The R Series

Implementing Reproducible Research

Edited by
Victoria Stodden
Friedrich Leisch
Roger D. Peng

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

The R Series

Implementing Reproducible Research

Edited by

Victoria Stodden

Columbia University
New York, New York, USA

Friedrich Leisch

University of Natural Resources and Life Sciences
Institute of Applied Statistics and Computing
Vienna, Austria

Roger D. Peng

Johns Hopkins University
Baltimore, Maryland, USA

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK

Chapman & Hall/CRC
The R Series

Series Editors

John M. Chambers
Department of Statistics
Stanford University
Stanford, California, USA

Torsten Hothorn
Division of Biostatistics
University of Zurich
Switzerland

Duncan Temple Lang
Department of Statistics
University of California, Davis
Davis, California, USA

Hadley Wickham
Department of Statistics
Rice University
Houston, Texas, USA

Aims and Scope

This book series reflects the recent rapid growth in the development and application of R, the programming language and software environment for statistical computing and graphics. R is now widely used in academic research, education, and industry. It is constantly growing, with new versions of the core software released regularly and more than 4,000 packages available. It is difficult for the documentation to keep pace with the expansion of the software, and this vital book series provides a forum for the publication of books covering many aspects of the development and application of R.

The scope of the series is wide, covering three main threads:

- Applications of R to specific disciplines such as biology, epidemiology, genetics, engineering, finance, and the social sciences.
- Using R for the study of topics of statistical methodology, such as linear and mixed modeling, time series, Bayesian methods, and missing data.
- The development of R, including programming, building packages, and graphics.

The books will appeal to programmers and developers of R software, as well as applied statisticians and data analysts in many fields. The books will feature detailed worked examples and R code fully integrated into the text, ensuring their usefulness to researchers, practitioners and students.

Published Titles

Analyzing Baseball Data with R, *Max Marchi and Jim Albert*

Customer and Business Analytics: Applied Data Mining for Business Decision Making Using R, *Daniel S. Putler and Robert E. Krider*

Dynamic Documents with R and knitr, *Yihui Xie*

Event History Analysis with R, *Göran Broström*

Implementing Reproducible Research, *Victoria Stodden, Friedrich Leisch, and Roger D. Peng*

Programming Graphical User Interfaces with R, *Michael F. Lawrence and John Verzani*

R Graphics, Second Edition, *Paul Murrell*

Reproducible Research with R and RStudio, *Christopher Gandrud*

Statistical Computing in C++ and R, *Randall L. Eubank and Ana Kupresanin*

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20131025

International Standard Book Number-13: 978-1-4665-6159-5 (Hardback)

Library of Congress Cataloging-in-Publication Data

Implementing reproducible research / [edited by] Victoria Stodden, Friedrich Leisch, and Roger D. Peng.
pages cm. -- (Chapman & Hall/CRC the R series)
Includes bibliographical references and index.
ISBN 978-1-4665-6159-5
1. Reproducible research. 2. Research--Statistical methods. I. Stodden, Victoria, editor of compilation. II. Leisch, Friedrich, editor of compilation. III. Peng, Roger D., editor of compilation.

Q180.55.S7I47 2013
507.2--dc23 2013036694

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

Science moves forward when discoveries are replicated and reproduced. In general, the more frequently a given relationship is observed by independent scientists, the more trust we have that such a relationship truly exists in nature. Replication, the practice of independently implementing scientific experiments to validate specific findings, is the cornerstone of discovering scientific truth. Related to replication is reproducibility, which is the calculation of quantitative scientific results by independent scientists using the original datasets and methods. Reproducibility can be thought of as a different standard of validity from replication because it forgoes independent data collection and uses the methods and data collected by the original investigator (Peng et al., 2006). Reproducibility has become an important issue for more recent research due to advances in technology and the rapid spread of computational methods across the research landscape.

Much has been written about the rise of computational science and the complications computing brings to the traditional practice of science (Bailey et al. 2013; Birney et al. 2009; Donoho et al. 2009; Peng 2011; Stodden 2012; Stodden et al. 2013; Yale Roundtable 2010). Large datasets, fast computers, and sophisticated statistical modeling make a powerful combination for scientific discovery. However, they can also lead to a lack of reproducibility in computational science findings when inappropriately applied to the discovery process. Recent examples show that improper use of computational tools and software can lead to spectacularly incorrect results (e.g., Coombes et al. 2007). Making computational research reproducible does not guarantee correctness of all results, but it allows for quickly building on sound results and for rapidly rooting out unsound ones.

The sharing of analytic data and the computer codes used to map those data into computational results is central to any comprehensive definition of reproducibility. Except for the simplest of analyses, the computer code used to analyze a dataset is the only record that permits others to fully understand what a researcher has done. The traditional materials and methods sections in most journal publications are simply too short to allow for the inclusion of critical details that make up an analysis. Often, seemingly innocuous details can have profound impacts on the results, particularly when the relationships being examined are inherently weak. Some concerns have been raised over the sharing of code and data. For example, the sharing of data may allow other competing scientists to analyze the data and scoop the scientists who originally published the data, or the sharing of code may lead to the inability to monetize software through proprietary versions of the code. While these concerns are real and have not been fully resolved by the scientific community, we do not dwell on them in this book.

This book is focused on a simple question. Assuming one agrees that reproducibility of a scientific result is a good thing, *how do we do it*? In computational science, reproducibility requires that one make code and data available to others so that they may analyze the original data in a similar manner as in the original publication. This task requires that the analysis be done in such a way that preserves the code and data, and permits their distribution in a format that is generally readable, and a platform be available to the author on which the data and code can be distributed widely. Both data and code need to be licensed permissively enough so that others can reproduce the work without a substantial legal burden.

In this book, we cover many of the ingredients necessary for conducting and distributing reproducible research. The book is divided into three parts that cover the three principal areas: tools, practices, and platforms. Each part contains contributions from leaders in the area of reproducible research who have materially contributed to the area with software or other products.

Tools

Literate statistical programming is a concept introduced by Rossini, which builds on the idea of literate programming as described by Donald Knuth. With literate statistical programming, one combines the description of a statistical analysis and the code for doing the statistical analysis into a single document. Subsequently, one can take the combined document and produce either a human-readable document (i.e., PDF) or a machine-readable code file. An early implementation of this concept was the Sweave system of Leisch, which uses R as its programming language and LaTeX as its documentation language. Yihui Xie describes his knitr package, which builds substantially on Sweave and incorporates many new ideas developed since the initial development of Sweave. Along these lines, Tanu Malik and colleagues describe the Science Object Linking and Embedding framework for creating interactive publications that allow authors to embed various aspects of computational research in a document, creating a complete research compendium.

There have been a number of systems developed recently that are designed to track the provenance of data analysis outputs and to manage a researcher's workflow. Juliana Freire and colleagues describe the VisTrails system for open source provenance management for scientific workflow creation. VisTrails interfaces with existing scientific software and captures the inputs, outputs, and code that produced a particular result, even presenting this workflow in flowchart form. Andrew Davison and colleagues describe the Sumatra toolkit for reproducible research. Their goal is to introduce a tool for reproducible research that minimizes the disruption to scientists' existing

workflows, therefore maximizing the uptake by current scientists. Their tool serves as a kind of "backend" to keep track of the code, data, and dependencies as a researcher works. This allows for easily reproducing specific analyses and for sharing with colleagues.

