Yihui Xie

Dynamic Documents with R and knitr

Note

This PDF document only contains the first three chapters, released with permission of Chapman & Hall. The complete book is available on Amazon (`http://www.amazon.com/gp/product/1482203537`).

For those who are interested in writing a book with Chapman & Hall, the LyX and Rnw source files are freely available in the repository `https://github.com/yihui/knitr-book`. Hopefully this template can help beginners eliminate 90% of the possible LaTeX problems, e.g.,

- make odd-numbered pages recto, and even-numbered pages verso
- leave a desired number of blank pages (note that eventually you should remove this page, since this page is here only for trim marks to be correctly generated on blank pages)
- different page numbering styles
- correct order of the items in the table of contents
- blank pages after each chapter
- ...

To my parents

Shaobai Xie *and* Guolan Xie

Contents

Preface

We import a dataset into a statistical software package, run a procedure to get all results, then copy and paste selected pieces into a typesetting program, add a few descriptions, and finish a report. This is a common practice of writing statistical reports. There are obvious dangers and disadvantages in this process:

1. it is error-prone due to too much manual work;
2. it requires lots of human effort to do tedious jobs such as copying results across documents;
3. the workflow is barely recordable especially when it involves GUI (Graphical User Interface) operations, therefore it is difficult to reproduce;
4. a tiny change of the data source in the future will require the author(s) to go through the same procedure again, which can take nearly the same amount of time and effort;
5. the analysis and writing are separate, so close attention has to be paid to the synchronization of the two parts.

In fact, a report can be generated dynamically from program code. Just like a software package has its source code, a dynamic document is the source code of a report. It is a combination of computer code and the corresponding narratives. When we compile the dynamic document, the program code in it is executed and replaced with the output; we get a final report by mixing the code output with the narratives. Because we only manage the source code, we are free of all the possible problems above. For example, we can change a single parameter in the source code, and get a different report on the fly.

In this book, *dynamic documents* refer to the kind of source documents containing both program code and narratives. Sometimes we may just call them *source documents* since "dynamic" may sound confusing and ambiguous to some people (it does not mean interactivity or animations). We also use the term *report* frequently throughout the book, which really means the output document compiled from a dynamic document.

Who Should Read This Book

This book is written for both beginners and advanced users. The main goal is to make writing reports easier: the "report" here can range from student homework or project reports, exams, books, blogs, and web pages to virtually any documents related to statistical graphics, computing, and data analysis.

For beginners, Chapter 1 to 8 should be enough for basic applications (which have already covered many features); for power users, Chapter 9 to 11 can be helpful for understanding the extensibility of the **knitr** package.

Familiarity with LaTeX and HTML can be helpful, but is not required at all; once we get the basic idea, we can write reports in simple languages such as Markdown. Unless otherwise noted, all features apply to all document formats, although we primarily use LaTeX for examples.

We recommend the readers to take a look at the Web site RPubs (`http://rpubs.com`), which contains a large number of user-contributed documents. Hopefully they are convincing enough to show it is quick and easy to write dynamic documents.

Software Information and Conventions

The main tools we introduce in this book are the R language (R Core Team, 2013) and the **knitr** package (Xie, 2013), with which this book was written, but the language in the documents is not restricted to R; for example, we can also integrate Python, awk, and shell scripts, etc., into the reports. For document formats, we mainly use LaTeX, HTML, and Markdown.

Both R and **knitr** are available on CRAN (Comprehensive R Archive Network) as free and open-source software: `http://cran.r-project.org`. Their version information for this book can be found in the session information:

```
print(sessionInfo(), FALSE)

## R version 3.0.1 (2013-05-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## attached base packages:
```

```
## [1] stats     graphics  grDevices utils     datasets
## [6] base
##
## other attached packages:
## [1] knitr_1.4.8
##
## loaded via a namespace (and not attached):
## [1] evaluate_0.4.7 formatR_0.9    highr_0.2.4
## [4] stringr_0.6.2  tools_3.0.1
```

The **knitr** package is documented on the Web site `http://yihui.name/knitr/`, and the most important page is perhaps `http://yihui.name/knitr/options`, where we can find the complete reference for chunk options (Section 5.1.1). The development version is hosted on Github: `https://github.com/yihui/knitr`; you can always check out the latest development version, file issues/feature requests, or even participate in the development by forking the repository and making changes by yourself. There are plenty of examples in the repository `https://github.com/yihui/knitr-examples`, including both minimal and advanced examples. There is also a wiki page maintained by Frank Harrell *et al.* from the Department of Biostatistics, Vanderbilt University, which introduced several tricks and useful experience of using **knitr**: `http://biostat.mc.vanderbilt.edu`.

Unlike many other books on R, we do not add prompts to R source code in this book, and we comment out the text output by two hashes `##` by default, as you can see above. The reason for this convention is explained in Chapter 6. Package names are in bold text (e.g., **rpart**), function names in italic (e.g., *paste()*), inline code is formatted in a typewriter font (e.g., `mean(1:10, trim = 0.1)`), and filenames are in sans serif fonts (e.g., figure/foo.pdf).

