, G., Tierney, L., Yang, J. Y. H., and Zhang, J. (2004), "Bioconductor: open software development for computational biology and bioinformatics," *Genome Biology*, 5, R80.

Gini, R. and Pasquini, J. (2006), "Automatic generation of documents," *The Stata Journal*, 6 (1), 22–39.

Ihaka, R. and Gentleman, R. (1996), "R: A Language for Data Analysis and Graphics," *Journal of Computational and Graphical Statistics*, 5, 299–314.

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

Laine, C., Goodman, S. N., Griswold, M. E., and Sox, H. C. (2007), "Reproducible Research: Moving toward Research the Public Can Really Trust," *Annals of Internal Medicine*, 146, 450–453.

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

MathWorks (2007), "MATLAB Report Generator," [Online; accessed 30-April-2007].

Newson, R. (2003), "Confidence intervals and p-values for delivery to the end user," *The Stata Journal*, 3 (3), 245–269.

Peng, R. D. (2006), "Interacting with data using the filehash package," *R News*, 6, 19–24.

Peng, R. D., Dominici, F., Pastor-Barriuso, R., Zeger, S. L., and Samet, J. M. (2005), "Seasonal Analyses of Air Pollution and Mortality in 100 US Cities," *American Journal of Epidemiology*, 161, 585–594.

Peng, R. D., Dominici, F., and Zeger, S. L. (2006), "Reproducible Epidemiologic Research,"
American Journal of Epidemiology, 163, 783–789, doi:10.1093/aje/kwj093.

Peng, R. D. and Welty, L. J. (2004), "The NMMAPSdata Package," *R News*, 4, 10–14.

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

Ripley, B. D. (2004), "Lazy Loading and Packages in R 2.0.0," *R News*, 4, 2–4.

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

Rossini, A. and Leisch, F. (2003), "Literate Statistical Practice," in *Proceedings of the 2nd
International Workshop on Distributed Statistical Computing*, eds. Hornik, K. and Leisch,
F., pp. 1–10.

Rossini, A. J., Heiberger, R. M., Sparapani, R. A., Mächler, M., and Hornik, K. (2004),
"Emacs Speaks Statistics: A Multiplatform, Multipackage Development Environment for
Statistical Analysis," *Journal of Computational and Graphical Statistics*, 13, 247–261.

Ruschhaupt, M., Huber, W., Poustka, A., and Mansmann, U. (2004), "A Compendium to
Ensure Computational Reproducibility in High-Dimensional Classification Tasks," *Statistical Applications in Genetics and Molecular Biology*, 3, Article 37.

Samet, J. M., Dominici, F., Curriero, F. C., Coursac, I., and Zeger, S. L. (2000a), "Fine Particulate Air Pollution and Mortality in 20 US Cities," *New England Journal of Medicine*,
343, 1742–1749.

Samet, J. M., Dominici, F., Zeger, S. L., Schwartz, J., and Dockery, D. W. (2000b), *The National Morbidity, Mortality, and Air Pollution Study, Part I: Methods and Methodological
Issues*, Health Effects Institute, Cambridge MA.

Samet, J. M., Zeger, S. L., Dominici, F., Curriero, F., Coursac, I., Dockery, D. W., Schwartz,
J., and Zanobetti, A. (2000c), *The National Morbidity, Mortality, and Air Pollution*

24

Study, Part II: Morbidity and Mortality from Air Pollution in the United States, Health Effects Institute, Cambridge, MA.

Sawitzki, G. (2002), "Keeping Statistics Alive in Documents," *Journal of Computational and Graphical Statistics*, 17, 65–88.

Zeger, S. L., McDermott, A., Dominici, F., Peng, R. D., and Samet, J. M. (2006), *Internet-Based Health and Air Pollution Surveillance System*, Communication 12, Health Effects Institute, Boston MA.

Zhang, J. and Gentleman, R. (2004), "Tools for Interactively Exploring R Packages," *R News*, 4, 20–25.