Philip Guo takes the "backend tracking" idea one step further and describes his Code, Data, Environment (CDE) package, which is a minimal "virtual machine" for reproducing the environment as well as the analysis. This package keeps track of all files used by a given program (i.e., a statistical analysis program) and bundles everything, including dependencies, into a single package. This approach guarantees that all requirements are included and that a given analysis can be reproduced on another computer.

Peter Murray-Rust and Dave Murray-Rust introduce The Declaraton, a tool for the precise mapping of mathematical expressions to computational implementations. They present an example from materials science, defining what reproducibility means in this field, in particular for unstable dynamical systems.

Practices and Guidelines

Conducting reproducible research requires more than the existence of good tools. Ensuring reproducibility requires the integration of useful tools into a larger workflow that is rigorous in keeping track of research activities. One metaphor is that of the lab notebook, now extended to computational experiments. Jarrod Millman and Fernando Pérez raise important points about how computational scientists should be trained, noting that many are not formally trained in computing, but rather pick up skills "on the go." They detail skills and tools that may be useful to computational scientists and describe a web-based notebook system developed in IPython that can be used to combine text, mathematics, computation, and results into a reproducible analysis. Titus Brown discusses tools that can be useful in the area of bioinformatics as well as good programming practices that can apply to a broad range of areas.

Holger Hoeing and Anthony Rossini present a case study in how to produce reproducible research in a commercial environment for large-scale data analyses involving teams of investigators, analysts, and stakeholders/clients. All scientific practice, whether in academia or industry, can be informed by the authors' experiences and the discussion of tools they used to organize their work.

Closely coupled with the idea of reproducibility is the notion of "open science," whereby results are made available to the widest audience possible through journal publications or other means. Luis Ibanez and colleagues give some thoughts on open science and reproducibility and trends that are

either encouraging or discouraging it. Bill Howe discusses the role of cloud computing in reproducible research. He describes how virtual machines can be used to replicate a researcher's entire software environment and allow researchers to easily transfer that environment to a large number of people. Other researchers can then copy this environment and conduct their own research without having to go through the difficult task of reconstructing the environment from scratch.

Members of the Open Science Collaboration outline the need for reproducibility in all science and detail why most scientific findings are rarely reproduced. Reasons include a lack of incentives on the part of journals and investigators to publish reproductions or null findings. He describes the Reproducibility Project, whose goal is to estimate the reproducibility of scientific findings in psychology. This massive undertaking represents a collaboration of over 100 scientists to reproduce a sample of findings in the psychology literature. By spreading the effort across many people, the project overcomes some of the disincentives to reproducing previous work.

Platforms

Related to the need for good research practices to promote reproducibility is the need for software and methodological platforms on which reproducible research can be conducted and distributed. Mikio Braun and Cheng Soon Ong discuss the area of machine learning and place it in the context of open source software and open science. Aspects of the culture of machine learning have led to many open source software packages and hence reproducible methods.

Christophe Hurlin and colleagues, in Chapter 14, describe the RunMyCode platform for sharing reproducible research. This chapter addresses a critical need in the area of reproducible research, which is the lack of central infrastructure for distributing results. A key innovation of this platform is the use of cloud computing to allow research findings to be reproduced through the RunMyCode web interface, or on the user's local system via code and data download.

Perhaps the oldest "platform" for distributing research is the journal. Iain Hrynaszkiewicz and colleagues describe some of the infrastructure available for publishing reproducible research. In particular, they review how journal policies and practices in the growing field of open access journals encourage reproducible research.

Victoria Stodden provides a primer on the current legal and policy framework for publishing reproducible scientific work. While the publication of traditional articles is rather clearly covered by copyright law, the publication

of data and code treads into murkier legal territory. Stodden describes the options available to researchers interested in publishing data and code and summarizes the recommendations of the Reproducible Research Standard.

Summary

We have divided this book into three parts: Tools, Practices and Guidelines, and Platforms. These mirror the composition of research happening in reproducibility today. Even just over the last two years, tool development for computational science has taken off. An early conference at Applied Mathematics Perspectives called "Reproducible Research: Tools and Strategies for Scientific Computing" in July of 2011 sought to encourage the nascent community of research tool builders (Stodden 2012). Recently, in December of 2012, a workshop entitled "Reproducibility in Computational and Experimental Mathematics" was held as part of the ICERM workshop series (ICERM Workshop 2012). A summary of the tools presented is available on the workshop wiki (Stodden 2012), and the growth of the field is evident. Additional material, including code and data, is available from the editor's website: www.ImplementingRR.org.

Journals are continuing to raise standards to ensure reproducibility in computational science (*Nature* Editorial 2013; Marcia 2014) and funding agencies have recently been instructed by the White House to develop plans for the open dissemination of data arising from federally funded research (Stebbins 2013). We feel that a book documenting available tools, practices, and dissemination platforms could not come at a better time.

References

Bailey, D.H., Borwein, J.M., LeVeque, R.J., Rider, B., Stein, W., and Stodden, V. (2012). Reproducibility in computational and experimental mathematics, in *ICERM Workshop*, December 10–14, 2012. http://icerm.brown.edu/tw12-5-.rcem.

Bailey, D.H., Borwein, J.M., Stodden, V., Set the default to 'Open,' notices of the American Mathematical Society, June/July 2013. http://www.ams.org/notices/201306/rnoti-p679.pdf.

Birney, E., Hudson, T. J., Green, E. D., Gunter, C., Eddy, S., Rogers, J., Harris, J. R. et al. (2009), Prepublication data sharing, *Nature*, 461, 168–170.

Coombes, K., Wang, J., and Baggerly, K. (2007), Microarrays: Retracing steps, *Nat. Med.*, 13, 1276–1277.

Donoho, D.L., Maleki, A., Rahman I.U., Shahram, M., and Stodden V., Reproducible research in computational harmonic analysis, computing in science and engineering, *IEEE Comput. Sci. Eng.*, 11(1), 8–18, Jan/Feb 2009, doi:10.1109/MCSE.2009.15.

McNutt, M. (2014, January 17), Reproducibility, *Science*, 343(6168), 229. http://www.sciencemag.org/content/343/6168/229.summary.

Nature Editorial (2013, April 24), Announcement: Reducing our irreproducibility, *Nature*, 496. http://www.nature.com/news/announcement-reducing-our-irreproducibility-1.12852.

Peng, R. D. (2011), Reproducible research in computational science, *Science*, 334, 1226–1227.

Peng, R. D., Dominici, F., and Zeger, S. L. (2006), Reproducible epidemiologic research, *Am. J. Epidemiol.*, 163, 783–789.

Stebbins, M. (2013), Expanding public access to the results of federally funded research, February 22, 2013. http://www.whitehouse.gov/blog/2013/02/22/expanding-public-access-results-federally-funded-research.

Stodden, V. (2012), Reproducible research: Tools and strategies for scientific computing, *Comput. Sci. Eng.*, 14(4), 11–12 July/August 2012. http://www.computer.org/csdl/mags/cs/2012/04/mcs2012040011-abs.html.

Stodden, V., Borwein, J., and Bailey, D.H. (2013), Setting the default to reproducible in computational science research, *SIAM News*, June 3, 2013. http://www.siam.org/news/news.php?id=2078.

Yale Roundtable (2010), Reproducible research: Addressing the need for data and code sharing in computational science, *IEEE Comput. Sci. Eng.*, 12, 8–13.