Structure of the Book

Chapter 1 is an overview of dynamic documents, introducing the idea of literate programming; Chapter 2 explains why dynamic documents are important to scientific research from the viewpoint of reproducible research; Chapter 3 gives a first complete example that covers basic concepts and what we can do with **knitr**; Chapter 4 introduces a few common text editors that support **knitr**, so that it is easier to compile reports from source documents; and Chapter 5 describes the syntax for different document formats such as LaTeX, HTML, and Markdown.

Chapter 6 to 11 explain the core functionality of the package. Chapter 6 and 7 present how to control text and graphics output from **knitr**, respectively; Chapter 8 talks about the caching mechanism that may significantly reduce the computation time; Chapter 9 shows how to reuse source code by chunk references and organize child documents; Chapter 10 consists of an advanced topic — chunk hooks, which make a literate programming document really programmable and extensible; and Chapter 11 illustrates how to integrate other languages, such as Python and awk, etc. into one report in the **knitr** framework.

Chapter 12 introduces some useful tricks that make it easier to write documents with **knitr**; Chapter 13 shows how to publish reports in a variety of formats including PDF, HTML, and HTML5 slides; Chapter 14 covers a few significant applications; and Chapter 15 introduces other tools for dynamic report generation, such as Sweave, other R packages, and software in other languages. Appendix A is a guide to some internal structures of **knitr**, which may be helpful to other package developers.

The topics from Chapter 6 to 11 are parallel to each other. For example, if you want to know more about graphics output, you can skip Chapter 6 and jump to Chapter 7 directly.

In all, we will show how to improve our efficiency in writing reports, fine tune every aspect of a report, and go from program output to publication quality reports.

Acknowledgments

First, I want to thank my wireless router, which was broken when I started writing the core chapters of this book (in the boring winter of Ames). Besides, I also thank my wife for not giving me the Ethernet cable during that period.

This book would certainly not have been possible without the powerful R language, for which I thank the R core team and its contributors. The seminal work of Sweave (by Friedrich Leisch and R-core) is the most important source of inspiration of **knitr**. Some additional features were inspired by other R packages including **cacheSweave** (Roger Peng), **pgfSweave** (Cameron Bracken and Charlie Sharpsteen), **weaver** (Seth Falcon), **SweaveListingUtils** (Peter Ruckdeschel), **highlight** (Romain Francois), and **brew** (Jeffrey Horner). The initial design was based

on Hadley Wickham's **decumar** package, and the evaluator is based on his **evaluate** package. Both LyX and RStudio quickly included support to **knitr** after it came out, which made it a lot easier to write source documents, and I'd like to thank their developers (especially Jean-Marc Lasgouttes, JJ Allaire, and Joe Cheng); similarly I thank the developers of other editors such as Emacs/ESS.

The R/**knitr** user community is truly amazing. There has been a lot of feedback since the beginning of its development in late 2011. I still remember some users shouted it from the rooftops when I released the first beta version. I appreciate this kind of excitement. Hundreds of questions and comments in the mailing list (`https://groups.google.com/group/knitr`) and on StackOverflow (`http://stackoverflow.com/questions/tagged/knitr`) made this package far more powerful than I imagined. The development repository is on Github, where I have received nearly 500 issues and more than 50 pull requests (patches) from several contributors (`https://github.com/yihui/knitr/pulls`), including Ramnath Vaidyanathan, Taiyun Wei, and J.J. Allaire.

```r
# to see a full list of contributors
packageDescription("knitr", fields = "Authors@R")
```

I thank my PhD advisors at Iowa State University, Di Cook and Heike Hofmann, for their open-mindedness and consistent support for my research in this "non-classical" area of statistics.

Lastly I thank the reviewers Frank Harrell, Douglas Bates, Carl Boettiger, Joshua Wiley, and Scott Kostyshak for their valuable advice on improving the quality of this book (which is the first book of my career), and I'm grateful to the editor John Kimmel, without whom I would not have been able to publish my first book quickly.

Yihui Xie
Ames, IA

Author

Yihui Xie (`http://yihui.name`) is a PhD student in the Department of Statistics, Iowa State University. His research interests include interactive statistical graphics and statistical computing. He is an active R user and the author of several R packages, such as **animation**, **formatR**, **Rd2roxygen**, and **knitr**, among which the **animation** package won the 2009 John M. Chambers Statistical Software Award (ASA), and the **knitr** package was awarded the "Honorable Mention" prize in the "Applications of R in Business Contest 2012" thanks to Revolution Analytics.

In 2006 he founded the "Capital of Statistics" (`http://cos.name`), which has grown into a large online community on statistics in China. He initiated the first Chinese R conference in 2008 and has been organizing R conferences in China since then. During his PhD training at the Iowa State University, he won the Vince Sposito Statistical Computing Award (2011) and the Snedecor Award (2012) in the Department of Statistics.

List of Figures

List of Tables

1

Introduction

The basic idea behind dynamic documents stems from *literate programming*, a programming paradigm conceived by Donald Knuth (Knuth, 1984). The original idea was mainly for writing software: mix the source code and documentation together; we can either extract the source code out (called *tangle*) or execute the code to get the compiled results (called *weave*). A dynamic document is not entirely different from a computer program: for a dynamic document, we need to run software packages to compile our ideas (often implemented as source code) into numeric or graphical output, and insert the output into our literal writings (like documentation).