25

A   Figures

Author

Presentation code

Processing code   Analytic code

Figures

Measured
Data

Analytic
Data

Computational
Results

Tables
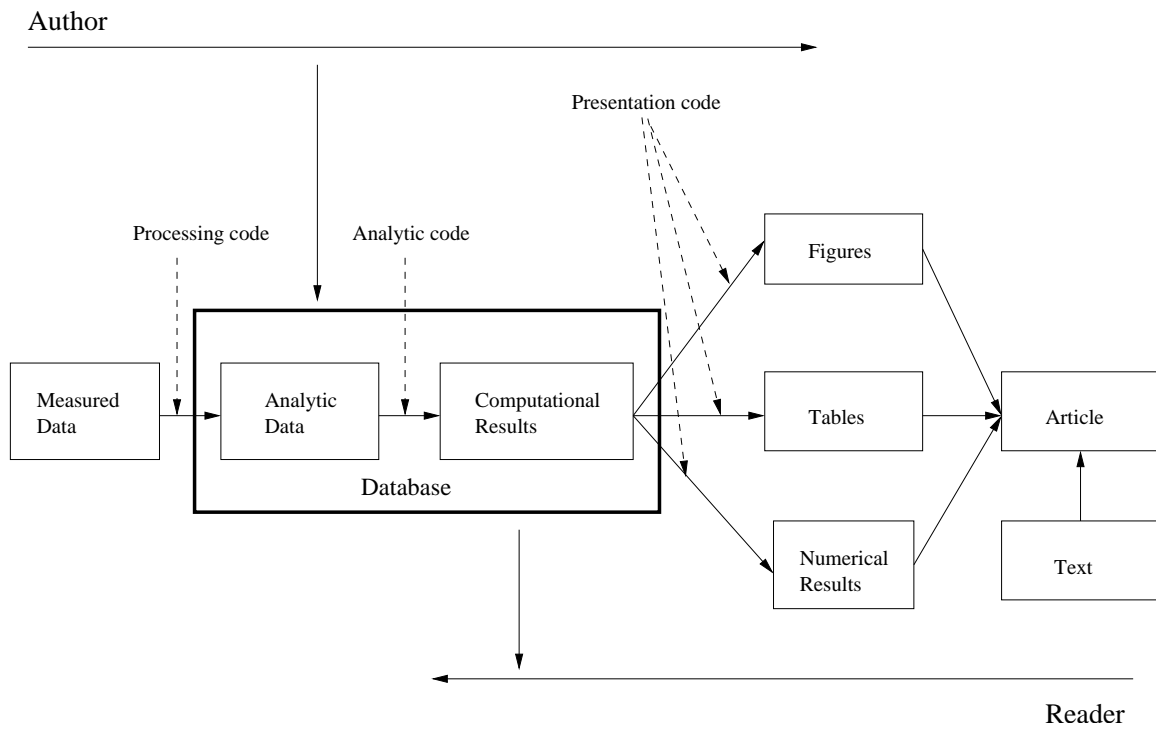
Article

Database

Numerical
Results

Text

Reader

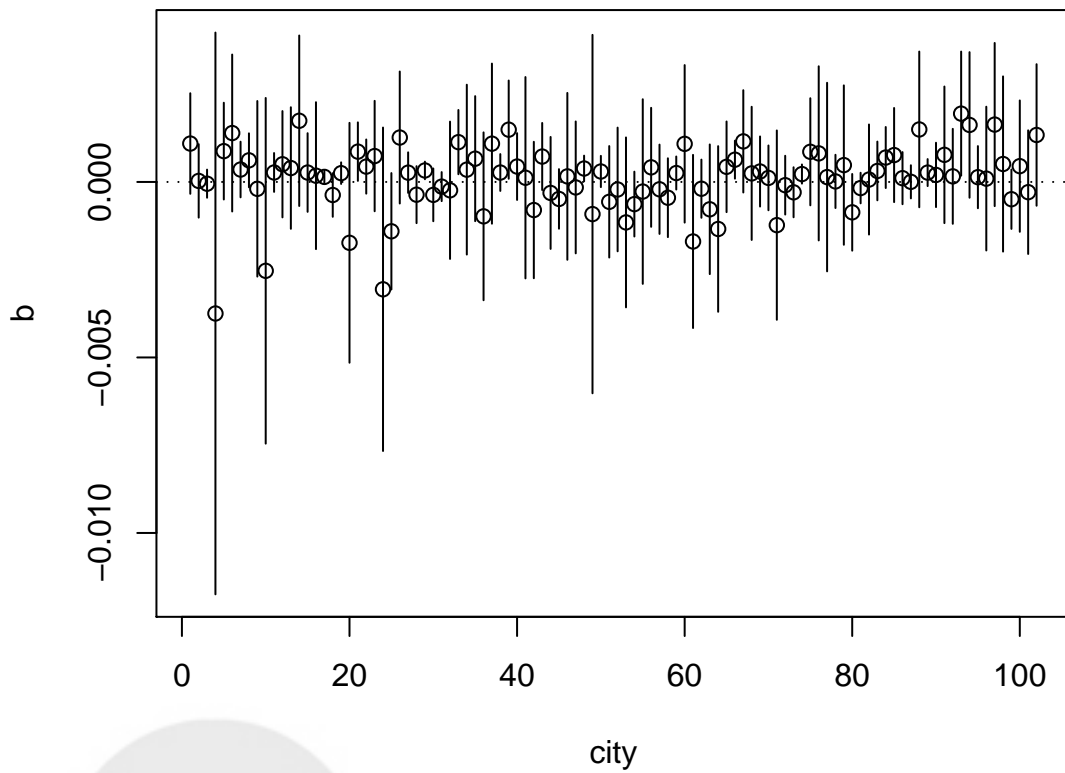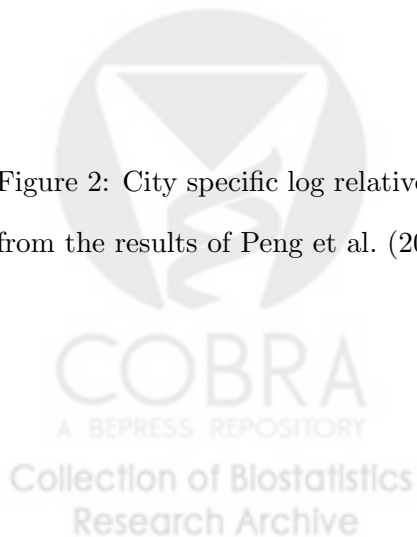Figure 1: The research pipeline as a model for reproducible research.

Figure 2: City specific log relative risks associated with a unit increase in PM_{10}, computed from the results of Peng et al. (2005).

B   Obtaining Toolkit R Packages

The three R packages described in this paper can be obtained from the Comprehensive R Archive Network (CRAN) at http://cran.r-project.org/ or at a nearby mirror (see http://cran.r-project.org/mirrors.html). The packages can be installed by running the following R function

```
install.packages(c("cacheSweave", "stashR", "SRPM"),
                 dependencies = TRUE)
```

which will download and install the `cacheSweave`, `stashR`, and SRPM packages as well as any packages on which they depend.