Acknowledgment

The editors acknowledge the generous support of Awards R01ES019560 and R21ES020152 from the National Institute of Environmental Health Sciences (Peng), NSF Award 1153384 "EAGER: Policy Design for Reproducibility and Data Sharing in Computational Science" (Stodden), and the Sloan Foundation Award "Facilitating Transparency in Scientific Publishing" (Stodden). The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute of Environmental Health Sciences, the National Institutes of Health, the National Science Foundation, or the Alfred P. Sloan Foundation.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Editors

Victoria Stodden is an assistant professor of statistics at Columbia University and affiliated with the Columbia University Institute for Data Sciences and Engineering, New York City, New York. Her research centers on the multifaceted problem of enabling reproducibility in computational science. This includes studying adequacy and robustness in replicated results, designing and implementing validation systems, developing standards of openness for data and code sharing, and resolving legal and policy barriers to disseminating reproducible research. She is the developer of the award-winning "Reproducible Research Standard," a suite of open licensing recommendations for the dissemination of computational results.

Friedrich Leisch is head of the Institute of Applied Statistics and Computing at the University of Natural Resources and Life Sciences in Vienna. He is a member of the R Core Team, the original creator of the Sweave system in R, and has published extensively about tools for reproducible research. He is also a leading researcher in the area of high-dimensional data analysis.

Roger D. Peng is an associate professor in the Department of Biostatistics at the Johns Hopkins Bloomberg School of Public Health, Baltimore, Maryland. He is a prominent researcher in the areas of air pollution and health risk assessment and statistical methods for environmental health data. Dr. Peng is the associate editor for reproducibility for the journal *Biostatistics* and is the author of numerous R packages.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contributors

Mikio L. Braun
Department of Computer Science
Technical University of Berlin
Berlin, Germany

C. Titus Brown
Department of Computer Science and Engineering
and
Department of Microbiology and Molecular Genetics
Michigan State University
East Lansing, Michigan

Fernando Chirigati
Department of Computer Science and Engineering
Polytechnic Institute of New York University
Brooklyn, New York

Andrew P. Davison
Unité de Neurosciences, Information & Complexité
Centre National de la Recherche Scientifique
Gif sur Yvette, France

Scott Edmunds
Beijing Genomics Institute
Beijing, People's Republic of China

Juliana Freire
Department of Computer Science and Engineering
Polytechnic Institute of New York University
Brooklyn, New York

Philip J. Guo
University of Rochester
Rochester, New York

Marcus D. Hanwell
Kitware, Inc.
Clifton Park, New York

Holger Hoefling
Novartis, Pharma
Basel, Switzerland

Bill Howe
Scalable Data Analytics
University of Calabria
Rende, Italy

and

eScience Institute
and
Department of Computer Science and Engineering
University of Washington
Seattle, Washington

Iain Hrynaszkiewicz
Outreach Director
Faculty of 1000
London, United Kingdom

Christophe Hurlin
Department of Economics
University of Orléans
Orléans, France

Luis Ibanez
Kitware, Inc.
Clifton Park, New York

David Koop
Department of Computer Science and Engineering
Polytechnic Institute of New York University
Brooklyn, New York

Peter Li
Giga Science
Beijing Genomics Institute
Beijing, People's Republic of China

Michele Mattioni
European Molecular Biology Laboratory
European Bioinformatics Institute
Hinxton, United Kingdom

K. Jarrod Millman
Division of Biostatistics
School of Public Health
University of California, Berkeley
Berkeley, California

Dave Murray-Rust
Department of Informatics
University of Edinburgh
Edinburgh, Scotland

Peter Murray-Rust
Department of Chemistry
University of Cambridge
Cambridge, United Kingdom

Cheng Soon Ong
Bioinformatics Group
National ICT Australia
University of Melbourne
Melbourne, Victoria, Australia

Open Science Collaboration
Charlottesville, Virginia

Fernando Pérez
Henry H. Wheeler Jr. Brain Imaging Center
Helen Wills Neuroscience Institute
University of California, Berkeley
Berkeley, California

Christophe Pérignon
Finance Department
Hautes études commerciales de Paris
Paris, France

Likit Preeyanon
Department of Microbiology and Molecular Genetics
Michigan State University
East Lansing, Michigan

Alexis Black Pyrkosz
Avian Disease and Oncology Laboratory
East Lansing, Michigan

Anthony Rossini
Novartis, Pharma
Basel, Switzerland

Dmitry Samarkanov
Ecole Centrale de Lille
Lille University of Science and Technology
Villeneuve-d'Ascq, France

William J. Schroeder
Kitware, Inc.
Clifton Park, New York

Cláudio T. Silva
Polytechnic Institute of New York University
Brooklyn, New York

Victoria Stodden
Department of Statistics
Columbia University
New York City, New York

Bartosz Teleńczuk
Unité de Neurosciences, Information & Complexité
Centre National de la Recherche Scientifique
Gif sur Yvette, France

and

Institute for Theoretical Biology
Humboldt University
Berlin, Germany

Yihui Xie
Department of Statistics
Iowa State University
Ames, Iowa

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Part I

Tools

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

knitr: A Comprehensive Tool for Reproducible Research in R

Yihui Xie

CONTENTS

Reproducibility is the ultimate standard by which scientific findings are judged. From the computer science perspective, reproducible research is often related to literate programming [13], a paradigm conceived by Donald Knuth, and the basic idea is to combine computer code and software

documentation in the same document; the code and documentation can be identified by different special markers. We can either compile the code and mix the results with documentation or extract the source code from the document. To some extent, this implies reproducibility because everything is generated automatically from computer code, and the code can reflect all the details about computing.

Early implementations like WEB [12] and Noweb [20] were not directly suitable for data analysis and report generation, which was partly overcome by later tools like Sweave [14]. There are still a number of challenges that were not solved by existing tools; for example, Sweave is closely tied to LaTeX and hard to extend. The **knitr** package [28,29] was built upon the ideas of previous tools with a framework redesigned, enabling easy and fine control of many aspects of a report. Sweave can be regarded as a subset of **knitr** in terms of the features.

In this chapter, we begin with a simple but striking example that shows how reproducible research can become natural practice to authors given a simple and appealing tool. We introduce the design of the package in Section 1.2 and how it works with a variety of document formats, including LaTeX, HTML, and Markdown. Section 1.3 lists the features that can be useful to data analysis such as the cache system and graphics support. Section 1.4 covers advanced features that extend **knitr** to a comprehensive environment for data analysis; for example, other languages such as Python, awk, and shell scripts can also be integrated into the **knitr** framework. We will conclude with a few significant examples, including student homework, data reports, blog posts, and websites built with **knitr**.

The main design philosophy of **knitr** is to make reproducible research easier and more enjoyable than the common practice of cut-and-paste results. This package was written in the R language [11,19]. It is freely available on CRAN (Comprehensive R Archive Network) and documented in its website http://yihui.name/knitr/; the development repository is on Github: https://github.com/yihui/knitr, where users can file bug reports and feature requests and participate in the development.

There are obvious advantages of writing a literate programming document over copying and pasting results across software packages and documents. An overview of literate programming applied to statistical analysis can be found in [22]; [8] introduced general concepts of literate programming documents for statistical analysis, with a discussion of the software architecture; [7] is a practical example based on [8], using an R package **GolubRR** to distribute reproducible analysis; and [2] 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 enough details for reproducibility (even with datasets supplied). Instead of separating results from computing, we can actually put everything in one document (called a *compendium* in [8]), including the computer code and narratives. When we compile this document, the computer code will be executed, giving us the results directly.

This is the central idea of this chapter—we go from the source code to the report in one step, and everything is automated by the source code.