We explain the idea with a trivial example: suppose we need to write the value of 2π into a report; of course, we can directly write the number 6.2832. Now, if I change my mind and I want 6π instead, I may have to find a calculator, erase the previous value, and write the new answer. Since it is extremely easy for the computer to calculate 6π, why not leave this job to the computer completely and free oneself from this kind of manual work? What we need to do is to leave the source code in the document instead of a hard-coded value, and tell the computer how to find and execute the source code. Usually we use special markers for computer code in the source report, e.g., we can write

```
The correct answer is {{ 6 * pi }}.
```

in which `{{` and `}}` is a pair of markers that tell the computer `6 * pi` is the source code and should be executed. Note here `pi` (π) is a constant in R.

If you know a web scripting language such as PHP (which can embed program code into HTML documents), this idea should look familiar. The above example shows the *inline* code output, which means source code is mixed inline with a sentence. The other type of output is the *chunk* output, which gives the results from a whole block of code. The chunk output has much more flexibility; for example, we can produce graphics and tables from a code chunk.

Figure 1.1 was dynamically created with a chunk of R code, which is printed below:

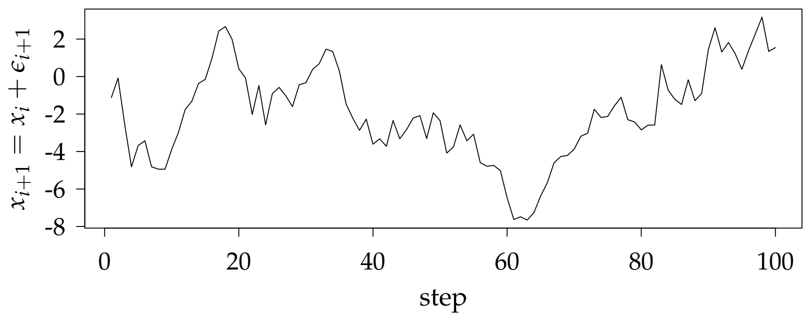


FIGURE 1.1
A simulation of the Brownian motion for 100 steps: $x_1 = \epsilon_1$, $x_{i+1} = x_i + \epsilon_{i+1}$, $\epsilon_i \overset{iid}{\sim} N(0,1)$, $i = 1, 2, \cdots, 100$

```
set.seed(1213)  # for reproducibility of random numbers
x <- cumsum(rnorm(100))
plot(x, type = "l", ylab = "$x_{i+1}=x_i + \\epsilon_{i+1}$",
    xlab = "step")
```

If we were to do this by hand, we would have to open R, paste the code into the R console to draw the plot, save it as a PDF file, and insert it into a LaTeX document with `\includegraphics{}`. This is both tedious for the author and difficult to maintain — supposing we want to change the random seed in *set.seed()*, increase the number of steps, or use a scatterplot instead of a line graph, we will have to update both the source code and the output. In practice, the computing and analysis can be far more complicated than the toy example in Figure 1.1, and more manual work will be required accordingly.

The spirit of dynamic documents may best be described by the philosophy of the ESS project (Rossini et al., 2004) for the S language:

> The source code is real.
>
> Philosophy for using ESS[S]

Since the output can be produced by the source code, we can maintain the source code only. However, in most cases, the direct output from the source code alone does not constitute a report that is readable for a human. That is why we need the literate programming paradigm. In this paradigm, an author has two tasks:

1. write program code to do computing, and
2. write narratives to explain what is being done by the program code

The traditional approach to doing the second task is to write comments for the code, but comments are often limited in terms of expressing the full thoughts of the authors. Normally we write our ideas in a paper or a report instead of hundreds of lines of code comments.

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.
>
> Donald E. Knuth
> Literate Programming, 1984

Technically, literate programming involves three steps:

1. parse the source document and separate code from narratives
2. execute source code and return results
3. mix results from the source code with the original narratives

These steps can be implemented in software packages, so the authors do not need to take care of these technical details. Instead, we only control what the output should look like. There are many details that we can tune for a report (especially for reports related to data analysis), although the idea of literate programming seems to be simple. For example, data reports often include tables, and Table 1.1 is a table generated from the R code below using the *kable()* function in **knitr**:

```r
library(knitr)
kable(head(mtcars[, 1:6]))
```

Think how easy it is to maintain two lines of R code compared to maintaining many lines of messy LaTeX code!

TABLE 1.1
A subset of the `mtcars` dataset: the first 6 rows and 6 columns.

	mpg	cyl	disp	hp	drat	wt
Mazda RX4	21.0	6	160	110	3.90	2.620
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875
Datsun 710	22.8	4	108	93	3.85	2.320
Hornet 4 Drive	21.4	6	258	110	3.08	3.215
Hornet Sportabout	18.7	8	360	175	3.15	3.440
Valiant	18.1	6	225	105	2.76	3.460

Generating reports dynamically by integrating computer code with narratives is not only easier, but also closely related to reproducible research, which we will discuss in the next chapter.

2

Reproducible Research

Results from scientific research have to be reproducible to be trustworthy. We do not want a finding to be merely due to an isolated occurrence, e.g., only one specific laboratory researcher can produce the results on one specific day, and nobody else can produce the same results under the same conditions.