1.1 Web Application

R Markdown (referred to as *Rmd* hereafter) is one of the document formats that **knitr** supports, and it is also the simplest one. Markdown [10] is a both easy-to-read and easy-to-write language that was designed primarily for writing web content easily and can be translated to HTML (e.g., `**text**` translates to `<strong>text</strong>`). What follows is a trivial example of how Rmd looks like:

````
# First section

Description of the methods.

```{r brownian-motion, fig.height=4, fig.cap='Brownian Motion'}
x <- cumsum(rnorm(100))
plot(x)
```

The mean of x is `r mean(x)`.
````

We can compile this document with **knitr**, and the output will be an HTML web page containing all the results from R, including numeric and graphical results. This is not only easier for authors to write a report but also guarantees a report is reproducible since no cut-and-paste operations are involved. To compile the report, we only need to load the **knitr** package in R and call the `knit()` function:

```
library(knitr)
knit("myfile.Rmd")  # suppose we saved the above file as
  myfile.Rmd
```

Based on this simple idea, **knitr** users have contributed hundreds of reports to the hosting website RPubs (http://rpubs.com) within a few months since it was launched, ranging from student homework, data analysis reports, HTML5 slides, and class quizzes. Traditionally, literate programming tools often choose LaTeX as the authoring environment, which has a steep learning curve for beginners. The success of R Markdown and RPubs

shows that one does not have to be a typesetting expert in order to make use of literate programming and write reproducible reports.

1.2 Design

The package design consists of three components: parser, evaluator, and renderer. The parser identifies and extracts computer code from the source document; the evaluator executes the code; and the renderer generates the final output by appropriately marking up the results according to the output format.

1.2.1 Parser

To include computer code into a document, we have to use special patterns to separate it from normal texts. For instance, the Rmd example in Section 1.1 has an R code chunk that starts with `` ```{r} `` and ends with `` ``` ``.

Internally, **knitr** uses the object `knit_patterns` to set or get the pattern rules, which are essentially regular expressions. Different document formats use different sets of regular expressions by default, and all built-in patterns are stored in the object `all_patterns` as a named list. For example, `all_patterns$rnw` is a set of patterns for the Rnw format, which has an R code embedded in a LaTeX document using the Noweb syntax. Similarly, **knitr** has default syntax patterns for other formats like Markdown (`md`), HTML (`html`), and reStructuredText (`rst`). We take the Rnw syntax for example.

```
library(knitr)
names(all_patterns)  # all built-in document formats

## [1] "rnw"  "brew" "tex"  "html" "md"   "rst"

all_patterns$rnw[c("chunk.begin", "chunk.end", "inline.code")]

## $chunk.begin
## [1] "^\\s*<<(.*)>>="
##
## $chunk.end
## [1] "^\\s*@\\s*(%+.*|)$"
##
## $inline.code
## [1] "\\\\Sexpr\\{([^}]+)\\}"
```

In the pattern list for the Rnw format, there are three major elements as shown earlier: `chunk.begin`, `chunk.end`, and `inline.code`, which are regular expressions indicating the patterns for the beginning and ending of a code chunk, and inline code, respectively. For example, the regular expression `^\s*<<(.*)>>=` means the pattern for the beginning of a code chunk is: in the beginning (`^`) of this line, there are at most some white spaces (`\s*`), then the chunk header starts with `<<`; inside the chunk header, there can be some texts denoting chunk options (`(.*)`), which can be regarded as metadata for a chunk (e.g., `fig.height=4` means the figure height will be 4 in. for this chunk); the chunk header is closed by `>>=`. The code chunk is usually closed by `@` (white spaces are allowed before it and TEX comments are allowed after it), and we can also write inline code inside the pseudo TEX command `\Sexpr{}`. What follows is an example of a fragment of an Rnw document:

```
\section{First section}

Description of the methods.

<<brownian-motion, fig.height=4, fig.cap='Brownian  Motion'>>=
x <- cumsum(rnorm(100))
plot(x)
@

The mean of x is \Sexpr{mean(x)}.
```

Based on the Rnw syntax, **knitr** will find out the code chunk, as well as the inline code `mean(x)`. Anything else in the document will remain untouched and will be mixed with the results from the computer code eventually. To show the parser can be easily generalized, we take a look at the Rmd syntax as well:

```
str(all_patterns$md[c("chunk.begin", "chunk.end",
  "inline.code")])

## List of 3
##  $ chunk.begin: chr "^\\s*`{3,}\\s*\\{r(.*)\\}\\s*$"
##  $ chunk.end  : chr "^\\s*`{3,}\\s*$"
##  $ inline.code: chr "`r +([^`\n]+)\\s*`"
```

Roughly speaking, the three major patterns are changed to `` ```{r *} `` (beginning), ``` ``` ``` (ending), and `` `r *` `` (inline), respectively. If we want to specify our own syntax, we can use the `knit_patterns$set()` function, which will override the default syntax, for example:

```
knit_patterns$set(chunk.begin = "^<<r(.*)", chunk.end =
  "^r>>$", inline.code = "\\{\\{([^}]+)\\}\\}")
```

Then, we will be able to parse a document like this with the custom syntax:

```
<<r brownian-motion, fig.height=4, fig.cap='Brownian  Motion'
x <- cumsum(rnorm(100))
plot(x)
r>>

The mean of x is {{mean(x)}}.
```

In practice, however, this kind of customization is often unnecessary. It is better to follow the default syntax, otherwise additional instructions will be required in order to compile a literate programming document. Table 1.1 shows all the document formats that are currently supported by **knitr**.

Among all chunk options, there is a special option called the chunk label. It is the only chunk option that does not have to be of the form `option = value`. The chunk label is supposed to be a unique identifier of a code chunk, which will be used as the filename for figure files, cache files, and also ids for chunk references. We will mention these later in Section 1.3.

1.2.2 Evaluator

Once we have the code chunks and inline code expressions extracted from the document, we need to evaluate them. The **evaluate** package [26] is used to execute code chunks, and the `eval()` function in base R is used to execute the inline R code. The latter is easy to understand and is made possible by

TABLE 1.1

Code Syntax for Different Document Formats

Format	Start	End	Inline	Output
Rnw	`<<*>>=`	`@`	`\Sexprx`	TEX
Rmd	```` ```{r *} ````	```` ``` ````	`` `r x` ``	Markdown
Rhtml	`<!-begin.rcode *`	`end.rcode->`	`<!-rinline x->`	HTML
Rrst	`.. {r *}`	`.. ..`	`:r:`x``	reST
Rtex	`% begin.rcode *`	`% end.rcode`	`\rinlinex`	TEX
brew			`<% x %>`	text

* Denotes local chunk options, for example, `<<label, eval=FALSE>>=`; `x` denotes inline R code, for example, `<% 1+2 %>`.

the power of "computing on the language" [18] of R. Suppose we have a code fragment `1+1` as a character string, we can parse and evaluate it as an R code:

```
eval(parse(text = "1+1"))
```

```
## [1] 2
```

For code chunks, it is more complicated. The **evaluate** package takes a piece of R source code, evaluates it, and returns a list containing the results of six possible classes: `character` (normal text output), `source` (source code), `warning`, `message`, `error`, and `recordedplot` (plots).

```
library(evaluate)
res <- evaluate(c("'hello world!'", "1:2+1:3"))
str(res, nchar.max = 37)
```

```
## List of 5
##  $ :List of 1
##   ..$ src: chr "'hello world!'\n"
##   ..- attr(*, "class")= chr "source"
##  $ : chr "[1] \"hello world!\"\n"
##  $ :List of 1
##   ..$ src: chr "1:2+1:3"
##   ..- attr(*, "class")= chr "source"
##  $ :List of 2
##   ..$ message: chr "longer object length is not a
     multip"| __truncated__
##   ..$ call   : language 1:2 + 1:3
##   ..- attr(*, "class")= chr [1:3] "simpleWarning"
     "warning" "condition"
##  $ : chr "[1] 2 4 4\n"
```

An internal S3 generic function *wrap()* in **knitr** is used to deal with different types of output using output hooks defined in the object `knit_hooks`, which constitutes the renderer. Before the final output is rendered, we may have to postprocess the output from **evaluate** according to the chunk options. For example, if the chunk option is `echo=FALSE`, we need to remove the source code. This is one advantage of using the **evaluate** package because we can easily filter out the result elements that we do not want according to the classes of the elements. Continuing the aforementioned example, we can remove the source code by

```
## filter out elements which are not source
res <- Filter(Negate(is.source), res)
str(res, nchar.max = 37)

## List of 3
##  $ : chr "[1] \"hello world!\"\n"
##  $ :List of 2
##   ..$ message: chr "longer object length is not a
     multip"| __truncated__
##   ..$ call   : language 1:2 + 1:3
##   ..- attr(*, "class")= chr [1:3] "simpleWarning" "warning"
     "condition"
##  $ : chr "[1] 2 4 4\n"
```

Similarly, we can process other elements according to the chunk options; for instance, `warning=FALSE` means to remove warning messages, and `results='hide'` means to remove elements of the class `character`; **knitr** has a large number of chunk options to tweak the output, which are documented at http://yihui.name/knitr/options.

One notable feature of the **evaluate** package that may be surprising to most R users is that it does not stop on errors by default. This is to mimic the behavior of R when we copy and paste R code in the console (or terminal): If an error occurs in a previous R expression, the rest of the code will still be pasted and executed. To completely stop on errors, we need to set a chunk option in **knitr**:

```
opts_chunk$set(error = FALSE)
```

1.2.3 Renderer

Unlike other implementations such as Sweave, **knitr** makes almost everything accessible to the users, including every piece of results returned from **evaluate**. The users are free to write these results in any formats they like via output hook functions. Consider the following simple example:

```
1 + 1

## [1] 2
```

There are two parts in the returned results: the source code `1 + 1` and the output `[1] 2`. Users may define a hook function for the source code like this to use the `lstlisting` environment in LaTeX:

```
knit_hooks$set(source = function(x, options) {
    paste("\\begin{lstlisting}\n", x, "\\end{lstlisting}\n",
        sep = "")
})
```

Or put it inside the <pre> tag with a CSS class source in HTML:

```
knit_hooks$set(source = function(x, options) {
    paste("<pre class='source'>", x, "</pre>", sep = "")
})
```

Here, the name of the hook function corresponds to the class of the element returned from **evaluate**; see Table 1.2 for the mapping between the two sets of names. The argument `x` of the hook denotes the corresponding output (a character string), and `options` is a list of chunk options for the current code chunk, for example, `options$fig.width` is a numeric value that determines the width of figures in the current chunk. Note that there are two additional output hooks called `chunk` and `document`. The `chunk` hook takes the output of the whole chunk as input, which has been processed by the previous six output hooks; the `document` hook takes the output of the whole document as input and allows further postprocessing of the output text.

Like the parser, **knitr** also has a series of default output hooks for different document formats, so users do not have to rewrite the renderer in most cases.

TABLE 1.2

Output Hook Functions and the Object Classes of Results from the **evaluate** Package

Class	Output Hook	Arguments
source	source	x, options
character	output	x, options
recordedplot	plot	x, options
message	message	x, options
warning	warning	x, options
error	error	x, options
	chunk	x, options
	document	x

1.3 Features

The **knitr** package borrowed features such as TikZ graphics [25] and cache from **pgfSweave** [3] and **cacheSweave** [16], respectively, but the implementations are completely different. New features like code reference from an external R script, as well as output customization, are also introduced. The feature of hook functions in Sweave was reimplemented and hooks have extended power now. Special emphasis was put on graphics: there can be any number of plots per chunk, there are more than 20 graphical devices to choose from (PDF, PNG, and Cairo devices), and it is also easy to specify the size and alignment of plots via chunk options.

There are several other small features that were motivated from the experience of using Sweave. For example, a progress bar is provided when knitting a file so we more or less know how long we still need to wait; output from inline R code (e.g., `\Sexpr{x[1]}`) is automatically formatted in scientific notation (like 1.2346×10^8) if the result is numeric (this applies to all document formats), and we will not get too many digits by default (the default number in R is 7, which is too long).

As we emphasize the ease of use, the concept of an "R Notebook" was also introduced in this package, which enables one to write a pure R script to create a report, and **knitr** will take care of the details of formatting and compilation.

1.3.1 Code Decoration

Syntax highlighting comes by default in **knitr** (chunk option `highlight=TRUE`) since we believe it enhances the readability of the source code. The **formatR** [27] is used to reformat R code (option `tidy=TRUE`), for example, add spaces and indentation, break long lines into shorter ones, and automatically replace the assignment operator = to <-; see the manual of **formatR** for details.

For LaTeX output, the **framed** package is used to decorate code chunks with a light gray background (as we can see in this document). If this LaTeX package is not found in the system, a version will be copied directly from **knitr**. The output for HTML documents is styled with CSS, which looks similar to LaTeX (with gray shadings and syntax highlighting).

The prompt characters are removed by default because they mangle the R source code in the output and make it difficult to copy the R code. The R output is masked in comments by default based on the same rationale. In fact, this was largely motivated from my experience of grading homework; with the default prompts, it is difficult to verify the results in the homework because it is so inconvenient to copy the source code. Anyway, it is easy to revert to the output with prompts (set option `prompt=TRUE`), and we will

quickly realize the inconvenience to the readers if they want to run the code in the output document:

```
> x <- rnorm(5)
> x
[1] -0.56048 -0.23018  1.55871  0.07051  0.12929
> var(x)
[1] 0.6578
```

The example below shows the effect of `tidy=TRUE/FALSE`:

```
## option tidy=FALSE
for(k in 1:10){j=cos(sin(k)*k^2)+3;print(j-5)}
```

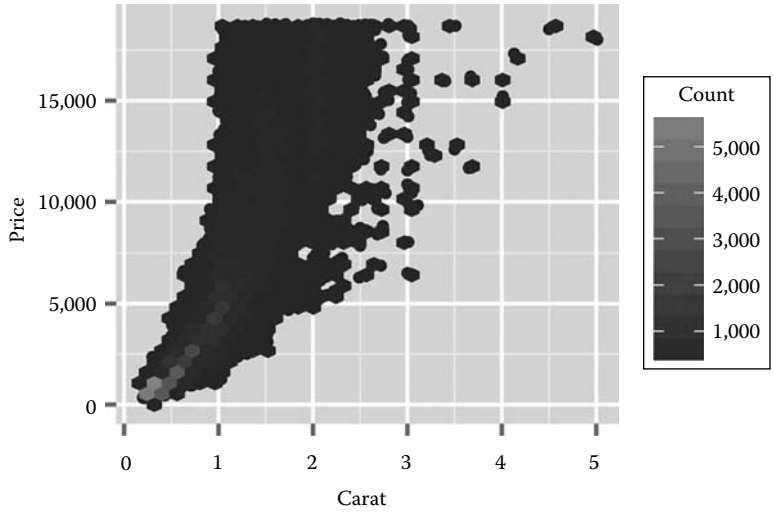
```
## option tidy=TRUE
for (k in 1:10) {
    j <- cos(sin(k) * k^2) + 3
    print(j - 5)
}
```

While this may seem to be irrelevant to reproducible research, we would argue that it is of great importance to design styles that look appealing and helpful at the first glance, which can encourage users to write reports in this way.