Reproducible research (RR) is one possible by-product of dynamic documents, but dynamic documents do not absolutely guarantee RR. Because there is usually no human intervention when we generate a report dynamically, it is likely to be reproducible since it is relatively easy to prepare the same software and hardware environment, which is everything we need to reproduce the results. However, the meaning of reproducibility can be beyond reproducing one result or one report. As a trivial example, one might have done a Monte Carlo simulation with a certain random seed and got a good estimate of a parameter, but the result was actually due to a "lucky" random seed. Although we can strictly reproduce the estimate, it is not actually reproducible in the general sense. Similar problems exist in optimization algorithms, e.g., different starting values can lead to different roots of the same equation.

Anyway, dynamic report generation is still an important step towards RR. In this chapter, we discuss a selection of the RR literature and practices of RR.

2.1 Literature

The term reproducible research was first proposed by Jon Claerbout at Stanford University (Fomel and Claerbout, 2009). The idea is that the final product of research is not only the paper itself, but also the full computational environment used to produce the results in the paper such as the code and data necessary for reproduction of the results and building upon the research.

Similarly, Buckheit and Donoho (1995) pointed out the essence of the scholarship of an article as follows:

> An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.
>
> D. Donoho
> WaveLab and Reproducible Research

That was well said! Fortunately, journals have been moving in that direction as well. For example, Peng (2009) provided detailed instructions to authors on the criteria of reproducibility and how to submit materials for reproducing the paper in the *Biostatistics* journal.

At the technical level, RR is often related to literate programming (Knuth, 1984), a paradigm conceived by Donald Knuth to integrate computer code with software documentation in one document. However, early implementations like WEB (Knuth, 1983) and Noweb (Ramsey, 1994) were not directly suitable for data analysis and report generation. There are other tools on this path of documentation generation, such as **roxygen2** (Wickham et al., 2013), which is an R implementation of Doxygen (van Heesch, 2008). Sweave (Leisch, 2002) was among the first implementations for dealing with dynamic documents in R (Ihaka and Gentleman, 1996; R Core Team, 2013). There are still a number of challenges that were not solved by the existing tools; for example, Sweave is closely tied to LaTeX and hard to extend. The **knitr** package (Xie, 2013) was built upon the ideas of previous tools with a framework redesign, enabling easy and fine control of many aspects of a report. We will introduce other tools in Chapter 15.

An overview of literate programming applied to statistical analysis can be found in Rossini (2002). Gentleman and Temple Lang (2004) introduced general concepts of literate programming documents for statistical analysis, with a discussion of the software architecture. Gentleman (2005) is a practical example based on Gentleman and Temple Lang (2004), using an R package **GolubRR** to distribute reproducible analysis. Baggerly et al. (2004) revealed several problems that may arise with the standard practice of publishing data analysis results, which can lead to false discoveries due to lack of details for reproducibility (even with datasets supplied). Instead of separating results from computing, we can put everything in one document (called a *compendium* in Gentleman and Temple Lang (2004)), including the computer code and narratives. When we compile this document, the computer code will be executed, giving us the results directly.

2.2 Good and Bad Practices

The key to keep in mind for RR is that other people should be able to reproduce our results, therefore we should try our best to make our computation *portable*. We discuss some good practices for RR below and explain why it can be bad not to follow them.

- Manage all source files under the same directory and use relative paths whenever possible: absolute paths can break reproducibility, e.g., a data file like C:/Users/someone/foo.csv or /home/someone/foo.csv may only exist in one computer, and other people may not be able to read it since the absolute path is likely to be different in their hard disk. If we keep everything under the same directory, we can read a data file with `read.csv('foo.csv')` (if it is under the current working directory) or `read.csv('../data/foo.csv')` (go one level up and find the file under the data/ directory); when we disseminate the results, we can make an archive of the whole directory (e.g., as a zip package).

- Do not change the working directory after the computing has started: *setwd()* is the function in R to set the working directory, and it is not uncommon to see `setwd('path/to/some/dir')` in user's code, which is bad because it is not only an absolute path, but also has a global effect on the rest of the source document. In that case, we have to keep in mind that all relative paths may need adjustments since the root directory has changed, and the software may write the output in an unexpected place (e.g., the figures are expected to be generated in the ./figures/ directory, but are actually written to ./data/figures/ instead if we `setwd('./data/')`). If we have to set the working directory at all, do it in the very beginning of an R session; most of the editors to be introduced in Chapter 4 follow this rule, and the working directory is set to the directory of the source document before **knitr** is called to compile documents.

- Compile the documents in a clean R session: existing R objects in the current R session may "contaminate" the results in the output. It is fine if we write a report by accumulating code chunks one by one and running them interactively to check the results, but in the end we should compile a report in the batch mode with a new R session so all the results are freshly generated from the code.

- Avoid the commands that require human interaction: human input can be highly unpredictable, e.g., we do not know for sure which file the user will choose if we pop up a dialog box asking the user to choose a data file. Instead of using functions like *file.choose()* to input a file to *read.table()*, we should write the filename explicitly, e.g., `read.table('a-specific-file.txt')`.

- Avoid environment variables for data analysis: while environment variables are often heavily used in programming for configuration purposes, it is ill-advised to use them in data analysis because they require additional instructions for users to set up, and humans can simply forget to do this. If there are any options to set up, do it inside the source document.

- Attach *sessionInfo()* and instructions on how to compile this document: the session information makes a reader aware of the software environment, such as the version of R, the operating system and add-on packages used. Sometimes it is not as simple as calling one single function to compile a document, and we have to make it clear how to compile it if additional steps are required; but it is better to provide the instructions in the form of a computer script, e.g., a shell script, a Makefile, or a batch file.

These practices are not necessarily restricted to the R language, although we used R for examples. The same rules also apply to other computing environments.

Note that literate programming tools often require users to compile the documents in batch mode, and it is good for reproducible research, but the batch mode can be cumbersome for exploratory data analysis. When we have not decided what to put in the final document, we may need to interact with the data and code frequently, and it is not worth compiling the whole document each time we update the code. This problem can be solved by a capable editor such as RStudio and Emacs/ESS, which are introduced in Chapter 4. In these editors, we can interact with the code and explore the data freely (e.g., send or write R code in an associated R session), and once we finish the coding work, we can compile the whole document in the batch mode to make sure all the code works in a clean R session.

2.3 Barriers

Despite all the advantages of RR, there are some practical barriers, and here is a non-exhaustive list:

- the data can be huge: for example, it is common that high energy physics and next-generation sequencing data in biology can produce tens of terabytes of data, and it is not trivial to archive the data with the reports and distribute them
- confidentiality of data: it may be prohibited to release the raw data with the report, especially when it is involved with human subjects due to the confidentiality issues
- software version and configuration: a report may be generated with an old version of a software package that is no longer available, or with a software package that compiles differently on different operating systems
- competition: one may choose not release the code or data with the report due to the fact that potential competitors can easily get everything for free, whereas the original authors have invested a large amount of money and effort

We certainly should not expect all reports in the world to be publicly available and strictly reproducible, but it is better to share even mediocre or flawed code or problematic datasets than not to share anything at all. Instead of persuading people into RR by policies, we may try to create tools that make RR easier than cut-and-paste, and **knitr** is such an attempt. The success of RPubs (`http://rpubs.com`) is evidence that an easy tool can quickly promote RR, because users enjoy using it. Readers can find hundreds of reports contributed by users in the above Web site. It is fairly common to see student homework and exercises there, and once the students are trained in this manner, we may expect more reproducible scientific research in the future.

3

A First Look

The **knitr** package is a general-purpose literate programming engine — it supports document formats including LaTeX, HTML, and Markdown (see Chapter 5), and programming languages such as R, Python, awk, C++, and shell scripts (Chapter 11). Before we get started, we need to install **knitr** in R. Then we will introduce the basic concepts with minimal examples. Finally, we will show how to generate reports quickly from pure R scripts, which can be useful for beginners who do not know anything about dynamic documents.

3.1 Setup

Since **knitr** is an R package, it can be installed from CRAN in the usual way in R:

```
install.packages("knitr", dependencies = TRUE)
```

Note here that `dependencies = TRUE` is optional, and will install all packages that are not absolutely necessary but can enhance this package with some useful features. The development version is hosted on Github: `https://github.com/yihui/knitr`, and you can always check out the latest development version, which may not be stable but contains the latest bug fixes and new features. If you have any problems with **knitr**, the first thing to check is its version:

```
packageVersion("knitr")
# if not the latest version, run
update.packages()
```

If you choose LaTeX as the typesetting tool, you may need to install MiKTeX (Windows, `http://miktex.org/`), MacTeX (Mac OS, `http://tug.org/mactex/`) or TeXLive (Linux, `http://tug.org/texlive/`). If you are going to work with HTML or Markdown, nothing else needs to be installed, since the output will be Web pages, which you can view with a Web browser.

Once we have **knitr** installed, we can compile source documents using the function *knit()*, e.g.,

```
library(knitr)
knit("your-file.Rnw")
```

A *.Rnw file is usually a LaTeX document with R code embedded in it, as we will see in the following section and Chapter 5, in which more types of documents will be introduced.

3.2 Minimal Examples

We use two minimal examples written in LaTeX and Markdown, respectively, to illustrate the structure of dynamic documents. We do not discuss the syntax of LaTeX or Markdown for the time being (see Chapter 5 instead). For the sake of simplicity, the `cars` dataset in base R is used to build a simple linear regression model. Type `?cars` in R to see detailed documentation. Basically it has two variables, speed and distance:

```
str(cars)

## 'data.frame': 50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

3.2.1 An Example in LaTeX

Figure 3.1 is a full example of R code embedded in LaTeX; we call this kind of documents *Rnw documents* hereafter because their filename extension is Rnw by convention. If we save it as a file minimal.Rnw and run `knit('minimal.Rnw')` in R as described before, **knitr** will generate a LaTeX output document named minimal.tex. For those who are familiar with LaTeX, you can compile this document to PDF via `pdflatex`. Figure 3.2 is the PDF document compiled from the Rnw document.

What is essential here is how we embedded R code in LaTeX. In an Rnw document, <<>>= marks the beginning of code chunks, and `@` terminates a code chunk (this description is not rigorous but is often easier

```
\documentclass{article}
\begin{document}
\title{A Minimal Example}
\author{Yihui Xie}
\maketitle

We examine the relationship between speed and stopping
distance using a linear regression model:
$Y = \beta_0 + \beta_1 x + \epsilon$.

<<model, fig.width=4, fig.height=3, fig.align='center'>>=
par(mar = c(4, 4, 1, 1), mgp = c(2, 1, 0), cex = 0.8)
plot(cars, pch = 20, col = 'darkgray')
fit <- lm(dist ~ speed, data = cars)
abline(fit, lwd = 2)
@

The slope of a simple linear regression is
\Sexpr{coef(fit)[2]}.
\end{document}
```

FIGURE 3.1
The source of a minimal Rnw document: see output in Figure 3.2.

to understand); we have four lines of R code between the two markers in this example to draw a scatterplot, fit a linear model, and add a regression line to the scatterplot. The command `\Sexpr{}` is used to embed inline R code, e.g., `coef(fit)[2]` in this example. We can write chunk options for a code chunk between << and >>=; the chunk options in this example specified the plot size to be 4 by 3 inches (`fig.width` and `fig.height`), and plots should be aligned in the center (`fig.align`).

In this minimal example, we have most basic elements of a report:

1. title, author, and date
2. model description
3. data and computation
4. graphics
5. numeric results

All the output is generated dynamically from R. Even if the data has

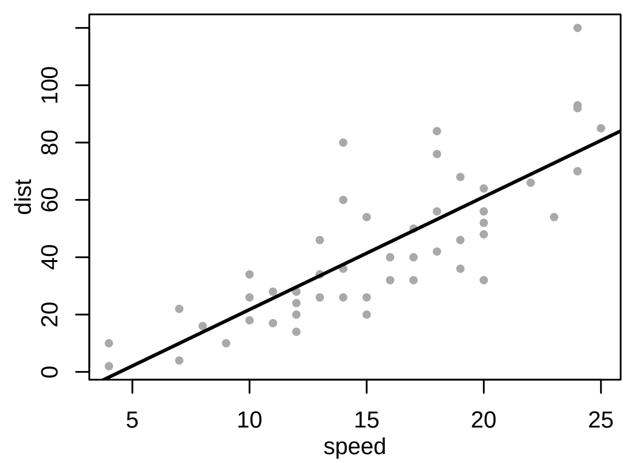
A Minimal Example

Yihui Xie

September 5, 2013

We examine the relationship between speed and stopping distance using a linear regression model: $Y = \beta_0 + \beta_1 x + \epsilon$.

```
par(mar = c(4, 4, 1, 1), mgp = c(2, 1, 0), cex = 0.8)
plot(cars, pch = 20, col = "darkgray")
fit <- lm(dist ~ speed, data = cars)
abline(fit, lwd = 2)
```



The slope of a simple linear regression is 3.9324.

FIGURE 3.2
A minimal example in LaTeX with an R code chunk, a plot, and numeric output (regression coefficient).

````
# A Minimal Example

We examine the relationship between speed and stopping
distance using a linear regression model:
$Y = \beta_0 + \beta_1 x + \epsilon$.

```{r model, fig.width=4, fig.height=3, fig.align='center'}
par(mar = c(4, 4, 1, 1), mgp = c(2, 1, 0), cex = 0.8)
plot(cars, pch = 20, col = 'darkgray')
fit <- lm(dist ~ speed, data = cars)
abline(fit, lwd = 2)
```

The slope of a simple linear regression is `r coef(fit)[2]`.
````

FIGURE 3.3
The source of a minimal Rmd document: see output in Figure 3.4.

changed, we do not need to redo the report from the ground up, and the output will be updated accordingly if we update the data and recompile the report.

3.2.2 An Example in Markdown

LaTeX may look overwhelming to beginners due to the large number of commands. By comparison, Markdown (Gruber, 2004) is a much simpler format. Figure 3.3 is a Markdown example doing the same analysis with the previous example:

The ideal output from Markdown is an HTML Web page, as shown in Figure 3.4 (in Mozilla Firefox). Similarly, we can see the syntax for R code in a Markdown document: ```{r} opens a code chunk, ``` terminates a chunk, and inline R code can be put inside `r `, where ` is a backtick.

A slightly longer example in **knitr** is a demo named `notebook`, which is based on Markdown. It shows not only the potential power of Markdown, but also the possibility of building Web applications with **knitr**. To watch the demo, run the code below:

```r
if (!require("shiny")) install.packages("shiny")
demo("notebook", package = "knitr")
```

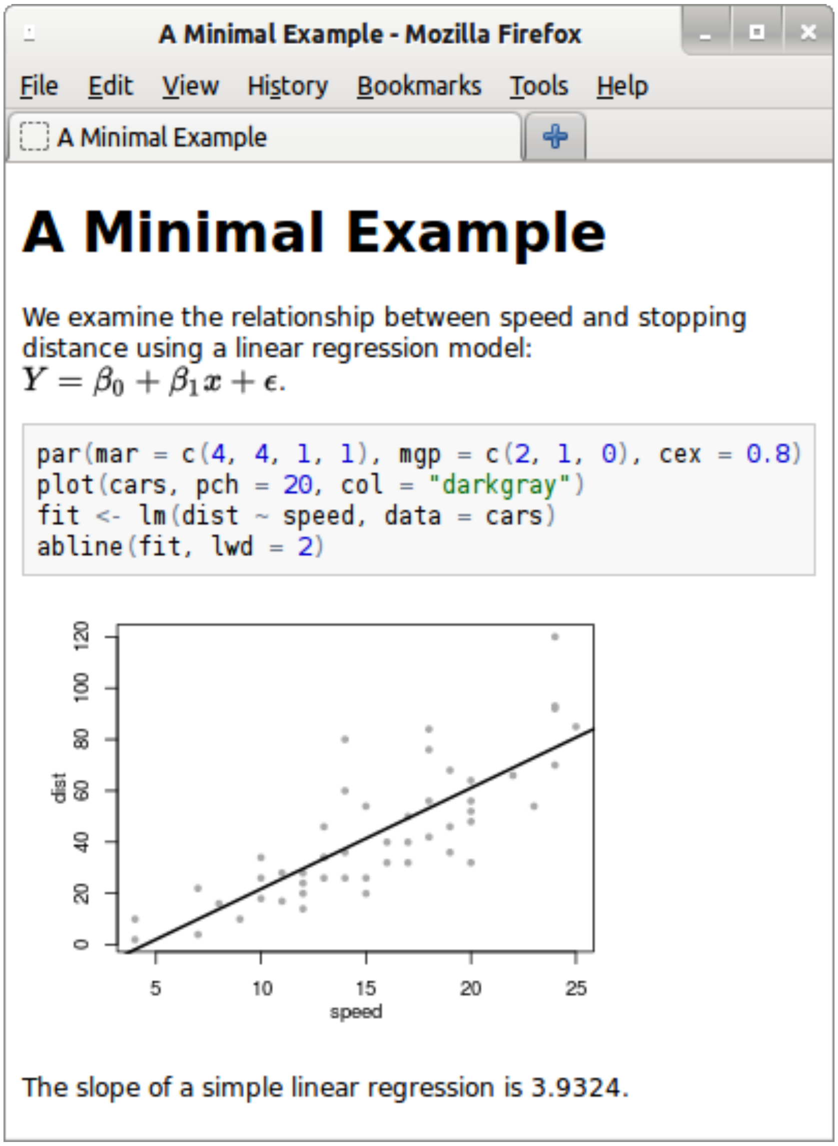


FIGURE 3.4
A minimal example in Markdown with the same analysis as in Figure 3.2, but the output is HTML instead of PDF now.

Your default Web browser will be launched to show a Web notebook. The source code is in the left panel, and the live results are in the right panel. You are free to experiment with the source code and re-compile the notebook.

3.3 Quick Reporting

If a user only has basic knowledge of R but knows nothing about **knitr**, or one does not want to write anything other than an R script, it is also possible to generate a quick report from this R script using the *stitch()* function.

The basic idea of *stitch()* is that **knitr** provides a template of the source document with some default settings, so that the user only needs to feed this template with an R script (as one code chunk); then **knitr** will compile the template to a report. Currently it has built-in templates for LaTeX, HTML, and Markdown. The usage is like this:

```
library(knitr)
stitch("your-script.R")
```

3.4 Extracting R Code

For a literate programming document, we can either compile it to a report (run the code), or extract the program code in it. They are called "weaving" and "tangling," respectively. Apparently the function *knit()* is for weaving, and the corresponding tangling function is *purl()* in **knitr**. For example,

```
library(knitr)
purl("your-file.Rnw")
purl("your-file.Rmd")
```

The result of tangling is an R script; in the above examples, the default output will be your-file.R, which consists of all code chunks in the source document.

So far we have been introducing the command line usage of **knitr**, and it is often tedious to type the commands repeatedly. In the next chapter, we show how a decent editor can help edit and compile the source document with one single mouse click or a keyboard shortcut.

4

Editors

4.1 RStudio

4.2 LyX

4.3 Emacs/ESS

4.4 Other Editors

5

Document Formats

6

Text Output

6.1 Inline Output

6.2 Chunk Output

6.2.1 Chunk Evaluation

6.2.2 Code Formatting

6.2.3 Code Decoration

6.2.4 Show/Hide Output

6.3 Tables

6.4 Themes

7

Graphics

7.1 Graphical Devices

7.1.1 Custom Device

7.1.2 Choose a Device

7.1.3 Device Size

7.1.4 More Device Options

7.1.5 Encoding

7.1.6 The Dingbats Font

7.2 Plot Recording

7.3 Plot Rearrangement

7.3.1 Animation

7.3.2 Alignment

7.4 Plot Size in Output

7.5 Extra Output Options

7.6 The tikz Device

7.7 Figure Environment

7.8 Figure Path

8

Cache

8.1 Implementation

8.2 Write Cache

8.3 When to Update Cache

8.4 Side Effects

8.5 Chunk Dependencies

8.5.1 Manual Dependency

8.5.2 Automatic Dependency

9

Cross Reference

9.1 Chunk Reference

9.1.1 Embed Code Chunks

9.1.2 Reuse Whole Chunks

9.2 Code Externalization

9.2.1 Labeled Chunks

9.2.2 Line-based Chunks

9.3 Child Documents

9.3.1 Input Child Documents

9.3.2 Child Documents as Templates

9.3.3 Standalone Mode

10

Hooks

10.1 Chunk Hooks

10.1.1 Create Chunk Hooks

10.1.2 Trigger Chunk Hooks

10.1.3 Hook Arguments

10.1.4 Hooks and Chunk Options

10.1.5 Write Output

10.2 Examples

10.2.1 Crop Plots

10.2.2 rgl Plots

10.2.3 Manually Save Plots

10.2.4 Optimize PNG Plots

10.2.5 Close an rgl Device

10.2.6 WebGL

11

Language Engines

11.1 Design

11.1.1 The Engine Function

11.1.2 Engine Options

11.2 Languages and Tools

11.2.1 C++

11.2.2 Interpreted Languages

11.2.3 TikZ

11.2.4 Graphviz

11.2.5 Highlight

12

Tricks and Solutions

12.1 Chunk Options

12.1.1 Option Aliases

12.1.2 Option Templates

12.1.3 Program Chunk Options

12.1.4 Code in Appendix

12.2 Package Options

12.3 Typesetting

12.3.1 Output Width

12.3.2 Message Colors

12.3.3 Box Padding

12.3.4 Beamer

12.3.5 Suppress Long Output

12.3.6 Escape Special Characters

12.3.7 The Example Environment

12.4 Utilities

12.4.1 R Package Citation

12.4.2 Image URI

12.4.3 Upload Images

12.4.4 Compile Documents

12.4.5 Construct Code Chunks

12.4.6 Extract Source Code

12.4.7 Reproducible Simulation

12.4.8 R Documentation

12.4.9 Rst2pdf

12.4.10 Package Demos

12.4.11 Pretty Printing

12.4.12 A Macro Preprocessor

12.5 Debugging

13

Publishing Reports

13.1 RStudio

13.2 Pandoc

13.3 HTML5 Slides

13.4 Jekyll

13.5 WordPress

14

Applications

14.1 Homework

14.2 Web Site and Blogging

14.2.1 Vistat and Rcpp Gallery

14.2.2 UCLA R Tutorial

14.2.3 The cda and RHadoop Wiki

14.2.4 The ggbio Package

14.2.5 Geospatial Data in R and Beyond

14.3 Package Vignettes

14.3.1 PDF Vignette

14.3.2 HTML Vignette

14.4 Books

14.4.1 This Book

14.4.2 The Analysis of Data

14.4.3 The Statistical Sleuth in R

15

Other Tools

15.1 Sweave

15.1.1 Syntax

15.1.2 Options

15.1.3 Problems

15.2 Other R Packages

15.3 Python Packages

15.3.1 Dexy

15.3.2 PythonTeX

15.3.3 IPython

15.4 More Tools

15.4.1 Org-mode

15.4.2 SASweave

15.4.3 Office

A

Internals

A.1 Documentation

A.2 Closures

A.3 Implementation

A.3.1 Parser

A.3.2 Chunk Hooks

A.3.3 Option Aliases

A.3.4 Cache

A.3.5 Compatibility with Sweave

A.3.6 Concordance

A.4 Syntax

Bibliography

Baggerly, K. A., Morris, J. S., and Coombes, K. R. (2004). Reproducibility of seldi-tof protein patterns in serum: comparing datasets from different experiments. *Bioinformatics*, 20(5):777–785.

Buckheit, J. and Donoho, D. (1995). Wavelab and reproducible research. *Wavelets and statistics*, 103:55.

Fomel, S. and Claerbout, J. (2009). Guest editors' introduction: Reproducible research. *Computing in Science & Engineering*, 11(1):5–7.

Gentleman, R. (2005). Reproducible research: A bioinformatics case study. *Statistical Applications in Genetics and Molecular Biology*, 4(1):1034.

Gentleman, R. and Temple Lang, D. (2004). Statistical analyses and reproducible research. *Bioconductor Project Working Papers*. URL: `http://biostats.bepress.com/bioconductor/paper2`.

Gruber, J. (2004). *The Markdown Project*. URL: `http://daringfireball.net/projects/markdown/`.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314.

Knuth, D. E. (1983). The WEB system of structured documentation. Technical report, Department of Computer Science, Stanford University.

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In *COMPSTAT 2002 Proceedings in Computational Statistics*, number 69, pages 575–580. Heidelberg: Physica Verlag.

Peng, R. (2009). Reproducible research and biostatistics. *Biostatistics*, 10(3):405–408.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ramsey, N. (1994). Literate programming simplified. *Software, IEEE*, 11(5):97–105.

Rossini, A. (2002). Literate statistical analysis. In *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, pages 15–17, Vienna, Austria.

Rossini, A., Heiberger, R., Sparapani, R., Maechler, M., and Hornik, K. (2004). Emacs speaks statistics: A multiplatform, multipackage development environment for statistical analysis. *Journal of Computational and Graphical Statistics*, 13(1):247–261.

van Heesch, D. (2008). Doxygen: Source code documentation generator tool. URL: `http://www.doxygen.org/`.

Wickham, H., Danenberg, P., and Eugster, M. (2013). *roxygen2: In-source documentation for R*. R package version 2.2.2.

Xie, Y. (2013). *knitr: A general-purpose package for dynamic report generation in R*. R package version 1.4.7.

Index