1.3.2 Graphics

Graphics is an important part of reports, and several enhancements have been made in **knitr**. For example, **grid** graphics [15] may not need to be explicitly printed as long as the same code can produce plots in the R console (in some cases, however, they have to be printed, e.g., in a loop, because we have to do so in an R console); what follows is a chunk of code that will produce a plot in both the R console and the **knitr**:

```
library(ggplot2)
p <- qplot(carat, price, data = diamonds) + geom_hex()
p  # no need to print(p)
```



1.3.2.1 Graphical Devices

Over a long time, a frequently requested feature for Sweave was the support for other graphics devices, which has been implemented since R 2.13.0. Instead of using several logical options like `png` or `jpeg`, **knitr** uses a single option `dev` (like `grdevice` in Sweave), which has support for more than 20 devices. For instance, `dev='png'` will use the `png()` device in the **grDevices** package in base R, and `dev='CairoJPEG'` uses the `CairoJPEG()` device in the add-on package **Cairo** (it has to be installed first, of course). Here are the possible values for `dev`:

```
##  [1] "bmp"           "postscript"   "pdf"          "png"
##  [5] "svg"           "jpeg"         "pictex"       "tiff"
##  [9] "win.metafile" "cairo_pdf"    "cairo_ps"     "quartz_pdf"
## [13] "quartz_png"   "quartz_jpeg"  "quartz_tiff"  "quartz_gif"
## [17] "quartz_psd"   "quartz_bmp"   "CairoJPEG"    "CairoPNG"
## [21] "CairoPS"      "CairoPDF"     "CairoSVG"     "CairoTIFF"
## [25] "Cairo_pdf"    "Cairo_png"    "Cairo_ps"     "Cairo_svg"
## [29] "tikz"
```

If none of these devices is satisfactory, we can provide the name of a customized device function, which must have been defined in this form before it is used:

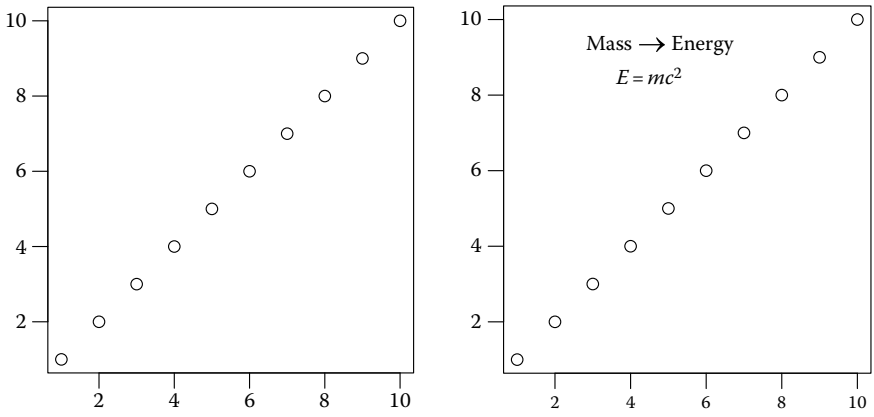
```
custom_dev <- function(file, width, height, ...) {
    # open the device here, e.g. pdf(file, width, height, ...)
}
```

Then, we can set the chunk option `dev='custom_dev'`.

1.3.2.2 Plot Recording

All the plots in a code chunk are first recorded as R objects and then "replayed" inside a graphical device to generate plot files. The **evaluate** package will record plots per *expression* basis; in other words, the source code is split into individual complete expressions and **evaluate** will examine the possible plot changes in snapshots after each single expression has been evaluated. For example, the following code consists of three expressions, out of which two are related to drawing plots, therefore **evaluate** will produce two plots by default:

```
par(mar = c(3, 3, 0.1, 0.1))
plot(1:10, ann = FALSE, las = 1)
text(5, 9, "mass $\\rightarrow$ energy\n$E=mc^2$")
```



This brings a significant difference with traditional tools in R for dynamic report generation since low-level plotting changes can also be recorded. The option `fig.keep` controls which plots to keep in the output; `fig.keep='all'` will keep low-level changes in separate plots; by default (`fig.keep='high'`), **knitr** will merge low-level plot changes into the previous high-level plot, like most graphics devices do. This feature may be useful for teaching R graphics step by step. Note, however, that low-level plotting commands in a single expression (a typical case is a loop) will not be recorded cumulatively, but high-level plotting commands, regardless of where they are, will always be recorded. For example, this chunk will only produce 2 plots instead of 21 plots because there are 2 complete expressions:

```
plot(0, 0, type = "n", ann = FALSE)
for (i in seq(0, 2 * pi, length = 20)) points(cos(i), sin(i))
```

But this will produce 20 plots as expected:

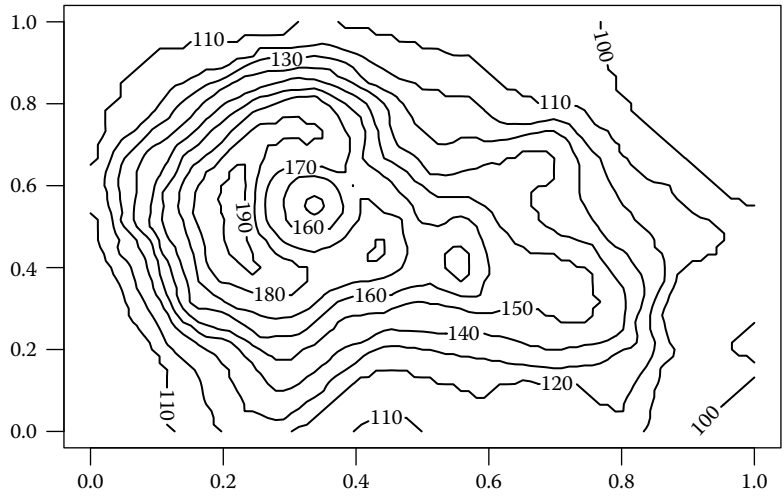
```
for (i in seq(0, 2 * pi, length = 20)) {
    plot(cos(i), sin(i), xlim = c(-1, 1), ylim = c(-1, 1))
}
```

We can discard all previous plots and keep the last one only by `fig.keep='last'`, or keep only the first plot by `fig.keep='first'`, or discard all plots by `fig.keep='none'`.

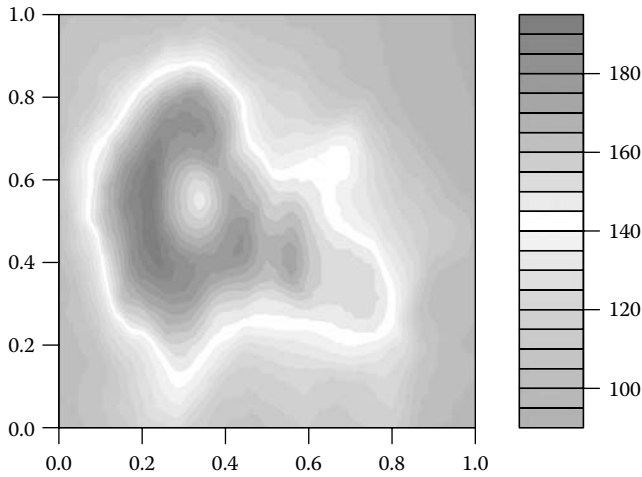
1.3.2.3 Plot Rearrangement

The chunk option `fig.show` can decide whether to hold all plots while evaluating the code and "flush" all of them to the end of a chunk (`fig.show='hold'`; see the previous plot example), or just insert them to the places where they were created (by default `fig.show='asis'`). Here is an example of `fig.show='asis'` for two plots in one chunk:

```
contour(volcano)  # contour lines
```



```
filled.contour(volcano)  # fill contour plot with colors
```



Besides `'hold'` and `'asis'`, the option `fig.show` can take a third value, `'animate'`, which makes it possible to insert animations into the output document. In LaTeX, the package **animate** is used to put together image frames as an animation. For animations to work, there must be more than one plot produced in a chunk. The option `interval` controls the time interval between animation frames; by default it is 1 s. Note that we have to add `\usepackage{animate}` in the LaTeX preamble because **knitr** will not add it automatically. Animations in the PDF output can only be viewed in Adobe Reader. There are animation examples in both the main manual and graphics manual of **knitr**, which can be found on the package website.

We can specify the figure alignment via the chunk option `fig.align` (`'left'`, `'center'`, and `'right'`). The plot example in the previous section used `fig.align='center'` so the two plots were centered.

1.3.2.4 Plot Size

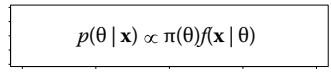
The `fig.width` and `fig.height` options specify the size of plots in the graphics device (units in inches), and the real size in the output document can be different (specified by `out.width` and `out.height`). When there are multiple plots per code chunk, it is possible to arrange multiple plots side by side. For example, in LaTeX, we only need to set `out.width` to be less than half of the current line width, for example, `out.width='.49\\linewidth'`.

1.3.2.5 Tikz Device

Besides PDF, PNG, and other traditional R graphical devices, **knitr** has special support to TikZ graphics via the **tikzDevice** package [24], which is similar to the feature of **pgfSweave**. If we set the chunk option `dev='tikz'`, the `tikz()` device in **tikzDevice** will be used to generate plots. The options `sanitize` (for escaping special TEX characters) and `external` are related to the tikz device: see the documentation of `tikz()` for details. Note that `external=TRUE` in **knitr** has a different meaning with **pgfSweave**—it means `standAlone=TRUE` in `tikz()`, and the TikZ graphics output will be compiled to PDF *immediately* after it is created, so the "externalization" does not depend on the official but complicated externalization commands in the **tikz** package in LATEX. To maintain consistency in (font) styles, **knitr** will read the preamble of the input document and pass it to the tikz device so that the font style in the plots will be the same as the style of the whole LATEX document.

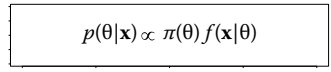
Besides consistency of font styles, the tikz device also enables us to write arbitrary LATEX expressions into R plots. A typical use is to write math expressions. The traditional approach in R is to use an `expression()` object to write math symbols in the plot, and for the tikz device, we only need to write normal LATEX code. What follows is an example of a math expression $p(\theta|\mathbf{x}) \propto \pi(\theta)f(\mathbf{x}|\theta)$ using the two approaches, respectively:

```
plot(0, type = "n", ann = FALSE)
text(0, expression(p(theta ~ "|" ~ bold(x)) %prop% pi(theta)
    * f(bold(x) ~ "|" ~ theta)), cex = 2)
```



With the tikz device, it is both straightforward (if we are familiar with LATEX) and more beautiful:

```
plot(0, type = "n", ann = FALSE)
text(0, "$p(\\theta|\\mathbf{x})\\propto\\pi(\\theta)
    f(\\mathbf{x}|\\theta)$", cex = 2)
```



One disadvantage of the tikz device is that LATEX may not be able to handle too large tikz files (it can run out of memory). For example, an R plot with tens of thousands of graphical elements may fail to compile in LATEX if we use the tikz device. In such cases, we can switch to the PDF or PNG device, or

reconsider our decision on the type of plots, for example, a scatter plot with millions of points is usually difficult to read, and a contour plot or a hexagon plot showing the 2D density can be a better alternative (they are smaller in size).

We emphasized the uniqueness of chunk labels in Section 1.2.1, and here is one reason why it has to be unique: the chunk label is used in the filenames of plots; if there are two chunks that share the same label, the latter chunk will override the plots generated in the previous chunk. The same is true for cache files in the next section.

1.3.3 Cache

The basic idea of cache is that we directly load results from a previous run instead of recompute everything from scratch if nothing has been changed since the last run. This is not a new idea—both **cacheSweave** [16] and **weaver** [6] have implemented it based on Sweave, with the former using **filehash** [17] and the latter using .RData images; **cacheSweave** also supports lazy-loading of objects based on **filehash**. The **knitr** package directly uses internal base R functions to save (`tools:::makeLazyLoadDB()`) and lazy-load objects (`lazyLoad()`). The **cacheSweave** vignette has clearly explained lazy-loading; roughly speaking, lazy-loading means an object will not be really loaded into memory unless it is really used somewhere. This is very useful for cache; sometimes, we read a large object and cache it, then take a subset for analysis and this subset is also cached; in the future, the initial large object will not be loaded into R if our computation is only based on the subset object.

The paths of cache files are determined by the chunk option `cache.path`; by default all cache files are created under a directory cache/ relative to the current working directory, and if the option value contains a directory (e.g., `cache.path='cache/abc-'`), cache files will be stored under the directory cache/ (automatically created if it does not exist) with a prefix abc-. The cache is invalidated and purged on any changes to the code chunk, including both the R code and chunk options; this means previous cache files of this chunk are removed (filenames are identified by the chunk label) and a new set of cache files will be written. The change is detected by verifying if the MD5 hash of the code and options has changed, which is calculated from the **digest** package [5].

Two new features that make **knitr** different from other packages are as follows: cache files will never accumulate since old cache files will always be removed, and **knitr** will also try to preserve side effects such as printing and loading add-on packages. However, there are still other types of side effects like setting `par()` or `options()`, which are not cached. Users should be aware of these special cases and make sure to clearly divide the code that is not meant to be cached into other chunks that are not cached, for example,

set all global options in the first chunk of a document and do not cache that chunk.

Sometimes, a cached chunk may need to use objects from other cached chunks, which can bring a serious problem—if objects in previous chunks have changed, this chunk will not be aware of the changes and will still use old cached results, unless there is a way to detect such changes from other chunks. There is an option called `dependson` in **cacheSweave**, which does this job. In **knitr**, we can also explicitly specify which other chunks this chunk depends on by setting an option like `dependson=c('chunkA', 'chunkB')` (a character vector of chunk labels). Each time the cache of a chunk is rebuilt, all other chunks that depend on this chunk will lose cache, hence their cache will be rebuilt as well.

There are two alternative approaches to specify chunk dependencies: `dep_auto()` and `dep_prev()`. For the former, we need to turn on the chunk option `autodep` (i.e., set `autodep=TRUE`), then put `dep_auto()` in the first chunk in a document. This is an experimental feature borrowed from **weaver** that frees us from setting chunk dependencies manually. The basic idea is, if a latter chunk uses any objects created from a previous chunk, the latter chunk is said to depend on the previous one. The function `findGlobals()` in the **codetools** package is used to find out all global objects in a chunk, and according to its documentation, the result is an approximation. Global objects roughly mean the ones that are not created locally, for example, in the expression `function() {y <- x}`, `x` should be a global object, whereas `y` is local. Meanwhile, we also need to save the list of objects created in each cached chunk so that we can compare them to the global objects in latter chunks. For example, if chunk A created an object `x` and chunk B uses this object, chunk B must depend on A, that is, whenever A changes, B must also be updated. When `autodep=TRUE`, **knitr** will write out the names of objects created in a cached chunk as well as those global objects in two files named __objects and __globals, respectively; later we can use the function `dep_auto()` to analyze the object names to figure out the dependencies automatically. For `dep_prev()`, it is a very conservative approach that sets the dependencies so that a cached chunk will depend on *all* of its previous chunks, that is, whenever a previous chunk is updated, all later chunks will be updated accordingly; similarly, this function needs to be called in the first code chunk in a document.

1.3.4 Code Externalization

It can be more convenient to write R code in a separate file rather than mixing it into a literate programming document; for example, we can run R code successively in a pure R script from one chunk to the other without jumping through other text chunks. This may not sound important for some editors that support interaction with R, such as RStudio (http://www.rstudio.com/ide) or Emacs with ESS [21], since we can send

R code chunks directly from the editor to R, but for other editors like LyX (http://www.lyx.org), we can only compile the whole report as a batch job, which can be inconvenient when we only want to know the results of a single chunk.

The second reason for the feature of code externalization is to be able to reuse code across different documents. Currently the setting is like this: the external R script also has chunk labels for the code in it (marked in the form `## @knitr chunk-label` by default); if the code chunk in the input document is empty, **knitr** will match its label with the label in the R script to input external R code. For example, suppose this is a code chunk labeled as `Q1` in an R script named mycode.R, which is under the same directory as the source document:

```
## @knitr Q1
#' find the greatest common divisor of m and n
gcd <- function(m, n) {
    while ((r <- m%%n) != 0) {
        m <- n
        n <- r
    }
    n
}
```

In the source document, we can first read the script using the function `read_chunk()`, which is available in **knitr**:

```
read_chunk("mycode.R")
```

This is usually done in an early chunk, and we can use the chunk `Q1` later in the source document (e.g., an Rnw document):

```
<<Q1, echo=TRUE, tidy=TRUE>>=
@
```

Different documents can read the same R script, so the R code can be reusable across different input documents. In a large project, however, this may not be an ideal approach to organizing code since there are too many code fragments. We may consider an R package to organize functions, which can be easier to call and test.

1.3.5 Chunk Reference

Code externalization is one way to reuse code chunks across documents, and for a single document, all its code chunks are also reusable in this document.

We can either reuse a whole chunk or embed one chunk into the other one. The former is done through the chunk option `ref.label`, for example.

```
<<chunkA>>=
x <- rnorm(100)
@
Now we reuse chunkA in another chunk:

<<chunkB, ref.label="chunkA">>=
@
```

Then, all the code in chunkA will be put into chunkB. Note only the code is reused; in this example, chunkB will generate a new batch of random numbers, regardless of the value of `x` in chunkA.

To embed a code chunk as a part of another chunk, we can use the syntax `<<label>>`, for example

```
<<chunkA>>=
x <- rnorm(100)
@
Now we embed chunkA into chunkB:

<<chunkB>>=
<<chunkA>>
mean(x)
@
```

The location of the chunks does not matter. We can even define a code chunk later, but reference it in an earlier chunk. We can also recursively embed chunks, and there is no limit on the levels of recursion. For example, we can embed A in B, and B in C, then C will reuse the code in A as well.

1.3.6 Evaluation of Chunk Options

By default **knitr** treats chunk options like function arguments instead of a text string to be split by commas to obtain option values. This gives the user much more power than the traditional syntax in Sweave; we can pass arbitrary R objects to chunk options besides simple ones like `TRUE`/`FALSE`, numbers, and character strings. The page http://yihui.name/knitr/demo/sweave/ has given two examples to show the advantages of the new syntax. Here, we show yet another useful application: conditional evaluation.

The idea is, instead of setting chunk options `eval` to be `TRUE` or `FALSE` (logical constants), their values can be controlled by a variable in the current R session. This enables **knitr** to conditionally evaluate code chunks

according to variables. For example, here we assign `TRUE` to a variable `dothis`:

```
dothis <- TRUE
```

In the next chunk, we set chunk options `eval=dothis` and `echo=!dothis`, both are valid R expressions since the variable `dothis` exists. As we can see, the source code is hidden, but it was indeed evaluated since we can see the output:

```
## [1] "you cannot see my source because !dothis is FALSE"
```

Then, we set `eval=dothis` and `echo=dothis` for another chunk:

```
if (dothis) print("you can see everything now because dothis
    is TRUE")
```

```
## [1] "you can see everything now because dothis is TRUE"
```

If we change the value of `dothis` to `FALSE`, neither of the aforementioned chunks will be evaluated any more. Therefore, we can control many chunks with a single variable and present results *selectively*. When chunk options are parsed and evaluated like function arguments, a literate programming document becomes really programmable.

1.3.7 Child Document

We do not have to put everything in one single document; instead, we can write smaller child documents and include them into a main document. This can be done through the `child` option, for example, `child=c('child1.Rnw', 'child2.Rnw')`. When **knitr** sees the `child` option is not empty, it will parse, evaluate, and render the child documents as usual and include the results back into the main document. Child documents can have a nested structure (one child can have a further child), and there is no limit on the depth of nesting. This feature enables us to better organize large projects, for example, one author can focus on one child document.

1.3.8 R Notebook

We can obtain a report based on a pure R script without taking care of the authoring tools such as LaTeX or HTML. This kind of R scripts is called R notebooks in **knitr**. There are two approaches to compile R notebooks: `stitch()`

and `spin()`. The idea of "stitch" is we fit an R script into a predefined template in **knitr** (choices of templates include LaTeX, HTML, and Markdown) and compile the mixed document to a report; all the code in the script will be put into one single chunk. The idea of "spin" is to write a specially formatted script, with normal texts masked in roxygen comments (i.e., after `#'`) and chunk options after `#+`. Here is an example for `spin()`:

```
#' This is a report.
#'
#+ chunkA, eval=TRUE
# generate data
x <- rnorm(100)
#'
#' The report is done.
```

This script will be parsed and translated to one of the document formats that **knitr** supports (Table 1.1), and then compiled to a report. This can be done through a single click in RStudio or we can also call the functions manually in R:

```
library(knitr)
stitch("mycode.R")  # stitch it, or spin it
spin("mycode.R")
```

1.4 Extensibility

The **knitr** package is highly extensible. We have seen in Section 1.2 that both the syntax patterns and output hooks can be customized. In this section, we introduce two new concepts: chunk hooks and language engines.

1.4.1 Hooks

A chunk hook (not to be confused with the output hooks) is a function to be called when a corresponding chunk option is not `NULL`, and the returned value of the function is written into the output if it is `character`. All chunk hooks are also stored in the object `knit_hooks`.

One common and tedious task when using R base graphics is we often have to call `par()` to set graphical parameters. This can be abstracted into a chunk hook, so that before a code chunk is evaluated, a set of graphical parameters can be automatically set. A chunk hook can be arbitrarily named as long as it does not conflict with existing hooks in `knit_hooks`. For example, we create a hook named `pars`: