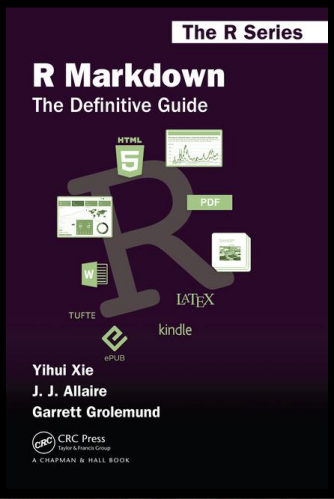
A **CRC Press** FREEBOOK

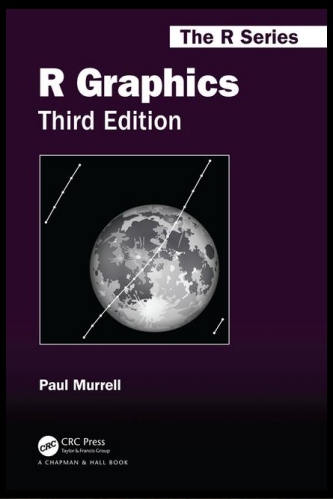
Using R

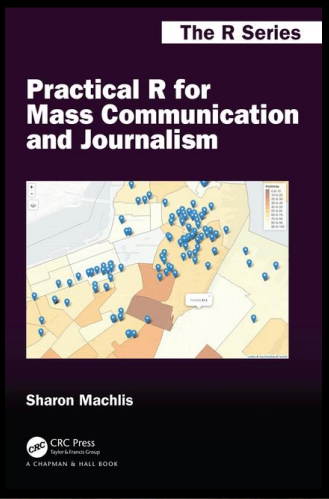
CRC Press
Taylor & Francis Group

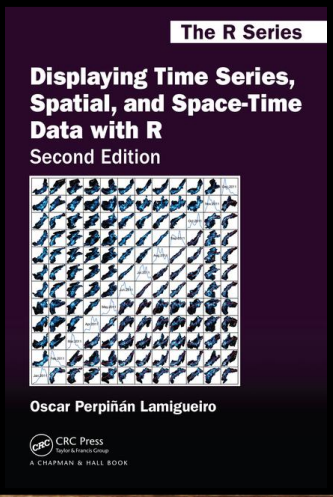
TABLE OF CONTENTS

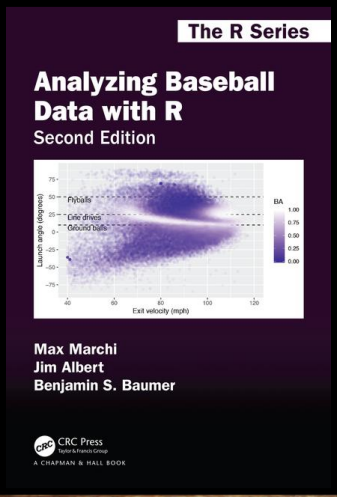
READ THE LATEST ON R PROGRAMMING WITH THESE KEY TITLES

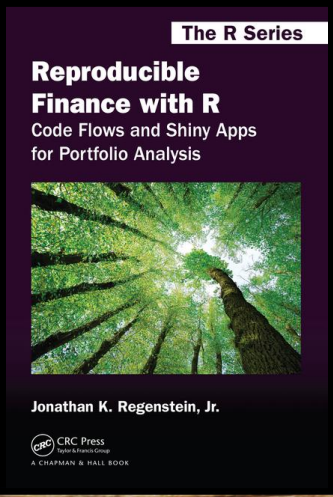












VIEW THE R SERIES HERE

SAVE 20% AND **FREE STANDARD SHIPPING** WITH DISCOUNT CODE **RSER1**

Introduction

R is a programming language and software environment for statistical computing and graphics. It 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 13,000 packages available. It is difficult for the documentation to keep pace with the expansion of the software, which led to the launch of the Chapman & Hall/CRC R Series, a forum for the publication of books covering many aspects of the development and application of R.

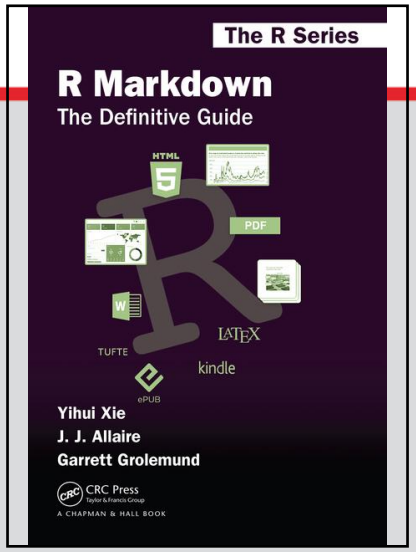
The scope of the series is wide, covering three main threads: applications of R to specific disciplines such as biology, epidemiology, genetics, engineering, finance, and the social sciences; using R for the study of topics of statistical methodology, such as linear and mixed modeling, time series, Bayesian methods, and missing data; and the development of R itself, including programming, building packages, and graphics. The books appeal to programmers and developers of R software, as well as applied statisticians and data analysts in many fields. They feature detailed worked examples and R code fully integrated into the text, ensuring their usefulness to researchers, practitioners and students

This FreeBook presents six chapters on 'Using R' from books recently published in The R Series. Each chapter presents a different aspect of using R, including the basics of RMarkdown, base graphics, and applications in baseball analytics and portfolio risk. We hope you will find the content useful, and that it gives you a snapshot of the quality of the books in the series.

CHAPTER

1

BASICS



This chapter is excerpted from

R Markdown The Definitive Guide

by Yihui Xie, J.J. Allaire, Garrett Grolemund.

Learn more

2

Basics

R Markdown provides an authoring framework for data science. You can use a single R Markdown file to both

- save and execute code, and
- generate high quality reports that can be shared with an audience.

R Markdown was designed for easier reproducibility, since both the computing code and narratives are in the same document, and results are automatically generated from the source code. R Markdown supports dozens of static and dynamic/interactive output formats.

If you prefer a video introduction to R Markdown, we recommend that you check out the website https://rmarkdown.rstudio.com, and watch the videos in the "Get Started" section, which cover the basics of R Markdown.

Below is a minimal R Markdown document, which should be a plain-text file, with the conventional extension `.Rmd`:

```
---
title: "Hello R Markdown"
author: "Awesome Me"
date: "2018-02-14"
output: html_document
---
```

````
This is a paragraph in an R Markdown document.

Below is a code chunk:

```{r}
fit = lm(dist ~ speed, data = cars)
b   = coef(fit)
```
````

````
plot(cars)
abline(fit)
```


The slope of the regression is `r b[1]`.
```
````

You can create such a text file with any editor (including but not limited to RStudio). If you use RStudio, you can create a new Rmd file from the menu `File -> New File -> R Markdown`.

There are three basic components of an R Markdown document: the metadata, text, and code. The metadata is written between the pair of three dashes `---`. The syntax for the metadata is YAML (YAML Ain't Markup Language, https://en.wikipedia.org/wiki/YAML), so sometimes it is also called the YAML metadata or the YAML frontmatter. Before it bites you hard, we want to warn you in advance that indentation matters in YAML, so do not forget to indent the sub-fields of a top field properly. See the Appendix B.2[1] of Xie (2016) for a few simple examples that show the YAML syntax.

The body of a document follows the metadata. The syntax for text (also known as prose or narratives) is Markdown, which is introduced in Section 2.5. There are two types of computer code, which are explained in detail in Section 2.6:

- A code chunk starts with three backticks like ```` ```{r} ```` where `r` indicates the language name,[2] and ends with three backticks. You can write chunk options in the curly braces (e.g., set the figure height to 5 inches: ```` ```{r, fig.height=5} ````).
- An inline R code expression starts with `` `r `` and ends with a backtick `` ` ``.

Figure 2.1 shows the above example in the RStudio IDE. You can click the `Knit` button to compile the document (to an HTML page). Figure 2.2 shows the output in the RStudio Viewer.

Now please take a closer look at the example. Did you notice a problem? The object `b` is the vector of coefficients of length 2 from the linear regression; `b[1]` is actually the intercept, and `b[2]` is the slope! This minimal example shows you why R Markdown is great for reproducible research: it includes

[1] https://bookdown.org/yihui/bookdown/r-markdown.html
[2] It is not limited to the R language; see Section 2.7 for how to use other languages.

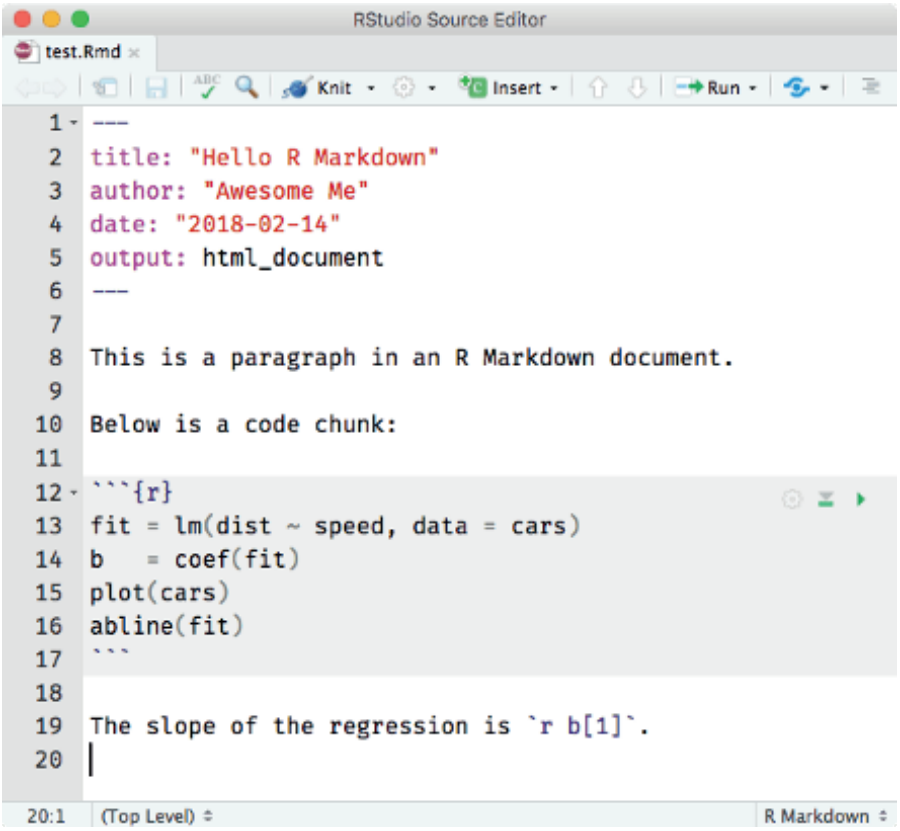


FIGURE 2.1: A minimal R Markdown example in RStudio.

the source code right inside the document, which makes it easy to discover and fix problems, as well as update the output document. All you have to do is change `b[1]` to `b[2]`, and click the `Knit` button again. Had you copied a number `-17.579` computed elsewhere into this document, it would be very difficult to realize the problem. In fact, I had used this example a few times by myself in my presentations before I discovered this problem during one of my talks, but I discovered it anyway.

Although the above is a toy example, it could become a horror story if it happens in scientific research that was not done in a reproducible way (e.g., cut-and-paste). Here are two of my personal favorite videos on this topic:

- "A reproducible workflow" by Ignasi Bartomeus and Francisco Rodríguez-Sánchez (`https://youtu.be/s3JldKoA0zw`). It is a 2-min

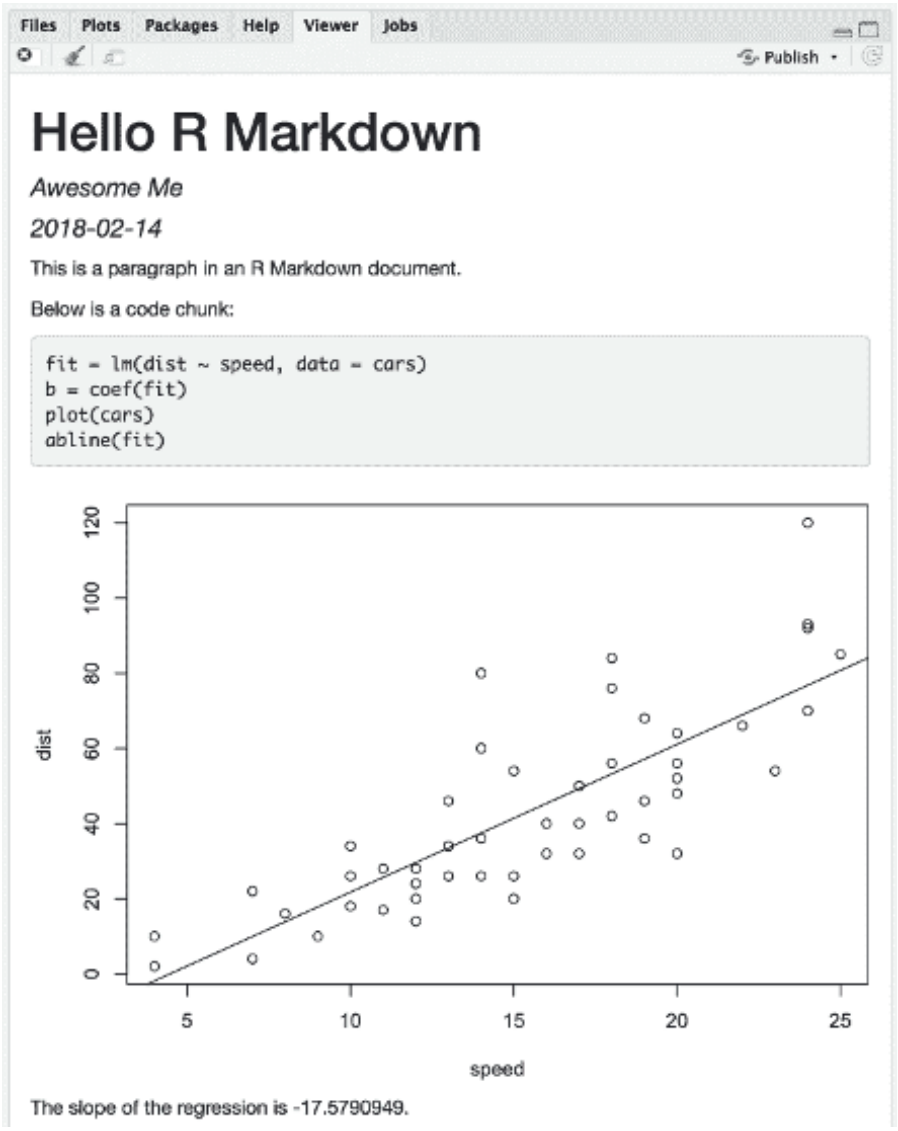


FIGURE 2.2: The output document of the minimal R Markdown example in RStudio.

video that looks artistic but also shows very common and practical problems in data analysis.

- "The Importance of Reproducible Research in High-Throughput Biology" by Keith Baggerly (https://youtu.be/7gYIs7uYbMo). You will be impressed by both the content and the style of this lecture. Keith Baggerly and Kevin Coombes were the two notable heroes in revealing the Duke/Potti scandal[3], which was described as "one of the biggest medical research frauds ever" by the television program "60 Minutes".

It is fine for humans to err (in computing), as long as the source code is readily available.

2.1 Example applications

Now you have learned the very basic concepts of R Markdown. The idea should be simple enough: interweave narratives with code in a document, knit the document to dynamically generate results from the code, and you will get a report. This idea was not invented by R Markdown, but came from an early programming paradigm called "Literate Programming" (Knuth, 1984).

Due to the simplicity of Markdown and the powerful R language for data analysis, R Markdown has been widely used in many areas. Before we dive into the technical details, we want to show some examples to give you an idea of its possible applications.

2.1.1 Airbnb's knowledge repository

Airbnb uses R Markdown to document all their analyses in R, so they can combine code and data visualizations in a single report (Bion et al., 2018). Eventually all reports are carefully peer-reviewed and published to a company knowledge repository, so that anyone in the company can easily find analyses relevant to their team. Data scientists are also able to learn as much

[3] https://en.wikipedia.org/wiki/Anil_Potti

as they want from previous work or reuse the code written by previous authors, because the full R Markdown source is available in the repository.

2.1.2 Homework assignments on RPubs

A huge number of homework assignments have been published to the website `https://RPubs.com` (a free publishing platform provided by RStudio), which shows that R Markdown is easy and convenient enough for students to do their homework assignments (see Figure 2.3). When I was still a student, I did most of my homework assignments using Sweave, which was a much earlier implementation of literate programming based on the S language (later R) and LaTeX. I was aware of the importance of reproducible research but did not enjoy LaTeX, and few of my classmates wanted to use Sweave. Right after I graduated, R Markdown was born, and it has been great to see so many students do their homework in the reproducible manner.

In a 2016 JSM (Joint Statistical Meetings) talk, I proposed that course instructors could sometimes intentionally insert some wrong values in the source data before providing it to the students for them to analyze the data in the homework, then correct these values the next time, and ask them to do the analysis again. This way, students should be able to realize the problems with the traditional cut-and-paste approach for data analysis (i.e., run the analysis separately and copy the results manually), and the advantage of using R Markdown to automatically generate the report.

2.1.3 Personalized mail

One thing you should remember about R Markdown is that you can programmatically generate reports, although most of the time you may be just clicking the `Knit` button in RStudio to generate a single report from a single source document. Being able to program reports is a super power of R Markdown.

Mine Çetinkaya-Rundel once wanted to create personalized handouts for her workshop participants. She used a template R Markdown file, and knitted it in a for-loop to generate 20 PDF files for the 20 participants. Each PDF contained both personalized information and common information. You may read the article `https://rmarkdown.rstudio.com/articles_mail_merge.html` for the technical details.

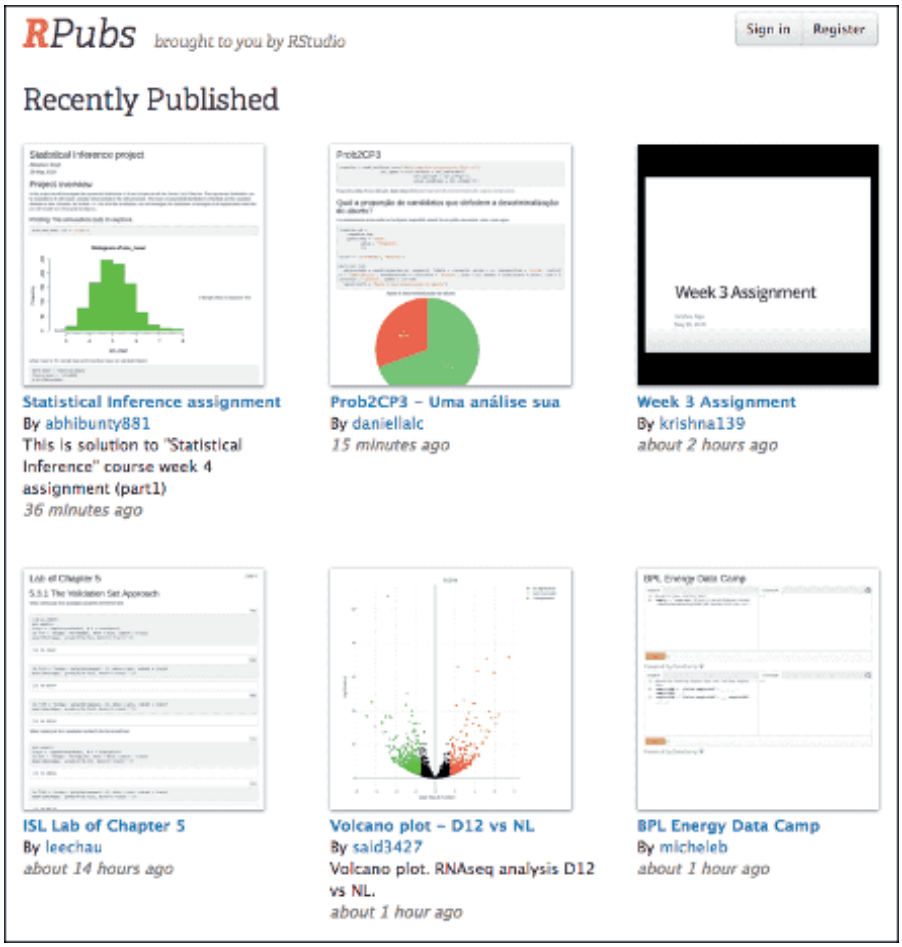


FIGURE 2.3: A screenshot of RPubs.com that contains some homework assignments submitted by students.

2.1.4 2017 Employer Health Benefits Survey

The 2017 Employer Health Benefits Survey[4] was designed and analyzed by the Kaiser Family Foundation, NORC at the University of Chicago, and Health Research & Educational Trust. The full PDF report was written in R Markdown (with the **bookdown** package). It has a unique appearance, which was made possible by heavy customizations in the LaTeX template. This ex-

[4] https://www.kff.org/health-costs/report/2017-employer-health-benefits-survey/

ample shows you that if you really care about typesetting, you are free to apply your knowledge about LaTeX to create highly sophisticated reports from R Markdown.

2.1.5 Journal articles

Chris Hartgerink explained how and why he used R Markdown to write dynamic research documents in the post at https://elifesciences.org/labs/cad57bcf/composing-reproducible-manuscripts-using-r-markdown. He published a paper titled "Too Good to be False: Nonsignificant Results Revisited" with two co-authors (Hartgerink et al., 2017). The manuscript was written in R Markdown, and results were dynamically generated from the code in R Markdown.

When checking the accuracy of P-values in the psychology literature, his colleagues and he found that P-values could be mistyped or miscalculated, which could lead to inaccurate or even wrong conclusions. If the P-values were dynamically generated and inserted instead of being manually copied from statistical programs, the chance for those problems to exist would be much lower.

Lowndes et al. (2017) also shows that using R Markdown (and version control) not only enhances reproducibility, but also produces better scientific research in less time.

2.1.6 Dashboards at eelloo

R Markdown is used at eelloo (https://eelloo.nl) to design and generate research reports. Here is one of their examples (in Dutch): https://eelloo.nl/groepsrapportages-met-infographics/, where you can find gauges, bar charts, pie charts, wordclouds, and other types of graphs dynamically generated and embedded in dashboards.

2.1.7 Books

We will introduce the R Markdown extension **bookdown** in Chapter 12. It is an R package that allows you to write books and long-form reports with multiple Rmd files. After this package was published, a large number of books

have emerged. You can find a subset of them at https://bookdown.org. Some of these books have been printed, and some only have free online versions.

There have also been students who wrote their dissertations/theses with **bookdown**, such as Ed Berry: https://eddjberry.netlify.com/post/writing-your-thesis-with-bookdown/. Chester Ismay has even provided an R package **thesisdown** (https://github.com/ismayc/thesisdown) that can render a thesis in various formats. Several other people have customized this package for their own institutions, such as Zhian N. Kamvar's **beaverdown** (https://github.com/zkamvar/beaverdown) and Ben Marwick's **huskydown** (https://github.com/benmarwick/huskydown).

2.1.8 Websites

The **blogdown** package to be introduced in Chapter 10 can be used to build general-purpose websites (including blogs and personal websites) based on R Markdown. You may find tons of examples at https://github.com/rbind or by searching on Twitter: https://twitter.com/search?q=blogdown. Here are a few impressive websites that I can quickly think of off the top of my head:

- Rob J Hyndman's personal website: https://robjhyndman.com (a very comprehensive academic website).
- Amber Thomas's personal website: https://amber.rbind.io (a rich project portfolio).
- Emi Tanaka's personal website: https://emitanaka.github.io (in particular, check out the beautiful showcase page).
- "Live Free or Dichotomize" by Nick Strayer and Lucy D'Agostino McGowan: http://livefreeordichotomize.com (the layout is elegant, and the posts are useful and practical).

2.2 Compile an R Markdown document

The usual way to compile an R Markdown document is to click the `Knit` button as shown in Figure 2.1, and the corresponding keyboard shortcut is `Ctrl + Shift + K` (`Cmd + Shift + K` on macOS). Under the hood, RStudio calls the function `rmarkdown::render()` to render the document *in a new R session*. Please note the emphasis here, which often confuses R Markdown users. Rendering an Rmd document in a new R session means that *none of the objects in your current R session (e.g., those you created in your R console) are available to that session*.[5] Reproducibility is the main reason that RStudio uses a new R session to render your Rmd documents: in most cases, you may want your documents to continue to work the next time you open R, or in other people's computing environments. See this StackOverflow answer[6] if you want to know more.

If you must render a document in the current R session, you can also call `rmarkdown::render()` by yourself, and pass the path of the Rmd file to this function. The second argument of this function is the output format, which defaults to the first output format you specify in the YAML metadata (if it is missing, the default is `html_document`). When you have multiple output formats in the metadata, and do not want to use the first one, you can specify the one you want in the second argument, e.g., for an Rmd document `foo.Rmd` with the metadata:

```
output:
  html_document:
    toc: true
  pdf_document:
    keep_tex: true
```

You can render it to PDF via:

```
rmarkdown::render('foo.Rmd', 'pdf_document')
```

The function call gives you much more freedom (e.g., you can generate a

[5] This is not strictly true, but mostly true. You may save objects in your current R session to a file, e.g., `.RData`, and load it in a new R session.

[6] https://stackoverflow.com/a/48494678/559676

series of reports in a loop), but you should bear reproducibility in mind when you render documents this way. Of course, you can start a new and clean R session by yourself, and call `rmarkdown::render()` in that session. As long as you do not manually interact with that session (e.g., manually creating variables in the R console), your reports should be reproducible.

Another main way to work with Rmd documents is the R Markdown Notebooks, which will be introduced in Section 3.2. With notebooks, you can run code chunks individually and see results right inside the RStudio editor. This is a convenient way to interact or experiment with code in an Rmd document, because you do not have to compile the whole document. Without using the notebooks, you can still partially execute code chunks, but the execution only occurs in the R console, and the notebook interface presents results of code chunks right beneath the chunks in the editor, which can be a great advantage. Again, for the sake of reproducibility, you will need to compile the whole document eventually in a clean environment.

Lastly, I want to mention an "unofficial" way to compile Rmd documents: the function `xaringan::inf_mr()`, or equivalently, the RStudio addin "Infinite Moon Reader". Obviously, this requires you to install the **xaringan** package (Xie, 2018g), which is available on CRAN. The main advantage of this way is LiveReload: a technology that enables you to live preview the output as soon as you save the source document, and you do not need to hit the `Knit` button. The other advantage is that it compiles the Rmd document *in the current R session*, which may or may not be what you desire. Note that this method only works for Rmd documents that output to HTML, including HTML documents and presentations.

A few R Markdown extension packages, such as **bookdown** and **blogdown**, have their own way of compiling documents, and we will introduce them later.

Note that it is also possible to render a series of reports instead of single one from a single R Markdown source document. You can parameterize an R Markdown document, and generate different reports using different parameters. See Chapter 15 for details.

2.3 Cheat sheets

RStudio has created a large number of cheat sheets, including the one-page R Markdown cheetahs, which are freely available at https://www.rstudio.com/resources/cheatsheets/. There is also a more detailed R Markdown reference guide. Both documents can be used as quick references after you become more familiar with R Markdown.

2.4 Output formats

There are two types of output formats in the **rmarkdown** package: documents, and presentations. All available formats are listed below:

- `beamer_presentation`
- `github_document`
- `html_document`
- `ioslides_presentation`
- `latex_document`
- `md_document`
- `odt_document`
- `pdf_document`
- `powerpoint_presentation`
- `rtf_document`
- `slidy_presentation`
- `word_document`

We will document these output formats in detail in Chapters 3 and 4. There are more output formats provided in other extension packages (starting from Chapter 5). For the output format names in the YAML metadata of an Rmd file, you need to include the package name if a format is from an extension package, e.g.,

```
output: tufte::tufte_html
```

If the format is from the **rmarkdown** package, you do not need the `rmarkdown::` prefix (although it will not hurt).

When there are multiple output formats in a document, there will be a dropdown menu behind the RStudio `Knit` button that lists the output format names (Figure 2.4).

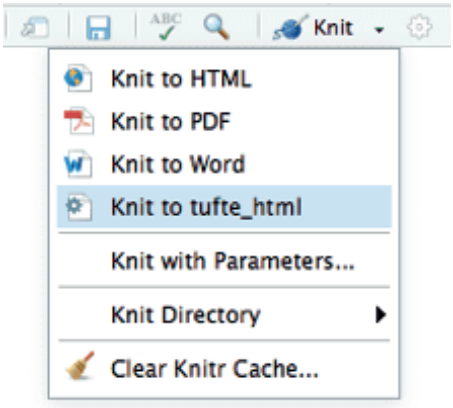


FIGURE 2.4: The output formats listed in the dropdown menu on the RStudio toolbar.

Each output format is often accompanied with several format options. All these options are documented on the R package help pages. For example, you can type `?rmarkdown::html_document` in R to open the help page of the `html_document` format. When you want to use certain options, you have to translate the values from R to YAML, e.g.,

```
html_document(toc = TRUE, toc_depth = 2, dev = 'svg')
```

can be written in YAML as:

```
output:
  html_document:
    toc: true
    toc_depth: 2
    dev: 'svg'
```

The translation is often straightforward. Remember that R's `TRUE`, `FALSE`, and `NULL` are `true`, `false`, and `null`, respectively, in YAML. Character strings in

YAML often do not require the quotes (e.g., `dev: 'svg'` and `dev: svg` are the same), unless they contain special characters, such as the colon `:`. If you are not sure if a string should be quoted or not, test it with the **yaml** package, e.g.,

```
cat(yaml::as.yaml(list(
  title = 'A Wonderful Day',
  subtitle = 'hygge: a quality of coziness'
)))
```

```
title: A Wonderful Day
subtitle: 'hygge: a quality of coziness'
```

Note that the subtitle in the above example is quoted because of the colon.

If a certain option has sub-options (which means the value of this option is a list in R), the sub-options need to be further indented, e.g.,

```
output:
  html_document:
    toc: true
    includes:
      in_header: header.html
      before_body: before.html
```

Some options are passed to **knitr**, such as `dev`, `fig_width`, and `fig_height`. Detailed documentation of these options can be found on the **knitr** documentation page: https://yihui.name/knitr/options/. Note that the actual **knitr** option names can be different. In particular, **knitr** uses `.` in names, but **rmarkdown** uses `_`, e.g., `fig_width` in **rmarkdown** corresponds to `fig.width` in **knitr**. We apologize for the inconsistencies—programmers often strive for consistencies in their own world, yet one standard plus one standard often equals three standards.[7] If I were to design the **knitr** package again, I would definitely use `_`.

Some options are passed to Pandoc, such as `toc`, `toc_depth`, and `number_sections`. You should consult the Pandoc documentation when in

[7] https://xkcd.com/927/

doubt. R Markdown output format functions often have a `pandoc_args` argument, which should be a character vector of extra arguments to be passed to Pandoc. If you find any Pandoc features that are not represented by the output format arguments, you may use this ultimate argument, e.g.,

```
output:
  pdf_document:
    toc: true
    pandoc_args: ["--wrap=none", "--top-level-division=chapter"]
```

2.5 Markdown syntax

The text in an R Markdown document is written with the Markdown syntax. Precisely speaking, it is Pandoc's Markdown. There are many flavors of Markdown invented by different people, and Pandoc's flavor is the most comprehensive one to our knowledge. You can find the full documentation of Pandoc's Markdown at https://pandoc.org/MANUAL.html. We strongly recommend that you read this page at least once to know all the possibilities with Pandoc's Markdown, even if you will not use all of them. This section is adapted from Section 2.1[8] of Xie (2016), and only covers a small subset of Pandoc's Markdown syntax.

2.5.1 Inline formatting

Inline text will be *italic* if surrounded by underscores or asterisks, e.g., `_text_` or `*text*`. **Bold** text is produced using a pair of double asterisks (`**text**`). A pair of tildes (`~`) turn text to a subscript (e.g., `H~3~PO~4~` renders H_3PO_4). A pair of carets (`^`) produce a superscript (e.g., `Cu^2+^` renders Cu^{2+}).

To mark text as `inline code`, use a pair of backticks, e.g., `` `code` ``. To include n literal backticks, use at least $n + 1$ backticks outside, e.g., you can use four backticks to preserve three backtick inside: ```` ```code``` ````, which is rendered as ```` ```code``` ````.

[8] https://bookdown.org/yihui/bookdown/markdown-syntax.html

Hyperlinks are created using the syntax `[text](link)`, e.g., `[RStudio](https://www.rstudio.com)`. The syntax for images is similar: just add an exclamation mark, e.g., `![alt text or image title](path/to/image)`. Footnotes are put inside the square brackets after a caret `^[]`, e.g., `^[This is a footnote.]`.

There are multiple ways to insert citations, and we recommend that you use BibTeX databases, because they work better when the output format is LaTeX/PDF. Section 2.8[9] of Xie (2016) has explained the details. The key idea is that when you have a BibTeX database (a plain-text file with the conventional filename extension `.bib`) that contains entries like:

```
@Manual{R-base,
  title = {R: A Language and Environment for Statistical
    Computing},
  author = {{R Core Team}},
  organization = {R Foundation for Statistical Computing},
  address = {Vienna, Austria},
  year = {2017},
  url = {https://www.R-project.org/},
}
```

You may add a field named `bibliography` to the YAML metadata, and set its value to the path of the BibTeX file. Then in Markdown, you may use `@R-base` (which generates "R Core Team (2018)") or `[@R-base]` (which generates "(R Core Team, 2018)") to reference the BibTeX entry. Pandoc will automatically generated a list of references in the end of the document.

2.5.2 Block-level elements

Section headers can be written after a number of pound signs, e.g.,

```
# First-level header

## Second-level header

### Third-level header
```

[9] https://bookdown.org/yihui/bookdown/citations.html

If you do not want a certain heading to be numbered, you can add `{-}` or `{.unnumbered}` after the heading, e.g.,

```
# Preface {-}
```

Unordered list items start with `*`, `-`, or `+`, and you can nest one list within another list by indenting the sub-list, e.g.,

```
- one item
- one item
- one item
    - one more item
    - one more item
    - one more item
```

The output is:

- one item
- one item
- one item
    - one more item
    - one more item
    - one more item

Ordered list items start with numbers (you can also nest lists within lists), e.g.,

```
1. the first item
2. the second item
3. the third item
    - one unordered item
    - one unordered item
```

The output does not look too much different with the Markdown source:

1. the first item
2. the second item
3. the third item
    - one unordered item
    - one unordered item

Blockquotes are written after >, e.g.,

```
> "I thoroughly disapprove of duels. If a man should challenge me,
  I would take him kindly and forgivingly by the hand and lead him
  to a quiet place and kill him."
>
> --- Mark Twain
```

The actual output (we customized the style for blockquotes in this book):

> "I thoroughly disapprove of duels. If a man should challenge me, I would take him kindly and forgivingly by the hand and lead him to a quiet place and kill him."
>
> — Mark Twain

Plain code blocks can be written after three or more backticks, and you can also indent the blocks by four spaces, e.g.,

````
```
This text is displayed verbatim / preformatted
```


Or indent by four spaces:

    This text is displayed verbatim / preformatted
````

In general, you'd better leave at least one empty line between adjacent but different elements, e.g., a header and a paragraph. This is to avoid ambiguity to the Markdown renderer. For example, does "#" indicate a header below?

```
In R, the character
# indicates a comment.
```

And does "-" mean a bullet point below?

```
The result of 5
- 3 is 2.
```

Different flavors of Markdown may produce different results if there are no blank lines.

2.5.3 Math expressions

Inline LaTeX equations can be written in a pair of dollar signs using the LaTeX syntax, e.g., `$f(k) = {n \choose k} p^{k} (1-p)^{n-k}$` (actual output: $f(k) = \binom{n}{k} p^k (1-p)^{n-k}$); math expressions of the display style can be written in a pair of double dollar signs, e.g., `$$f(k) = {n \choose k} p^{k} (1-p)^{n-k}$$`, and the output looks like this:

$$f\left(k\right) = \binom{n}{k} p^k \left(1-p\right)^{n-k}$$

You can also use math environments inside `$ $` or `$$ $$`, e.g.,

```
$$\begin{array}{ccc}
x_{11} & x_{12} & x_{13}\\
x_{21} & x_{22} & x_{23}
\end{array}$$
```

$$\begin{array}{ccc}
x_{11} & x_{12} & x_{13}\\
x_{21} & x_{22} & x_{23}
\end{array}$$

```
$$X = \begin{bmatrix}1 & x_{1}\\
1 & x_{2}\\
1 & x_{3}
\end{bmatrix}$$
```

$$X = \begin{bmatrix}1 & x_{1}\\
1 & x_{2}\\
1 & x_{3}
\end{bmatrix}$$

```
$$\Theta = \begin{pmatrix}\alpha & \beta\\
\gamma & \delta
\end{pmatrix}$$
```

$$\Theta = \begin{pmatrix}\alpha & \beta\\ \gamma & \delta \end{pmatrix}$$

```
$$\begin{vmatrix}a & b\\
c & d
\end{vmatrix}=ad-bc$$
```

$$\begin{vmatrix}a & b\\ c & d \end{vmatrix}=ad-bc$$

2.6 R code chunks and inline R code

You can insert an R code chunk either using the RStudio toolbar (the `Insert` button) or the keyboard shortcut `Ctrl + Alt + I` (`Cmd + Option + I` on macOS).

There are a lot of things you can do in a code chunk: you can produce text output, tables, or graphics. You have fine control over all these output via chunk options, which can be provided inside the curly braces (between ```` ```{r ```` and `}`). For example, you can choose hide text output via the chunk option `results = 'hide'`, or set the figure height to 4 inches via `fig.height = 4`. Chunk options are separated by commas, e.g.,

```
```{r, chunk-label, results='hide', fig.height=4}
```

The value of a chunk option can be an arbitrary R expression, which makes chunk options extremely flexible. For example, the chunk option `eval` controls whether to evaluate (execute) a code chunk, and you may conditionally evaluate a chunk via a variable defined previously, e.g.,
```

````
```{r}
# execute code if the date is later than a specified day
do_it = Sys.Date() > '2018-02-14'
```

```{r, eval=do_it}
x = rnorm(100)
```
````

There are a large number of chunk options in **knitr** documented at https://yihui.name/knitr/options. We list a subset of them below:

- `eval`: Whether to evaluate a code chunk.
- `echo`: Whether to echo the source code in the output document (someone may not prefer reading your smart source code but only results).
- `results`: When set to `'hide'`, text output will be hidden; when set to `'asis'`, text output is written "as-is", e.g., you can write out raw Markdown text from R code (like `cat('**Markdown** is cool.\n')`). By default, text output will be wrapped in verbatim elements (typically plain code blocks).
- `collapse`: Whether to merge text output and source code into a single code block in the output. This is mostly cosmetic: `collapse = TRUE` makes the output more compact, since the R source code and its text output are displayed in a single output block. The default `collapse = FALSE` means R expressions and their text output are separated into different blocks.
- `warning`, `message`, and `error`: Whether to show warnings, messages, and errors in the output document. Note that if you set `error = FALSE`, `rmarkdown::render()` will halt on error in a code chunk, and the error will be displayed in the R console. Similarly, when `warning = FALSE` or `message = FALSE`, these messages will be shown in the R console.
- `include`: Whether to include anything from a code chunk in the output document. When `include = FALSE`, this whole code chunk is excluded in the output, but note that it will still be evaluated if `eval = TRUE`. When you are trying to set `echo = FALSE`, `results = 'hide'`, `warning = FALSE`, and `message = FALSE`, chances are you simply mean a single option `include = FALSE` instead of suppressing different types of text output individually.

- `cache`: Whether to enable caching. If caching is enabled, the same code chunk will not be evaluated the next time the document is compiled (if the code chunk was not modified), which can save you time. However, I want to honestly remind you of the two hard problems in computer science (via Phil Karlton): naming things, and cache invalidation. Caching can be handy but also tricky sometimes.
- `fig.width` and `fig.height`: The (graphical device) size of R plots in inches. R plots in code chunks are first recorded via a graphical device in **knitr**, and then written out to files. You can also specify the two options together in a single chunk option `fig.dim`, e.g., `fig.dim = c(6, 4)` means `fig.width = 6` and `fig.height = 4`.
- `out.width` and `out.height`: The output size of R plots in the output document. These options may scale images. You can use percentages, e.g., `out.width = '80%'` means 80% of the page width.
- `fig.align`: The alignment of plots. It can be `'left'`, `center`, or `'right'`.
- `dev`: The graphical device to record R plots. Typically it is `'pdf'` for LaTeX output, and `'png'` for HTML output, but you can certainly use other devices, such as `'svg'` or `'jpeg'`.
- `fig.cap`: The figure caption.
- `child`: You can include a child document in the main document. This option takes a path to an external file.

Chunk options in **knitr** can be surprisingly powerful. For example, you can create animations from a series of plots in a code chunk. I will not explain how here because it requires an external software package[10], but encourage you to read the documentation carefully to discover the possibilities. You may also read Xie (2015), which is a comprehensive guide to the **knitr** package, but unfortunately biased towards LaTeX users for historical reasons (which was one of the reasons why I wanted to write this R Markdown book).

There is an optional chunk option that does not take any value, which is the chunk label. It should be the first option in the chunk header. Chunk labels are mainly used in filenames of plots and cache. If the label of a chunk is missing, a default one of the form `unnamed-chunk-i` will be generated, where `i` is incremental. I strongly recommend that you only use alphanumeric characters (`a-z`, `A-Z` and `0-9`) and dashes (`-`) in labels, because they are not spe-

[10] https://blogdown-demo.rbind.io/2018/01/31/gif-animations/

cial characters and will surely work for all output formats. Other characters, spaces and underscores in particular, may cause trouble in certain packages, such as **bookdown**.

If a certain option needs to be frequently set to a value in multiple code chunks, you can consider setting it globally in the first code chunk of your document, e.g.,

````
```{r, setup, include=FALSE}
knitr::opts_chunk$set(fig.width = 8, collapse = TRUE)
```
````

Besides code chunks, you can also insert values of R objects inline in text. For example:

````
```{r}
x = 5  # radius of a circle
```

For a circle with the radius `r x`,
its area is `r pi * x^2`.
````

2.6.1 Figures

By default, figures produced by R code will be placed immediately after the code chunk they were generated from. For example:

````
```{r}
plot(cars, pch = 18)
```
````

You can provide a figure caption using `fig.cap` in the chunk options. If the document output format supports the option `fig_caption: true` (e.g., the output format `rmarkdown::html_document`), the R plots will be placed into figure environments. In the case of PDF output, such figures will be automatically numbered. If you also want to number figures in other formats (such as HTML), please see the **bookdown** package in Chapter 12 (in particular, see Section 12.4.4).

PDF documents are generated through the LaTeX files generated from R Markdown. A highly surprising fact to LaTeX beginners is that figures float by default: even if you generate a plot in a code chunk on the first page, the whole figure environment may float to the next page. This is just how LaTeX works by default. It has a tendency to float figures to the top or bottom of pages. Although it can be annoying and distracting, we recommend that you refrain from playing the "Whac-A-Mole" game in the beginning of your writing, i.e., desparately trying to position figures "correctly" while they seem to be always dodging you. You may wish to fine-tune the positions once the content is complete using the `fig.pos` chunk option (e.g., `fig.pos = 'h'`). See https://www.sharelatex.com/learn/Positioning_images_and_tables for possible values of `fig.pos` and more general tips about this behavior in LaTeX. In short, this can be a difficult problem for PDF output.

To place multiple figures side-by-side from the same code chunk, you can use the `fig.hold='hold'` option along with the `out.width` option. Figure 2.5 shows an example with two plots, each with a width of `50%`.

```
par(mar = c(4, 4, 0.2, 0.1))
plot(cars, pch = 19)
plot(pressure, pch = 17)
```

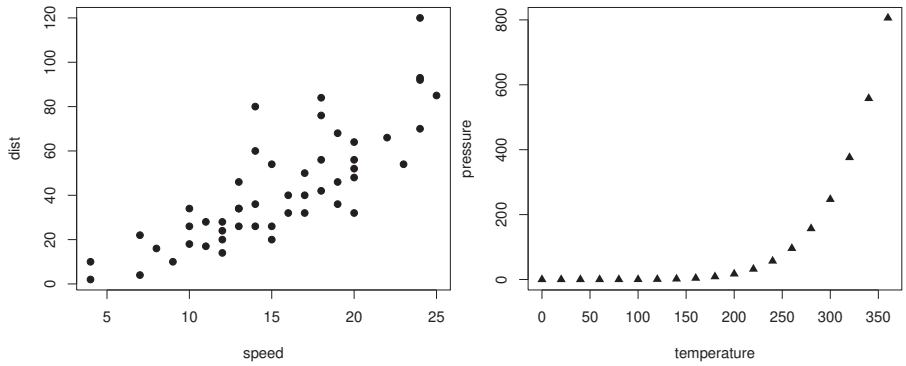


FIGURE 2.5: Two plots side-by-side.

If you want to include a graphic that is not generated from R code, you may use the `knitr::include_graphics()` function, which gives you more control over the attributes of the image than the Markdown syntax of `![alt`

`text or image title](path/to/image)` (e.g., you can specify the image width via `out.width`). Figure 2.6 provides an example of this.

````
```{r, out.width='25%', fig.align='center', fig.cap='...'}
knitr::include_graphics('images/hex-rmarkdown.png')
```
````

FIGURE 2.6: The R Markdown hex logo.

2.6.2 Tables

The easiest way to include tables is by using `knitr::kable()`, which can create tables for HTML, PDF and Word outputs.[11] Table captions can be included by passing `caption` to the function, e.g.,

````
```{r tables-mtcars}
knitr::kable(iris[1:5, ], caption = 'A caption')
```
````

Tables in non-LaTeX output formats will always be placed after the code block. For LaTeX/PDF output formats, tables have the same issue as figures: they may float. If you want to avoid this behavior, you will need to use the LaTeX package longtable[12], which can break tables across multiple pages. This can be achieved by adding `\usepackage{longtable}` to your LaTeX preamble, and passing `longtable = TRUE` to `kable()`.

If you are looking for more advanced control of the styling of tables, you are

[11] You may also consider the **pander** package. There are several other packages for producing tables, including **xtable**, **Hmisc**, and **stargazer**, but these are generally less compatible with multiple output formats.

[12] https://www.ctan.org/pkg/longtable

recommended to use the **kableExtra**[13] package, which provides functions to customize the appearance of PDF and HTML tables. Formatting tables can be a very complicated task, especially when certain cells span more than one column or row. It is even more complicated when you have to consider different output formats. For example, it is difficult to make a complex table work for both PDF and HTML output. We know it is disappointing, but sometimes you may have to consider alternative ways of presenting data, such as using graphics.

We explain in Section 12.3 how the **bookdown** package extends the functionality of **rmarkdown** to allow for figures and tables to be easily cross-referenced within your text.

2.7 Other language engines

A less well-known fact about R Markdown is that many other languages are also supported, such as Python, Julia, C++, and SQL. The support comes from the **knitr** package, which has provided a large number of *language engines*. Language engines are essentially functions registered in the object `knitr::knit_engine`. You can list the names of all available engines via:

```
names(knitr::knit_engines$get())
```

```
##  [1] "awk"         "bash"        "coffee"
##  [4] "gawk"        "groovy"      "haskell"
##  [7] "lein"        "mysql"       "node"
## [10] "octave"      "perl"        "psql"
## [13] "Rscript"     "ruby"        "sas"
## [16] "scala"       "sed"         "sh"
## [19] "stata"       "zsh"         "highlight"
## [22] "Rcpp"        "tikz"        "dot"
## [25] "c"           "fortran"     "fortran95"
## [28] "asy"         "cat"         "asis"
## [31] "stan"        "block"       "block2"
## [34] "js"          "css"         "sql"
```

[13] https://cran.r-project.org/package=kableExtra

```
## [37] "go"          "python"      "julia"
## [40] "theorem"     "lemma"       "corollary"
## [43] "proposition" "conjecture"  "definition"
## [46] "example"     "exercise"    "proof"
## [49] "remark"      "solution"
```

Most engines have been documented in Chapter 11 of Xie (2015). The engines from `theorem` to `solution` are only available when you use the **bookdown** package, and the rest are shipped with the **knitr** package. To use a different language engine, you can change the language name in the chunk header from `r` to the engine name, e.g.,

````
```{python}
x = 'hello, python world!'
print(x.split(' '))
```
````

For engines that rely on external interpreters such as `python`, `perl`, and `ruby`, the default interpreters are obtained from `Sys.which()`, i.e., using the interpreter found via the environment variable `PATH` of the system. If you want to use an alternative interpreter, you may specify its path in the chunk option `engine.path`. For example, you may want to use Python 3 instead of the default Python 2, and we assume Python 3 is at `/usr/bin/python3` (may not be true for your system):

````
```{python, engine.path = '/usr/bin/python3'}
import sys
print(sys.version)
```
````

You can also change the engine interpreters globally for multiple engines, e.g.,

```
knitr::opts_chunk$set(engine.path = list(
  python = '~/anaconda/bin/python',
  ruby = '/usr/local/bin/ruby'
))
```

Note that you can use a named list to specify the paths for different engines.

Most engines will execute each code chunk in a separate new session (via a `system()` call in R), which means objects created in memory in a previous code chunk will not be directly available to latter code chunks. For example, if you create a variable in a `bash` code chunk, you will not be able to use it in the next `bash` code chunk. Currently the only exceptions are `r`, `python`, and `julia`. Only these engines execute code in the same session throughout the document. To clarify, all `r` code chunks are executed in the same R session, all `python` code chunks are executed in the same Python session, and so on, but *the R session and the Python session are independent*.[14]

I will introduce some specific features and examples for a subset of language engines in **knitr** below. Note that most chunk options should work for both R and other languages, such as `eval` and `echo`, so these options will not be mentioned again.

2.7.1 Python

The `python` engine is based on the **reticulate** package (Allaire et al., 2018b), which makes it possible to execute all Python code chunks in the same Python session. If you actually want to execute a certain code chunk in a new Python session, you may use the chunk option `python.reticulate = FALSE`. If you are using a **knitr** version lower than 1.18, you should update your R packages.

Below is a relatively simple example that shows how you can create/modify variables, and draw graphics in Python code chunks. Values can be passed to or retrieved from the Python session. To pass a value to Python, assign to `py$name`, where `name` is the variable name you want to use in the Python session; to retrieve a value from Python, also use `py$name`.

```
---
title: "Python code chunks in R Markdown"
date: 2018-02-22
---

## A normal R code chunk
```

[14] This is not strictly true, since the Python session is actually launched from R. What I mean here is that you should not expect to use R variables and Python variables interchangeably without explicitly importing/exporting variables between the two sessions.

````
```{r}
library(reticulate)
x = 42
print(x)
```

## Modify an R variable

In the following chunk, the value of `x` on the right hand side
is `r x`, which was defined in the previous chunk.

```{r}
x = x + 12
print(x)
```

## A Python chunk

This works fine and as expected.

```{python}
x = 42 * 2
print(x)
```

The value of `x` in the Python session is `r py$x`.
It is not the same `x` as the one in R.

## Modify a Python variable

```{python}
x = x + 18
print(x)
```

Retrieve the value of `x` from the Python session again:

```{r}
````
````

````
py$x
```

Assign to a variable in the Python session from R:

```{r}
py$y = 1:5
```

See the value of `y` in the Python session:

```{python}
print(y)
```

## Python graphics

You can draw plots using the **matplotlib** package in Python.

```{python}
import matplotlib.pyplot as plt
plt.plot([0, 2, 1, 4])
plt.show()
```
````

You may learn more about the **reticulate** package from https://rstudio.github.io/reticulate/.

2.7.2 Shell scripts

You can also write Shell scripts in R Markdown, if your system can run them (the executable `bash` or `sh` should exist). Usually this is not a problem for Linux or macOS users. It is not impossible for Windows users to run Shell scripts, but you will have to install additional software (such as Cygwin[15] or the Linux Subsystem).

[15] https://www.cygwin.com
````

```
```{bash}
echo "Hello Bash!"
cat flights1.csv flights2.csv flights3.csv > flights.csv
```
```

Shell scripts are executed via the `system2()` function in R. Basically **knitr** passes a code chunk to the command `bash -c` to run it.

2.7.3 SQL

The `sql` engine uses the **DBI**[16] package to execute SQL queries, print their results, and optionally assign the results to a data frame.

To use the `sql` engine, you first need to establish a DBI connection to a database (typically via the `DBI::dbConnect()` function). You can make use of this connection in a `sql` chunk via the `connection` option. For example:

```
```{r}
library(DBI)
db = dbConnect(RSQLite::SQLite(), dbname = "sql.sqlite")
```

```{sql, connection=db}
SELECT * FROM trials
```
```

By default, `SELECT` queries will display the first 10 records of their results within the document. The number of records displayed is controlled by the `max.print` option, which is in turn derived from the global **knitr** option `sql.max.print` (e.g., `knitr::opts_knit$set(sql.max.print = 10)`; N.B. it is `opts_knit` instead of `opts_chunk`). For example, the following code chunk displays the first 20 records:

```
```{sql, connection=db, max.print = 20}
SELECT * FROM trials
```
```

[16] https://cran.rstudio.com/package=DBI

You can specify no limit on the records to be displayed via `max.print = -1` or `max.print = NA`.

By default, the `sql` engine includes a caption that indicates the total number of records displayed. You can override this caption using the `tab.cap` chunk option. For example:

````
```{sql, connection=db, tab.cap = "My Caption"}
SELECT * FROM trials
```
````

You can specify that you want no caption all via `tab.cap = NA`.

If you want to assign the results of the SQL query to an R object as a data frame, you can do this using the `output.var` option, e.g.,

````
```{sql, connection=db, output.var="trials"}
SELECT * FROM trials
```
````

When the results of a SQL query are assigned to a data frame, no records will be printed within the document (if desired, you can manually print the data frame in a subsequent R chunk).

If you need to bind the values of R variables into SQL queries, you can do so by prefacing R variable references with a `?`. For example:

````
```{r}
subjects = 10
```

```{sql, connection=db, output.var="trials"}
SELECT * FROM trials WHERE subjects >= ?subjects
```
````

If you have many SQL chunks, it may be helpful to set a default for the `connection` chunk option in the setup chunk, so that it is not necessary to specify the connection on each individual chunk. You can do this as follows:

````
```{r setup}
library(DBI)
db = dbConnect(RSQLite::SQLite(), dbname = "sql.sqlite")
knitr::opts_chunk$set(connection = "db")
```
````

Note that the `connection` option should be a string naming the connection object (not the object itself). Once set, you can execute SQL chunks without specifying an explicit connection:

````
```{sql}
SELECT * FROM trials
```
````

2.7.4 Rcpp

The `Rcpp` engine enables compilation of C++ into R functions via the **Rcpp** `sourceCpp()` function. For example:

````
```{Rcpp}
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```
````

Executing this chunk will compile the code and make the C++ function `timesTwo()` available to R.

You can cache the compilation of C++ code chunks using standard **knitr** caching, i.e., add the `cache = TRUE` option to the chunk:

````
```{Rcpp, cache=TRUE}
#include <Rcpp.h>
````
````

````
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```
````

In some cases, it is desirable to combine all of the `Rcpp` code chunks in a document into a single compilation unit. This is especially useful when you want to intersperse narrative between pieces of C++ code (e.g., for a tutorial or user guide). It also reduces total compilation time for the document (since there is only a single invocation of the C++ compiler rather than multiple).

To combine all Rcpp chunks into a single compilation unit, you use the `ref.label` chunk option along with the `knitr::all_rcpp_labels()` function to collect all of the `Rcpp` chunks in the document. Here is a simple example:

````
All C++ code chunks will be combined to the chunk below:

```{Rcpp, ref.label=knitr::all_rcpp_labels(), include=FALSE}
```

First we include the header `Rcpp.h`:

```{Rcpp, eval=FALSE}
#include <Rcpp.h>
```

Then we define a function:

```{Rcpp, eval=FALSE}
// [[Rcpp::export]]
int timesTwo(int x) {
  return x * 2;
}
```
````
````

The two `Rcpp` chunks that include code will be collected and compiled together in the first `Rcpp` chunk via the `ref.label` chunk option. Note that we set the `eval = FALSE` option on the `Rcpp` chunks with code in them to prevent them from being compiled again.

2.7.5 Stan

The `stan` engine enables embedding of the Stan probabilistic programming language[17] within R Markdown documents.

The Stan model within the code chunk is compiled into a `stanmodel` object, and is assigned to a variable with the name given by the `output.var` option. For example:

````
```{stan, output.var="ex1"}
parameters {
  real y[2];
}
model {
  y[1] ~ normal(0, 1);
  y[2] ~ double_exponential(0, 2);
}
```

```{r}
library(rstan)
fit = sampling(ex1)
print(fit)
```
````

2.7.6 JavaScript and CSS

If you are using an R Markdown format that targets HTML output (e.g., `html_document` and `ioslides_presenation`, etc.), you can include JavaScript to be executed within the HTML page using the JavaScript engine named `js`.

[17] http://mc-stan.org

For example, the following chunk uses jQuery (which is included in most R Markdown HTML formats) to change the color of the document title to red:

````
```{js, echo=FALSE}
$('.title').css('color', 'red')
```
````

Similarly, you can embed CSS rules in the output document. For example, the following code chunk turns text within the document body red:

````
```{css, echo=FALSE}
body {
  color: red;
}
```
````

Without the chunk option `echo = FALSE`, the JavaScript/CSS code will be displayed verbatim in the output document, which is probably not what you want.

2.7.7 Julia

The Julia[18] language is supported through the **JuliaCall** package (Li, 2018). Similar to the `python` engine, the `julia` engine runs all Julia code chunks in the same Julia session. Below is a minimal example:

````
```{julia}
a = sqrt(2);  # the semicolon inhibits printing
```
````

2.7.8 C and Fortran

For code chunks that use C or Fortran, **knitr** uses `R CMD SHLIB` to compile the code, and load the shared object (a `*.so` file on Unix or `*.dll` on Windows). Then you can use `.C()` / `.Fortran()` to call the C / Fortran functions, e.g.,

[18] https://julialang.org

````
```{c, test-c, results='hide'}
void square(double *x) {
  *x = *x * *x;
}
```

Test the `square()` function:

```{r}
.C('square', 9)
.C('square', 123)
```
````

You can find more examples on different language engines in the GitHub repository https://github.com/yihui/knitr-examples (look for filenames that contain the word "engine").

2.8 Interactive documents

R Markdown documents can also generate interactive content. There are two types of interactive R Markdown documents: you can use the HTML Widgets framework, or the Shiny framework (or both). They will be described in more detail in Chapter 16 and Chapter 19, respectively.

2.8.1 HTML widgets

The HTML Widgets framework is implemented in the R package **htmlwidgets** (Vaidyanathan et al., 2018), interfacing JavaScript libraries that create interactive applications, such as interactive graphics and tables. Several widget packages have been developed based on this framework, such as **DT** (Xie, 2018c), **leaflet** (Cheng et al., 2018), and **dygraphs** (Vanderkam et al., 2017). Visit https://www.htmlwidgets.org to know more about widget packages as well as how to develop a widget package by yourself.

Figure 2.7 shows an interactive map created via the **leaflet** package, and the source document is below:

````
---
title: "An Interactive Map"
---

Below is a map that shows the location of the
Department of Statistics, Iowa State University.

```{r out.width='100%', echo=FALSE}
library(leaflet)
leaflet() %>% addTiles() %>%
  setView(-93.65, 42.0285, zoom = 17) %>%
  addPopups(
    -93.65, 42.0285,
    'Here is the <b>Department of Statistics</b>, ISU'
  )
```
````

Although HTML widgets are based on JavaScript, the syntax to create them in R is often pure R syntax.

If you include an HTML widget in a non-HTML output format, such as a PDF, **knitr** will try to embed a screenshot of the widget if you have installed the R package **webshot** (Chang, 2017) and the PhantomJS package (via `webshot::install_phantomjs()`).

2.8.2 Shiny documents

The **shiny** package (Chang et al., 2018) builds interactive web apps powered by R. To call Shiny code from an R Markdown document, add `runtime: shiny` to the YAML metadata, like in this document:

```
---
title: "A Shiny Document"
output: html_document
runtime: shiny
---
```

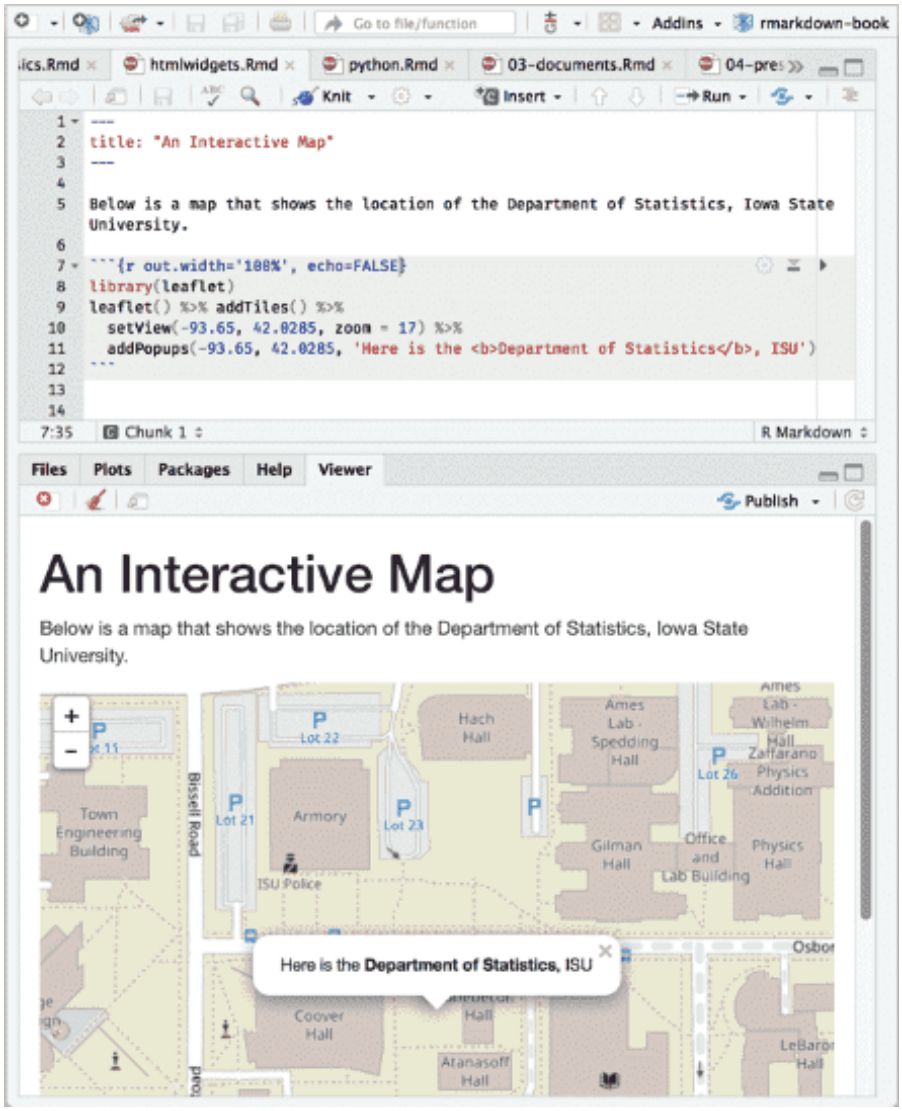


FIGURE 2.7: An R Markdown document with a leaflet map widget.

````
A standard R plot can be made interactive by wrapping
it in the Shiny `renderPlot()` function. The `selectInput()`
function creates the input widget to drive the plot.

```{r eruptions, echo=FALSE}
selectInput(
  'breaks', label = 'Number of bins:',
  choices = c(10, 20, 35, 50), selected = 20
)

renderPlot({
  par(mar = c(4, 4, .1, .5))
  hist(
    faithful$eruptions, as.numeric(input$breaks),
    col = 'gray', border = 'white',
    xlab = 'Duration (minutes)', main = ''
  )
})
```
````

Figure 2.8 shows the output, where you can see a dropdown menu that allows you to choose the number of bins in the histogram.

You may use Shiny to run any R code that you like in response to user actions. Since web browsers cannot execute R code, Shiny interactions occur on the server side and rely on a live R session. By comparison, HTML widgets do not require a live R session to support them, because the interactivity comes from the client side (via JavaScript in the web browser).

You can learn more about Shiny at https://shiny.rstudio.com.

HTML widgets and Shiny elements rely on HTML and JavaScript. They will work in any R Markdown format that is viewed in a web browser, such as HTML documents, dashboards, and HTML5 presentations.

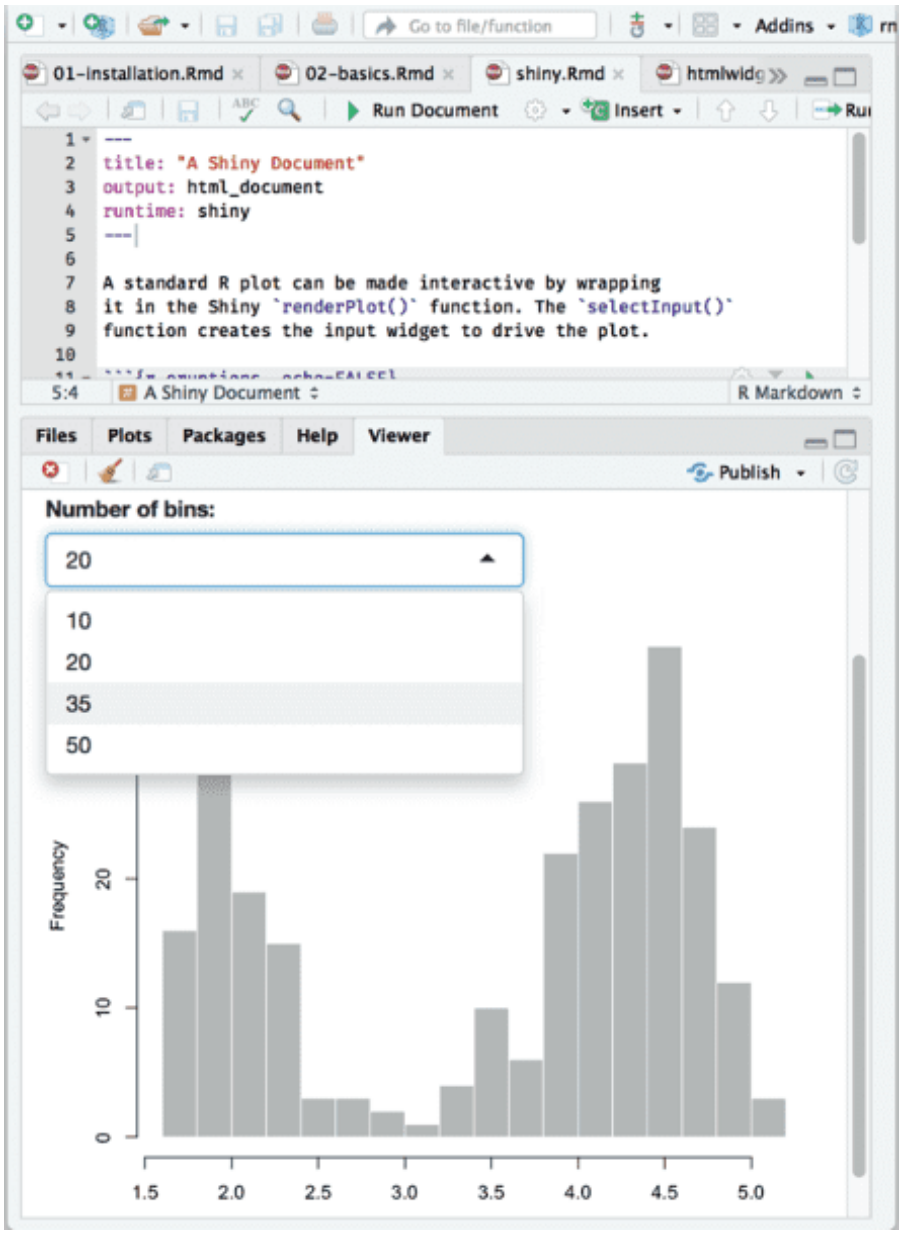
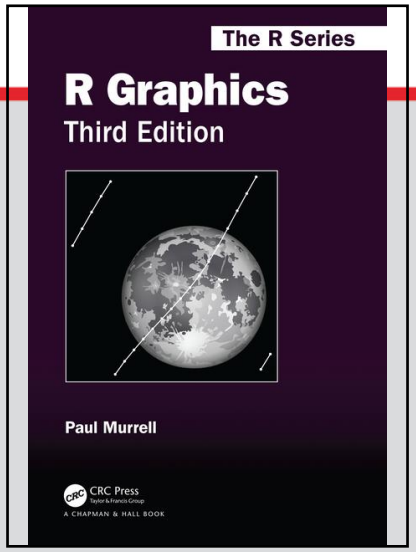


FIGURE 2.8: An R Markdown document with a Shiny widget.

CHAPTER

2

CUSTOMIZING BASE GRAPHICS



This chapter is excerpted from

R Graphics

by Paul Murrell.

Learn more

3

Customizing Base Graphics

Chapter preview

It is very often the case that a high-level plotting function does not produce exactly the final result that is desired. This chapter describes *low-level* base graphics functions that are useful for controlling the fine details of a plot and for adding further output to a plot (e.g., adding descriptive labels).

In order to utilize these low-level functions effectively, this chapter also includes a description of the regions and coordinate systems that are used to locate the output from low-level functions. For example, there is a description of which function to use to draw text in the margins of a plot as opposed to drawing text in the data region (where the data symbols are plotted). There is also a discussion of ways to arrange several plots together on a single page.

Sometimes it is not possible to achieve a final result by modifying an existing high-level plot. In such cases, the user might need to create a plot using only low-level functions. This case is also addressed in this chapter together with some discussion of how to write a new graphics function for other people to use.

It is often the case that the default or standard output from a high-level function is not exactly what the user requires, particularly when producing graphics for publication. Various aspects of the output often need to be modified or completely replaced. This chapter describes the various ways in which the output from a base graphics high-level function can be customized and extended.

The real power of the base graphics system lies in the ability to control many aspects of the appearance of a plot, to add extra output to a plot, and even to build a plot from scratch in order to produce precisely the right final output.

Section 3.1 introduces important concepts of *drawing regions*, *coordinate systems*, and *graphics state* that are required for properly working with base graphics at a lower level. Section 3.2 describes how to control aspects of output such as colors, fonts, line styles, and plotting symbols, and Section 3.3 addresses the problem of placing several plots on the same page. Section 3.4 describes how to customize a plot by adding extra output and Section 3.5 looks at ways to develop entirely new types of plots.

3.1 The base graphics model in more detail

In order to explain some of the facilities for customizing plots, it is necessary to describe more about the model underlying base graphics plots.

3.1.1 Plotting regions

In the base graphics system, every page is split up into three main regions: the *outer margins*, the current *figure region*, and the current *plot region*. Figure 3.1 shows these regions when there is only one figure on the page and Figure 3.2 shows the regions when there are multiple figures on the page.

The region obtained by removing the outer margins from the device is called the *inner region*. When there is only one figure, this usually corresponds to the figure region, but when there are multiple figures the inner region corresponds to the union of all figure regions.

The area outside the plot region, but inside the figure region is referred to as the *figure margins*. A typical high-level function draws data symbols and lines within the plot region and axes and labels in the figure margins or outer margins (see Section 3.4 for information on the functions used to draw output in the different regions). The margins are numbered 1 to 4 in the order bottom, left, top, then right. For example, "margin 3" means the top margin.

The size and location of the different regions are controlled either via the `par()` function, or using special functions for arranging plots (see Section 3.3). Specifying an arrangement of the regions does not usually affect the current plot as the settings only come into effect when the next plot is started.

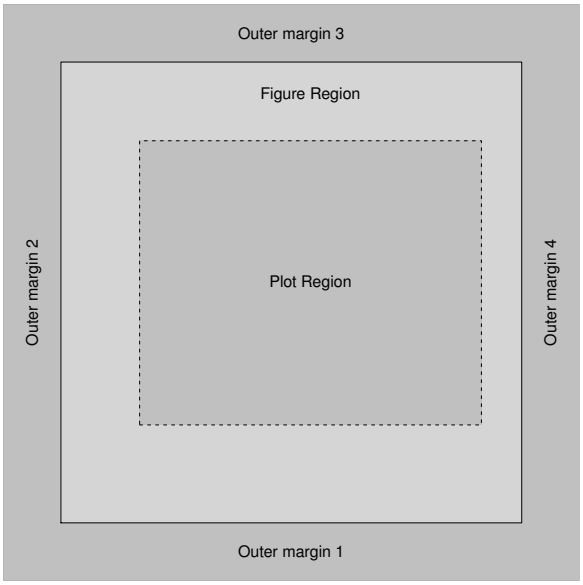


Figure 3.1
The plot regions in base graphics — the outer margins, figure region, and plot region — when there is a single plot on the page.

Coordinate systems

Each plotting region has one or more coordinate systems associated with it. Drawing in a region occurs relative to the relevant coordinate system. The coordinate system in the plot region, referred to as *user coordinates*, is probably the easiest to understand as it simply corresponds to the range of values on the axes of the plot (see Figure 3.3). The drawing of data symbols, lines, and text in the plot region occurs relative to this user coordinate system.

The scales on the axes of a plot are often set up automatically by R, but Sections 2.6 and 3.4.4 describe ways to set the scales manually.

The figure margins contain the next most commonly used coordinate systems. The coordinate systems in these margins are a combination of x- or y-ranges (like user coordinates) and lines of text away from the boundary of the plot region. Figure 3.4 shows two of the four figure margin coordinate systems. Axes are drawn in the figure margins using these coordinate systems.

There is a further set of "normalized" coordinate systems available for the figure margins in which the x- and y-ranges are replaced with a range from 0 to 1. In other words, it is possible to specify locations along the axes as a proportion of the total axis length. Axis labels and plot titles are drawn relative

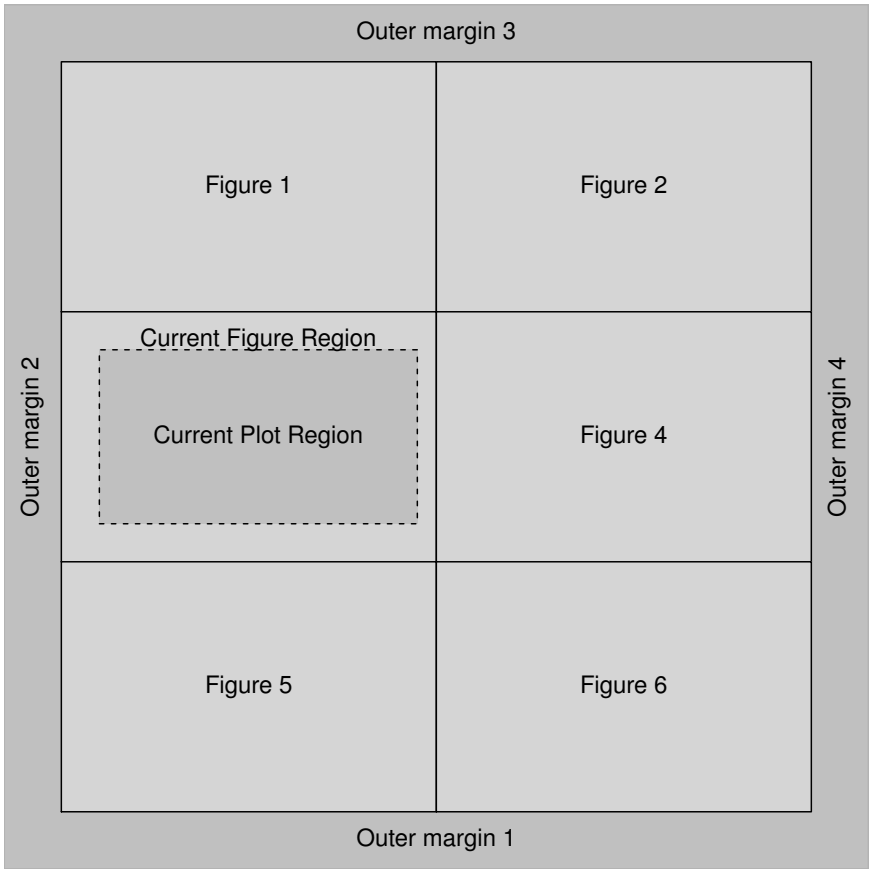


Figure 3.2
Multiple figure regions in base graphics — the outer margins, *current* figure region, and *current* plot region — when there are multiple plots on the page.

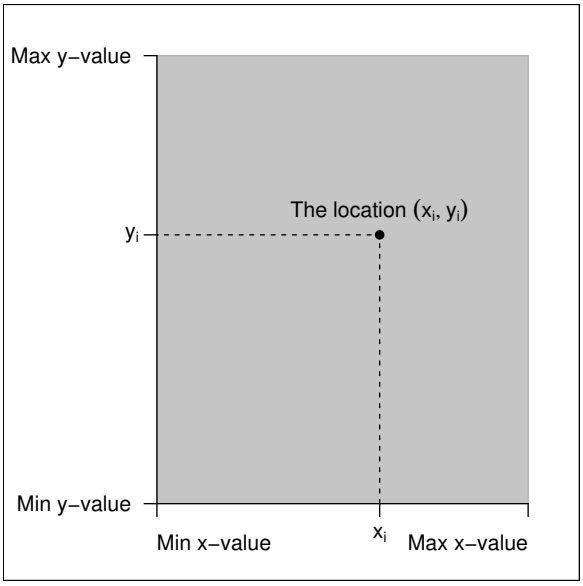


Figure 3.3
The user coordinate system in the plot region. Locations within this coordinate system are relative to the scales on the plot axes.

to this coordinate system. All of these figure margin coordinate systems are created implicitly from the arrangement of the figure margins and the setting of the user coordinate system.

The outer margins have similar sets of coordinate systems, but locations along the boundary of the inner region can only be specified in normalized coordinates (always relative to the extent of the complete outer margin). Figure 3.5 shows two of the four outer margin coordinate systems.

Sections 3.4.2 and 3.4.4 describe functions that draw output relative to the figure margin and outer margin coordinate systems.

3.1.2 The base graphics state

The base graphics system maintains a graphics "state" for the graphics window and, when drawing occurs, this state is consulted to determine where output should be drawn, what color to use, what fonts to use, and so on.

The graphics state consists of a large number of settings. Some of these settings describe the size and placement of the plot regions and coordinate systems that were described in the previous section. Some settings describe

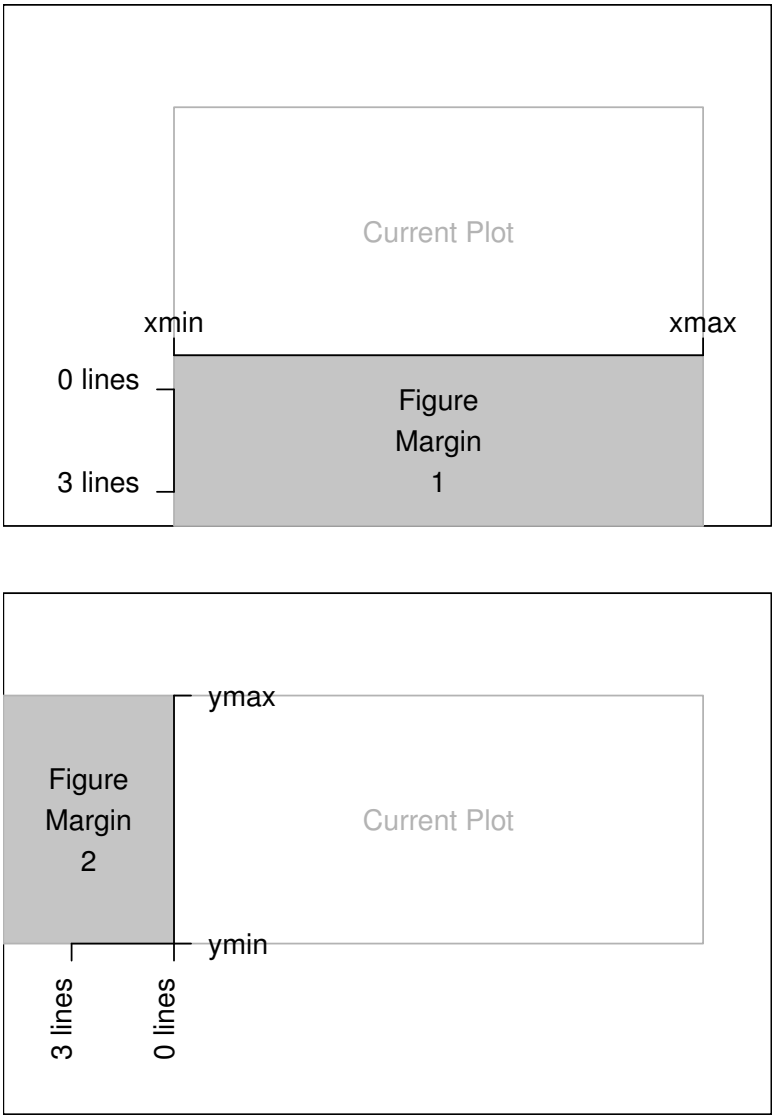


Figure 3.4
Figure margin coordinate systems. The typical coordinate systems for figure margin 1 (top plot) and figure margin 2 (bottom plot). Locations within these coordinate systems are a combination of position along the axis scale and distance away from the axis in multiples of lines of text.

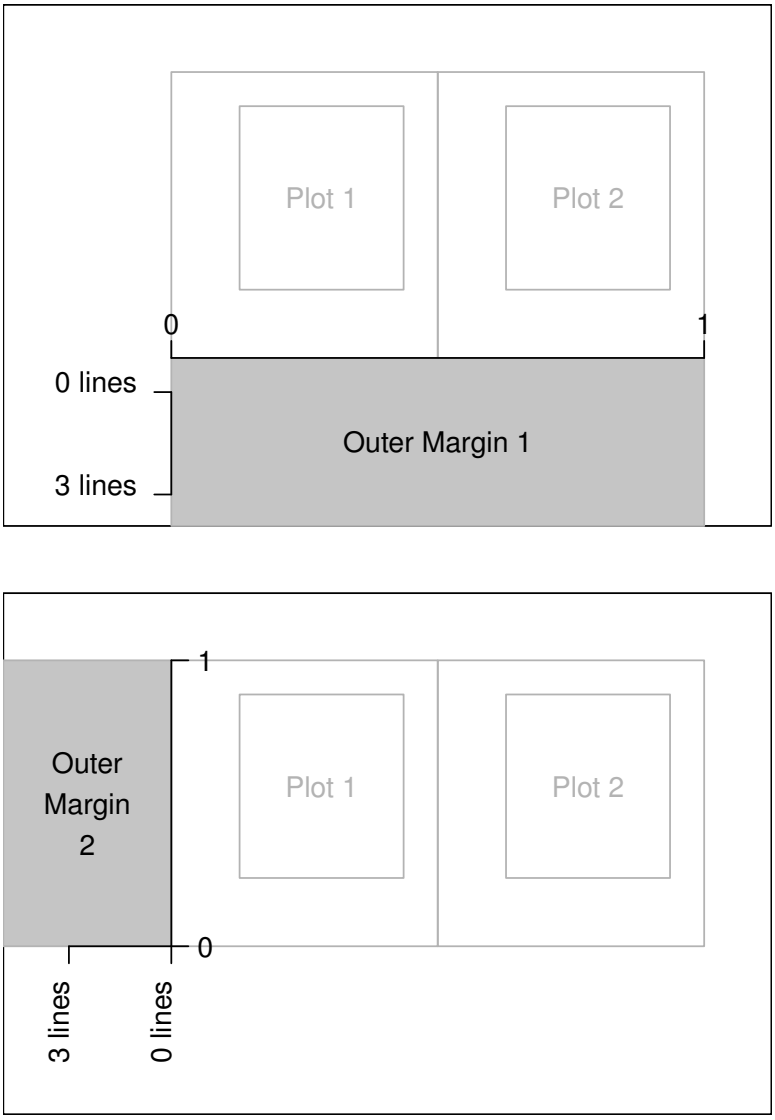


Figure 3.5
Outer margin coordinate systems. The typical coordinate systems for outer margin 1 (top plot) and outer margin 2 (bottom plot). Locations within these coordinate systems are a combination of a proportion along the inner region and distance away from the inner region in multiples of lines of text.

the general appearance of graphical output (e.g., the colors and line types that are used to draw lines and the fonts that are used to draw text) and some settings describe aspects of the output device (e.g., the physical size of the device and the current clipping region).

Tables 3.1 to 3.3 together provide a list of all of the graphics state settings and a very brief indication of their meaning. Most of the settings are described in detail in Sections 3.2 and 3.3.

The main function used to access the graphics state is the `par()` function. Simply typing `par()` will result in a complete listing of the current graphics state. A specific state setting can be queried by supplying specific setting names as arguments to `par()`. The following code queries the current state of the `col` and `lty` settings. In this case, we are asking what the current drawing color is (black) and what the current line type is (solid).

```
> par(c("col", "lty"))

$col
[1] "black"

$lty
[1] "solid"
```

The `par()` function can be used to modify base graphics state settings by specifying a value via an argument with the appropriate setting name. The following code sets new values for the `col` and `lty` settings. In this case, we are changing the drawing color to red and the line type to dashed.

```
> par(col="red", lty="dashed")
```

Modifying base graphics state settings via `par()` has a persistent effect. Settings specified in this way will hold until a different setting is specified. Settings may also be *temporarily* modified by specifying a new value in a call to a high-level graphics function such as `plot()` or a low-level graphics function such as `lines()`. The following code demonstrates this idea. First of all, the line type is permanently set to dashed using `par()`, then a plot is drawn and the lines drawn between data points in this plot are dashed. Next, a plot is drawn with a temporary line type setting of `lty="solid"` and the lines in this plot are solid. When the third plot is drawn, the permanent line type setting of `lty="dashed"` is back in effect so the lines are again dashed (see Figure 3.6).*

*The data used in this example are daily closing prices of major European stock indices available as the `EuStockMarkets` object from the **datasets** package.

Table 3.1
High-level base graphics state settings. This set of graphics state settings can be queried and set via the `par()` function *and* can be used as arguments to other graphics functions (e.g., `plot()` or `lines()`). Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`adj`	Justification of text	3.2.3
`ann`	Draw plot labels and titles?	3.2.3
`bg`	Background color	3.2.1
`bty`	Type of box drawn by `box()`	3.2.5
`cex`	Size of text (multiplier)	3.2.3
	also `cex.axis`, `cex.lab`, `cex.main`, `cex.sub`	
`col`	Color of lines and data symbols	3.2.1
	also `col.axis`, `col.lab`, `col.main`, `col.sub`	
`family`	Font family for text	3.2.3
`fg`	Foreground color	3.2.1
`font`	Font face (bold, italic) for text	3.2.3
	also `font.axis`, `font.lab`, `font.main`, `font.sub`	
`lab`	Number of ticks on axes	3.2.5
`las`	Rotation of text in margins	3.2.3
`lend`	Line end/join style	3.2.2
	also `ljoin`, `lmitre`	
`lty`	Line type (solid, dashed)	3.2.2
`lwd`	Line width	3.2.2
`mgp`	Placement of axis ticks and tick labels	3.2.5
`pch`	Data symbol type	3.2.4
`srt`	Rotation of text in plot region	3.2.3
`tck`	Length of axis ticks (relative to plot size)	3.2.5
`tcl`	Length of axis ticks (relative to text size)	3.2.5
`xaxp`	Number of ticks on x-axis	3.2.5
`xaxs`	Calculation of scale range on x-axis	3.2.5
`xaxt`	X-axis style (standard, none)	3.2.5
`xpd`	Clipping region	3.2.7
`yaxp`	Number of ticks on y-axis	3.2.5
`yaxs`	Calculation of scale range on y-axis	3.2.5
`yaxt`	Y-axis style (standard, none)	3.2.5

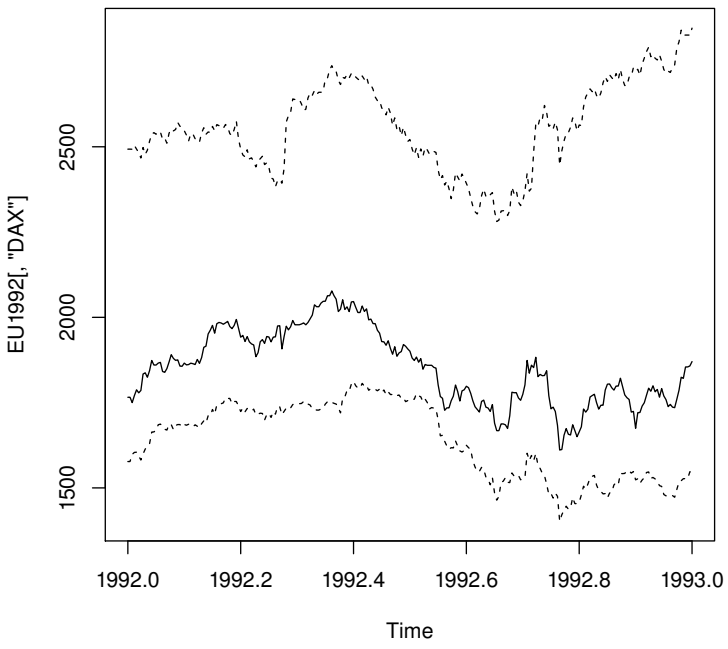


Figure 3.6
Persistent versus temporary graphical settings. The line type was set permanently to dashed with the `par()` function to draw a plot containing the top line, then the line type was temporarily set to solid in a call to `lines()` for the middle line, then the line type reverted to the permanent dashed setting for the call to `lines()` for the bottom line.

```
> EU1992 <- window(EuStockMarkets, 1992, 1993)
> par(lty="dashed")
> plot(EU1992[,"DAX"], ylim=range(EU1992))
> lines(EU1992[,"CAC"], lty="solid")
> lines(EU1992[,"FTSE"])
```

Only some of the graphics state settings can be set temporarily in calls to graphics functions. For example, the `mfrow` setting may not be set in this way and can only be set using `par()`. These "low-level" settings are listed in Table 3.2.

Table 3.2
Low-level base graphics state settings. This set of graphics state settings can only be queried and set via the `par()` function. Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`fig`	Location of figure region (normalized)	3.2.6
`fin`	Size of figure region (inches)	3.2.6
`lheight`	Line spacing (multiplier)	3.2.3
`mai`	Size of figure margins (inches)	3.2.6
`mar`	Size of figure margins (lines of text)	3.2.6
`mex`	Line spacing in margins	3.2.6
`mfcol`	Number of figures on a page	3.3.1
`mfg`	Which figure is used next	3.3.1
`mfrow`	Number of figures on a page	3.3.1
`new`	Has a new plot been started?	3.2.8
`oma`	Size of outer margins (lines of text)	3.2.6
`omd`	Location of inner region (normalized)	3.2.6
`omi`	Size of outer margins (inches)	3.2.6
`pin`	Size of plot region (inches)	3.2.6
`plt`	Location of plot region (normalized)	3.2.6
`ps`	Size of text (points)	3.2.3
`pty`	Aspect ratio of plot region	3.2.6
`usr`	Range of scales on axes	3.4.5
`xlog`	Logarithmic scale on x-axis?	3.2.5
`ylog`	Logarithmic scale on y-axis?	3.2.5

Table 3.3
Read-only base graphics state settings. This set of graphics state settings can only be queried (via the `par()` function). Each setting is described in more detail in the relevant **Section**.

Setting	Description	Section
`cin`	Size of a character (inches)	3.4.5
`cra`	Size of a character ("pixels")	3.4.5
`cxy`	Size of a character (user coordinates)	3.4.5
`din`	Size of graphics device (inches)	3.4.5
`page`	Will the next plot start a new page?	3.3.1

A small set of graphics state settings cannot be modified at all and can only be queried using `par()`. For example, there is no function to allow the user to modify the size of the current device (after the device has been created), but its size (in inches) may be obtained using `par("din")`. These "read-only" settings are listed in Table 3.3.

It is possible to have more than one graphics window open at the same time (see Section 9.1). Every graphics window has its own graphics state and calls to `par()` only affect the base graphics state of the currently active graphics window (see Section 9.1).

3.2 Controlling the appearance of plots

This section is concerned with the appearance of plots, which means the colors, line types, fonts and so on that are used to draw a plot. As described in Section 3.1.2, these features are controlled via base graphics state settings and values are specified for the settings either with a call to the `par()` function or as arguments to a specific graphics function such as `plot()`. For example, there is a setting called `col` to control the color of output (see Section 3.2.1). This can be set permanently using `par()` with an expression of the form:

```
par(col="red")
```

This will affect all subsequent graphical output. Alternatively, the setting can be specified as an argument to a high-level function using an expression of the form:

```
plot(..., col="red")
```

This will affect the output just for that plot. Finally, the setting can be used as an argument to a low-level function, as in the expression below.

```
lines(..., col="red")
```

This demonstrates that the setting can be used to control the appearance of just a single piece of graphical output.

There are many individual settings that affect the appearance of a plot, but they can be grouped in terms of what aspects of a plot the settings affect. Each of the following sections details a particular group of settings, including a description of the role of individual settings. There are sections on specifying colors; how to control the appearance of lines, text, data symbols, and axes; how to control the size and location of the various plotting regions; clipping (only drawing output on certain parts of the page); and specifying what should happen when a high-level function is called to start a new plot.

The appearance of plots is also affected by the location and size of the plotting regions, but this is dealt with separately in Section 3.3.

The following sections provide some simple examples of how to specify the settings for the base graphical parameters, but much more detail is provided in Chapter 10.

3.2.1 Colors

There are three main color settings in the base graphics state: `col`, `fg`, and `bg`.

The `col` setting is the most commonly used. The primary use is to specify the color of data symbols, lines, text, and so on that are drawn in the plot region. Unfortunately, when specified via a graphics function, the effect can vary. For example, a standard scatterplot produced by the `plot()` function will use `col` for coloring data symbols and lines, but the `barplot()` function will use `col` for filling the contents of its bars. In the `rect()` function (see Section 3.4), the `col` argument provides the color to fill the rectangle and there is a `border` argument specific to `rect()` that gives the color to draw the border of the rectangle. The effect of `col` on graphical output drawn in the margins also varies. It does not affect the color of axes and axis labels, but it does affect the output from the `mtext()` function. There are specific settings for affecting axes, labels, titles, and subtitles called `col.axis`, `col.lab`, `col.main`, and `col.sub`.

The `fg` setting is primarily intended for specifying the color of axes and borders on plots. There is some overlap between this and the specific `col.axis`, `col.main`, etc. settings described above.

The `bg` setting is primarily intended to specify the color of the background for base graphics output. This color is used to fill the entire page. As with the `col` setting, when `bg` is specified in a graphics function it can have a quite different meaning. For example, the `plot()` and `points()` functions use `bg` to specify the color for the interior of the data symbols, which can have different colors on the border (`pch` values `21` to `25`; see Section 3.2.4).

Colors may be specified in a number of different ways. The most simple is to use a color name, such as `"red"` and `"blue"`, but there are many alternatives, including generating sets of colors by calling a function. Section 10.1 describes the specification of colors in R in complete detail.

Fill patterns

In some cases (e.g., when printing in black and white), it is difficult to make use of different colors to distinguish between different elements of a plot. Using different levels of gray can be effective, but another option is to make use of some sort of fill pattern, such as cross-hatching. These should be used with caution because it is very easy to create visual effects that are distracting.

In base graphics, there is only limited support for fill patterns and they can only be applied to rectangles and polygons. It is possible to fill a rectangle or polygon with a set of lines drawn at a certain angle, with a specific separation between the lines. A `density` argument controls the separation between the lines (in terms of lines per inch) and an `angle` argument controls the angle of the lines (in terms of degrees anti-clockwise from 3 o'clock). Examples of the use of fill patterns are given in Figures 2.8, 3.20, and their associated code.

These settings can only be controlled via arguments to the functions `rect()`, `polygon()`, `hist()`, `barplot()`, `pie()`, and `legend()` (and *not* via `par()`).

3.2.2 Lines

There are five graphics state settings for controlling the appearance of lines. The `lty` setting describes the type of line to draw (e.g., solid, dashed, or dotted), the `lwd` setting describes the width of lines, and the `ljoin`, `lend`, and `lmitre` settings control how the ends and corners in lines are drawn (rounded or pointy).

The line type can be specified as a character value, for example, `"solid"`, `"dashed"`, or `"dotted"`, and the line width is given as a number, where `1`

corresponds to 1/96 inch (which is roughly 1 pixel on many computer screens).

The scope of these settings again differs depending on the graphics function being called. For example, for standard scatterplots, the setting only applies to lines drawn within the plot region. In order to affect the lines drawn as part of the axes, the `lty` setting must be passed directly to the `axis()` function (see Section 3.4.4).

Section 10.2 describes the specification of line styles in R in complete detail.

3.2.3 Text

There are a large number of base graphics state settings for controlling the appearance of text. The size of text is controlled via `ps` and `cex`; the font is controlled via `font` and `family`; the justification of text is controlled via `adj`; and the angle of rotation is controlled via `srt`.

There is also an `ann` setting, which indicates whether titles and axis labels should be drawn on a plot. This is intended to apply to high-level functions, but is not guaranteed to work with all such functions (especially functions from extension packages). There are examples of the use of `ann` as an argument to high-level plotting functions in Section 3.4.1.

Text size

The size of text is ultimately a numerical value specifying the size of the font in "points." The font size is controlled by two settings: `ps` specifies an absolute font size setting (e.g., `ps=9`), and `cex` specifies a multiplicative modifier (e.g., `cex=1.5`). The final font size specification is simply `fontsize * cex`.

As with specifying color, the scope of a `cex` setting can vary depending on where it is given. When `cex` is specified via `par()`, it affects most text. However, when `cex` is specified via `plot()`, it only affects the size of data symbols. There are special settings for controlling the size of text that is drawn as axis tick labels (`cex.axis`), text that is drawn as axis labels (`cex.lab`), text in the title (`cex.main`), and text in the subtitle (`cex.sub`).

Specifying fonts

The font used for drawing text is controlled by the settings `family` and `font`.

The `family` setting is a character value giving the name of a specific font family, such as `"Times Roman"`, or a generic family style, such as `"serif"`, `"sans"` (sans-serif), or `"mono"` (monospaced). Specific font families will only

Figure 3.7
Font families and font faces. The appearance of the twelve font family and font face combinations that are available in the base graphics system.

be available if they are installed on the operating system that R is run on, but the generic family styles are always available.

The `font` setting is a numeric value that selects between normal text (`1`), **bold** (`2`), *italic* (`3`), and ***bold-italic*** (`4`). Similar to color and text size, the `font` setting applies mostly to text drawn in the plot region. There are additional settings specifically for labels (`font.lab`), and titles (`font.main` and `font.sub`). Figure 3.7 demonstrates the 12 basic font family and face combinations.

The specification of fonts in R is described in great detail in Section 10.4.

Justification of text

The `adj` setting is a value from `0` to `1` indicating the horizontal justification of text strings (`0` means left-justified, `1` means right-justified and a value of `0.5` centers text).

The meaning of the `adj` setting depends on whether text is being drawn in the plot region, in the figure margins, or in the outer margins. In the plot region, the justification is relative to the `(x, y)` location at which the text is being drawn. In this context, it is also possible to specify two values for the setting and the second value is taken as a vertical justification for the text. Furthermore, non-finite values (`NA`, `NaN`, or `Inf`) may be specified for the justification and this is taken to mean "exact" centering (see below).

There is only a difference between a justification value of `0.5` and a non-finite justification value for vertical justification. In this case, a setting of `0.5` means text is vertically centered based on the height of the text above the text baseline (i.e., ignoring "descenders" like the tail on a "y"). A non-finite value means that text is vertically centered based on the full height of the text (including descenders). Figure 3.8 shows how various `adj` settings affect the alignment of text in the plot region.

In the figure margins and outer margins, the meaning of the `adj` setting depends on the `las` setting (see below). When margin text is parallel to the axis, `adj` specifies *both* the location and the justification of the text. For example, a value of `0` means that the text is left-justified *and* that the text is located at the left end of the margin. When text is perpendicular to the axis, the `adj` setting only affects justification. Furthermore, the `adj` setting only affects "horizontal" justification (justification in the reading direction) for text in the margins. Section 3.4.2 contains more information about the justification of text in the plot margins.

Rotating text

The `srt` setting specifies a rotation angle anti-clockwise from the positive x-axis, in degrees. This will only affect text drawn in the plot region (text drawn by the `text()` function; see Section 3.4.1). Text can be drawn at any angle within the plot region.

In the figure and outer margins, text may only be drawn at angles that are multiples of 90°, and this angle is controlled by the `las` setting. A value of `0` means text is always drawn parallel to the relevant axis (i.e., horizontal in margins 1 and 3, and vertical in margins 2 and 4). A value of `1` means text is always horizontal, `2` means text is always perpendicular to the relevant axis, and `3` means text is always vertical.

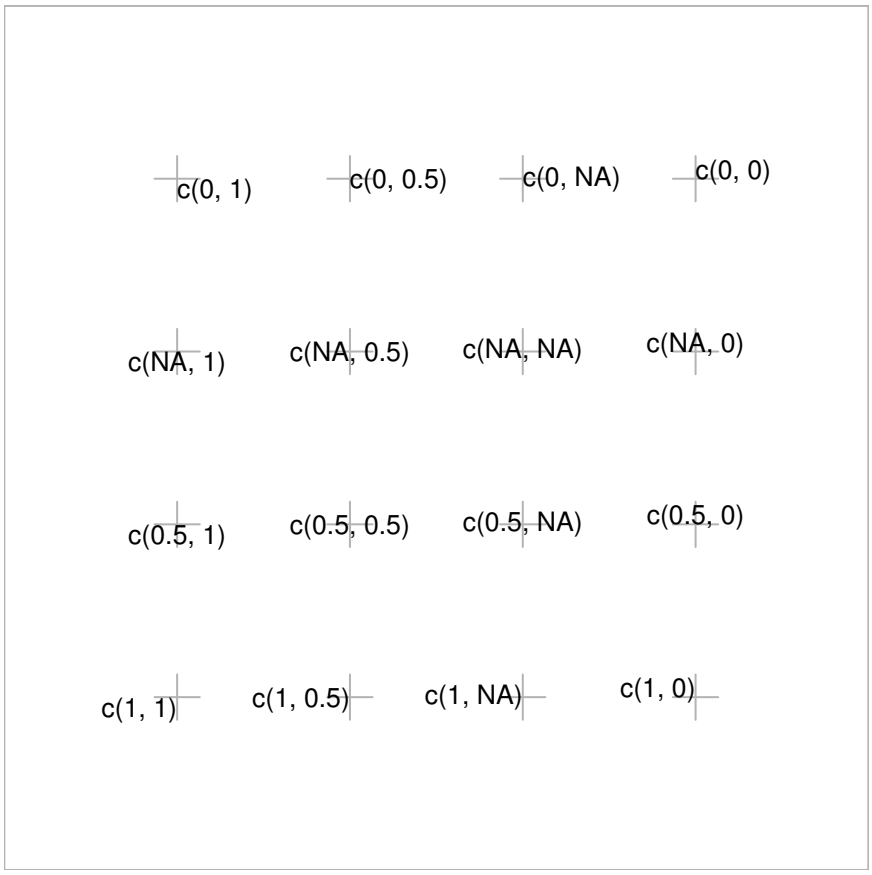


Figure 3.8
Alignment of text in the plot region. The `adj` graphical setting may be given two values, `c(`*hjust*, *vjust*`)`, where *hjust* specifies horizontal justification and *vjust* specifies vertical justification. Each piece of text in the diagram is justified relative to a gray cross to represent the effect of the relevant `adj` setting. The vertical adjustment for `NA` is subtly different from the vertical adjustment for `0.5`.

Figure 3.9
The first six data symbols that are available in base graphics. In the diagram, the relevant integer value for the `pch` setting is shown in gray to the left of the corresponding symbol.

Multi-line text

The spacing between multiple lines of text is controlled by the `lheight` setting, which is a multiplier applied to the natural height of a line of text. For example, `lheight=2` specifies double-spaced text. This setting can only be specified via `par()`.

3.2.4 Data symbols

The data symbol used for plotting points is controlled by the `pch` setting. This can be an integer value to select one of a fixed set of data symbols, or a single character. For example, specifying `pch=0` produces an open square, `pch=1` produces an open circle, and `pch=2` produces an open triangle (see Figure 3.9). Specifying `pch="#"` means that a hash character will be plotted at each data location.

Some of the predefined data symbols (`pch` between 21 and 25) allow a fill color separate from the border color, with the `bg` setting controlling the fill color in these cases.

Section 10.3 describes the possible set of data symbols in more detail.

The size of the data symbols is linked to the size of text and is affected by the `cex` setting. If the data symbol is a character, the size will also be affected by the `ps` setting.

The `type` setting controls how data are represented in a plot. A value of `"p"` means that data symbols are drawn at each `(x, y)` location. The value `"l"` means that the `(x, y)` locations are connected by lines. A value of `"b"` means that both data symbols and lines are drawn. The `type` setting may also have the value `"o"`, which means that data symbols are "over-plotted" on lines (with the value `"b"`, the lines stop short of each data symbol). It is also

possible to specify the value "h", which means that vertical lines are drawn from the x-axis to the (x, y) locations (the appearance is like a barplot with very thin bars). Two further values, "s" and "S" mean that (x, y) locations are joined in a city-block fashion with lines going horizontally then vertically (or vertically then horizontally) between each data location. Finally, the value "n" means that nothing is drawn at all.

Figure 3.10 shows simple examples of the different plot types. This setting is most often specified within a call to a high-level function (e.g., plot()) rather than via par().

3.2.5 Axes

By default, the base graphics system produces axes with sensible labels and tick marks at sensible locations. If the axis does not look right, there are a number of graphical state settings specifically for controlling aspects such as the number of tick marks and the positioning of labels. These are described below. If none of these gives the desired result, the user may have to resort to drawing the axis explicitly using the axis() function (see Section 3.4.4).

The lab setting in the base graphics state is used to control the number of tick marks on the axes. The setting is only used as a starting point for the algorithm R uses to determine sensible tick locations so the final number of tick marks that are drawn could easily differ from this specification. The setting takes two values: the first specifies the number of tick marks on the x-axis and the second specifies the number of tick marks on the y-axis.

The xaxp and yaxp settings also relate to the number and location of the tick marks on the axes of a plot. This setting is almost always calculated by R for each new plot so user settings are usually overridden (see Section 3.4.4 for an exception to this rule). In other words, it only makes sense to query this setting for its current value. The settings consist of three values: the first two specify the location of the left-most and right-most tick marks (bottom and top tick marks for the y-axis), and the third value specifies how many intervals there are between tick marks. When a log transformation is in effect for an axis, the three values have a different and much more complicated meaning altogether (see the on-line help page for par()).

The mgp setting controls the distance that the components of the axes are drawn away from the edge of the plot region. There are three values representing the positioning of the overall axis label, the tick mark labels, and the lines for the ticks. The values are in terms of lines of text away from the edges of the plot region. The default value is c(3, 1, 0). Figure 3.11 gives an example of different mgp settings.

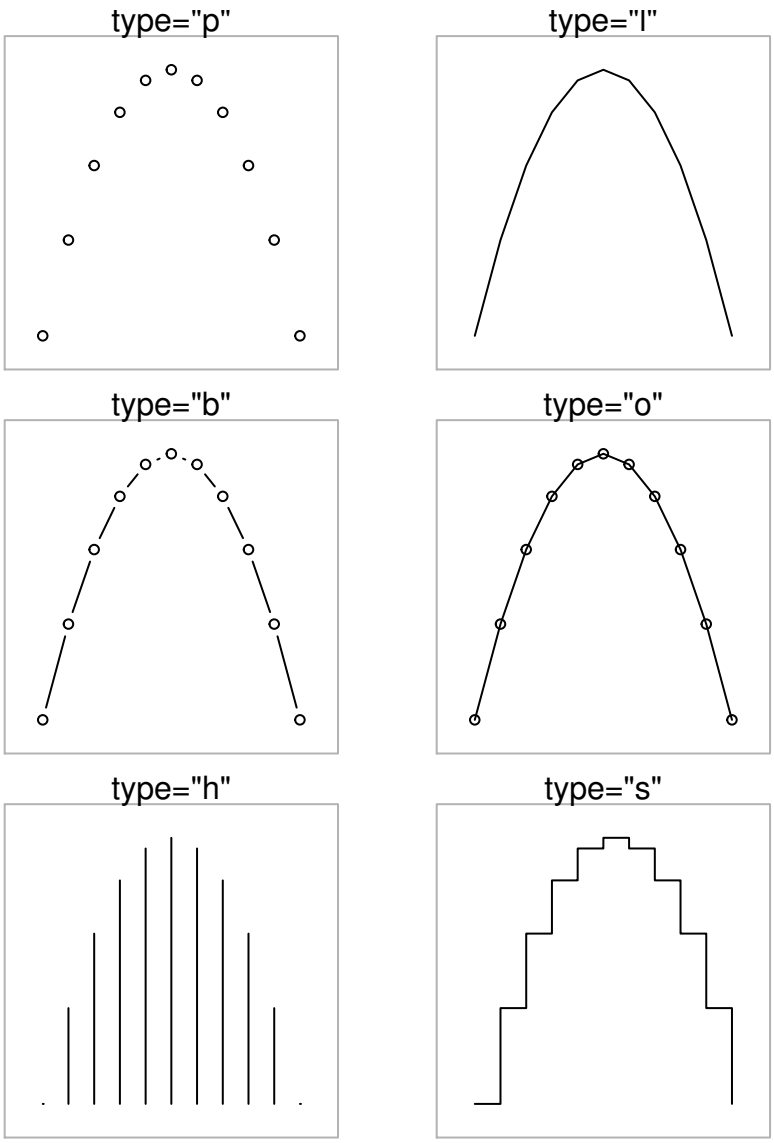


Figure 3.10
Basic plot types. Plotting the same data with different plot `type` settings. In each case, the output is produced by an expression of the form `plot(x, y, type=`*something*`)`, where the relevant value of `type` is shown above each plot.

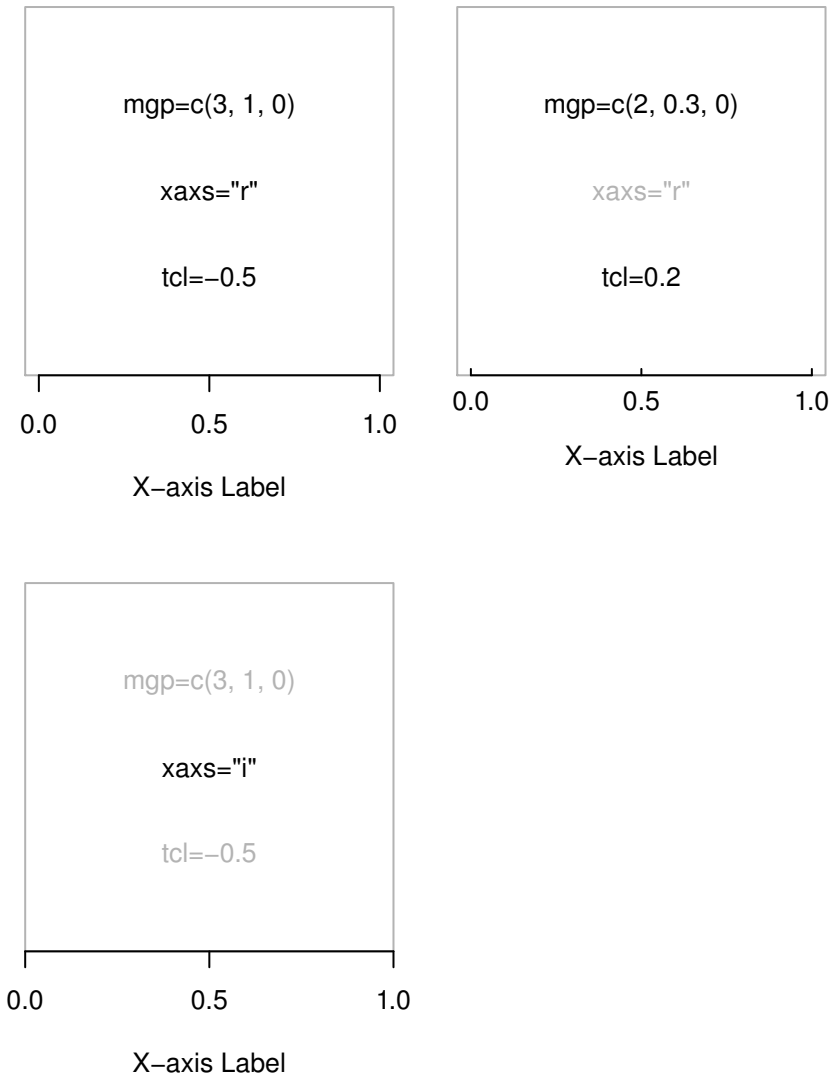


Figure 3.11
Different axis styles. The top-left plot demonstrates the default axis settings for an x-axis. The top-right plot shows the effects of specifying different positions for the axis labels (the tick labels and axis labels are closer to the plot region) and different lengths for the tick marks and the bottom-left plot shows the effect of specifying an "internal" axis range calculation.

The `tck` and `tcl` settings control the length of tick marks. The `tcl` setting specifies the length of tick marks as a fraction of the height of a line of text. The sign dictates the direction of the tick marks — a negative value draws tick marks outside the plot region and a positive value draws tick marks inside the plot region. The `tck` setting specifies tick mark lengths as a fraction of the smaller of the physical width or height of the plotting region, but it is only used if its value is not `NA` (and it is `NA` by default). Figure 3.11 gives an example of different `tcl` settings.

The `xaxs` and `yaxs` settings control the "style" of the axes of a plot. By default, the setting is `"r"`, which means that R calculates the range of values on the axis to be wider than the range of the data being plotted (so that data symbols do not collide with the boundaries of the plot region). It is possible to make the range of values on the axis exactly match the range of values in the data, by specifying the value `"i"`. This can be useful if the range of values on the axes are being explicitly controlled via `xlim` or `ylim` arguments to a function. Figure 3.11 gives an example of different `xaxs` settings.

The `xaxt` and `yaxt` settings control the "type" of axes. The default value, `"s"`, means that the axis is drawn. Specifying a value of `"n"` means that the axis is not drawn.

The `xlog` and `ylog` settings control the transformation of values on the axes. The default value is `FALSE`, which means that the axes are linear and values are not transformed. If this value is `TRUE` then a logarithmic transformation is applied to any values on the relevant dimension in the plot region. This also affects the calculation of tick mark locations on the axes.

When data of a special nature are being plotted (e.g., time series data), some of these settings may not apply (and may not have any sensible interpretation).

The `bty` setting is not strictly to do with axes, but it controls the output of the `box()` function, which is most commonly used in conjunction with drawing axes. This function draws a bounding box around the edges of the plot region (by default). The `bty` setting controls the type of box that the `box()` function draws. The value can be `"n"`, which means that no box is drawn, or it can be one of `"o"`, `"l"`, `"7"`, `"c"`, `"u"`, or `"]"`, which means that the box drawn resembles the corresponding uppercase character. For example, `bty="c"` means that the bottom, left, and top borders will be drawn, but the right border will not be drawn.

In addition to these graphics state settings, many high-level plotting functions, e.g., `plot()`, provide arguments `xlim` and `ylim` to control the range of the scale on the axes. Section 2.6.1 has an example.

3.2.6 Plotting regions

As described in Section 3.1.1, the base graphics system defines several different regions on the graphics device. This section describes how to control the size and layout of these regions using graphics state settings. Figure 3.12 shows a diagram of some of the settings that affect the widths and horizontal placement of the regions.

The size of each margin can be controlled independently, but R will check whether an overall specification is consistent. For example, if the margins are made too big, so that there is not room left on the page for the plot region, then R will give an error message like the following:

```
Error in plot.new() : figure margins too large
```

Outer margins

By default, there are no outer margins on a page. Outer margins can be specified using the `oma` graphics state setting. This consists of four values for the four margins in the order `(bottom, left, top, right)` and values are interpreted as lines of text (a value of 1 provides space for one line of text in the margin). The margins can also be specified in inches using `omi` or in normalized device coordinates (i.e., as a proportion of the device region) using `omd`. If `omd` is used, the margins are specified in the order `(left, right, bottom, top)`.

Figure regions

By default, the figure region is calculated from the settings for the outer margins and the number of figures on the page. The figure region can be specified explicitly instead, using either the `fig` setting or the `fin` state setting. The `fig` setting specifies the location, `(left, right, bottom, top)`, of the figure region where each value is a proportion of the "inner" region (the page less the outer margins). The `fin` setting specifies the size, `(width, height)`, of the figure region in inches and the resulting figure region is centered within the inner region.

Figure margins

The figure margins can be controlled using the `mar` state setting. This consists of four values for the four margins in the order `(bottom, left, top, right)` where each value represents a number of lines of text. The default values are

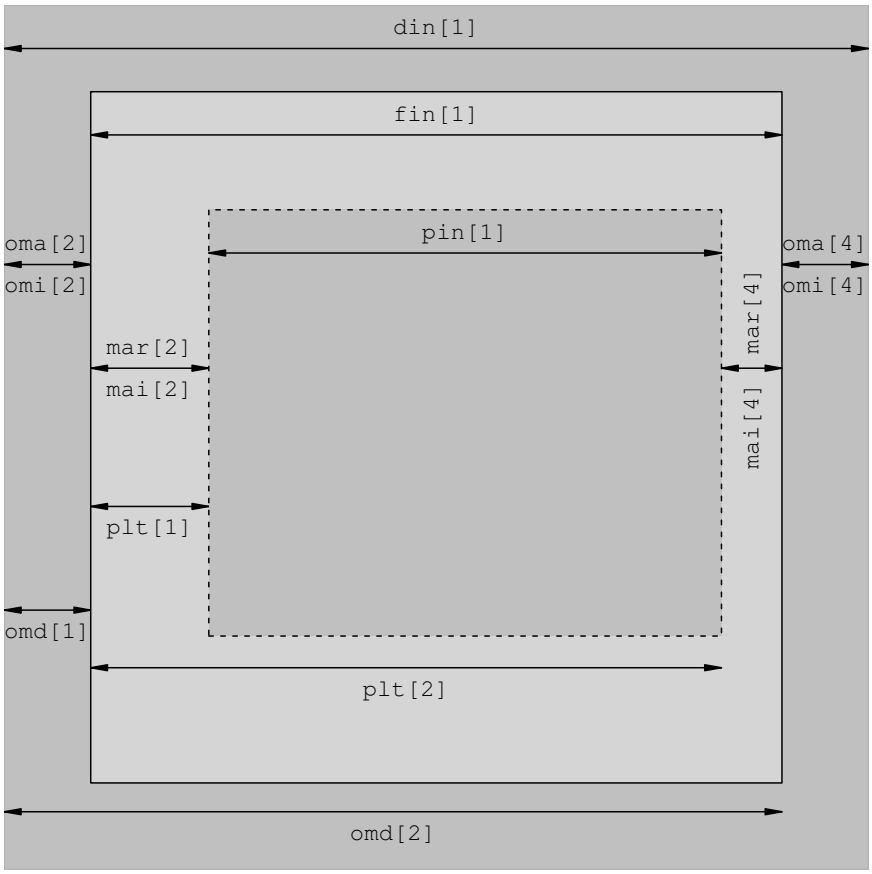


Figure 3.12
Graphics state settings controlling plot regions. These are some of the settings that control the widths and horizontal locations of the plot regions. For ease of comparison, this diagram has the same layout as Figure 3.1: the central gray rectangle represents the plot region, the lighter gray rectangle around that is the figure region, and the darker gray rectangle around that is the outer margins. A similar diagram could be produced for settings controlling heights and vertical locations.

`c(5, 4, 4, 2) + 0.1`. The margins may also be specified in terms of inches using `mai`.

The `mex` setting controls the size of a "line" in the margins. This does not affect the size of text drawn in the margins, but is used to multiply the size of text to determine the height of one line of text in the margins.

Plot regions

By default, the plot region is calculated from the figure region less the figure margins. The location and size of the plot region may be controlled explicitly instead, using the `plt`, `pin`, or `pty` settings. The `plt` setting allows the user to specify the location of the plot region, `(left, right, bottom, top)`, where each value is a proportion of current figure region. The `pin` setting specifies the size of the plot region, `(width, height)`, in terms of inches.

The `pty` setting controls how much of the available space (figure region less figure margins) the plot region occupies. The default value is `"m"`, which means that the plot region occupies all of the available space. A value of `"s"` means that the plot region will take up as much of the available space as possible, but it must be "square" (i.e., its physical width will be the same as its physical height).

3.2.7 Clipping

Base graphics output is usually clipped to the plot region. This means that any output that would appear outside the plot region is not drawn. For example, in the default behavior, data symbols for `(x, y)` locations which lie outside the ranges of the axes are not drawn. Base graphics functions that draw in the margins clip output to the current figure region or to the device. Section 3.4 has information about which functions draw in which regions.

It can be useful to override the default clipping region. For example, this is necessary to draw a legend outside the plot region using the `legend()` function.

The base clipping region is controlled via the `xpd` setting. Clipping can occur either to the whole device (an `xpd` value of `NA`), to the current figure region (a value of `TRUE`), or to the current plot region (a value of `FALSE`, which is the default).

There is also a `clip()` function for setting the clipping region to be *smaller* than the plot region.

3.2.8 Moving to a new plot

As described in Section 2.1, high-level graphics functions usually start a new plot.

The `devAskNewPage()` function can be used to control whether the user is prompted before the graphics system starts a new page of output.

The graphics state includes a setting called `new`, which controls whether a function that starts a new plot will move on to the next figure region (possibly a new page). Every plot sets the value to `FALSE` so that the next plot will move on by default, but if this setting has the value `TRUE` then a new plot does not move on to the next figure region. This can be used to overlay several plots on the same figure (Section 3.4.5 has an example).

3.3 Arranging multiple plots

There are a number of ways to produce multiple plots on a single page.

The number of plots on a page, and their placement on the page, can be controlled directly by specifying the base graphics state settings `mfrow` or `mfcol` using the `par()` function, or through a higher-level interface provided by the `layout()` function. The `split.screen()` function provides yet another approach, where a figure region can itself be treated as a complete page to split into further figure and plot regions.

These three approaches are mutually incompatible. For example, a call to the `layout()` function will override any previous `mfrow` and `mfcol` settings. Also, some high-level functions (e.g., `coplot()`) call `layout()` or `par()` themselves to create a plot arrangement, which means that the output from such functions cannot be arranged with other plots on a page (see Section 3.4.6 for further discussion; Section 12.2 describes one way to work around this limitation).

3.3.1 Using the base graphics state

The number of figure regions on a page can be controlled via the `mfrow` and `mfcol` graphics state settings. Both of these consist of two values indicating a number of rows, *nr*, and a number of columns, *nc*; these settings result in $nr \times nc$ figure regions of equal size.

The top-left figure region is used first. If the setting is made via `mfrow` then

the figure regions along the top row are used next from left to right, until that row is full. After that, figure regions are used in the next row down, from left to right, and so on. When all rows are full, a new page is started. For example, the following code creates six figure regions on the page, arranged in three rows and two columns and the regions are used in the order shown in Figure 3.13(a).

```
> par(mfrow=c(3, 2))
```

If the setting is made via `mfcol`, figure regions are used in a column-first order instead of a row-first order.

The order in which figure regions are used can be controlled explicitly by using the `mfg` setting to specify the next figure region. This setting consists of two values that indicate the row and column of the next figure to use.

The read-only `page` setting can be queried to determine whether the next high-level graphics function is going to start a new page.

3.3.2 Layouts

The `layout()` function provides an alternative to the `mfrow` and `mfcol` settings. The primary difference is that the `layout()` function allows the creation of multiple figure regions of *unequal* size.

The simple idea underlying the `layout()` function is that it divides the inner region of the page into a number of rows and columns, but the heights of rows and the widths of columns can be independently controlled, *and* a figure can occupy more than one row or more than one column.

The first argument (and the only required argument) to the `layout()` function is a matrix. The number of rows and columns in the matrix determines the number of rows and columns in the layout.

The contents of the matrix are integer values that determine which rows and columns each figure will occupy. The following layout specification is identical to `par(mfrow=c(3, 2))`.

```
> layout(matrix(c(1, 2, 3, 4, 5, 6), byrow=TRUE, ncol=2))
```

It may be easier to imagine the arrangement of figure regions if the matrix is specified using `cbind()` or `rbind()`. The code below repeats the previous example, but uses `rbind()` to specify the layout matrix.

```
> layout(rbind(c(1, 2),
               c(3, 4),
               c(5, 6)))
```

The function `layout.show()` may be helpful for visualizing the figure regions that are created. The following code creates a figure visualizing the layout created in the previous example (see Figure 3.13(a)).

```
> layout.show(6)
```

The contents of the layout matrix determine the order in which the resulting figure regions will be used. The following code creates a layout with exactly the same rows and columns as the previous one, but the figure regions will be used in the reverse order (see Figure 3.13(b)).

```
> layout(rbind(c(6, 5),
               c(4, 3),
               c(2, 1)))
```

By default, all row heights are the same and all column widths are the same size and the available inner region is divided up equally. The `heights` arguments can be used to specify that certain rows are given a greater portion of the available height (for all of what follows, the `widths` argument works analogously for column widths). When the available height is divided up, the proportion of the available height given to each row is determined by dividing the row heights by the sum of the row heights. For example, in the following layout there are two rows and one column. The top row is given two thirds of the available height, $2/(2+1)$, and the bottom row is given one third, $1/(2+1)$. Figure 3.13(c) shows the resulting layout.

```
> layout(matrix(c(1, 2)), heights=c(2, 1))
```

In the examples so far, the division of row heights has been completely independent of the division of column widths. The widths and heights can be forced to correspond as well so that, for example, a height of `1` corresponds to the same physical distance as a width of `1`. This allows control over the aspect ratio of the resulting figure. The `respect` argument is used to force this correspondence. The following code is the same as the previous example except that the `respect` argument is set to `TRUE` (see Figure 3.13(d)).

```
> layout(matrix(c(1, 2)), heights=c(2, 1),
         respect=TRUE)
```

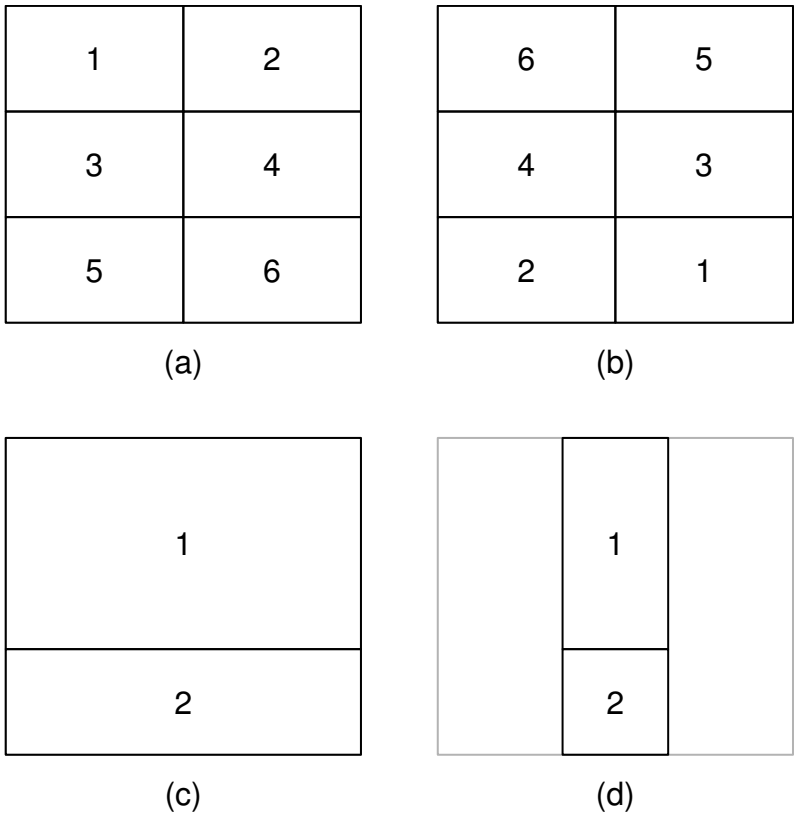


Figure 3.13
Some basic layouts: (a) A layout that is identical to `par(mfrow=c(3, 2))`; (b) Same as (a) except the figures are used in the reverse order; (c) A layout with unequal row heights; (d) Same as (c) except the layout widths and heights "respect" each other.

It is also possible to specify heights of rows and widths of columns in absolute terms. The `lcm()` function can be used to specify heights and widths for a layout in terms of centimeters. The following code is the same as the previous example, except that a third, empty region is created to provide a vertical gap of 0.5 cm between the two figures (see Figure 3.14(a)). The `0` in the first matrix argument means that no figure occupies that region.

```
> layout(matrix(c(1, 0, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=TRUE)
```

This next piece of code demonstrates that a figure may occupy more than one row or column in the layout. This extends the previous example by adding a second column and creating a figure region that occupies both columns of the bottom row. In the matrix argument, the value `2` appears in both columns of row 3 (see Figure 3.14(b)).

```
> layout(rbind(c(1, 3),
               c(0, 0),
               c(2, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=TRUE)
```

Finally, it is possible to specify that only certain rows and columns should respect each other's heights/widths. This is done by specifying a matrix for the `respect` argument. In the following code, the previous example is modified by specifying that only the first column and the last row should respect each other's widths/heights. In this case, the effect is to ensure that the width of figure region 1 is the same as the height of figure region 2, but the width of figure region 3 is free to expand to the available width (see Figure 3.14(c)).

```
> layout(rbind(c(1, 3),
               c(0, 0),
               c(2, 2)),
         heights=c(2, lcm(0.5), 1),
         respect=rbind(c(0, 0),
                       c(0, 0),
                       c(1, 0)))
```

3.3.3 The split-screen approach

The `split.screen()` function provides yet another way to divide the page into a number of figure regions. The first argument, `figs`, is either two

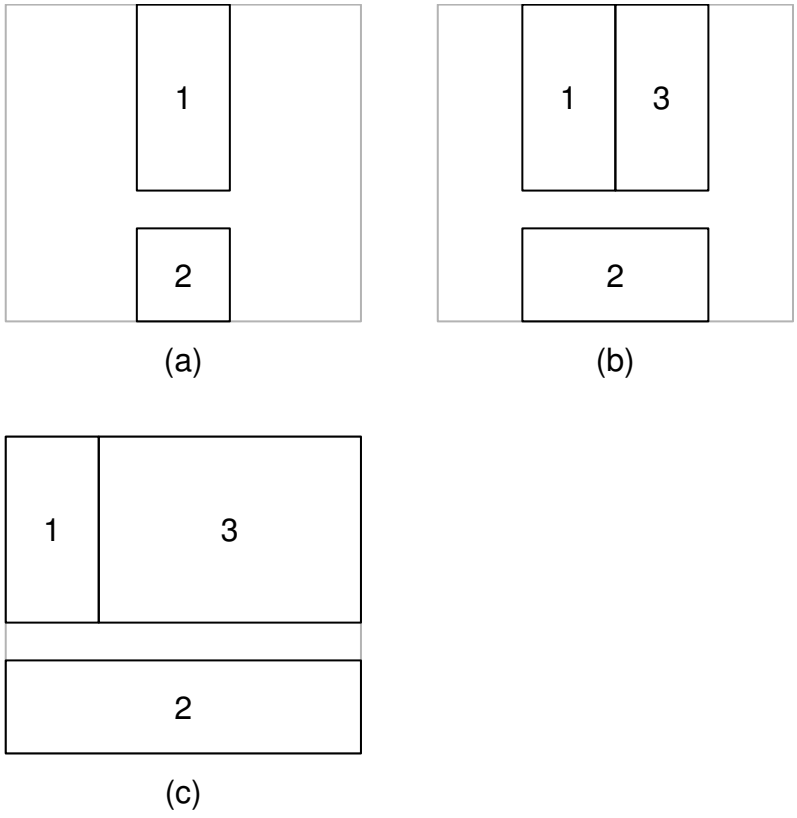


Figure 3.14
Some more complex layouts: (a) A layout with a row height specified in centimeters; (b) A layout with a figure occupying more than one column; (c) Same as (b), but with only column 1 and row 3 respected.

values specifying a number of rows and columns of figures (i.e., like the `mfrow` setting), or a matrix containing a figure region location, `(left, right, bottom, top)`, on each row (i.e., like a `par(fig)` setting on each row).

Having established figure regions in this manner, a figure region is used by calling the `screen()` function to select a region. This means that the order in which figures are used is completely under the user's control, and it is possible to reuse a figure region, though there are dangers in doing so (the on-line help for `split.screen()` provides further discussion). The function `erase.screen()` can be used to clear a defined screen and `close.screen()` can be used to remove one or more screen definitions.

An even more useful feature of this approach is that each figure region can itself be divided up by a further call to `split.screen()`. This allows complex arrangements of plots to be created.

The downside to this approach is that it does not fit very nicely with the underlying base graphics system model (see Section 3.1). The recommended way to achieve complex arrangements of plots is via the `layout()` function from the previous section or by using the **grid** graphics system (see Part II), possibly in combination with base graphics high-level functions (see Chapter 12).

3.4 Annotating plots

Sometimes it is not enough to be able to modify the default output from high-level functions and further graphical output must be added, using low-level functions, to achieve the desired result (see, for example, Figure 1.3). R graphics in general is fundamentally oriented to supporting the annotation of plots — the ability to add graphical output to an existing plot. In particular, the regions and coordinate systems used in the construction of a plot remain available for adding further output to the plot. For example, it is possible to position a text label relative to the scales on the axes of a plot.

3.4.1 Annotating the plot region

Most low-level graphics functions that add output to an existing plot, add the output to the plot region. In other words, locations are specified relative to the user coordinate system (see Section 3.1.1).

Table 3.4
The low-level base graphics functions for drawing basic graphical primitives.

Function	Description
`points()`	Draw data symbols at locations (`x`, `y`)
`lines()`	Draw lines between locations (`x`, `y`)
`segments()`	Draw line segments between (`x0`, `y0`) and (`x1`, `y1`)
`arrows()`	Draw line segments with arrowheads at the end(s)
`xspline()`	Draw a smooth curve relative to control points (`x`, `y`)
`rect()`	Draw rectangles with bottom-left corner at (`xl`, `yb`) and top-right corner at (`xr`, `yt`)
`polygon()`	Draw one or more polygons with vertices (`x`, `y`)
`polypath()`	Draw a single polygon made up of one or more paths with vertices (`x`, `y`)
`rasterImage()`	Draw a bitmap image
`text()`	Draw text at locations (`x`, `y`)

Graphical primitives

This section describes the graphics functions that provide the most basic graphics output (lines, rectangles, text, etc). Table 3.4 provides a complete list.

The most common use of this facility is to add extra sets of data to a plot. The `lines()` function draws lines between `(x, y)` locations, and the `points()` function draws data symbols at `(x, y)` locations. The following code demonstrates a common situation where three different sets of y-values, recorded at the same set of x-values, are plotted together on the same plot (see the left-hand plot in Figure 3.15).

First, we extract just a few days of data from the `EuStockMarkets` time series and plot the closing price from one market as a gray line (`type="l"` and `col="gray"`). The scale on the y-axis is set, using `ylim`, to ensure that there will be room on the plot for all of the data series.

```
> EUdays <- window(EuStockMarkets, c(1992,1), c(1992,10))
> plot(EUdays[,"DAX"], ylim=range(EUdays), ann=FALSE,
       axes=FALSE, type="l", col="gray")
```

Now a set of points are added for the first set of closing prices, then lines and points are added for the closing prices of two other markets.

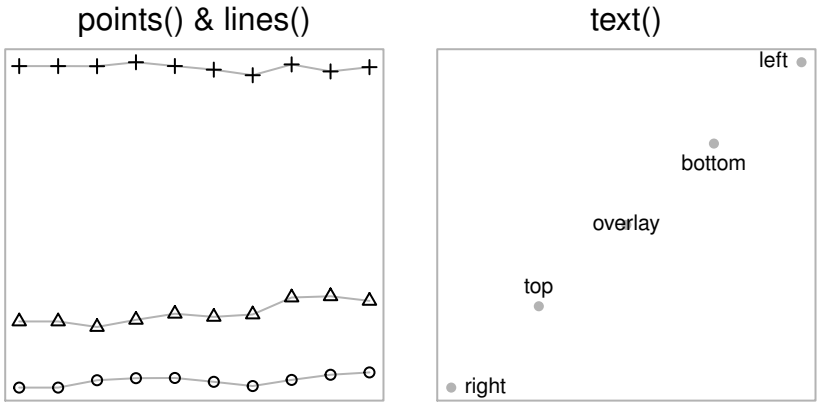


Figure 3.15
Annotating the plot region of a base graphics plot. The left-hand plot shows points and extra lines being added to an initial line plot. The right-hand plot shows text being added to an initial scatterplot.

```
> points(EUdays[,"DAX"])
> lines(EUdays[,"CAC"], col="gray")
> points(EUdays[,"CAC"], pch=2)
> lines(EUdays[,"FTSE"], col="gray")
> points(EUdays[,"FTSE"], pch=3)
```

It is also possible to draw text at `(x, y)` locations with the `text()` function. This is useful for labeling data locations, particularly using the `pos` argument to offset the text so that it does not overlay the corresponding data symbols. The following code creates a diagram demonstrating the use of `text()` (see the right-hand plot in Figure 3.15). Again, some data are created and (gray) data symbols are plotted at the `(x, y)` locations.

```
> x <- 1:5
> y <- x
> plot(x, y, ann=FALSE, axes=FALSE, col="gray", pch=16)
```

Now some text labels are added, with each one offset in a different way from the `(x, y)` location. Notice that the arguments to `text()` may be vectors so that several pieces of text are drawn by the one function call.

```
> text(x[-3], y[-3], c("right", "top", "bottom", "left"),
       pos=c(4, 3, 1, 2))
> text(3, 3, "overlay")
```

Like the `plot()` function, the `text()`, `lines()`, and `points()` functions are generic. This means that they have flexible interfaces for specifying the data for the `(x, y)` locations, or they produce different output when given objects of a particular class in the `x` argument. For example, both `lines()`, and `points()` will accept formulae for specifying the `(x, y)` locations and the `lines()` function will behave sensibly when given a `ts` (time series) object to draw.

The `text()` function normally takes a character value to draw, but it will also accept an R expression (as produced by the `expression()` function), which can be used to produce a mathematical formula with special symbols and formatting. For example, the following code draws the formula $\sqrt{2\pi\sigma^2}$. Section 10.5 describes this facility in more detail.

```
> text(0.5, 0.5, expression(sqrt(2*pi*sigma^2)))
```

As a parallel to the `matplot()` function (see Section 2.5), there are functions `matpoints()` and `matlines()` specifically for adding lines and data symbols to a plot, given `x` or `y` as matrices.

Having access to graphical primitives not only makes it easy to add new data series to a plot and to add labels, but it also makes it possible to add arbitrary drawing to a plot. In addition to lines, points, and text, there are graphical primitives for drawing more complex shapes.

In order to demonstrate these other graphical primitives, the following code produces a simple set of `x`- and `y`-values. These points will be plotted and used to draw a variety of shapes (see Figure 3.16).

```
> t <- seq(60, 360, 30)
> x <- cos(t/180*pi)*t/360
> y <- sin(t/180*pi)*t/360
```

The `lines()` function draws a single line through several points. Missing values in the `(x, y)` locations will create breaks in the line.

```
> lines(x, y)
```

An alternative is provided by the `segments()` function, which will draw several different straight lines between pairs of end points. In the following code, a straight line is drawn from `(0, 0)` to each of the `(x, y)` locations. Notice that R's normal *recycling rule* behavior is applied to most arguments of graphics functions.

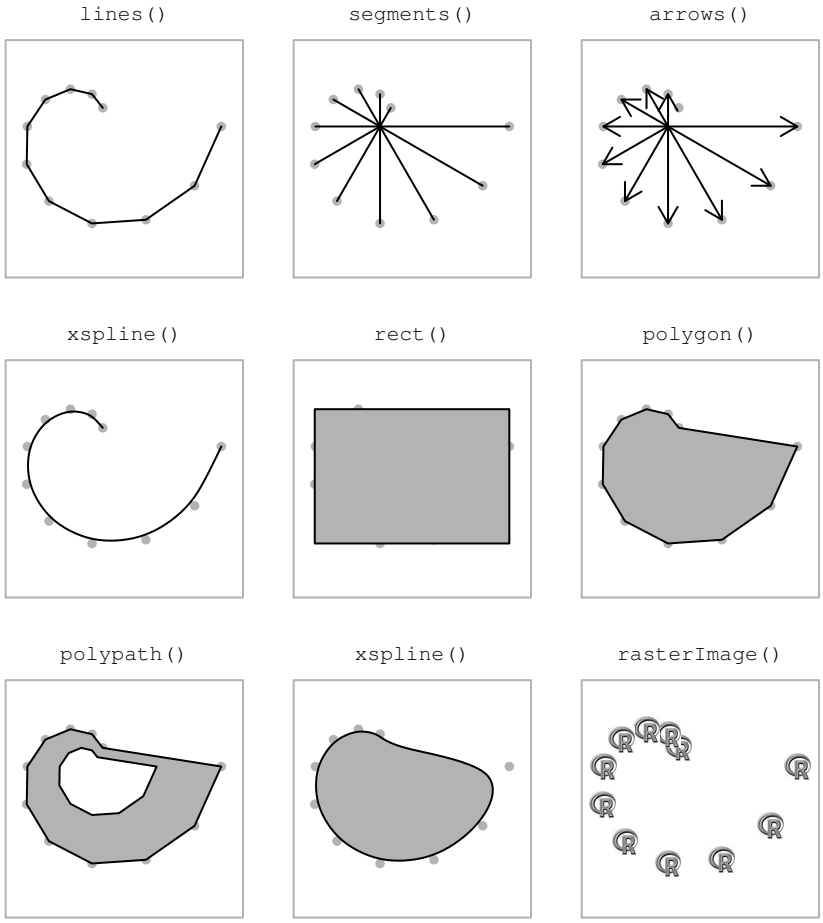


Figure 3.16
Drawing in the plot region of a base graphics plot. These pictures show some of the functions that draw more complex graphical shapes. The shapes are based on a set of `(x, y)` points which are drawn as light gray dots.

```
> segments(0, 0, x, y)
```

The `arrows()` function produces the same output as `segments()`, but also adds simple arrowheads at either end of the line segments. The `length` argument is used here to control the size of the arrowheads.

```
> arrows(0, 0, x[-1], y[-1], length=.1)
```

The `xspline()` function also produces a line, but the line is an X-spline, which treats the `(x, y)` locations as *control points* from which to produce a smooth curve. The smoothness of the curve is controlled by a `shape` parameter.

```
> xspline(x, y, shape=1)
```

There are also several functions for producing closed shapes. The simplest is `rect()`, which only requires a left, bottom, right, and top value to draw a rectangle (though all values can be vectors, which will result in several rectangles being drawn).

```
> rect(min(x), min(y), max(x), max(y), col="gray")
```

The `polygon()` function produces more complex shapes, using the `(x, y)` locations as vertices. Multiple polygons may be drawn using `polygon()` by inserting an `NA` value between each set of polygon vertexes. For both `rect()` and `polygon()`, the `col` argument specifies the color to *fill* the interior of the shape and the argument `border` controls the color of the line around the boundary of the shape.

```
> polygon(x, y, col="gray")
```

The `polygon()` function can draw self-intersecting polygons, but cannot represent polygons with holes. For the latter case, there is `polypath()`, which only draws a single polygon, but the polygon can be composed of more than one subpath. This allows for polygons consisting of distinct paths as well as polygons with holes.

```
> polypath(c(x, NA, .5*x), c(y, NA, .5*y),
           col="gray", rule="evenodd")
```

The `xspline()` function can also be used to create closed shapes, by specifying `open=FALSE`.

```
> xspline(x, y, shape=1, open=FALSE, col="gray")
```

Finally, there is a function, `rasterImage()`, for drawing bitmap images on a plot. The bitmap can be an external file, or it can just be a vector, matrix, or array. The following code draws the R logo at each of the `(x, y)` locations (code to read in the R logo is not shown; see Chapter 11 for more information).

```
> rasterImage(rlogo,
              x - .1, y - .1,
              x + .1, y + .1)
```

These examples only provide a tiny glimpse of what is possible with these graphical primitives. The possibilities are endless and a number of the examples in the remainder of this chapter provide some further demonstrations of what can be achieved by adding basic graphical shapes to a plot (see, for example, Figure 3.24).

Graphical utilities

In addition to the low-level graphical primitives of the previous section, there are a number of utility functions that provide a set of slightly more complex shapes.

The `grid()` function adds a series of grid lines to a plot. This is simply a series of line segments, but the default appearance (light gray and dotted) is suited to the purpose of providing visual cues to the viewer without interfering with the primary data symbols.

The `abline()` function provides a number of convenient ways to add a line (or lines) to a plot. The line(s) can be specified either by a slope and y-axis intercept, or as a series of x-locations for vertical lines or as a series of y-locations for horizontal lines. The function will also accept the coefficients from a linear regression analysis (even as an `"lm"` object), thereby providing a simple way to add a line of best fit to a scatterplot.

The following code annotates a basic scatterplot with a line and arrows (see the left-hand plot of Figure 3.17).

First, we plot some points in an unadorned plot.*

```
> plot(cars, ann=FALSE, axes=FALSE, col="gray", pch=16)
```

*The data used in this example are vehicle speeds and stopping distances that were recorded in the 1920s available as the data set `cars` in the `datasets` package.

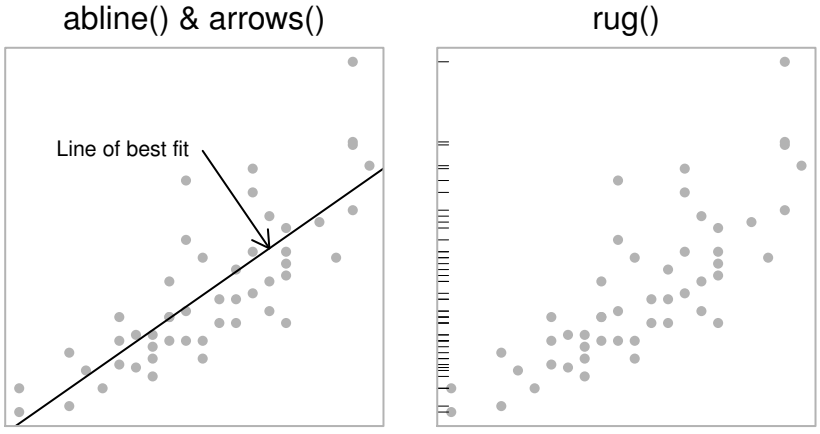


Figure 3.17
More examples of annotating the plot region of a base graphics plot. The left-hand plot shows a line of best fit (plus a text label and arrow) being added to an initial scatterplot. The right-hand plot shows a series of ticks being added as a rug plot on an initial histogram.

Now a line of best fit is drawn through the data using `abline()` and a text label and arrow are added using `text()` and `arrows()`.

```
> lmfit <- lm(dist ~ speed, cars)
> abline(lmfit)
> arrows(15, 90, 19, predict(lmfit, data.frame(speed=19)),
         length=0.1)
> text(15, 90, "Line of best fit", pos=2)
```

The `box()` function draws a rectangle around the boundary of the plot region. The `which` argument makes it possible to draw the rectangle around the current figure region, inner region, or outer region instead. The following code draws a gray box around the plot region in the plot above.

```
> box(col="gray")
```

The `rug()` function produces a "rug" plot along one of the axes, which consists of a series of tick marks representing data locations. This can be useful to represent an additional one-dimensional plot of data (e.g., in combination with a density curve). The following code uses this function to annotate the same scatterplot as above, with a set of tick marks on the y-axis to show the distribution of stopping distances (see the right-hand plot of Figure 3.17).

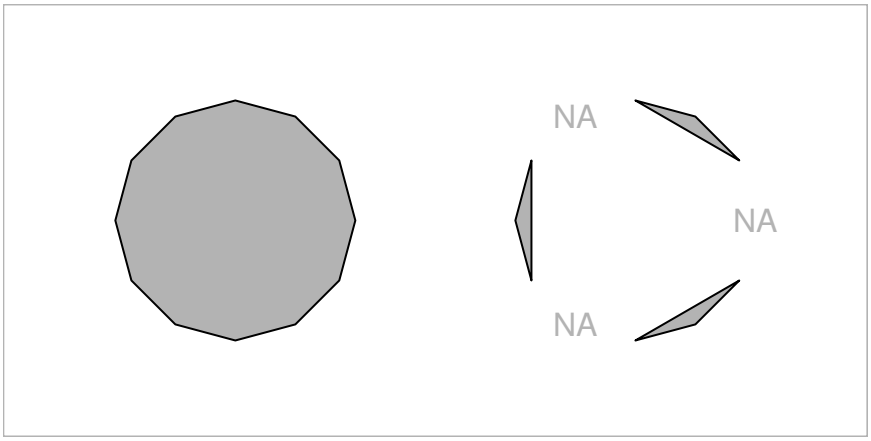


Figure 3.18
Drawing polygons using the `polygon()` function. On the left, a single polygon (dodecagon) is produced from multiple `(x, y)` locations. On the right, the first, fifth, and ninth values have been set to `NA`, which splits the output into three separate polygons. The `polygon()` function does not draw the gray `NA` values; those have been drawn using the `text()` function purely for the purposes of illustration.

```
> rug(cars$dist, side=2)
```

Missing values and non-finite values

R has special values representing missing observations (`NA`) and non-finite values (`NaN` and `Inf`). Most base graphics functions allow such values within `(x, y)` locations and handle them by not drawing the relevant location. For drawing data symbols or text, this means the relevant data symbol or piece of text will not be drawn. For drawing lines, this means that lines to or from the relevant location are not drawn; a gap is created in the line. For drawing rectangles, an entire rectangle will not be drawn if any of the four boundary locations is missing or non-finite.

Polygons are a slightly more complex case. For drawing polygons, a missing or non-finite value in `x` or `y` is interpreted as the end of one polygon and the start of another. Figure 3.18 shows an example. On the left, a polygon is drawn through 12 locations evenly spaced around a circle. On the right, the first, fifth, and ninth locations have been set to `NA` so the output is split into three separate polygons.

Missing or non-finite values can also be specified for some base graphics state settings. For example, if a color setting is missing or non-finite, then nothing is drawn (this is a brute-force way to specify a completely transparent color).

Similarly, specifying a missing value or non-finite value for `cex` means that the relevant data symbol or piece of text is not drawn.

3.4.2 Annotating the margins

There are only two functions that produce output in the figure or outer margins, relative to the margin coordinate systems (Section 3.1.1).

The `mtext()` function draws text at any location in any of the margins. The `outer` argument controls whether output goes in the figure or outer margins. The `side` argument determines which margin to draw in: `1` means the bottom margin, `2` means the left margin, `3` means the top margin, and `4` means the right margin.

Text is drawn a number of lines of text away from the edges of the plot region for figure margins or a number of lines away from the edges of the inner region for outer margins. In the figure margins, the location of the text along the margin can be specified relative to the user coordinates on the relevant axis using the `at` argument. In some cases it is possible to specify the location as a proportion of the length of the margin using the `adj` argument, but this is dependent on the value of the `las` state setting (see page 65). For certain `las` settings, the `adj` argument instead controls the justification of the text relative to a position chosen by the `las` argument. There is also a `padj` argument for controlling the "vertical" justification of text in the margins (the justification of the text **p**erpendicular to the reading direction of the text).

The `title()` function is essentially a specialized version of `mtext()`. It is more convenient for producing a few specific types of output, but much less flexible than `mtext()`. This function can be used to produce a main title for a plot (in the top figure margin), axis labels (in the left and bottom figure margins), and a subtitle for a plot (in the bottom margin below the x-axis label). The output from this function is heavily influenced by various graphics state settings, such as `cex.main` and `col.main`, which control the size and color of the title.

Just like the `text()` function, which draws text in the plot region, the functions that draw text in the margins all accept not only a character value, but also an R expression, so that axis labels and plot titles can include special symbols and formatting (see Section 10.5).

With a little extra effort, it is also possible to produce graphical output in the figure or outer margins using the functions that normally draw in the plot region (e.g., `points()` and `lines()`). In order to do this, the clipping region of the plot must first be set using the `xpd` state setting (see Section 3.2.7). This approach is not very convenient because the functions are drawing relative to user coordinates rather than locations relative to the margin co-

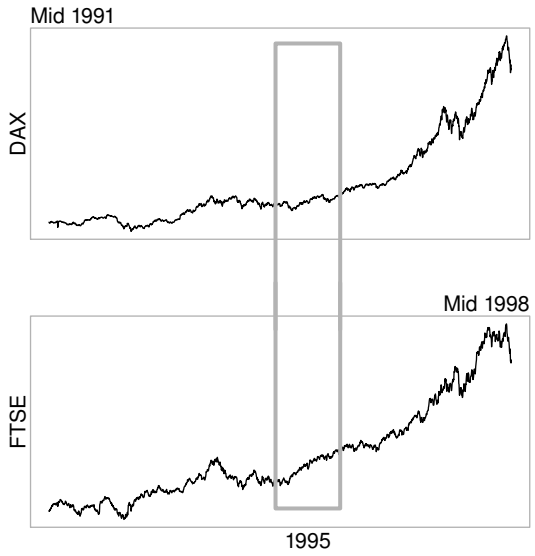


Figure 3.19
Annotating the margins of a base graphics plot. Text has been added in margin 3 of the top plot and in margins 1 and 3 in the bottom plot. Thick gray lines have been added to both plots (and overlapped so that it appears to be a single rectangle across the plots).

ordinate systems. Nevertheless, it can sometimes be useful and the functions `grconvertX()` and `grconvertY()` can help with converting locations between coordinate systems.

The following code demonstrates the use of `mtext()` and a simple application of using `lines()` outside the plot region for drawing what appears to be a rectangle extending across two plots (see Figure 3.19).*

First of all, the `mfrow` setting is used to set up an arrangement of two figure regions, one above the other. The clipping region is set to the entire device using `xpd=NA`.

```
> par(mfrow=c(2, 1), xpd=NA)
```

The first data set is plotted as a line on the top plot and a label is added at the left end of figure margin 3. In addition, thick gray lines are drawn to

*This example was motivated by a question to R-help on December 14, 2004 with subject: "drawing a rectangle through multiple plots".

represent the top of the rectangle, with the lines deliberately extending well below the bottom of the plot. The label "DAX" is drawn in figure margin 2.

```
> plot(EuStockMarkets[,"DAX"], type="l", axes=FALSE,
       xlab="", ylab="", main="")
> box(col="gray")
> mtext("Mid 1991", adj=0, side=3)
> lines(x=c(1995, 1995, 1996, 1996),
        y=c(-1000, 6000, 6000, -1000),
        lwd=3, col="gray")
> mtext("DAX", side=2, line=0)
```

The second data set is plotted as a line in the bottom plot, a label is added to this plot at the right end of figure margin 3, and another label is drawn beneath the x-location 1995.5 in figure margin 1. Finally, thick gray lines are drawn to represent the bottom of the rectangle, again deliberately extending these above the plot, and the label "FTSE" is drawn in figure margin 2. The thick gray lines overlap the lines drawn with respect to the top plot to create the impression of a single rectangle traversing both plots.

```
> plot(EuStockMarkets[,"FTSE"], type="l", axes=FALSE,
       xlab="", ylab="", main="")
> box(col="gray")
> mtext("Mid 1998", adj=1, side=3)
> mtext("1995", at=1995.5, side=1)
> lines(x=c(1995, 1995, 1996, 1996),
        y=c(7000, 2500, 2500, 7000),
        lwd=3, col="gray")
> mtext("FTSE", side=2, line=0)
```

3.4.3 Legends

The base graphics system provides the `legend()` function for adding a legend or key to a plot. The legend is usually drawn within the plot region, and is located relative to user coordinates. The function has many arguments, which allow for a great deal of flexibility in the specification of the contents and layout of the legend. The following code demonstrates a couple of typical uses.

The first example shows a scatterplot with a legend to relate group names to different symbols (see the top plot in Figure 3.20). The first two arguments give the position of the top-left corner of the legend, relative to the user coordinate system. The third argument provides labels for the legend and,

because the `pch` argument is also specified, data symbols are drawn beside each label.

```
> with(iris,
       plot(Sepal.Length, Sepal.Width,
            pch=as.numeric(Species), cex=1.2))
> legend(6.1, 4.4, c("setosa", "versicolor", "virginica"),
         cex=1.5, pch=1:3)
```

The next example shows a barplot with a legend to relate group names to different fill patterns (see the bottom plot in Figure 3.20). In this example, the `angle`, `density`, and `fill` arguments are specified, so small rectangles with fill patterns are drawn beside each label in the legend.

```
> barplot(VADeaths[1:2,], angle=c(45, 135), density=30,
          col="black", names=c("RM", "RF", "UM", "UF"))
> legend(0.4, 38, c("55-59", "50-54"), cex=1.5,
         angle=c(135, 45), density=30)
```

It should be noted that it is entirely the responsibility of the user to ensure that the legend corresponds to the plot. There is no automatic checking that data symbols in the legend match those in the plot, or that the labels in the legend have any correspondence with the data. This is one area where the **lattice** and **ggplot2** graphics systems provide a significant convenience (see Part II).

Some high-level functions draw their own legend specific to their purpose (e.g., `filled.contour()`).

3.4.4 Axes

In most cases, the axes that are automatically generated by the base graphics system will be sufficient for a plot. This is true even when the data being plotted on an axis are not numeric. For example, the axes of a boxplot or barplot are labeled appropriately using group names (see Figure 3.20).

Section 3.2.5 describes ways in which the default appearance of automatically generated axes can be modified, but it is more often the case that the user needs to inhibit the production of the automatic axis and draw a customized axis using the `axis()` function.

The first step is to inhibit the default axes. Most high-level functions should provide an `axes` argument which, when set to `FALSE`, indicates that the high-

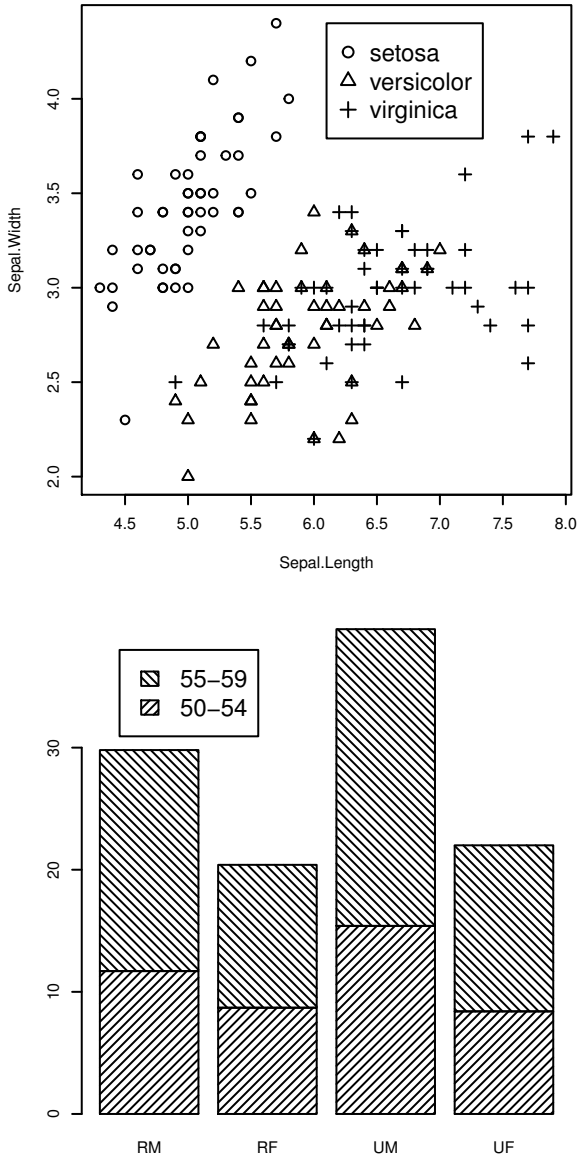


Figure 3.20
Some simple legends. Legends can be added to any kind of plot and can relate text labels to different symbols or different fill colors or patterns.

level function should not draw axes. Specifying the base graphics setting `xaxt="n"` (or `yaxt="n"`) may also do the trick.

The `axis()` function can draw axes on any side of a plot (chosen by the `side` argument), and the user can specify the location along the axis of tick marks and the text to use for tick labels (using the `at` and `labels` arguments, respectively). The following code demonstrates a simple example of a plot where the automatic axes are inhibited and custom axes are drawn, including a "secondary" y-axis on the right side of the plot (see Figure 3.21).*

First of all, a line plot is drawn with no axes.

```
> plot(nhtempCelsius, axes=FALSE, ann=FALSE, ylim=c(0, 13))
```

Next, the main y-axis is drawn with specific tick locations to represent the Centigrade scale. The number `2` means that the axis should be drawn in margin 2 (the left margin) and the `at` argument specifies the locations of the tick marks for the axis.

```
> axis(2, at=seq(0, 12, 4))
> mtext("Degrees Centigrade", side=2, line=3)
```

Now the default bottom axis is drawn and a secondary y-axis is drawn to represent the Fahrenheit scale. In the second expression, the `labels` argument is used to draw special tick mark labels on the secondary y-axis and this axis is drawn to the right of the plot by specifying `4` as the axis margin number.

```
> axis(1)
> axis(4, at=seq(0, 12, 4), labels=seq(0, 12, 4)*9/5 + 32)
> mtext(" Degrees Fahrenheit", side=4, line=3)
> box()
```

The `axis()` function is not generic, but there are special alternative functions for plotting time-related data. The functions `axis.Date()` and `axis.POSIXct()` take an object containing dates and produce an axis with appropriate labels representing times, days, months, and years (e.g., `10:15`, `Jan 12` or `1995`).

In some cases, it may be useful to draw tick marks at the locations that the default axis would use, but with different labels. The `axTicks()` function can

*The data used in this plot are (a Celsius version of) mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971, available as the data set `nhtemp` in the `datasets` package.

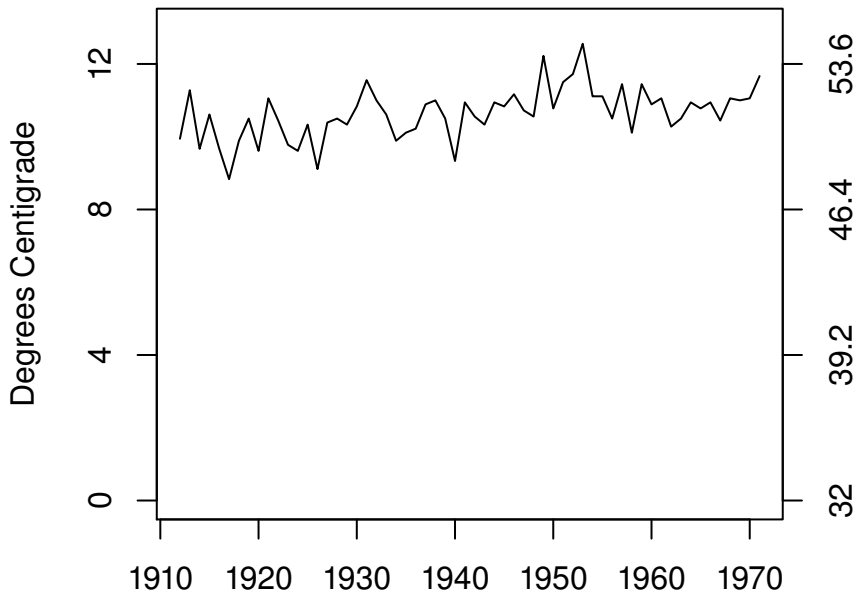


Figure 3.21
Customizing axes. On top is a data set drawn with the default axes. On the bottom, an initial plot is drawn with a y-scale in degrees Centigrade, including zero on the scale, then a secondary y-axis is drawn with a scale in degrees Fahrenheit. The labels on the secondary y-axis are specified explicitly, rather than just being the default numeric locations of the tick marks.

be used to calculate these default locations. This function is also useful for enforcing an `xaxp` (or `yaxp`) graphics state setting, which control the number and placement of tick marks. If these settings are specified via `par()`, they usually have no effect because the base graphics system almost always calculates the settings itself. The user can choose these settings by passing them as arguments to `axTicks()`, then passing the resulting locations via the `at` argument to `axis()`.

3.4.5 Coordinate systems

The base graphics system provides a number of coordinate systems for conveniently locating graphical output (see Section 3.1.1). Graphical output in the plot region is automatically positioned relative to the scales on the axes and text in the figure margins is placed in terms of a number of lines away from the edge of the plot (i.e., a scale that naturally corresponds to the size of the text).

It is also possible to locate output according to other coordinate systems that are not automatically supplied, but a little more work is required from the user. The basic principle is that the base graphics state can be queried to determine features of existing coordinate systems, then new coordinate systems can be calculated from this information.

The `par()` function

As well as being used to enforce new graphics state settings, the function `par()` can also be used to query current graphics state settings. The most useful settings are: `din`, `fin`, and `pin`, which reflect the current size, `(width, height)`, of the graphics device, figure region, and plot region, in inches; and `usr`, which reflects the current user coordinate system (i.e., the ranges on the axes). The values of `usr` are in the order `(xmin, xmax, ymin, ymax)`. When a scale has a logarithmic transformation, the values are `(10^xmin, 10^xmax, 10^ymin, 10^ymax)`.

There are also settings that reflect the size, `(width, height)`, of a "standard" character. The setting `cin` gives the size in inches, `cra` in "rasters" or pixels, and `cxy` in "user coordinates." However, these values are not very useful because they only refer to a `cex` value of `1` (i.e., they ignore the current `cex` setting) *and* they only refer to the `ps` value when the current graphics device was first opened. Of more use are the `strheight()` function and the `strwidth()` function. These calculate the height and width of a given piece of text in inches, or in terms of user coordinates, or as a proportion of the current figure region (taking into account the current `cex` and `ps` settings).

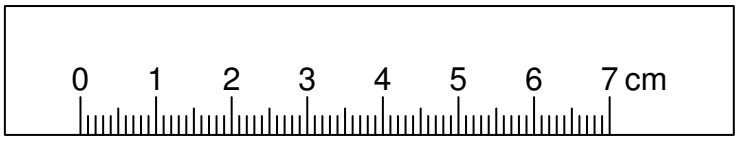


Figure 3.22
Custom coordinate systems. The lines and text are drawn relative to real physical centimeters (rather than the default coordinate system defined by the scales on plot axes).

The following code demonstrates a simple example of making use of customized coordinates where a ruler is drawn showing centimeter units (see Figure 3.22).

A blank plot region is set up first and calculations are performed to establish the relationship between user coordinates in the plot and physical centimeters.*

```
> plot(0:1, 0:1, type="n", axes=FALSE, ann=FALSE)
> usr <- par("usr")
> pin <- par("pin")
> xcm <- diff(usr[1:2])/(pin[1]*2.54)
> ycm <- diff(usr[3:4])/(pin[2]*2.54)
```

Now drawing can occur with positions expressed in terms of centimeters. The ruler itself is drawn with a call to `rect()` to draw the edges of the ruler, a call to `segments()` to draw the scale, and calls to `text()` to label the scale.

```
> rect(0, 0, 1, 1, col="white")
> segments(seq(1, 8, 0.1)*xcm, 0,
           seq(1, 8, 0.1)*xcm,
           c(rep(c(0.5, rep(0.25, 4),
                   0.35, rep(0.25, 4)),
                 7), 0.5)*ycm)
> text(1:8*xcm, 0.6*ycm, 0:7, adj=c(0.5, 0))
> text(8.2*xcm, 0.6*ycm, "cm", adj=c(0, 0))
```

*R graphics relies on having accurate information on the physical size of the natural units on the page or screen (e.g., the physical size of pixels on a computer screen). The physical size of output when producing PostScript and PDF files (see Section 9.1) should always be correct, but small inaccuracies may occur when specifying output with a physical size (such as inches) on a graphics window on screen.

Table 3.5
The coordinate systems recognized by the base graphics system.

Name	Description
`"user"`	The scales on the plot axes
`"inches"`	Inches, with (0, 0) at bottom-left
`"device"`	Pixels for screen or bitmap output, otherwise 1/72in.
`"ndc"`	Normalized coordinates, with (0, 0) at bottom-left and (1, 1) at top-right, within the entire device
`"nic"`	Normalized coordinates within the inner region
`"nfc"`	Normalized coordinates within the figure region
`"npc"`	Normalized coordinates within the plot region

There are utility functions, `xinch()` and `yinch()`, for performing the inches-to-user coordinates transformation (plus `xyinch()` for converting a location in one step and `cm()` for converting inches to centimeters). More powerful still are the `grconvertX()` and `grconvertY()` functions, which can be used to convert locations between any of the coordinate systems that the base graphics engine recognizes (see Table 3.5).

One problem with performing coordinate transformations like these is that the locations and sizes being drawn have no memory of how they were calculated. They are specified as locations and dimensions in user coordinates. This means that if the graphics window is resized (so that the relationship between physical dimensions and user coordinates changes), the locations and sizes will no longer have their intended meaning. If, in the above example, the graphics window is resized, the ruler will no longer accurately represent centimeter units. This problem will also occur if output is copied from one device to another device that has different physical dimensions. The `legend()` function performs calculations like these when arranging the components of a legend and its output is affected by resizing a device and copying between devices.*

Overlaying output

It is sometimes useful to plot two data sets on the same plot where the data sets share a common x-variable, but have very different y-scales. This can be achieved in at least two ways. One approach is simply to use `par(new=TRUE)` to overlay two distinct plots on top of each other, though care must be taken to avoid conflicting axes overwriting each other. Another approach is to explicitly

*It is possible to work around these problems in by using the `recordGraphics()` function, although this function should be used with extreme care.

reset the `usr` state setting before plotting a second set of data. The following code demonstrates both approaches to produce exactly the same result (see the top plot of Figure 3.23).

The data are yearly numbers of drunkenness-related arrests* and mean annual temperature in New Haven, Connecticut from 1912 to 1971. The temperature data are available as the data set `nhtemp` in the `datasets` package. There are only arrests data for the first 9 years.

```
> drunkenness <- ts(c(3875, 4846, 5128, 5773, 7327,
                      6688, 5582, 3473, 3186,
                      rep(NA, 51)),
                    start=1912, end=1971)
```

The first approach is to draw a plot of the drunkenness data, call `par(new=TRUE)`, then draw a complete second plot of the temperature data on top of the first plot. The second plot does not draw default axes (`axes=FALSE`), but uses the `axis()` function to draw a secondary y-axis to represent the temperature scale.

```
> par(mar=c(5, 6, 2, 4))
> plot(drunkenness, lwd=3, col="gray", ann=FALSE, las=2)
> mtext("Drunkenness\nRelated Arrests", side=2, line=3.5)
> par(new=TRUE)
> plot(nhtemp, ann=FALSE, axes=FALSE)
> mtext("Temperature (F)", side=4, line=3)
> title("Using par(new=TRUE)")
> axis(4)
```

The second approach draws only one plot (for the drunkenness data). The user coordinate system is then redefined by specifying a new `usr` setting and the second "plot" is produced simply using `lines()`. Again, a secondary axis is drawn using the `axis()` function.

*These data were obtained as "Crime Statistics and Department Demographics" from the New Haven Police Department:
`http://www.cityofnewhaven.com/police/html/stats/crime/yearly/1863-1920.htm`.

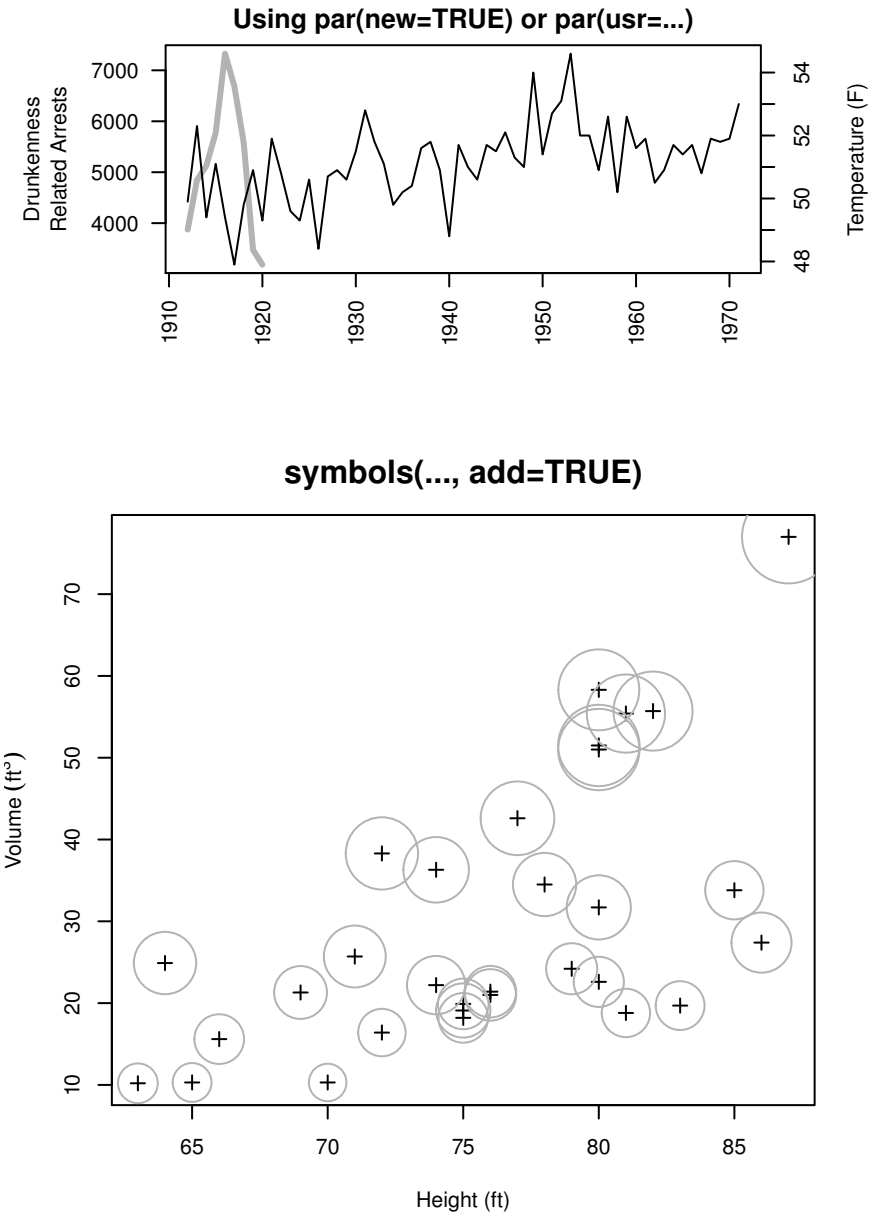


Figure 3.23
Overlaying plots. In the top plot, two line plots are drawn one on top of the other to produce aligned plots of two data sets with very different scales. In the bottom plot, the plotting function `symbols()` is used in "annotating mode" so that it adds circles to an existing scatterplot rather than producing a complete plot itself.

```
> par(mar=c(5, 6, 2, 4))
> plot(drunkenness, lwd=3, col="gray", ann=FALSE, las=2)
> mtext("Drunkenness\nRelated Arrests", side=2, line=3.5)
> usr <- par("usr")
> par(usr=c(usr[1:2], 47.6, 54.9))
> lines(nhtemp)
> mtext("Temperature (F)", side=4, line=3)
> title("Using par(usr=...)")
> axis(4)
```

Some high-level functions (e.g., `symbols()` and `contour()`) provide an argument called `add` which, if set to `TRUE`, will add the function output to the current plot, rather than starting a new plot. The following code shows the `symbols()` function being used to annotate a basic scatterplot (see the bottom plot of Figure 3.23). The data used in this example are physical measurements of black cherry trees available as the `trees` data frame from the **datasets** package.

```
> with(trees,
       {
         plot(Height, Volume, pch=3,
              xlab="Height (ft)",
              ylab=expression(paste("Volume ", (ft^3))))
         symbols(Height, Volume, circles=Girth/12,
                 fg="gray", inches=FALSE, add=TRUE)
       })
```

Another function of this type is the `bxp()` function. This function is called by `boxplot()` to draw the individual boxplots and is specifically set up to add boxplots to an existing plot (although it can also produce a complete plot).

It is also worth remembering that R follows a painters model, with later output obscuring earlier output. The following example makes use of this feature to fill a complex region within a plot (see Figure 3.24).

The first step is to prepare the data and calculate some important features of the data.

```
> x <- as.numeric(time(nhtemp))
> y <- as.numeric(nhtemp)
> n <- length(x)
> mean <- mean(y)
```

The first thing to draw is a plot with a filled polygon beneath the y-values (see the top-left plot of Figure 3.24).

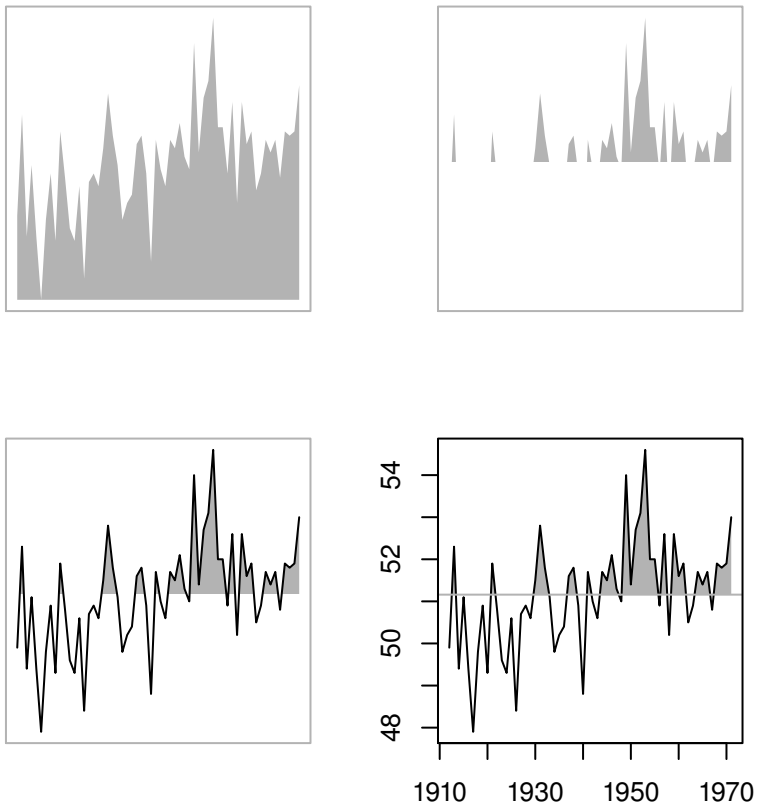


Figure 3.24
Overlaying output (making use of the painters model). The final complex plot, shown at bottom-right, is the result of overlaying several basic pieces of output: a gray polygon at top-left, with a white rectangle over the top (top-right), a black line on top of that (bottom-left), and a gray line on top of it all (plus axes and a bounding box).

```
> plot(x, y, type="n", axes=FALSE, ann=FALSE)
> polygon(c(x[1], x, x[n]), c(min(y), y, min(y)),
          col="gray", border=NA)
```

The next step is to draw a rectangle over the top of the polygon up to a fixed y-value. The expression `par("usr")` is used to obtain the current x-scale and y-scale ranges (see the top-right plot of Figure 3.24).

```
> usr <- par("usr")
> rect(usr[1], usr[3], usr[2], mean, col="white", border=NA)
```

Now a line through the y-values is drawn over the top of the rectangle (see the bottom-left plot of Figure 3.24).

```
> lines(x, y)
```

Finally, a horizontal line is drawn to indicate the y-value cut-off, and axes are added to the plot (see the bottom-right plot of Figure 3.24).

```
> abline (h=mean, col="gray")
> box()
> axis(1)
> axis(2)
```

3.4.6 Special cases

Some high-level functions are a little more difficult to annotate than others because the plotting regions that they set up either are not immediately obvious or are not available after the function has run. This section describes a number of high-level functions where additional knowledge is required to perform annotations.

Obscure scales on axes

It is not immediately obvious how to add extra annotation to a barplot or a boxplot in base R graphics because the scale on the categorical axis is not obvious.

The difficulty with the `barplot()` function is that, because the scale on the x-axis is not labeled at all by default, the numeric scale is not obvious (and calling `par("usr")` is not much help because the scale that the function sets

up is not intuitive either). In order to add annotations sensibly to a barplot it is necessary to capture the value returned by the function. This return value gives the x-locations of the mid-points of each bar that the function has drawn. These midpoints can then be used to locate annotations relative to the bars in the plot.

The code below shows an example of adding extra horizontal reference lines to the bars of a barplot. The mid-points of the bars are saved to a variable called `midpts`, then locations are calculated from those mid-points (and the original counts) to draw horizontal white line segments within each bar using the `segments()` function (see the left plot of Figure 3.25).

```
> y <- sample(1:10)
> midpts <- barplot(y, col=" light gray")
> width <- diff(midpts[1:2])/4
> left <- rep(midpts, y - 1) - width
> right <- rep(midpts, y - 1) + width
> heights <- unlist(apply(matrix(y, ncol=10),
                          2, seq))[-cumsum(y)]
> segments(left, heights, right, heights,
           col="white")
```

The `boxplot()` function is similar to the `barplot()` function in that the x-scale is typically labeled with category names so the numeric scale is not obvious from looking at the plot. Fortunately, the scale set up by the `boxplot()` function is much more intuitive. The individual boxplots are drawn at x-locations `1:n`, where `n` is the number of boxplots being drawn.

The following code provides a simple example of annotating boxplots to add a jittered dotplot of individual data points on top of the boxplots. This provides a detailed view of the data with individual points and shows the main features of the data via the boxplot. It is also a useful way to show how interesting features of the data, such as small clusters of points, can be hidden by a boxplot. In this example, the jittered data are centered upon the x-locations `1:2` to correspond to the centers of the relevant boxplots (see the right plot of Figure 3.25).

```
> with(ToothGrowth,
       {
         boxplot(len ~ supp, border="gray",
                 col="light gray", boxwex=0.5)
         points(jitter(rep(1:2, each=30), 0.5),
                unlist(split(len, supp)),
                cex=0.5, pch=16)
       })
```

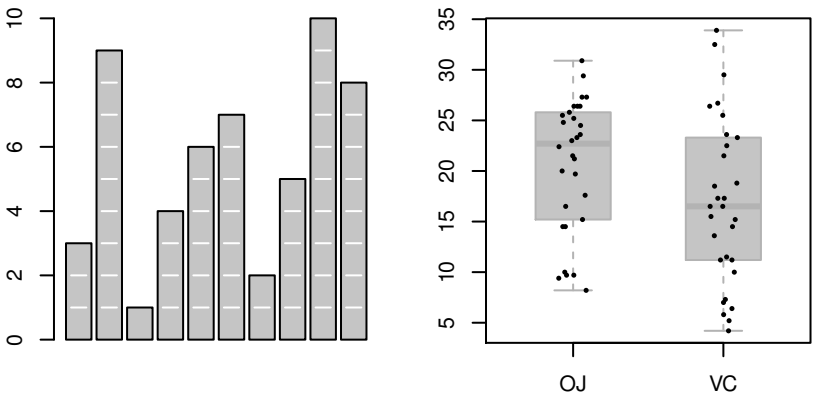


Figure 3.25
Special-case annotations. Some examples of functions where annotation requires special care. In the barplot at left, the value returned by the `barplot()` function is used to add horizontal white lines within the bars. Jittered points are added to the boxplot (right) using the knowledge that the ith box is located at position i on the x-axis.

Functions that draw several plots

The `pairs()` function is an example of a high-level function that draws more than one plot. This function draws a matrix of scatterplots. Such functions tend to save the base graphics state before drawing, call `par(mfrow)` or `layout()` to arrange the individual plots, and restore the base graphics state once all of the individual plots have been drawn. This means that it is not possible to annotate any of the plots drawn by the `pairs()` function once the function has completed drawing. The regions and coordinate systems that the function set up to draw the individual plots have been thrown away. The only way to annotate the output from such functions is by way of *panel functions*.

The `pairs()` function has a number of arguments that allow the user to specify a function: `panel`, `diag.panel`, `upper.panel`, `lower.panel`, and `text.panel`. The functions specified via these arguments are run as each individual plot is drawn. In this way, the panel function has access to the plot regions that are set up for each individual plot.

The following code shows a `pairs()` plot of the first two variables in the `iris` data set. The `diag.panel` argument is used to draw boxplots in the diagonal panels, instead of the default variable names. Notice that the panel function must only add extra output, *not* start its own plot and this is achieved in this

case by called `boxplot()` with `add=TRUE`. Because `axes=FALSE`, the normal boxplot axes are not drawn, and the `at` argument is used to make sure the boxplots are centered horizontally within the panels. Because the normal diagonal panels have variable names drawn in them, a `text.panel` function is also specified. This panel function calls `mtext()` so that the normal text is drawn in the top margin of the panel instead. The resulting plot is shown in Figure 3.26.

```
> pairs(iris[1:2],
        diag.panel=function(x, ...) {
            boxplot(x, add=TRUE, axes=FALSE,
                    at=mean(par("usr")[1:2]))
        },
        text.panel=function(x, y, labels, ...) {
            mtext(labels, side=3, line=0)
        })
```

The `filled.contour()` function and the `coplot()` function have the same problem as `pairs()` because the legends that they draw are actually separate plots. Again, those functions allow annotation via panel function arguments.

The `panel.smooth()` function provides a predefined panel function to add a smoothed trend line to a scatterplot of points.

3D plots

It is possible to annotate a plot that was produced using the `persp()` function, but it is more difficult than for most other high-level functions. The important step is to acquire the transformation matrix that the `persp()` function returns. This can be used to transform 3D locations into 2D locations, using the `trans3d()` function. The result can then be given to the standard annotation functions such as `lines()` and `text()`. The `persp()` function also has an `add` argument, which allows multiple `persp()` plots to be over-plotted.

The following code demonstrates annotation of `persp()` output to add a contour plot beneath a 3D plot of the Maunga Whau volcano in Auckland New Zealand (see Figure 3.27). The data are from the `volcano` matrix in the **datasets** package.

The first step is to draw the 3D surface. The important features of this code are that the `zlim` is specified to leave room for the contour plot and the result of the call to `persp()` is assigned to a variable called `trans`.

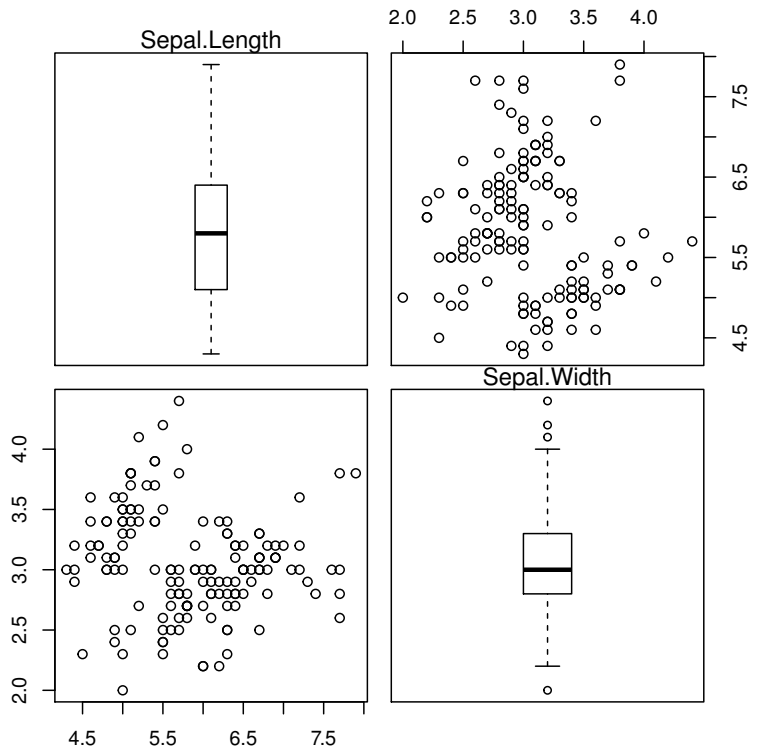


Figure 3.26
A panel function example. An example of using a panel function to add customized output to each of the diagonal panels of a `pairs()` plot.

```
> z <- 2 * volcano
> x <- 10 * (1:nrow(z))
> y <- 10 * (1:ncol(z))
> trans <- persp(x, y, z, zlim=c(0, max(z)),
                 theta = 150, phi = 12, lwd=.5,
                 scale = FALSE, axes=FALSE)
```

The next code calculates contour lines from the 3D data and then adds them to the plot. The result of `contourLines()` is a list, so `lapply()` is used to draw each contour line separately. The locations of the contour lines in the 3D plot are calculated using `trans3d()`, which is given the `x` and `y` vertices for a contour line, plus the `z`-position of zero (below the 3D surface). The `trans3d()` function converts the 3D locations into 2D locations which are drawn with the `lines()` function.

```
> clines <- contourLines(x, y, z)
> lapply(clines,
         function(contour) {
             lines(trans3d(contour$x, contour$y, 0, trans))
         })
```

A major limitation with annotating `persp()` output is that there is no support for automatically hiding output that should not be seen. In the above example, the view point was carefully chosen so that the entire contour plot was visible beneath the 3D surface. If the viewing angle is changed so that the surface and the contour lines overlap, the contour lines will be drawn *on top of* the 3D surface because they are drawn second. In simple cases, this sort of problem can be worked around through careful ordering of drawing operations, but in the general case a more sophisticated 3D graphics system would be required (e.g., the **rgl** package).

3.5 Creating new plots

There are cases where no existing plot provides a sensible starting point for creating the final plot that the user requires; situations where simply drawing more shapes on the plot is not sufficient. This section describes how to construct a new plot entirely from scratch for such cases.

The `plot.new()` function is the most basic starting point for producing a base graphics plot (the `frame()` function is equivalent). This function starts a new

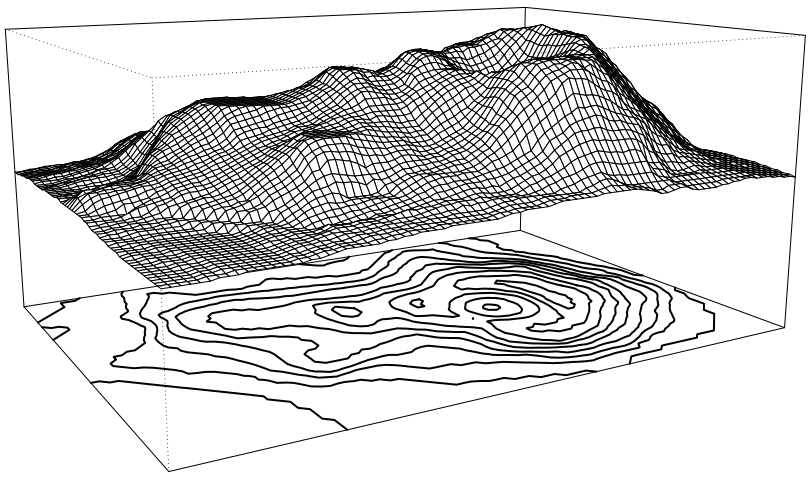


Figure 3.27
Annotating a 3D surface created by `persp()`. The contour lines are added to the 3D plot using the transformation matrix returned by the `persp()` function.

plot and sets up the various plotting regions described in Section 3.1.1, with both the x-scale and y-scale set to $(0, 1)$.* The size and position of the regions that are set up depend on the current graphics state settings (see Section 3.2.6).

The `plot.window()` function resets the scales in the user coordinate system, given x- and y-ranges via the arguments `xlim` and `ylim`, and the `plot.xy()` function draws data symbols and lines between locations within the plot region.

3.5.1 A simple plot from scratch

In order to demonstrate the use of these functions, the following code produces a very simple scatterplot like Figure 1.1 from scratch. The result is shown in Figure 3.28.

*The actual scale setup depends on the current settings for `xaxs` and `yaxs`. With the default settings, the scales are $(-0.04, 1.04)$.

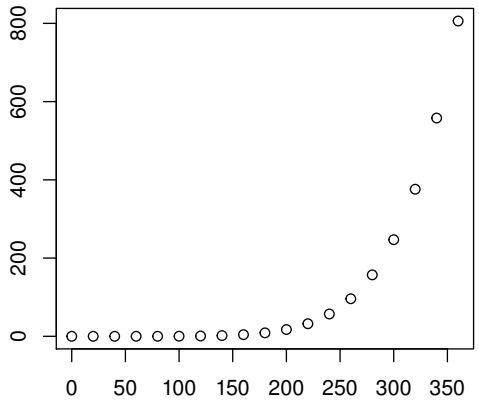


Figure 3.28
A simple scatterplot of vapor pressure of mercury as a function of temperature. This is similar to Figure 1.1, but where that figure was generated with a single call to the `plot()` function, this plot is produced from scratch using low-level plotting functions.

```
> plot.new()
> plot.window(range(pressure$temperature),
              range(pressure$pressure))
> plot.xy(pressure, type="p")
> box()
> axis(1)
> axis(2)
```

The call to `plot.new()` starts a new, completely blank, plot and the call to `plot.window()` sets the scales on the axes to fit the range of the data to be plotted. At this point, there is still nothing drawn. The `plot.xy()` function draws data symbols (`type="p"`) at the data locations, then `box()` draws a rectangle around the plot region, and `axis()` is used to draw the axes.

The output could be produced by the simple expression `plot(pressure)`, but this code shows that the steps in building a plot are available as separate functions as well, which allows the user to have fine control over the construction of a plot.

3.5.2 A more complex plot from scratch

This section describes a slightly more complex example of creating a plot from scratch. The final goal is represented in Figure 3.29 and the steps involved are described below.

The first chunk of code prepares some data to plot. These are the counts of (adult) male and female survivors of the sinking of the *Titanic*.

```
> groups <- dimnames(Titanic)[[1]]
> males <- Titanic[, 1, 2, 2]
> females <- Titanic[, 2, 2, 2]

> males

 1st  2nd  3rd Crew
  57   14   75  192

> females

 1st  2nd  3rd Crew
 140   80   76   20
```

There are several ways that the plot could be created, the main idea being that it fundamentally consists of just a collection of graphical primitives that have been arranged in a meaningful way.

For this example, the approach will be to create a single plot. The labels to the left of the plot will be drawn in the margins of the plot, but everything else will be drawn inside the plot region. This next bit of code sets up the figure margins so that there is enough room for the labels in the left margin, but all other margins are nice and small (to avoid lots of empty space around the plot).

```
> par(mar=c(0.5, 3, 0.5, 1))
```

Inside the plot region there are six different rows of output to draw: the four main pairs of bars, the x-axis, and the legend at the bottom. The axis will be drawn at a y-location of `0`, the main bars at the y-locations `1:4`, and the legend at `-1`. The following code starts the plot and sets up the appropriate y-scale and x-scale.

```
> plot.new()
> plot.window(xlim=c(-200, 200), ylim=c(-1.5, 4.5))
```

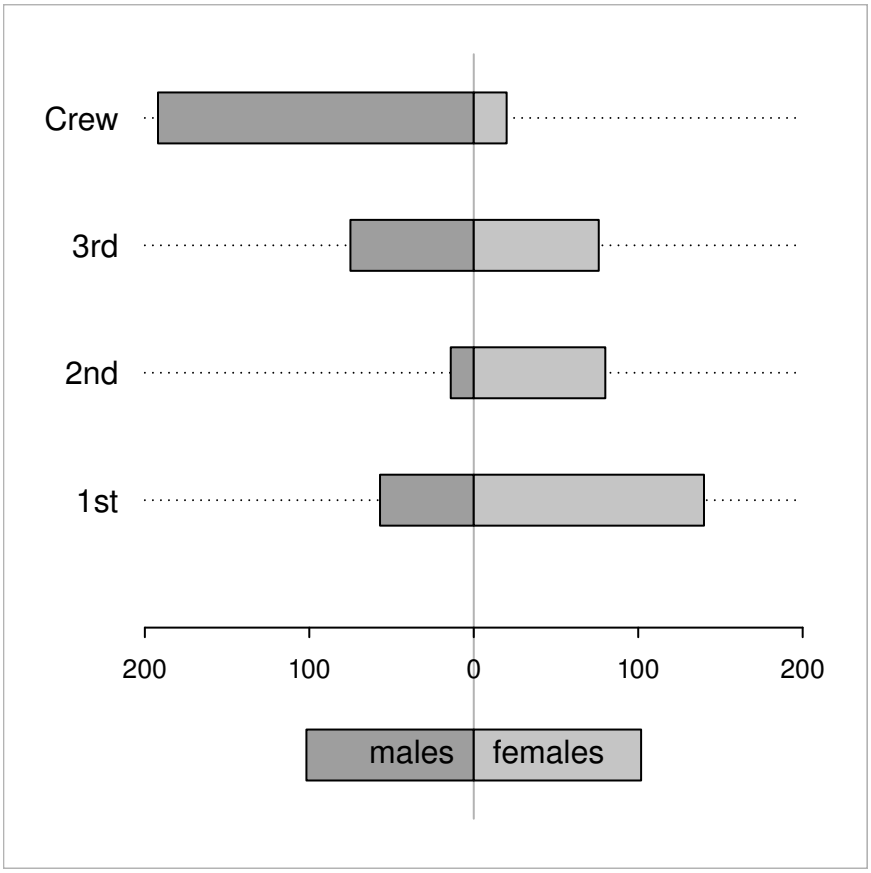


Figure 3.29
A back-to-back barplot from scratch. This demonstrates the use of lower-level plotting functions to produce a novel plot that cannot be produced by an existing high-level function.

This next bit of code assigns some useful values to variables, including the x-locations of tick marks on the x-axis, the y-locations of the main bars, and a value representing half the height of the bars.

```
> ticks <- seq(-200, 200, 100)
> y <- 1:4
> h <- 0.2
```

Now some drawing can occur. This next code draws the main part of the plot. Everything is drawn using calls to the low-level functions such as `lines()`, `segments()`, `mtext()`, and `axis()`. In particular, the main bars are just rectangles produced using `rect()`. Notice that the x-axis is drawn within the plot region (`pos=0`).

```
> lines(rep(0, 2), c(-1.5, 4.5), col="gray")
> segments(-200, y, 200, y, lty="dotted")
> rect(-males, y-h, 0, y+h, col="dark gray")
> rect(0, y-h, females, y+h, col="light gray")
> mtext(groups, at=y, adj=1, side=2, las=2)
> par(cex.axis=0.8, mex=0.5)
> axis(1, at=ticks, labels=abs(ticks), pos=0)
```

The final step is to produce the legend at the bottom of the plot. Again, this is just a series of calls to low-level functions, although the bars are sized using `strwidth()` to ensure that they contain the labels.

```
> tw <- 1.5*strwidth("females")
> rect(-tw, -1-h, 0, -1+h, col="dark gray")
> rect(0, -1-h, tw, -1+h, col="light gray")
> text(0, -1, "males", pos=2)
> text(0, -1, "females", pos=4)
```

This example is particularly customized to the data set involved. It could be made much more general by replacing some constants with variable values (e.g., instead of using `4` because there are four groups in the data set, the code could have a variable `numGroups`). If more than one such plot needs to be made, it makes good sense to also wrap the code within a function. That task is discussed in the next section.

3.5.3 Writing base graphics functions

Having made the effort to construct a plot from scratch, the next step is to encapsulate the calls within a new function and possibly even make it available

for others to use. This section briefly describes some of the things to consider when creating a new graphics function built on the base graphics system.

There are many advantages to developing new graphics functions in the **grid** graphics system (see Part II) rather than using base graphics. See Chapter 8 for a more complete discussion of the issues involved in developing new graphics functions.

Helper functions

There are some helper functions that do no drawing, but are used by the predefined high-level plots to do some of the work in setting up a plot.

The `xy.coords()` function is useful for allowing `x` and `y` arguments to your new function to be flexibly specified (just like the `plot()` function where `y` can be left unspecified and `x` can be a `data.frame`, and so on). This function takes `x` and `y` arguments and creates a standard object containing x-values, y-values, and sensible labels for the axes. There is also an `xyz.coords()` function for plots of three variables.

If your plotting function generates multiple subplots, the `n2mfrow()` function may be helpful to generate a sensible number of rows and columns of plots, based on the total number of plots to fit on a page.

Another set of useful helper functions are those that calculate values to plot from the raw data (but do no actual drawing). Examples of these sorts of functions are: `boxplot.stats()` used by `boxplot()` to generate five-number summaries; `contourLines()` used by `contour()` to generate contour lines; `nclass.Sturges()`, `nclass.scott()`, and `nclass.FD()` used by `hist()` to generate the number of intervals for a histogram; and `co.intervals()` used by `coplot()` to generate ranges of values for conditioning a data set into panels.

Some high-level functions invisibly return this sort of information too. For example, `boxplot()` returns the combined results from `boxplot.stats()` for all of the boxplots that it produces, and `hist()` returns information on the intervals that it creates including the number of data values in each interval. The `hist()` function is also useful (with `plot=FALSE`) simply to perform binning of continuous data.

Argument lists

A common technique when writing a base graphics function is to provide an ellipsis argument (`...`) instead of individual graphics state arguments (such as `col` and `lty`). This allows users to specify any state settings (e.g., `col="red"`

and `lty="dashed"`) and the new function can pass them straight on to the base graphics functions that the new function calls. This avoids having to specify all individual state settings as arguments to the new function. Some care must be taken with this technique because sometimes different graphics functions interpret the same graphics state setting in different ways (the `col` setting is a good example; see Section 3.2). In such cases, it becomes necessary to name the individual graphics state setting as an argument and explicitly pass it on only to other graphics calls that will accept it and respond to it in the desired manner.

Sometimes it is useful for a graphics function to deliberately override the current graphics state settings. For example, a new plot may want to force the `xpd` setting to be `NA` in order to draw lines and text outside of the plot region. In such cases, it is polite for the graphics function to revert the graphics state settings at the end of the function so that users do not get a nasty surprise! A standard technique is to put the following expressions at the start of the new function to restore the graphics state to the settings that existed before the function was called.

```
opar <- par(no.readonly=TRUE)
on.exit(par(opar))
```

Because some of the base graphics state settings interact with each other, such a wholesale save-and-replace approach is actually unlikely to return the graphics state to exactly what it was before, so an even better solution is to save and restore only those parameters that the function modifies.

Care should be taken to ensure that a new graphics function takes notice of appropriate graphics state settings (e.g., `ann`). This can be a little complicated to implement because it is necessary to be aware of the possibility that the user might specify a setting in the call to the function and that such a setting should override the main graphics state setting. The standard approach is to name the state setting explicitly as an argument to the graphics function and provide the permanent state setting as a default value. See the new graphics function template below for an example of this technique using the `ann` argument. An additional complication is that now there is a state setting that will not be part of the `...` argument, so the state setting must be explicitly passed on to any other functions that might make use of it.

Another good technique is to provide arguments that users are used to seeing in other graphics functions — the `main`, `sub`, `xlim`, and `ylim` arguments are good examples of this sort of thing — and a new graphics function should be able to handle missing and non-finite values. The functions `is.na()`, `is.finite()`, and `na.omit()` may be useful for this purpose.

Plot methods

If a new function is for use with a particular type of data, then it is convenient for users if the function is provided as a method for the generic `plot()` function. This allows users to simply call the new function by calling `plot(x)`, where `x` is an object of the relevant class.

A graphics function template

The code in Figure 3.30 is a simple shell that combines some of the basic guidelines from this section. This is just a simplified version of the default `plot()` method. It is far from complete and will not gracefully accept all possible inputs (especially via the `...` argument), but it could be used as the starting template for writing a new base graphics function.

3.6 Interactive graphics

The strength of the base graphics system lies in the production of static graphics and that is the focus of this book. However, for completeness, this section briefly mentions the limited facilities for interacting with base graphics output.

The `locator()` function allows the user to click within a plot and returns the coordinates where the mouse click occurred. It will also optionally draw data symbols at the clicked locations or draw lines between the clicked locations.

The `identify()` function can be used to add labels to data symbols on a plot. The data point closest to the mouse click gets labeled.

There is also a more general-purpose mechanism for defining interactions with the output in a graphics window (though at the time of writing only for the Windows, X Window, and Cairo graphics devices; see Chapter 9). The `setGraphicsEventHandlers()` function can be used to define R functions that will be called whenever events such as keystrokes or mouse clicks occur within the graphics window and the `getGraphicsEvent()` function can be called to start listening to events within the graphics window. This provides a more flexible basis for developing simple interactive base graphicsplots.

```
plot.newclass <- function(x, y=NULL,
                          main="", sub="",
                          xlim=NULL, ylim=NULL,
                          axes=TRUE, ann=par("ann"),
                          col=par("col"),
                          ...) {
    xy <- xy.coords(x, y)
    if (is.null(xlim))
        xlim <- range(xy$x[is.finite(xy$x)])
    if (is.null(ylim))
        ylim <- range(xy$y[is.finite(xy$y)])
    opar <- par(no.readonly=TRUE)
    on.exit(par(opar))
    plot.new()
    plot.window(xlim, ylim, ...)
    points(xy$x, xy$y, col=col, ...)
    if (axes) {
        axis(1)
        axis(2)
        box()
    }
    if (ann)
        title(main=main, sub=sub,
              xlab=xy$xlab, ylab=xy$ylab, ...)
}
```

Figure 3.30
A graphics function template. This code provides a starting point for producing a new graphics function for others to use.

Chapter summary

High-level base graphics functions produce complete plots, and low-level base graphics functions add output to existing plots. There are low-level functions for producing simple output such as lines, rectangles, text, and polygons and also functions for producing more complex output such as axes and legends.

The base graphics system creates several regions for drawing the various components of a plot: a plot region for drawing data symbols and lines, figure margins for axes and labels, and so on. Each low-level graphics function produces output in a particular drawing region and most work in the plot region.

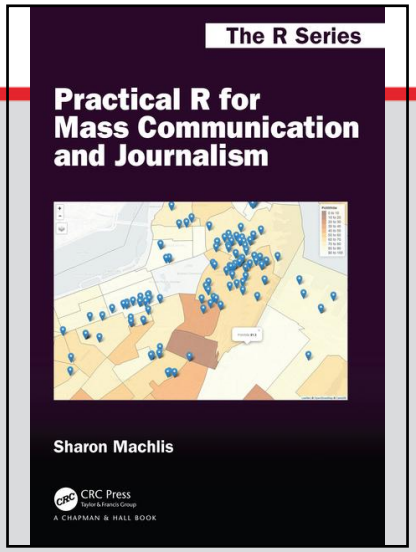
There is a base graphics system state that consists of settings to control the appearance of output and the arrangement of the drawing regions. There are settings for controlling color, fonts, line styles, data symbol style, and the style of axes. There are several mechanisms for arranging multiple plots on a single page.

It is straightforward to create a complete plot using only low-level graphics functions. This makes it possible to produce a completely new type of plot. It is also possible for the user to define an entirely new graphics function.

CHAPTER

3

SEE HOW MUCH YOU CAN DO IN A FEW LINES OF CODE



This chapter is excerpted from

Practical R for Mass Communication and Journalism

by Sharon Machlis.

Learn more

Chapter 3

See How Much You Can Do in a Few Lines of Code

The sole purpose of this chapter is instant gratification. Fundamentals are important, but so is seeing a few of the cool things you can do in R *with very little code.* I do realize that adding 7 + 52 in a console, as I demonstrated last chapter, isn't all that compelling. I want to get you excited about R's potential! But if you're the type of person who gets frustrated doing things without fully understanding them, you may want to skip ahead to Chapter 4 and come back here later. This chapter is designed to show you some eye candy, not give detailed explanations.

3.1 Packages needed this chapter

(I'll include installation code when required)

- quantmod
- dygraphs
- htmlwidgets

3.2 What we'll cover

- Quick interactive graphs with the quantmod and dygraphs packages
- Interactive maps with a few lines of code.

You'll first need to install two packages, which you can do by running this code in your interactive R console (bottom left pane):

```
install.packages(c("quantmod", "dygraphs"))
```

quantmod is a library for financial analysis. dygraphs creates interactive Web graphics of data over time.

A note about installing packages: Usually, this is pretty seamless in R. Occasionally, you might get a message that the package you're trying to install *won't* install, because another package is missing. R *should* automatically install needed packages by default, but sometimes there's a glitch. If you get an error message that some other package you didn't know about is missing, try installing *that* one manually with `install.packages("missingPackageName")` and then run the install on the package you want.

In addition, sometimes you may be asked whether you want to "install from sources the package which needs compilation?" Most of the time, that just means, "Would you like the absolute latest version that doesn't have a handy, single download file yet?" For all packages we're using in this book, choosing n for no is easier and should work just fine.

Once installation is done, load both packages into your current R session with

```
library("quantmod")
library("dygraphs")
```

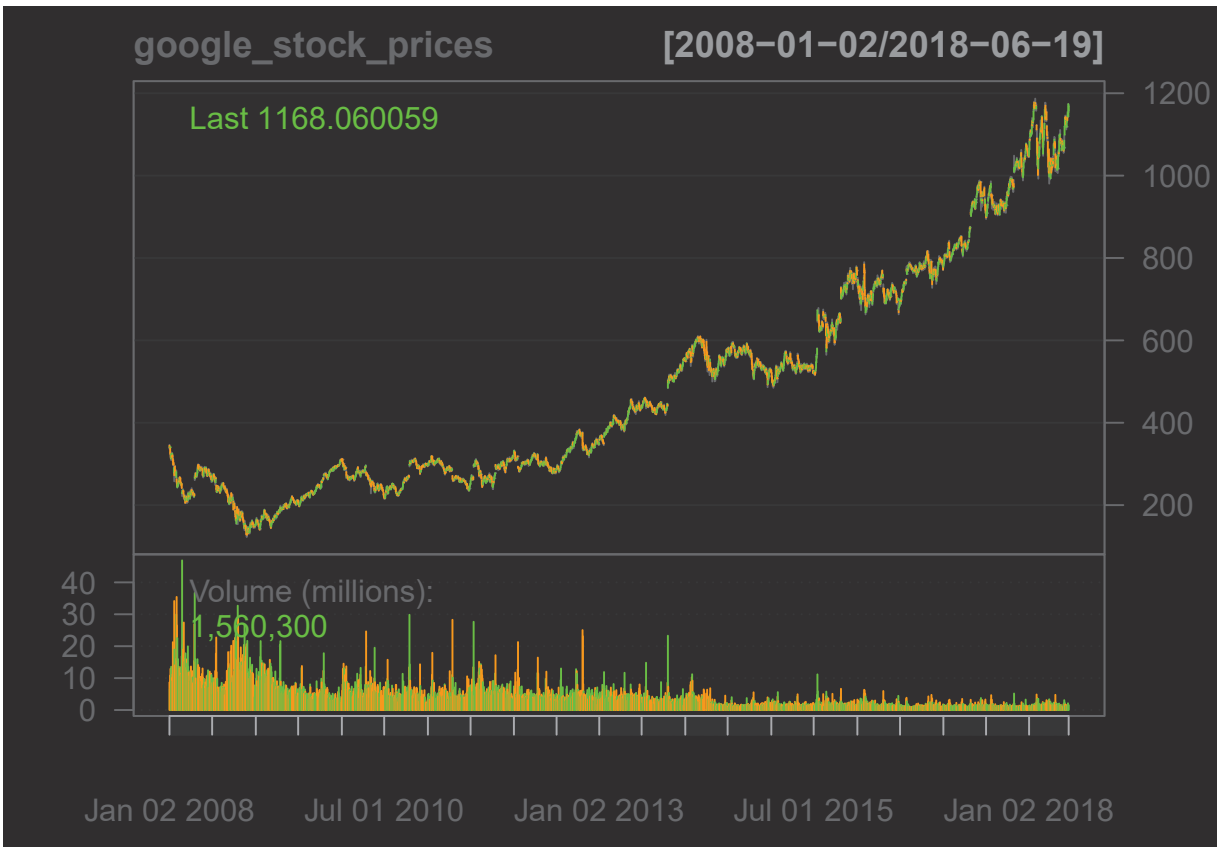
Finally, if you haven't yet set up an RStudio project for your code related to this book, as I described in the last chapter, create one for this work by going to File > New Project.

Now, let's give some those packages a try.

3.3 Simple stock market graphing

How did Google stock do since the 2008 stock market crash? Let's take a look ... with these two lines of code:

```
google_stock_prices <- getSymbols("GOOG", src = "yahoo", from = "2008-01-01", auto.assign = FALSE)
chartSeries(google_stock_prices)
```



Setup notes: You can type each line of code into the console, but it will be more convenient in the long run to set up a script file for this chapter. Go to File > New File > R Script (or use the keyboard shortcut

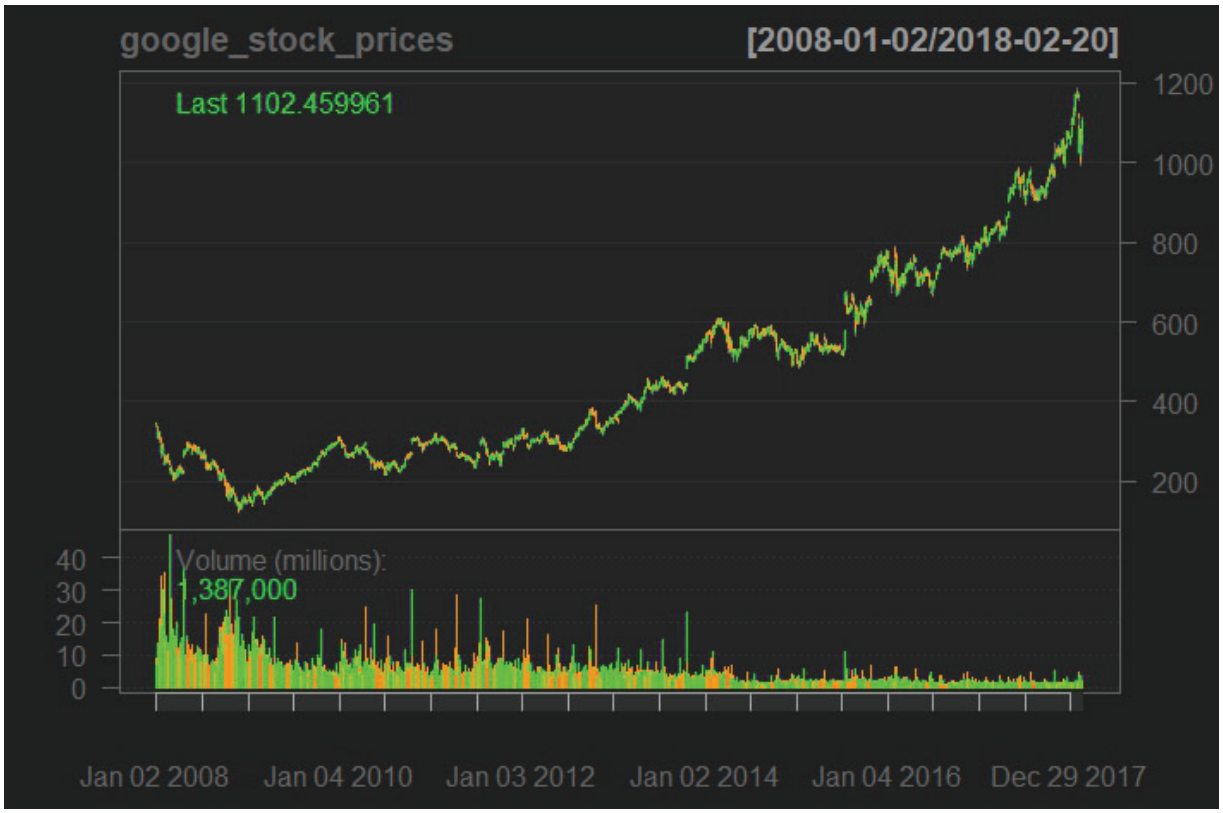

google_stock_prices
[2008-01-02/2018-02-20]
Last 1102.459961
1200
1000
800
600
400
200
Volume (millions):
1,387,000
40
30
20
10
0
Jan 02 2008
Jan 04 2010
Jan 03 2012
Jan 02 2014
Jan 04 2016
Dec 29 2017

Figure 3.1:

Ctrl-Shift-N on Windows / Cmnd-Shift-N on Mac). Type the commands into the script file, save it with the disk save icon, File > Save, or the ctrl/cmnd-S keyboard shortcut.

To run all *the code in the file, you can click the Source link above and to the right of the script panel (which is the top left panel). To run a few lines of code, select them the usual way (click and drag with your mouse) and hit control-Enter (command-Enter on Mac). To run one line of code, put your cursor anywhere on that line and hit control-Enter or command-Enter. When you do that, your cursor will jump to the next line of code and you can hit control/command-Enter again to quickly run the next line of code.*

getSymbols() is quantmod's powerful function for getting historical financial data from the Internet and importing it directly into R. `src = yahoo` tells quantmod that I want to pull the data from Yahoo; I could also have specified google to import from Google Finance. `from` tells quantmod when to start pulling the data. And auto.assign=FALSE deals with a quirk in older versions of quantmod, so it behaves like most R functions and can store functions in an R variable with the `<-` assignment operator.

The dygraphs package can make this graph interactive, so that you can mouse over the line and see underlying data, as well as click and drag to zoom in on a portion of the graph. Again, just one line of code, this one specifying I just want the GOOG.Close column (with closing prices) and a title of "Google Stock Price Starting in 2008"

```
dygraph(google_stock_prices[,c("GOOG.Close")], main = "Google Stock Price Starting in 2008")
```

3.4 Download and graph a city's median income

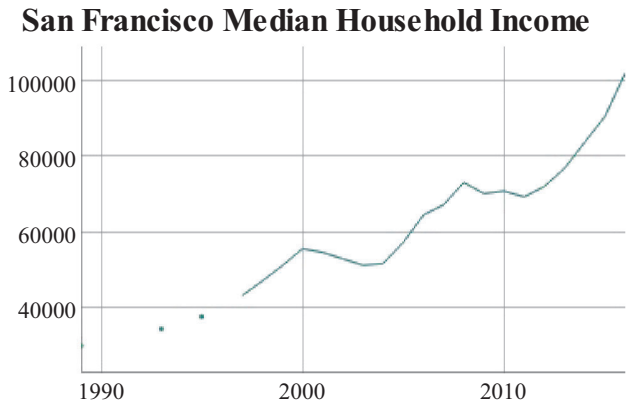
quantmod includes a function that lets you import data directly from the U.S. Federal Reserve – more specifically the St. Louis Federal Reserve's FRED database.

I went to FRED at https://fred.stlouisfed.org/ and searched for "median household income for San Francisco." This wasn't to get the data, but to find out which table contains the data. The URL for FRED's "Estimate of Median Household Income for San Francisco County/City, CA." was https://fred.stlouisfed.org/series/MHICA06075A052NCEN. The character/number string after fred.stlouisfed.org/series/ was what I needed, it's the St. Louis Fed's symbol for this data.

You may want to do the same search, so you can copy the MHICA06075A052NCEN portion of it into your clipboard from the FRED url instead of typing it manually.

Now try running the following code (I'll explain it in a bit) for some instant R gratification (gRatification?). Being able to paste MHICA06075A052NCEN from your clipboard should make this less onerous.

```
sfincome <- getSymbols("MHICA06075A052NCEN", src="FRED", auto.assign = FALSE)
names(sfincome) <- "Income"
dygraph(sfincome, main = "San Francisco Median Household Income")
```



You should see a graph that looks like this in the Viewer tab of RStudio's lower right pane.

The zoom button above the graph on the left will open the pane into a larger window. The icon showing an arrow pointing to the upper right will open the graph in a browser window (in the image above, it's the icon directly over sco in San Francisco).

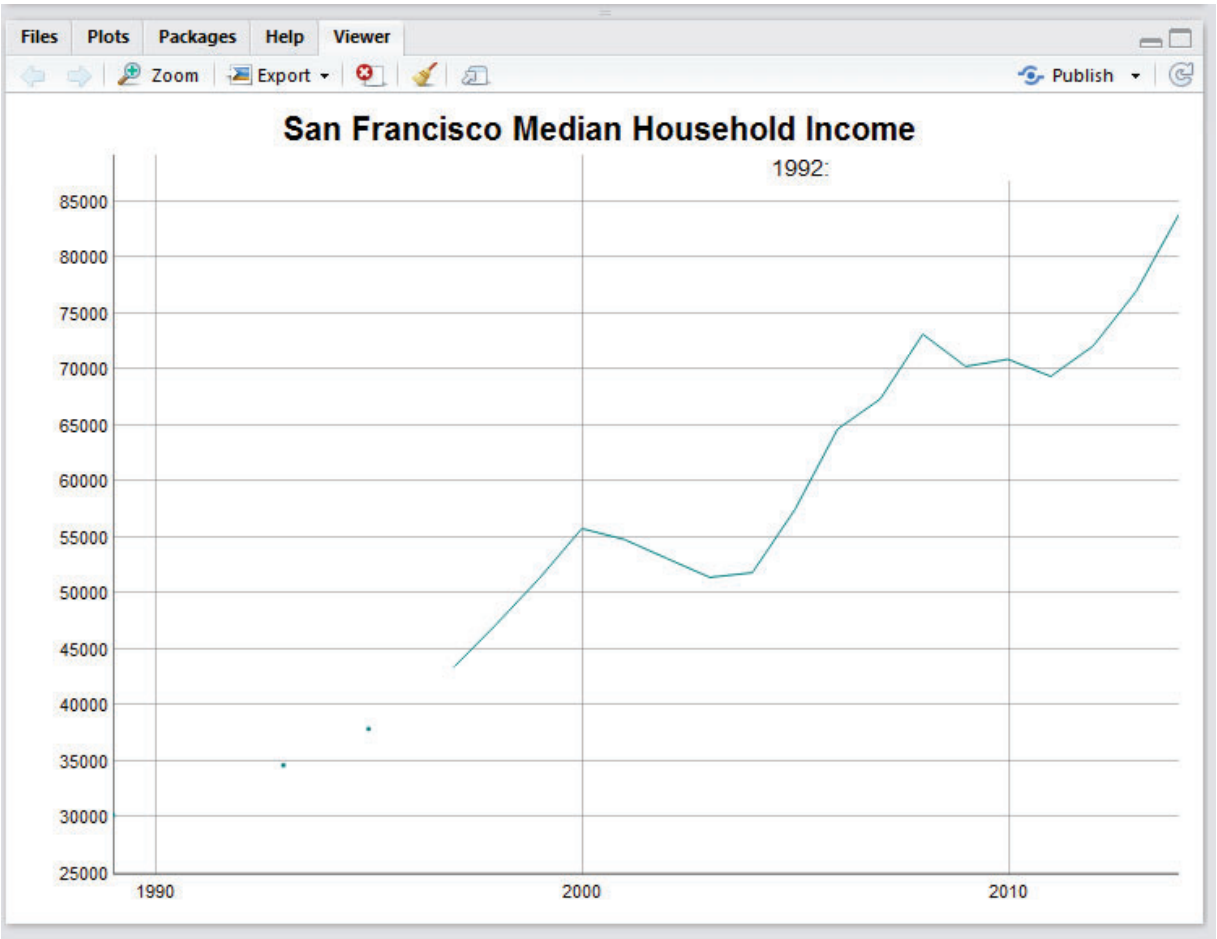

Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
San Francisco Median Household Income
1992:
85000
80000
75000
70000
65000
60000
55000
50000
45000
40000
35000
30000
25000
1990
2000
2010

Figure 3.2:

The graph is interactive. If you move your mouse from left to right anywhere on the graph, you'll see information about the closest underlying data point in the top right of the window. And if you click and drag from left to right inside the graph to define a portion of it - say from 2000 to 2010 - the graph will zoom in. (Double-clicking zooms out again, or you can always refresh the page.) This graph can be saved as an HTML "widget" or exported to a static JPG or PNG image using the RStudio Export menu item in this Viewer pane.

To save this graph as an HTML widget, first store it in an R variable, such as `sfmediangraph <- dygraph(MHICA06075A052NCEN, main = "San Francisco Median Household Income")`. Next, install the htmlwidget package with `install.packages("htmlwidgets")` and load it with `library("htmlwidgets")`. Finally, use the `saveWidget()` function to save your sfmediangraph object to a file, such as

```
saveWidget(sfmediangraph, file="sfmediangraph.html")
```

Find the sfmediangraph.html file on your computer and open it in a browser. You should see the same interactive HTML graph as you saw in RStudio, but now reusable on another Web site.

If by any chance you're thinking "This looks like a Web graphic I could make with JavaScript," you're right. The dygraphs package is an R wrapper for a JavaScript library, also called dygraphs. But with the R package, you can generate dygraphs JavaScript completely in R.

Now, the explanation of the code that I promised:

The first two lines load the quantmod and dygraphs packages into your current working session.

The third line uses quantmod's getSymbols() function to pull data from an external source. The function takes two "arguments" – options that the function needs to do its job. That first argument, "MHICA06075A052NCEN", is the symbol (the one we looked up at FRED) for retrieving data we want. The second argument, src="FRED", sets the data source to FRED, the St. Louis Federal Reserve's database. And you saw "autoassign = FALSE" in the previous section.

If you check the structure of this sfincome object with str(), you'll see that it's a special type of R object called `xts` (for extensible time series). Data starts in 1989, and those NA data listings you see stand for "Not Available," meaning that some points are missing (which you probably noticed on the graph).

```
str(sfincome)
```

```
## An 'xts' object on 1989-01-01/2016-01-01 containing:
##   Data: int [1:28, 1] 30166 NA NA NA 34623 NA 37854 NA 43405 47239 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr "Income"
##   Indexed by objects of class: [Date] TZ: UTC
##   xts Attributes:
## List of 2
##  $ src    : chr "FRED"
##  $ updated: POSIXct[1:1], format: "2018-06-20 20:19:32"
```

The name of the lone data column with the median-income data is "MHICA06075A052NCEN". The data column title shows up on the graph, and I don't like that character string as my data label. So, in the fourth line, I change the name of that data column to "Income" with `names(sfincome) <- "Income"`. You can also run names() on an R object *without* the assignment operator, such as `names(sfincome)`, to see existing names.

(If you're wondering why no name for the *date* column shows up, that's how R time series are structured.)

The fifth line creates the graph, using the dygraph() function from the dygraphs package. The first argument, sfdata, tells dygraph what data set to graph. The second argument, main, is just the headline for the graph.

3.5 So many packages!

At this point, you might want to ask me: How did you know about these packages? How did you know what functions to use, and how they work? And, as you learn about different packages, how do you remember which ones to use when?

All good questions (and ones I'm often asked at workshops). I try to keep up with package developments by looking at top tweets with the #rstats Twitter hashtag, scanning a number of R blogs, and following posts on the Google Plus R group and Reddit R subreddit. This may be easier for me than a lot of other R users because 1) I cover developments in data analysis for my day job as a tech journalist, and 2) I'm somewhat R-obsessed. If you don't follow R developments for *your* job, a good shortcut to keep up with the latest R developments is the community-sourced R Weekly, which tries to round up the most interesting R news in a fairly easy-to-scan post at rweekly.org. You can also follow my tweets with the #rstats hashtag from my @sharon000 account.

When I learn about a very useful package, I add it to an interactive, searchable table published by Computerworld, which is available at http://bit.ly/Rpackages. You might want to keep your own spreadsheet of favorite packages and functions, whether or not it's published for others to read. I'd suggest keeping it somewhere in the cloud even if it's not public, like in a Google sheet or Excel spreadsheet on OneDrive, so you can access it from different computers.

There's another way to store some of your favorite functions right in RStudio, called code snippets. I'll be covering them in Chapter 6.

After you discover a package, reading the introductory vignette can help you figure out how to use it. Also, even if a package is on CRAN, the code may be on GitHub as well, and package authors will often add useful information there. In fact, there's an extremely helpful tutorial on the dygraph package at https://rstudio.github.io/dygraphs/. If you Google "R" and a package name you may come across other useful content (when I wrote this, the RStudio dygraphs tutorial on GitHub came up first when Googling R dygraphs).

3.6 Running functions without loading packages

Returning to the project at hand, I'd like to show you a slightly different way to generate the same graph:

```
sfincome <- quantmod::getSymbols("MHICA06075A052NCEN", src="FRED", auto.assign = FALSE)
names(sfincome) <- "Income"
dygraphs::dygraph(sfincome, main = "San Francisco Median Household Income")
```

In this code above, I use `quantmod::getSymbols()` instead of just `getSymbols()` and `dygraphs::dygraph()` instead of `dygraph()`. By adding `packagename::` before a function, I'm able to access a function from an external package _without having to load the package into memory first with library(packageName). This syntax `PackageName::FunctionName` works for any package that *exists on your computer's hard drive but isn't loaded into memory.* There can be a couple of advantages to this. First, if the package is large and you're only using one function once, you can save system memory by not loading all the package's functions into your session. (However, if you're using several of a package's functions multiple times throughout a script, it's often easier to just load it into memory.)

Second, when you look at a code snippet months later, it's clear which package each function belongs to.

Finally, if you've got multiple external packages loaded into memory, it's possible that the author of package1 named a function the same thing as the author of package2. Using package1::myfun() lets R know you want the myfun() function in package1, and not any other function named myfun() from some other package. We'll encounter that exact situation with two different functions called `describe()`.

3.7 Comparing one city's data to the US median

Whether you're a journalist or government staffer looking at a trend line like this, it's helpful to keep a key statistical question in mind: *compared to what?* A 6% increase in a city's median household income over several years might be impressive if overall national income stayed flat in the same period, but could be viewed as sluggish if the US household median rose 10% over that same period.

So let's add national median income to the graph. I looked up US income at FRED, and it's "MHIUS00000A052NCEN". Get the national data with

```
usincome <- getSymbols("MHIUS00000A052NCEN", src="FRED")
names(usincome) <- "US Income"
```

and then add the national data columns to the San Francisco data set with base R's `cbind()` function. cbind means "bind" two data sets together by adding one data column to another, side by side. You can also `rbind()` to add rows from one data set below another.

```
mygraphdata <- cbind(sfincome, usincome)
```

```
## Warning in merge.xts(..., all = all, fill = fill, suffixes = suffixes): NAs
## introduced by coercion
```

Now, use the `dygraph()` function to graph the mygraphdata data set the same way you created the first graph:

```
dygraph(mygraphdata, main = "Median Household Income")
```

One of the best things about scripting this: When new data is available from the Fed, you can just run the same code and you'll have an updated graph! Now, imagine how useful this is if you work with data that updates monthly or weekly.

Another advantage: Once you've got the basic code for pulling data from the Fed, it's easy to change to another data point such as unemployment. The code for US unemployment data on FRED is "UNRATE" and San Francisco is "SANF806UR", so you can just swap those in for "MHICA06075A052NCEN" and "MHIUS00000A052NCEN" and get your data set:

```
sfunemp <- getSymbols("SANF806UR", src="FRED", auto.assign = FALSE)
names(sfunemp) <- "SFRate"
usunemp <- getSymbols("UNRATE", src="FRED", auto.assign = FALSE)
names(usunemp) <- "USRate"
unemploymentdata <- cbind(sfunemp, usunemp)
dygraph(unemploymentdata, main = "Monthly Unemployment Rates, US and San Francisco")
```

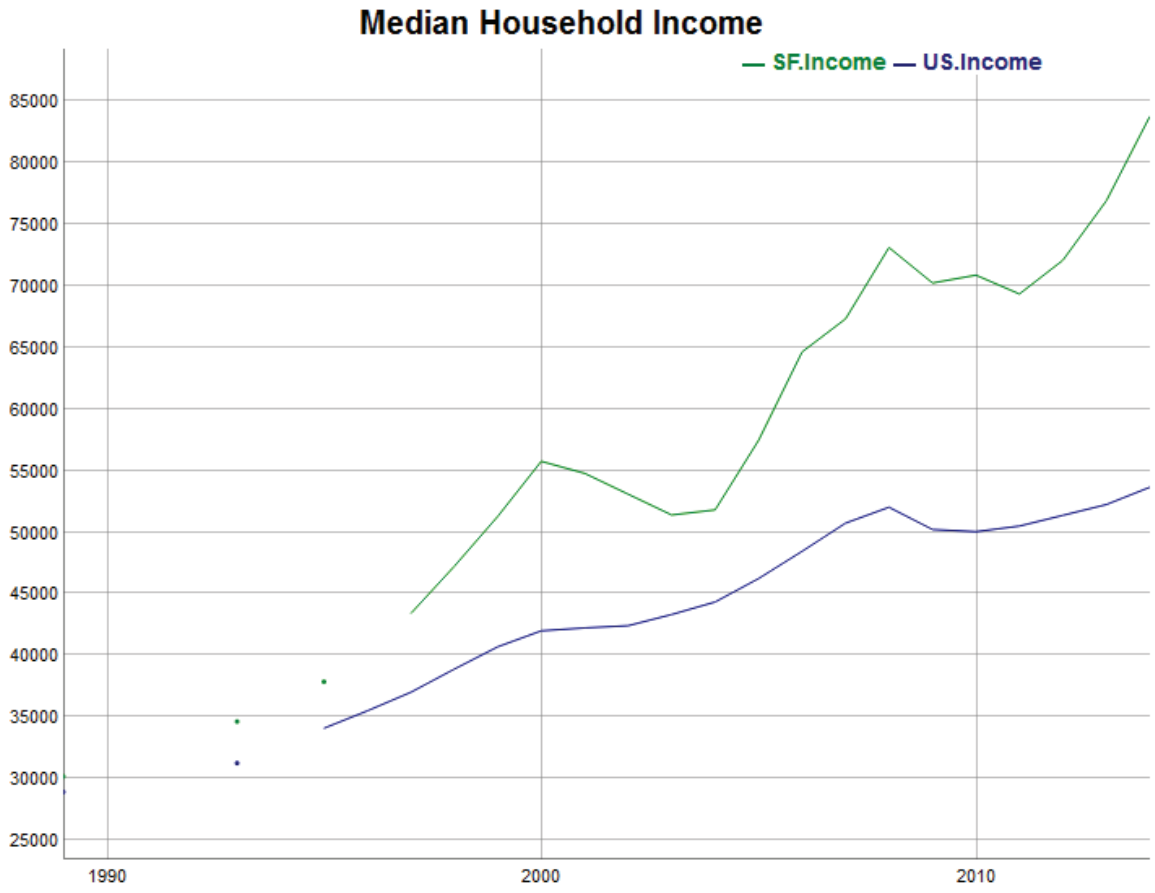
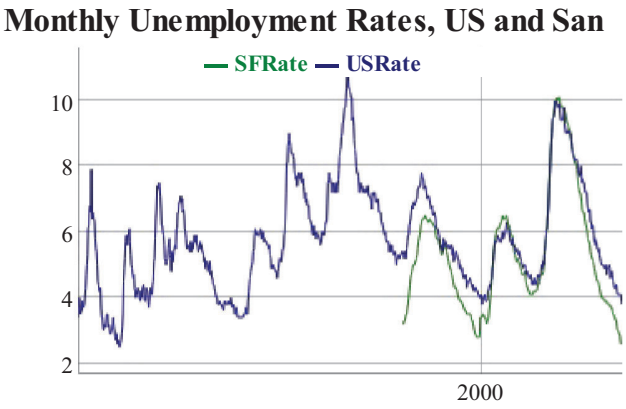


Figure 3.3: Graph of San Francisco and US median household income



Because San Francisco unemployment data is only available since 1990, it could be better to just show all data from 1990 onward instead of displaying earlier national data. Time series have their own unusual way of subsetting data; this code

```
unemploymentdata <- unemploymentdata["1990/"]
```

will update the unemploymentdata variable so it contains only information from 1990 and later.

Now you can draw the graph with:

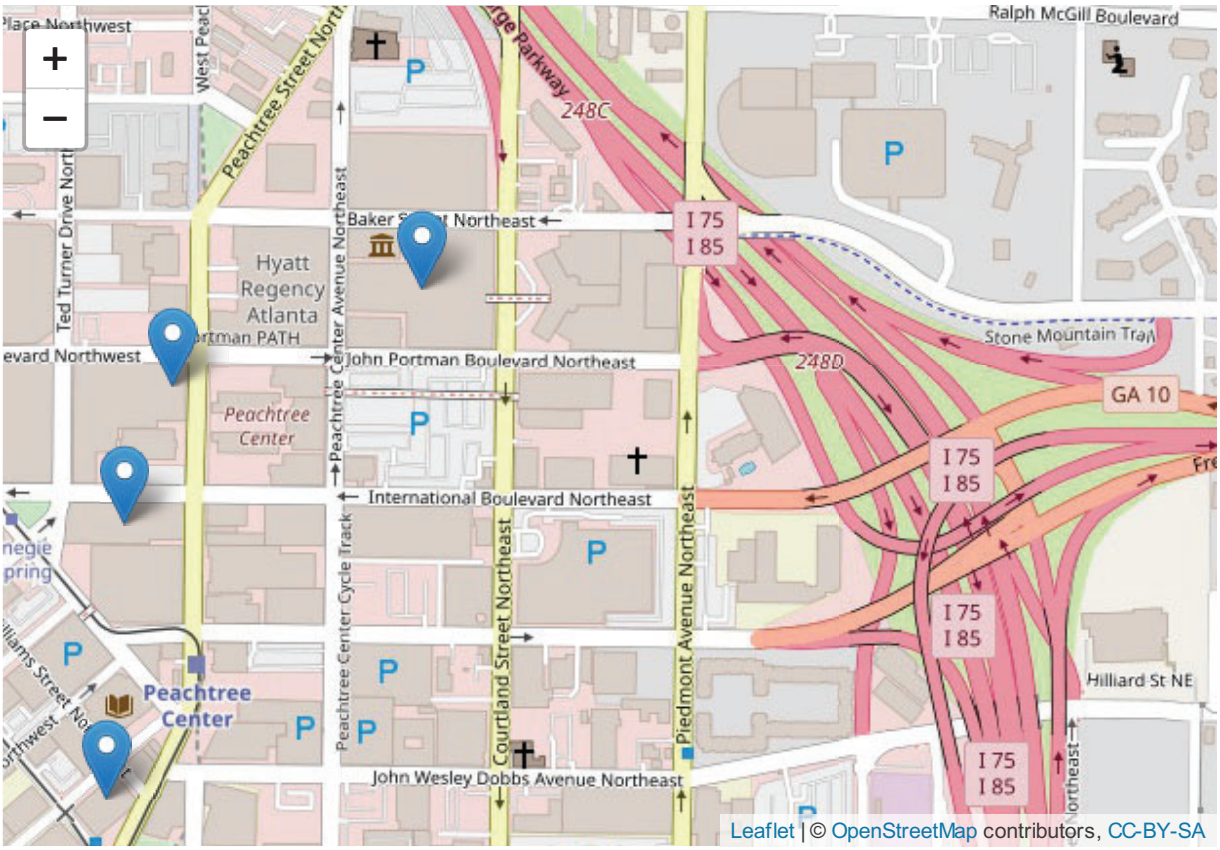
```
dygraph(unemploymentdata, main = "Monthly Unemployment Rates, US and San Francisco")
```

3.8 Run a remote script to make an interactive map

I've posted most of the data files from this book in a public repository on GitHub. Soon we'll create a new RStudio project for all these files. For now, though, I'd like you to run one file from that "repo" at https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_map.R . If that's somewhat onerous to type, go to the bit.ly link http://bit.ly/R4JournalismLinks and find and copy the URL under the *Chapter 3* heading.

Then run

```
source("https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_map.R")
make_mymap()
```



and see what happens. Eventually, you should see an interactive map in your bottom-right RStudio Viewer pane, showing a few Starbucks in Atlanta. This is a national map, though – you can zoom out and see other areas of the U.S.

Before you use this to find a coffee shop near you, though, be warned that the latest available data file for Starbucks locations is from 2012. The point is to demonstrate making a cool interactive map with pop-ups, and I've found that the Starbucks data set tends to resonate in a crowd of journalists.

I also wanted to show you how to include code from an external file in your scripts. `source("myfile")` lets you run code from an external file in your current script, whether that file is somewhere else on your own system or a file that you've stored remotely.

3.9 Bonus map: Mapping income data

We'll be doing a lot more work with maps later. Meanwhile, though, if you'd like a sneak peek, try running the following remote code. Note that it will be loading several packages on your system the first time it executes as well as a data file, so it may take a little while to execute.

You can find and copy that lengthy URL by heading to http://bit.ly/R4JournalismLinks and copying the second URL under the Chapter 3 heading.

```
source("https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_manhattan_income_map.R")
get_household_income_in_county()
```

Click on different portions of the map to get pop-ups with data details.

3.10 Wrap-Up

I hope that's gotten you enthused about some of the power of R. This chapter covered a lot of code somewhat quickly, but the point was to see that R can do some cool things, not to learn and understand every nuance of programming presented here.

Here's what's worth remembering from this chapter:

Load an entire R package into your working session's memory with `library(mypackage)`.

You can use a function from an external package *without* loading the entire package into memory, using the syntax `mypackage::myfunction()`.

Run R code from an external file with `source("myfile.R")`.

Combine data frames by column with `cbind()` (putting them together side by side) and by row with `rbind()` (putting them together one below the other). We'll cover more sophisticated merging by common columns in later chapters.

`names(myobject)` will display an object's existing names. You can also *change* the names of one or more items in an R object with names(myobject), such as names(mydataframe) <- c("Column1", "Column2").

NA stands for Not Available and indicates missing data in R.

There are some very elegant ways of importing and visualizing data with R.

Next up: How to import all sorts of data into R.

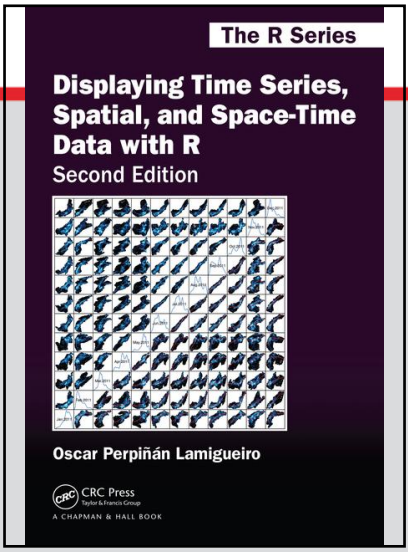
3.11 Additional resources

- **5 data visualizations in 5 minutes:** Each in 5 lines or less of R http://bit.ly/5LinesOrLess. This is a version of my 5-minute lightning talk at the 2015 National Institute for Computer Assisted Reporting conference. With video.
- **htmlwidgets for R** htmlwidgets.org. Find out more about interactive HTML for R including dozens of packages and a showcase of examples.

CHAPTER

4

TIME AS A COMPLEMENTARY VARIABLE



This chapter is excerpted from

Displaying Time Series, Spatial, and Space-Time Data with R

by Oscar Perpinan Lamigueiro.

Learn more

Chapter 5

Time as a Complementary Variable

In this chapter, time will be used as a complementary variable which adds information to a graph where several variables are confronted. We will illustrate this approach with the evolution of the relationship between Gross National Income (GNI) and carbon dioxide (CO_2) emissions for a set of countries extracted from the database of the World Bank Open Data. We will try several solutions to display the relationship between CO_2 emissions and GNI over the years using time as a complementary variable.
Along this chapter these subjects are covered: qualitative palettes, visual discrimination, panel functions, label positioning, small multiples, class intervals, and animation. Last section is devoted to interactive graphics.
The most relevant packages in this chapter are: `zoo` for reading and arranging data as time series; `reshape2` for converting data from the wide format to the long format; `RColorBrewer` for defining color palettes; `directlabels` for label positioning; `classInt` for computing class intervals; `plotly`, `googleVis`, and `gridSVG` for interactive graphics.

5.1 Polylines

Our first approach is to display the entire data in a panel with a scatterplot using country names as the grouping factor. Points of each country are connected with polylines to reveal the time evolution (Figure 5.1).

```
library(zoo)

load('data/CO2.RData')
```

```
## lattice version
xyplot(GNI.capita ~ CO2.capita, data = CO2data,
      xlab = "Carbon dioxide emissions (metric tons per capita)",
      ylab = "GNI per capita, PPP (current international $)",
      groups = Country.Name, type = 'b')
```

```
## ggplot2 version
ggplot(data = CO2data, aes(x = CO2.capita, y = GNI.capita,
                     color = Country.Name)) +
   xlab("Carbon dioxide emissions (metric tons per capita)") +
   ylab("GNI per capita, PPP (current international $)") +
   geom_point() + geom_path() + theme_bw()
```

Three improvements can be added to this graphical result:

1. Define a better palette to enhance visual discrimination between countries.
2. Display time information with labels to show year values.
3. Label each polyline with the country name instead of a legend.

5.1.1 Choosing Colors

The `Country.Name` categorical variable will be encoded with a qualitative palette, namely the first five colors of `Set1` palette[1] from the `RColorBrewer` package (Neuwirth 2014). Because there are more countries than colors, we have to repeat some colors to complete the number of levels of the variable `Country.Name`. The result is a palette with non-unique colors, and thus some countries will share the same color. This is not a problem because the curves will be labeled, and countries with the same color will be displayed at enough distance.

[1] http://colorbrewer2.org/

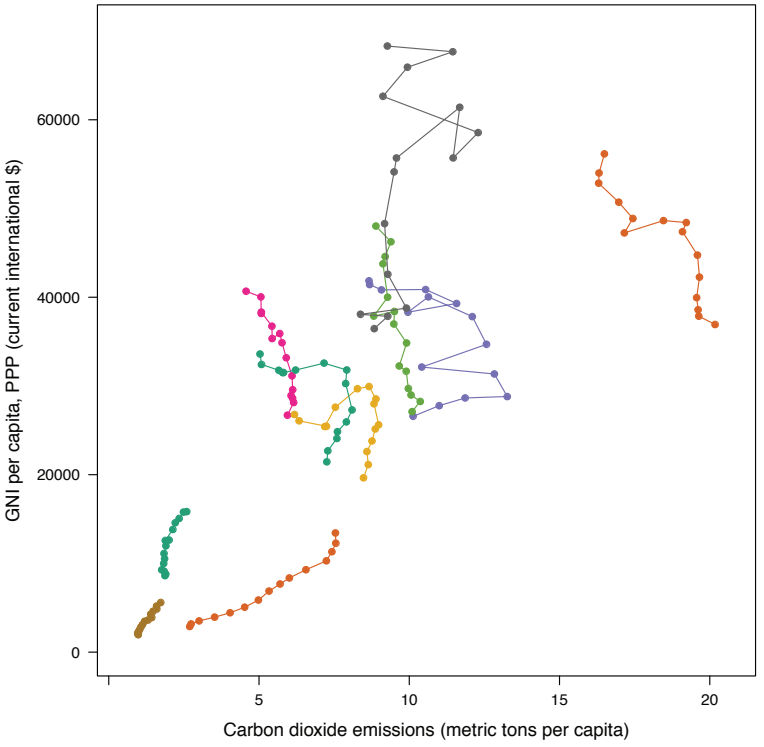


FIGURE 5.1: GNI per capita versus CO_2 emissions per capita (`lattice` version).

```
library(RColorBrewer)

nCountries <- nlevels(CO2data$Country.Name)
pal <- brewer.pal(n = 5, 'Set1')
pal <- rep(pal, length = nCountries)
```

Adjacent colors of this palette are chosen to be easily distinguishable. Therefore, the connection between colors and countries must be in such a way that nearby lines are encoded with adjacent colors of the palette.

A simple approach is to calculate the annual average of the variable to be represented along the x-axis (`CO2.capita`), and extract colors from the palette according to the order of this value.

```
## Rank of average values of CO2 per capita
CO2mean <- aggregate(CO2.capita ~ Country.Name,
                     data = CO2data, FUN = mean)
palOrdered <- pal[rank(CO2mean$CO2.capita)]
```

A more sophisticated solution is to use the ordered results of a hierarchical clustering of the time evolution of the CO_2 per capita values (Figure 5.2). The data is extracted from the original CO_2 `data.frame`.

```
library(reshape2)

CO2capita <- CO2data[, c('Country.Name',
                         'Year',
                         'CO2.capita')]
CO2capita <- dcast(CO2capita, Country.Name ~ Year)

summary(CO2capita)
```

```
Using CO2.capita as value column: use value.var to override.
 Country.Name 2000 2001 2002
Brazil :1 Min. : 0.9799 Min. : 0.9717 Min. : 0.9674
China :1 1st Qu.: 3.5093 1st Qu.: 3.5949 1st Qu.: 3.7725
Finland:1 Median : 7.8681 Median : 7.9634 Median : 7.9849
France :1 Mean : 7.6468 Mean : 7.7977 Mean : 7.7982
Germany:1 3rd Qu.: 9.7802 3rd Qu.:10.0960 3rd Qu.: 9.6920
Greece :1 Max. :20.1788 Max. :19.6365 Max. :19.6134
(Other):4
    2003 2004 2005 2006
Min. : 0.9924 Min. : 1.025 Min. : 1.069 Min. : 1.122
1st Qu.: 4.1721 1st Qu.: 4.559 1st Qu.: 4.917 1st Qu.: 5.212
Median : 8.1860 Median : 8.388 Median : 8.539 Median : 8.356
```

```
 Mean : 8.1468 Mean : 8.146 Mean : 7.948 Mean : 8.163
 3rd Qu.: 9.9536 3rd Qu.: 9.746 3rd Qu.: 9.545 3rd Qu.: 9.808
 Max. :19.5641 Max. :19.658 Max. :19.592 Max. :19.094

     2007 2008 2009 2010
 Min. : 1.193 Min. : 1.310 Min. : 1.432 Min. : 1.397
 1st Qu.: 5.443 1st Qu.: 5.693 1st Qu.: 5.581 1st Qu.: 5.526
 Median : 8.407 Median : 7.914 Median : 7.247 Median : 7.050
 Mean : 8.139 Mean : 8.082 Mean : 7.665 Mean : 7.947
 3rd Qu.: 9.553 3rd Qu.:10.354 3rd Qu.: 9.671 3rd Qu.:11.001
 Max. :19.218 Max. :18.462 Max. :17.158 Max. :17.442

     2011 2012 2013 2014
 Min. : 1.477 Min. : 1.598 Min. : 1.591 Min. : 1.730
 1st Qu.: 5.255 1st Qu.: 5.222 1st Qu.: 5.068 1st Qu.: 4.688
 Median : 7.216 Median : 7.336 Median : 6.947 Median : 6.862
 Mean : 7.475 Mean : 7.387 Mean : 7.396 Mean : 7.097
 3rd Qu.: 9.125 3rd Qu.: 9.168 3rd Qu.: 9.213 3rd Qu.: 8.832
 Max. :16.972 Max. :16.304 Max. :16.316 Max. :16.494
```

```
hCO2 <- hclust(dist(CO2capita[, -1]))

oldpar <- par(mar = c(0, 2, 0, 0) + .1)
plot(hCO2, labels = CO2capita$Country.Name,
    xlab = '', ylab = '', sub = '', main = '')
par(oldpar)
```

The colors of the palette are assigned to each country with `match`, which returns a vector of the positions of the matches of the country names in alphabetical order in the country names ordered according to the hierarchical clustering.

```
idx <- match(levels(CO2data$Country.Name),
          CO2capita$Country.Name[hCO2$order])
palOrdered <- pal[idx]
```

It must be highlighted that this palette links colors with the levels of `Country.Name` (country names in alphabetical order), which is exactly what the `groups` argument provides. The following code produces a curve for each country using different colors to distinguish them.

```
## simpleTheme encapsulates the palette in a new theme for xyplot
myTheme <- simpleTheme(pch = 19, cex = 0.6, col = palOrdered)
```

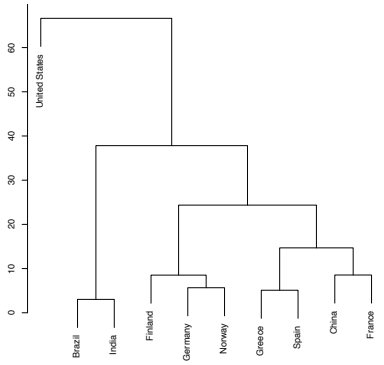


FIGURE 5.2: Hierarchical clustering of the time evolution of CO_2 per capita values.

```
## lattice version
pCO2.capita <- xyplot(GNI.capita ~ CO2.capita,
                  data = CO2data,
                  xlab = "Carbon dioxide emissions (metric tons
                      per capita)",
                  ylab = "GNI per capita, PPP (current
                      international $)",
                  groups = Country.Name,
                  par.settings = myTheme,
                  type = 'b')
```

```
## ggplot2 version
gCO2.capita <- ggplot(data = CO2data,
                  aes(x = CO2.capita,
                      y = GNI.capita,
                      color = Country.Name)) +
   geom_point() + geom_path() +
   scale_color_manual(values = palOrdered, guide = FALSE) +
   xlab('CO2 emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

5.1.2 Labels to Show Time Information

This result can be improved with labels displaying the years to show the time evolution. A panel function with `panel.text` to print the year labels and `panel.superpose` to display the lines for each group is a solution. In the panel function, `subscripts` is a vector with the integer indices representing the rows of the `data.frame` to be displayed in the panel.

```
xyplot(GNI.capita ~ CO2.capita,
      data = CO2data,
      xlab = "Carbon dioxide emissions (metric tons per capita)",
      ylab = "GNI per capita, PPP (current international $)",
      groups = Country.Name,
      par.settings = myTheme,
      type = 'b',
      panel = function(x, y, ..., subscripts, groups){
         panel.text(x, y, ...,
                  labels = CO2data$Year[subscripts],
                  pos = 2, cex = 0.5, col = 'gray')
         panel.superpose(x, y, subscripts, groups,...)
      })
```

The same result with a clearer code is obtained with the combination of `+.trellis`, `glayer_` and `panel.text`. Using `glayer_` instead of `glayer`, we ensure that the labels are printed below the lines.

```
## lattice version
pCO2.capita <- pCO2.capita +
   glayer_(panel.text(...,
                   labels = CO2data$Year[subscripts],
                     pos = 2, cex = 0.5, col = 'gray'))
```

```
## ggplot2 version
gCO2.capita <- gCO2.capita + geom_text(aes(label = Year),
                                  colour = 'gray',
                                  size = 2.5,
                                  hjust = 0, vjust = 0)
```

5.1.3 Country Names: Positioning Labels

The common solution to link each curve with the group value is to add a legend. However, a legend can be confusing with too many items. In addition, the reader must carry out a complex task: Choose the line, memorize its color, search for it in the legend, and read the country name.

A better approach is to label each line using nearby text with the same color encoding. A suitable method is to place the labels avoiding the overlapping between labels and lines. The package `directlabels` (Hocking 2017) includes a wide repertory of positioning methods to cope with overlapping. The main function, `direct.label`, is able to determine a suitable method for each plot, although the user can choose a different method from the collection or even define a custom method. For the `pCO2.capita` object, the best results are obtained with `extreme.grid` (Figure 5.3).

```
library(directlabels)

## lattice version
direct.label(pCO2.capita,
             method = 'extreme.grid')
```

```
## ggplot2 version
direct.label(gCO2.capita, method = 'extreme.grid')
```

5.2 A Panel for Each Year

Time can be used as a conditioning variable (as shown in previous sections) to display subsets of the data in different panels. Figure 5.4 is produced with the same code as in Figure 5.1, now including `|factor(Year)` in the lattice version and `facet_wrap(~ Year)` in the ggplot2 version.

```
## lattice version
xyplot(GNI.capita ~ CO2.capita | factor(Year),
       data = CO2data,
       xlab = "Carbon dioxide emissions (metric tons per capita)",
       ylab = "GNI per capita, PPP (current international $)",
       groups = Country.Name, type = 'b',
       auto.key = list(space = 'right'))
```

```
## ggplot2 version
ggplot(data = CO2data,
       aes(x = CO2.capita,
           y = GNI.capita,
           colour = Country.Name)) +
   facet_wrap(~ Year) + geom_point(pch = 19) +
   xlab('CO2 emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

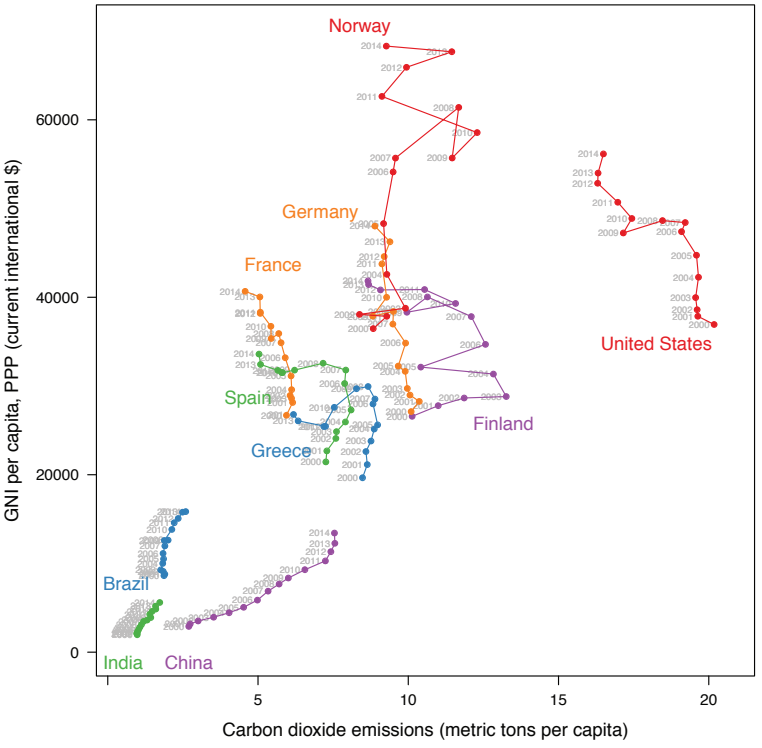


FIGURE 5.3: CO_2 emissions versus GNI per capita. Labels are placed with the `extreme.grid` method of the `directlabels` package.

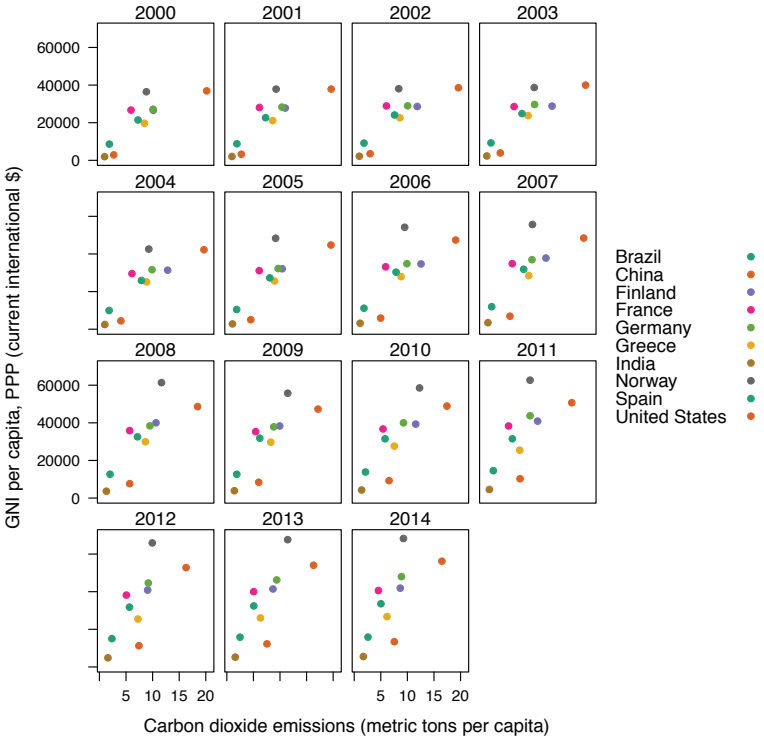


FIGURE 5.4: CO_2 emissions versus GNI per capita with a panel for each year.

Because the grouping variable, `Country.Name`, has many levels, the legend is not very useful. Once again, point labeling is recommended (Figure 5.5).

```
## lattice version
xyplot(GNI.capita ~ CO2.capita | factor(Year),
      data = CO2data,
      xlab = "Carbon dioxide emissions (metric tons per capita)",
      ylab = "GNI per capita, PPP (current international $)",
      groups = Country.Name, type = 'b',
      par.settings = myTheme) +
   glayer(panel.pointLabel(x, y,
                           labels = group.value,
                           col = palOrdered[group.number],
                           cex = 0.7))
```

5.2.1 Using Variable Size to Encode an Additional Variable

Instead of using simple points, we can display circles of different radius to encode a new variable. This new variable is `CO2.PPP`, the ratio of CO_2 emissions to the Gross Domestic Product with purchasing power parity (PPP) estimations.

To use this numeric variable as an additional grouping factor, its range must be divided into different classes. The typical solution is to use `cut` to coerce the numeric variable into a `factor` whose levels correspond to uniform intervals, which could be unrelated to the data distribution. The `classInt` package (Bivand 2017) provides several methods to partition data into classes based on natural groups in the data distribution.

```
library(classInt)
z <- CO2data$CO2.PPP
intervals <- classIntervals(z, n = 4, style = 'fisher')
```

Although the functions of this package are mainly intended to create color palettes for maps, the results can also be associated to point sizes. `cex.key` defines the sequence of sizes (to be displayed in the legend) associated with each `CO2.PPP` using the `findCols` function.

```
nInt <- length(intervals$brks) - 1
cex.key <- seq(0.5, 1.8, length = nInt)

idx <- findCols(intervals)
CO2data$cexPoints <- cex.key[idx]
```

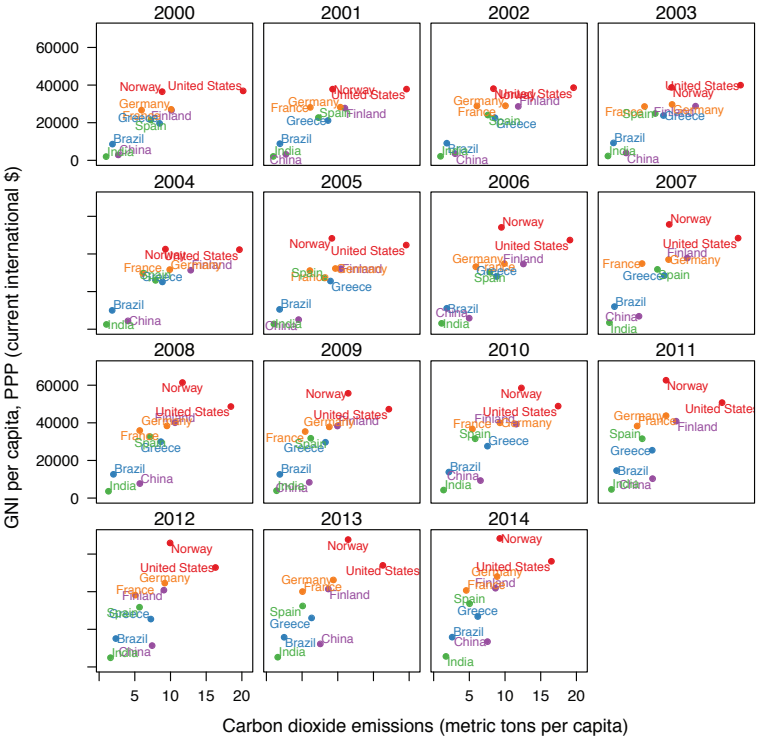


FIGURE 5.5: CO_2 emissions versus GNI per capita with a panel for each year.

The graphic will display information on two variables (`GNI.capita` and `CO2.capita` in the vertical and horizontal axes, respectively) with a conditioning variable (`Year`) and two grouping variables (`Country.Name`, and `CO2.PPP` through `cexPoints`) (Figure 5.6).

```
ggplot(data = CO2data,
      aes(x = CO2.capita,
          y = GNI.capita,
          colour = Country.Name)) +
   facet_wrap(~ Year) +
   geom_point(aes(size = cexPoints), pch = 19) +
   xlab('Carbon dioxide emissions (metric tons per capita)') +
   ylab('GNI per capita, PPP (current international $)') +
   theme_bw()
```

The `auto.key` mechanism of the `lattice` version is not able to cope with two grouping variables. Therefore, the legend, whose main componens are the labels (`intervals`) and the point sizes (`cex.key`), should be defined manually (Figure 5.7).

```
op <- options(digits = 2)
tab <- print(intervals)
options(op)
```

```
key <- list(space = 'right',
          title = expression(CO[2]/GNI.PPP),
          cex.title = 1,
          ## Labels of the key are the intervals strings
          text = list(labels = names(tab), cex = 0.85),
          ## Points sizes are defined with cex.key
          points = list(col = 'black',
                        pch = 19,
                        cex = cex.key,
                        alpha = 0.7))
```

```
xyplot(GNI.capita ~ CO2.capita|factor(Year), data = CO2data,
      xlab = "Carbon dioxide emissions (metric tons per capita)",
      ylab = "GNI per capita, PPP (current international $)",
      groups = Country.Name, key = key, alpha = 0.7,
      panel = panel.superpose,
      panel.groups = function(x, y,
          subscripts, group.number, group.value, ...){
          panel.xyplot(x, y,
                       col = palOrdered[group.number],
```

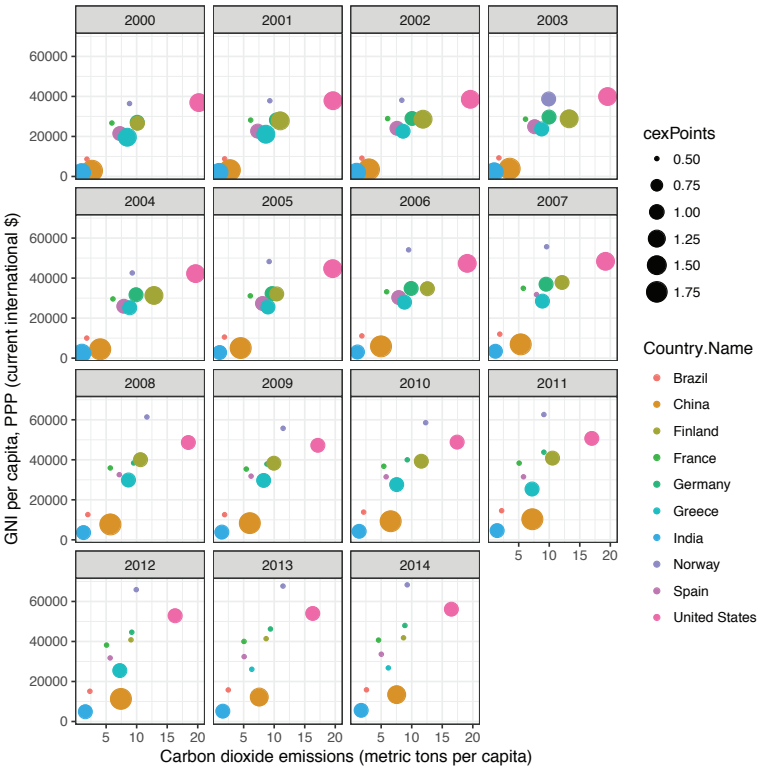


FIGURE 5.6: CO_2 emissions versus GNI per capita for different intervals of the ratio of CO_2 emissions to the GDP PPP estimations.

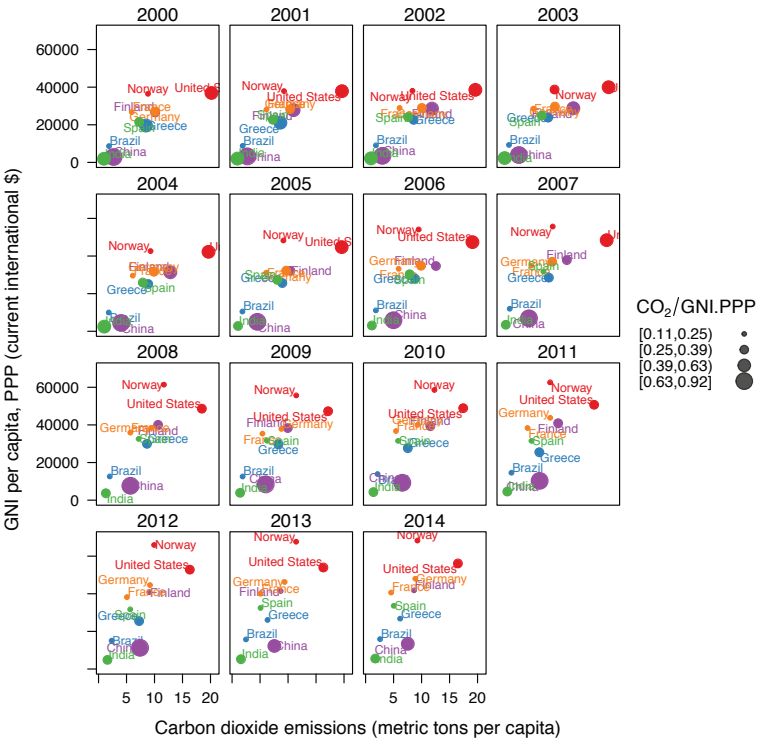


FIGURE 5.7: CO_2 emissions versus GNI per capita for different intervals of the ratio of CO_2 emissions to the GDP PPP estimations.

```
                    cex = CO2data$cexPoints[subscripts])
        panel.pointLabel(x, y, labels = group.value,
                         col = palOrdered[group.number],
                         cex = 0.7)
}
)
```

5.3 Interactive Graphics: Animation

Previous sections have been focused on static graphics. This section describes several solutions to display the data through animation with interactive functionalities.

Gapminder [2] is an independent foundation based in Stockholm, Sweden. Its mission is "to debunk devastating myths about the world by offering free access to a fact-based world view." They provide free online tools, data, and videos "to better understand the changing world." The initial development of Gapminder was the Trendalyzer software, used by Hans Rosling in several sequences of his documentary "The Joy of Stats."

The information visualization technique used by Trendalyzer is an interactive bubble chart. By default it shows five variables: two numeric variables on the vertical and horizontal axes, bubble size and color, and a time variable that may be manipulated with a slider. The software uses brushing and linking techniques for displaying the numeric value of a highlighted country.

This software was acquired by Google in 2007, and is now available as a Motion Chart gadget and as the Public Data Explorer.

We will mimic the Trendalyzer/Motion Chart solution, using traveling bubbles of different colors and with radius proportional to the values of the variable `CO2.PPP`. Previously, you should watch the magistral video "200 Countries, 200 Years, 4 Minutes"[3].

Three packages are used here: `googleVis`, `plotly`, and `gridSVG`.

5.3.1 plotly

The `plotly` package has already been used in Section 3.4.3 to create an interactive graphic representing time in the x-axis. In this section this package produces an animation piping the result of the `plot_ly` and `add_markers` functions through the `animation_slider` function.

Variables `CO2.capita` and `GNI.capita` are represented in the x-axis and y-axis, respectively.

```
library(plotly)
p <- plot_ly(CO2data,
             x = ~CO2.capita,
             y = ~GNI.capita)
```

[2] http://www.gapminder.org/

[3] http://www.gapminder.org/videos/200-years-that-changed-the-world-bbc/

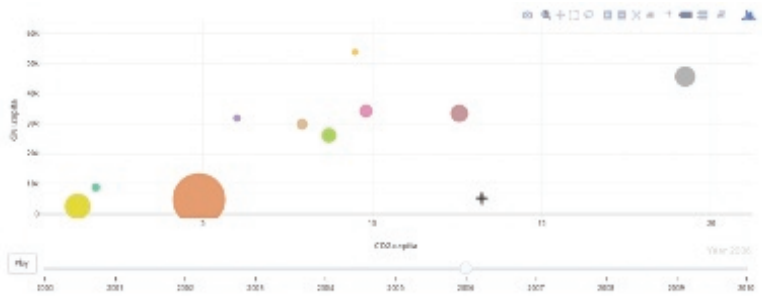


FIGURE 5.8: Snapshot of a Motion Chart produced with `plotly`.

`CO2.PPP` is encoded with the circle sizes, while `Country.Name` is represented with colours and labels.

```
p <- add_markers(p,
                 size = ~CO2.PPP,
                 color = ~Country.Name,
                 text = ~Country.Name, hoverinfo = "text",
                 ids = ~Country.Name,
                 frame = ~Year,
                 showlegend = FALSE)
```

Finally, animation is created with `animation_opts`, to customize the frame and transition times, and with `animation_slider` to define the slider. Figure 5.8 is an snapshot of this animation.

```
p <- animation_opts(p,
                    frame = 1000,
                    transition = 800,
                    redraw = FALSE)

p <- animation_slider(p,
                      currentvalue = list(prefix = "Year "))

p
```

5.3.2 `googleVis`

The `googleVis` package (Gesmann and Castillo 2011) is an interface between R and the Google Visualisation API. With its `gvisMotionChart` func-

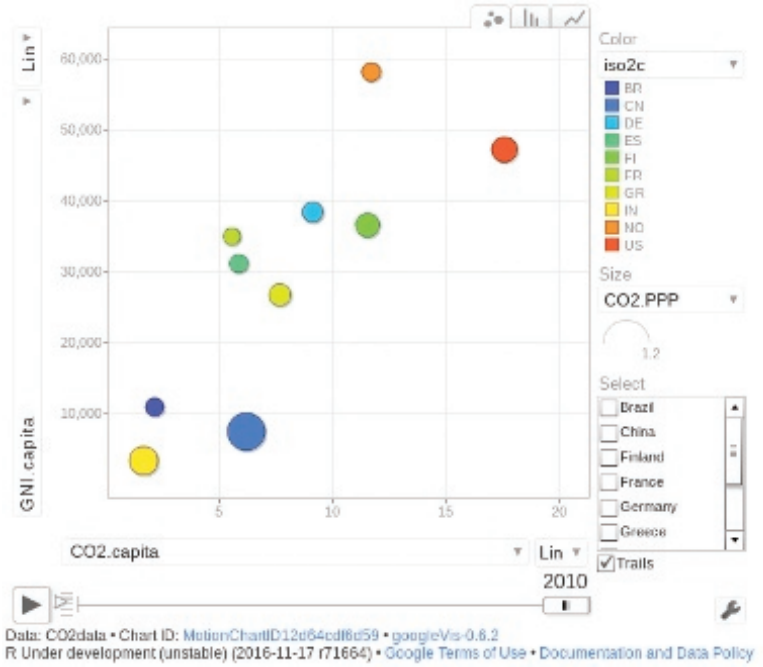


FIGURE 5.9: Snapshot of a Motion Chart produced with googleVis.

tion it is easy to produce a Motion Chart that can be displayed using a browser with Flash enabled (Figure 5.9).

```
library(googleVis)

pgvis <- gvisMotionChart(CO2data,
                         xvar = 'CO2.capita',
                         yvar = 'GNI.capita',
                         sizevar = 'CO2.PPP',
                         idvar = 'Country.Name',
                         timevar = 'Year')
```

Although the `gvisMotionChart` is quite easy to use, the global appearance and behavior are completely determined by Google API[4]. Moreover,

[4]You should read the Google API Terms of Service before using `googleVis`: https://developers.google.com/terms/

you should carefully read their Terms of Use before using it for public distribution.

5.3.3 gridSVG

The final solution to create an animation is based on the function `grid.animate` of the `gridSVG` package.

The first step is to draw the initial state of the bubbles. Their colors are again defined by the `palOrdered` palette (line 15), although the `adjustcolor` function is used for a ligther `fill` color. Because there will not be a legend, there is no need to define class intervals, and thus the radius is directly proportional to the value of `CO2data$CO2.PPP` (line 16).

```
1  library(gridSVG)
2  library(grid)
3
4  xyplot(GNI.capita ~ CO2.capita,
5        data = CO2data,
6        xlab = "Carbon dioxide emissions (metric tons per capita)",
7        ylab = "GNI per capita, PPP (current international $)",
8        subset = Year==2000,
9        groups = Country.Name,
10       ## The limits of the graphic are defined
11       ## with the entire dataset
12       xlim = extendrange(CO2data$CO2.capita),
13       ylim = extendrange(CO2data$GNI.capita),
14       panel = function(x, y, ..., subscripts, groups) {
15          color <- palOrdered[groups[subscripts]]
16          radius <- CO2data$CO2.PPP[subscripts]
17          ## Size of labels
18          cex <- 1.1*sqrt(radius)
19          ## Bubbles
20          grid.circle(x, y, default.units = "native",
21                      r = radius*unit(.25, "inch"),
22                      name = trellis.grobname("points", type = "panel
                            "),
23                      gp = gpar(col = color,
24                                ## Fill color ligther than border
25                                fill = adjustcolor(color, alpha = .5),
26                                lwd = 2))
27          ## Country labels
28          grid.text(label = groups[subscripts],
29                  x = unit(x, 'native'),
```

```
30                  ## Labels above each bubble
31                  y = unit(y, 'native') + 1.5 * radius *unit(.25,
                        'inch'),
32                  name = trellis.grobname('labels', type = 'panel'
                        ),
33                  gp = gpar(col = color, cex = cex))
34      })
```

From this initial state, `grid.animate` creates a collection of animated graphical objects with the result of `animUnit` (lines 9, 11, 14 and 18). This function produces a set of values that will be interpreted by `grid.animate` as intermediate states of a feature of the graphical object (lines 21 and 29). Thus, the bubbles will travel across the values defined by `x_points` and `y_points`, while their labels will use `x_points` and `x_labels`.

The use of `rep=TRUE` ensures that the animation will be repeated indefinitely (lines 27 and 34).

```
1  ## Duration in seconds of the animation
2  duration <- 20
3
4  nCountries <- nlevels(CO2data$Country.Name)
5  years <- unique(CO2data$Year)
6  nYears <- length(years)
7
8  ## Intermediate positions of the bubbles
9  x_points <- animUnit(unit(CO2data$CO2.capita, 'native'),
10                  id = rep(seq_len(nCountries), each = nYears))
11 y_points <- animUnit(unit(CO2data$GNI.capita, 'native'),
12                  id = rep(seq_len(nCountries), each = nYears))
13 ## Intermediate positions of the labels
14 y_labels <- animUnit(unit(CO2data$GNI.capita, 'native') +
15                  1.5 * CO2data$CO2.PPP * unit(.25, 'inch'),
16                  id = rep(seq_len(nCountries), each = nYears))
17 ## Intermediate sizes of the bubbles
18 size <- animUnit(CO2data$CO2.PPP * unit(.25, 'inch'),
19               id = rep(seq_len(nCountries), each = nYears))
20
21 grid.animate(trellis.grobname("points", type = "panel",
22                           row = 1, col = 1),
23           duration = duration,
24           x = x_points,
25           y = y_points,
26           r = size,
27           rep = TRUE)
```

```
28
29 grid.animate(trellis.grobname("labels", type = "panel",
30                           row = 1, col = 1),
31            duration = duration,
32            x = x_points,
33            y = y_labels,
34            rep = TRUE)
```

A bit of interactivity can be added with the `grid.hyperlink` function. For example, the following code adds the corresponding Wikipedia link to a mouse click on each bubble.

```
countries <- unique(CO2data$Country.Name)
URL <- paste('http://en.wikipedia.org/wiki/', countries, sep = ''
    )
grid.hyperlink(trellis.grobname('points', type = 'panel', row =
    1, col = 1),
            URL, group = FALSE)
```

Finally, the time information: The year is printed in the lower right corner, using the `visibility` attribute of an animated `textGrob` object to show and hide the values.

```
visibility <- matrix("hidden", nrow = nYears, ncol = nYears)
diag(visibility) <- "visible"
yearText <- animateGrob(garnishGrob(textGrob(years, .9, .15,
                                    name = "year",
                                    gp = gpar(cex = 2, col = "
                                        grey")),
                                visibility = "hidden"),
                    duration = 20,
                    visibility = visibility,
                    rep = TRUE)
grid.draw(yearText)
```

The SVG file produced with `grid.export` is available at the website of the book (Figure 5.10). Because this animation does not trace the paths, Figure 5.3 provides this information as a static complement.

```
grid.export("figs/bubbles.svg")
```

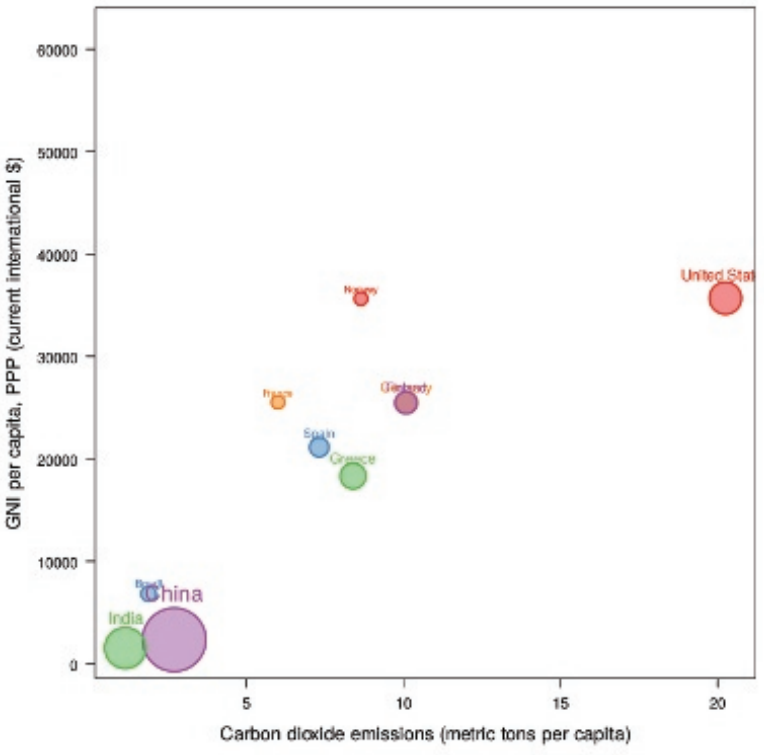
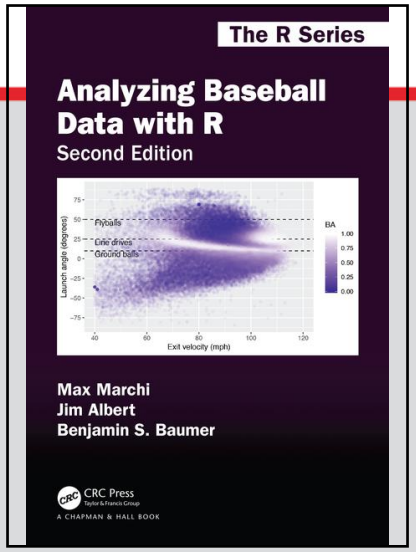


FIGURE 5.10: Animated bubbles produced with `gridSVG`.

CHAPTER

5

BALLS AND STRIKES EFFECTS



This chapter is excerpted from

Analyzing Baseball Data with R

by Max Marchi, Jim Albert, Benjamin S. Baumer.

Learn more

CHAPTER 6

Balls and Strikes Effects

6.1 Introduction

In this chapter we explore the effect of the ball/strike count on the behavior of players and umpires and on the final outcome of a plate appearance. We use Retrosheet data from the 2016 season to estimate how the ball/strike count affects the run expectancy. We also use PITCHf/x data to explore how one pitcher modifies his pitch selection, how one batter alters his swing zone, and how umpires judge pitches based on the count. Along the way, we introduce functions for string manipulation that are useful for managing the pitch sequences from the Retrosheet play-by-play files. Level plots and contour plots, created with the use of the `ggplot2` package, will be used for the explorations of batters' swing tendencies and umpires' strike zones.

6.2 Hitter's Counts and Pitcher's Counts

When watching a broadcast of a baseball game, one often hears an announcer's concern for a pitcher who is repeatedly "falling behind" in the count, or his/her anticipation for a particular pitch because it's a "hitter's count" and the batter has a chance to do some damage. We will see if there is actual evidence that the so-called hitter's count really leads to more favorable outcomes for batters, while "getting ahead" in the count (a pitcher's count) is beneficial for pitchers.

6.2.1 An example for a single pitcher

The Baseball Reference[1] website provides various splits for every player—in particular, it gives splits by ball/strike counts for all seasons since 1988. We find Mike Mussina's split statistics by entering the player's profile page (typing "Mussina" on the search box brings one there), clicking on the "Splits" tab in the "Standard Pitching" table, and clicking on "Career" (or whatever season we are interested in) on the pop-up menu that appears. One finds the "Count

[1] `baseball-reference.com`

Balls/Strikes" table scrolling down on the splits page. Alternatively, the table can be reached by a direct link: in this case the career splits by count for Mike Mussina are currently available at `www.baseball-reference.com/players/split.cgi?id=mussimi01&year=Career&t=p#count`.

The first series of lines (from "First Pitch" to "Full Count") shows the statistics for events happening in that particular count. Thus, for example, a batting average (BA) of .338 on 1-0 counts indicates batters hit safely 34% of the time when putting the ball in play *on a 1-0 count* against Mussina. We are more interested in the second group of rows, those beginning with the word "After". In fact, in these cases the statistics are relative to every plate appearance that *goes through* that count. Thus a .337 on-base percentage (OBP) after 1-0 means that, whenever Mike Mussina started a batter with a ball, the batter successfully got on base 34% of the time, no matter how many pitches it took to end the plate appearance.

The last column on every table in the splits page is "tOPS+". It's an index for comparing the player's OPS in that particular situation (the sum of on base percentage and slugging percentage[2]) to his overall OPS. A value over 100 is interpreted as a higher OPS in the situation compared to his OPS on all counts; conversely, values below 100 indicate a OPS value that is lower than his overall OPS.

Figure 6.1 uses a heat map to display Mussina's tOPS+ through the various counts. If one focuses on a particular number of strikes (moving up vertically), a higher number of balls in the count makes the outcome more likely to be favorable to the hitter (darker shades). Conversely, if one fixes the number of balls (moving right horizontally), the balance moves towards the pitcher (lighter shades) as one increases the number of strikes.

Figure 6.1 emphasizes the importance from a pitcher's perspective of beginning the duel with a strike. When Mussina fell behind 1-0 in his career, batters performed 18% better than usual in the plate appearance; conversely, after a first pitch strike, they were limited to 72% of their potential performance.

How is the heat map display of Figure 6.1 created in R? First a data frame `mussina` is prepared with all the possible balls/strikes counts, using the `expand.grid()` function as previously illustrated in Section 4.5. We then add a new variable `value` with the tOPS+ values taken from the Baseball-Reference website[3].

```
mussina <- expand.grid(balls = 0:3, strikes = 0:2) %>%
  mutate(value = c(100, 118, 157, 207, 72, 82,
                   114, 171, 30, 38, 64, 122))
mussina
```

[2]OPS is widely used as a measure of offensive production because—while being very easy to calculate—it correlates very well with runs scored at the team level.

[3]One might alternatively scrape these data using the `rvest` package

```
   balls strikes value
1      0       0   100
2      1       0   118
3      2       0   157
4      3       0   207
5      0       1    72
6      1       1    82
7      2       1   114
8      3       1   171
9      0       2    30
10     1       2    38
11     2       2    64
12     3       2   122
```

We create a `ggplot2` called `count_plot` by mapping strikes to the `x` aesthetic, balls to the `y` aesthetic, and `fill` color to the value. The tiles that we draw are colored based on this value. We use the `scale_fill_gradient2()` function to set a diverging color palette for the value of tOPS+. Since 100 is a neutral value, we set that to the midpoint and assign that value to the color white. For reasons that will become clear later on, we round the labels displayed in each tile (even though they are integers!).

```
count_plot <- mussina %>%
  ggplot(aes(x = strikes, y = balls, fill = value)) +
  geom_tile() +
  geom_text(aes(label = round(value, 3))) +
  scale_fill_gradient2("tOPS+", low = "grey10", high = crcblue,
                       mid = "white", midpoint = 100)
count_plot
```

6.2.2 Pitch sequences from Retrosheet

From viewing Figure 6.1, we obtain an initial view of hitter's counts (darker shades) and pitcher's counts (lighter shades) on the basis of offensive production. However this figure is based on data for a single pitcher—does a similar pattern emerge when using league-wide data?

Retrosheet provides pitch sequences beginning with the 1988 season. Sequences are stored in strings such as `FBSX`. Each character encodes the description of a pitch. In this example, the pitch sequence is a foul ball (`F`), followed by a ball (`B`), a swinging strike (`S`), and the ball put into play (`X`). Table 6.1 provides the description for every code used in Retrosheet pitch sequences.[4]

[4]Source: `www.retrosheet.org/eventfile.htm`.

TABLE 6.1 Pitch codes used by Retrosheet.

Symbol	description
+	following pickoff throw by the catcher
*	indicates the following pitch was blocked by the catcher
.	marker for play not involving the batter
1	pickoff throw to first
2	pickoff throw to second
3	pickoff throw to third
>	indicates a runner going on the pitch
B	ball
C	called strike
F	foul
H	hit batter
I	intentional ball
K	strike (unknown type)
L	foul bunt
M	missed bunt attempt
N	no pitch (on balks and interference calls)
O	foul tip on bunt
P	pitchout
Q	swinging on pitchout
R	foul ball on pitchout
S	swinging strike
T	foul tip
U	unknown or missed pitch
V	called ball because pitcher went to his mouth
X	ball put into play by batter
Y	ball put into play on pitchout

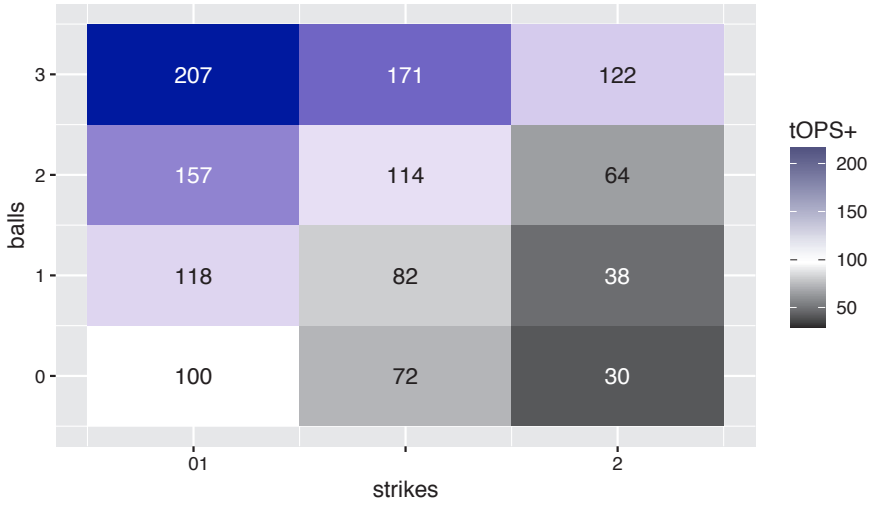


FIGURE 6.1 Heat map of tOPS+ for Mike Mussina through each ball/strike count. Data from Baseball Reference website.

6.2.2.1 *Functions for string manipulation*

Sequence strings from Retrosheet often require some initial processing before they can be suitably analyzed. In this section we provide a quick tutorial on some R functions for the manipulation of strings. Readers not interested in string manipulation functions may skip to Section 6.2.3.

The function `nchar()` returns the number of characters in a string. This function is helpful for obtaining the number of pitches delivered in a Retrosheet pitch sequence. For example, the number of pitches in the sequence `BBSBFFFX` is given by

```
nchar("BBSBFFFX")

[1] 8
```

However, as indicated in Table 6.1, there are some characters in the Retrosheet strings denoting actions that are not pitches, such as pickoff attempts.

The functions `grep()` and `grepl()` are used to find patterns within elements of character vectors. The function `grep()` returns the indices of the elements for which a match is found, and the function `grepl()` returns a logical vector, indicating for each element of the vector whether a match is found. For both functions, the first argument is the string pattern to search and the second argument is the vector of strings where matches are sought. For example, we apply the two functions to the vector of pitch sequences `sequences` to search for pickoff attempts to first base denoted by the code `1`.

```
sequences <- c("BBX", "C11BBC1S", "1X")
grep("1", sequences)

[1] 2 3

grepl("1", sequences)

[1] FALSE  TRUE  TRUE
```

The function `grep()` tells us that "1" is contained in the second (2) and third (3) components of the character vector `sequences`, and `grepl()` outputs this same information by means of a logical vector.

The pattern to search for does not have to be a single character. In fact, it can be any *regular expression*. For example we may want to look for consecutive pickoff attempts to first, which is the pattern "11". The output below shows that "11" is contained in the second component of `sequences`.

```
grepl("11", sequences)

[1] FALSE  TRUE FALSE
```

The function `gsub()` allows for the substitution of the pattern found with a replacement. The replacement can be an empty string, in which case the pattern is simply removed. For example, the following code removes the pickoff attempts to first from the pitch sequences.

```
gsub("1", "", sequences)

[1] "BBX"   "CBBCS" "X"
```

6.2.2.2 *Finding plate appearances going through a given count*

Since we are interested only in pitch counts, we should remove the characters not corresponding to actual pitches from the pitch sequences. Regular expressions are the computing tool needed for this particular task. While it's beyond the scope of this book to fully explain how regular expressions work, we will instead show a few examples on how to use them.[5]

We begin by loading the `all2016.csv` file containing Retrosheet's play-by-play for the 2016 season using the `read_csv()` function.

[5]The website `www.regular-expressions.info` is a comprehensive online resource on regular expressions, featuring examples, tutorials, references for syntax, and a list of related books.

```
headers <- read_csv("data/fields.csv")
pbp2016 <- read_csv("data/all2016.csv",
                    col_names = pull(headers, Header),
                    na = character())
```

We use the `gsub()` function to create the variable `pseq` of pitch sequences after removing the symbols from the Retrosheet pitch sequence variable `PITCH_SEQ_TX` that don't correspond to actual pitches.

```
pbp2016 <- pbp2016 %>%
  mutate(pseq = gsub("[.>123N+*]", "", PITCH_SEQ_TX))
```

In a regular expression, the square brackets indicate the collection of characters to search. The above code removes pickoff attempts at any base (`1`, `2`, `3`) either by the pitcher or the catcher (`+`), balks and interference calls (`N`), plays not involving the batter (`.`), indicators of runners going on the pitch (`>`), and of catchers blocking the pitch (`*`).[6]

We need another special character to identify the plate appearances that go through a 1-0 count. In a regular expression, the ^ character means the pattern has to be matched at the beginning of the string. Looking at Table 6.1 there are four different ways a ball can be coded (`B`, `I`, `P`, `V`). A plate appearance that goes though a 1-0 count must therefore begin with one of these characters. The following code creates the desired variable `c10`.

```
pbp2016 <- pbp2016 %>%
  mutate(c10 = grepl("^[BIPV]", pseq))
```

Similarly, plate appearances going through a 0-1 count must start with a strike. Thus, we create a new variable `c01`.

```
pbp2016 <- pbp2016 %>%
  mutate(c01 = grepl("^[CFKLMOQRST]", pseq))
```

Let's check our work by checking the values of `PITCH_SEQ_TX`, `c10`, and `c01` for the first ten lines of the data frame.

```
pbp2016 %>%
  select(PITCH_SEQ_TX, c10, c01) %>%
  head(10)

# A tibble: 10 x 3
   PITCH_SEQ_TX c10   c01
```

[6]Note that applying the `nchar()` function to the newly created variable `pseq` gives the number of pitches delivered in each at-bat.

```
   <chr>        <lgl> <lgl>
 1 BX           TRUE  FALSE
 2 X            FALSE FALSE
 3 SFS          FALSE TRUE
 4 BCX          TRUE  FALSE
 5 BSS*B1S      TRUE  FALSE
 6 BBX          TRUE  FALSE
 7 BCX          TRUE  FALSE
 8 CX           FALSE TRUE
 9 BCCS         TRUE  FALSE
10 SBFX         FALSE TRUE
```

Writing regular expressions for every pitch count is a tedious task and we will refer the reader to Appendix A for the full code. For the purpose of this chapter, a play-by-play file containing additional information on ball/strike counts is provided.

6.2.3 Expected run value by count

The data frame `pbp16rc` contains an enhanced version of the play-by-play data for the 2016 season. Other than the typical information provided by Retrosheet, this data file reports the change in expected run value for each play as calculated in Chapter 5 and additional variables such as `c00` and `c10`, indicating for each possible ball/strike count whether the at-bat has gone through that particular count. See Section A.3 for more detail on how to compute these new variables.

```
pbp16rc %>%
  select(GAME_ID, EVENT_ID, RUNS.VALUE, c00, c10, c20,
         c11, c01, c30, c21, c31, c02, c12, c22, c32) %>%
  head()

# A tibble: 6 x 15
  GAME_ID EVENT_ID RUNS.VALUE c00   c10   c20   c11   c01
  <chr>      <int>      <dbl> <lgl> <lgl> <lgl> <lgl> <lgl>
1 ANA201~        1      0.634 TRUE  TRUE  FALSE FALSE FALSE
2 ANA201~        2     -0.196 TRUE  FALSE FALSE FALSE FALSE
3 ANA201~        3     -0.565 TRUE  FALSE FALSE FALSE TRUE
4 ANA201~        4      0.849 TRUE  TRUE  FALSE TRUE  FALSE
5 ANA201~        5     -0.220 TRUE  TRUE  FALSE TRUE  FALSE
6 ANA201~        6     -0.230 TRUE  TRUE  TRUE  FALSE FALSE
# ... with 7 more variables: c30 <lgl>, c21 <lgl>, c31 <lgl>,
#   c02 <lgl>, c12 <lgl>, c22 <lgl>, c32 <lgl>
```

For example, the at-bat in the fourth line of the data frame (a two-out

RBI single) started with a 1-0 count (value TRUE in column c10), then moved to the count 1-1, and finally generated a change in expected runs of 0.849. The pbp16rc data frame has all the necessary information to calculate the run values of the various balls/strikes counts, in the same way the value of a home run and of a single were calculated in Chapter 5.

As an illustration, we can measure the importance of getting ahead on the first pitch. We calculate the mean change in expected run value for at-bats starting with a ball and for the at-bats starting with a strike.

```
pbp16rc %>%
  filter(c10 == 1 | c01 == 1) %>%
  group_by(c10, c01) %>%
  summarize(N = n(), mean_run_value = mean(RUNS.VALUE))

# A tibble: 2 x 4
# Groups:   c10 [?]
  c10   c01       N mean_run_value
  <lgl> <lgl> <int>          <dbl>
1 FALSE TRUE  94109        -0.0394
2 TRUE  FALSE 76165         0.0371
```

The conclusion is that the difference between a first pitch strike and a first pitch ball, as estimated with data from the 2016 season, is over 0.07 expected runs.

We can calculate the run value for each possible ball/strike count. First, we use the select() function and the starts_with() operator to grab only those columns that start with the letter c. In this case, this matches the columns c00, c01, etc. that we defined previously. Additionally, we grab the RUNS.VALUE column.

```
pbp_counts <- pbp16rc %>%
  select(starts_with("c"), RUNS.VALUE)
```

Now, we want to apply the group_by()-summarize() idiom that we used previously to calculate the mean run value across *all* of the 12 possible counts. One way to do this would be to write a function that will perform that operation for a given variable name, and then iterate over the 12 variable names, and indeed, that is the approach taken in the first edition of this book. Here, we employ an alternative strategy that is more in keeping with the tidyverse philosophy and involves much less code. However, it may be conceptually less intuitive.

pbp_counts has $n = 190717$ rows and $p = 13$ columns. The variable named RUNS.VALUE contains a measurement of runs, and the other 12 columns contain logical indicators as to whether the plate appearance passed through a particular count. Thus, we really have three different kinds of information in this data frame: a count, whether the plate appearance passed through that

count, and the run value. To *tidy* these data [Wickham, 2014], we need to create a long data frame with $12n$ rows and those three columns. We do this using the `gather()` function. We provide a name to the `key` argument, which becomes the name of the new variable that records the count. Similarly, the `value` argument takes a name for the new variable that records the data that was in the column that was gathered. We don't want to gather the `RUNS.VALUE` column, since that records a different kind of information.

```
pbp_counts_tidy <- pbp_counts %>%
  gather(key = "count", value = "passed_thru", -RUNS.VALUE)
sample_n(pbp_counts_tidy, 6)

# A tibble: 6 x 3
  RUNS.VALUE count passed_thru
       <dbl> <chr> <lgl>
1      0.114 c01   TRUE
2     -0.106 c12   FALSE
3     -1.19  c12   FALSE
4     -0.312 c10   FALSE
5     -0.695 c31   FALSE
6     -0.921 c10   FALSE
```

Note that every plate appearance appears $p = 12$ times in `pbp_counts_tidy`: one row for each count. To compute the average change in expected runs (i.e., the mean run value), we have to `filter()` for only those plate appearances that actually passed through that count. Then we simply apply our `group_by()-summarize()` operation as before. Thus, in the `mean()` operation, the data is limited to only those plate appearances that have gone through each particular ball-strike count.

```
run_value_by_count <- pbp_counts_tidy %>%
  filter(passed_thru == 1) %>%
  group_by(count) %>%
  summarize(N = n(), value = mean(RUNS.VALUE))
```

Finally, we can then update our `count_plot` to use these new data instead of the old `mussina` data. To do this, we have to re-compute the balls and strikes based on the count variable. We can do this by picking out the values of balls and strikes using the `str_sub()` function. We use the `scale_fill_gradient2()` function again to reset our diverging palette to colors more appropriate for these data (i.e., a midpoint at 0).

```
run_value_by_count <- run_value_by_count %>%
  mutate(balls = str_sub(count, 2, 2),
         strikes = str_sub(count, 3, 3))
```

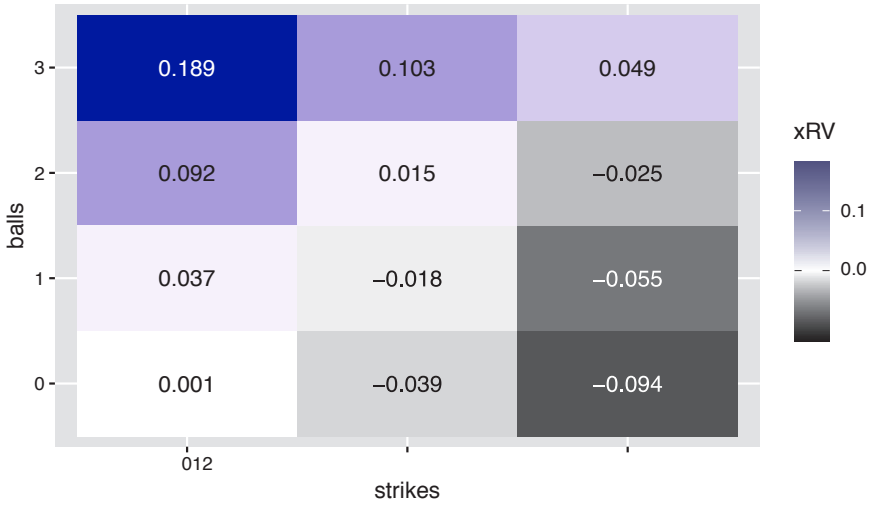


FIGURE 6.2 Average change in expected runs for plate appearances passing through each ball/strike count. Values estimated on data from the 2016 season.

```
count_plot %+% run_value_by_count +
  scale_fill_gradient2("xRV", low = "grey10", high = crcblue,
                       mid = "white")
```

By glancing at the values and color shades in Figure 6.2, one can construct reasonable definitions for the terms "hitter's count" and "pitcher's count." Note that since all plate appearances pass through the 0-0 count, the average change in expected run value for this count is 0. Ball/strike counts can be roughly divided in the following four categories[7]:

- Pitcher's counts: 0-2, 1-2, 2-2, 0-1;
- Neutral counts: 0-0, 1-1;
- Modest hitter's counts: 3-2, 2-1, 1-0;
- Hitter's counts: 3-0, 3-1, 2-0.

[7]The proposed categorization, based on observation of Figure 6.2, reflects the one proposed by analyst Tom Tango (see `www.insidethebook.com/ee/index.php/site/comments/plate_counts/`).

6.2.4 The importance of the previous count

In the previous section we calculated run values for any ball/strike count. In performing this calculation we simply looked at whether a plate appearance went through a particular count, without considering how it got there. In other words, we considered, for example, all the at-bats going through a 2-2 count as having the same run expectancy, no matter if the pitcher started ahead 0-2 or fell behind 2-0. The implicit assumption in these calculations is that the previous counts have no influence on the outcome on a particular count[8]. However, a pitcher who got ahead 0-2 is likely to "waste some pitches." That is, he would likely throw a few balls out of the strike zone with the sole intent of inducing the batter (who cannot afford another strike) to swing at them and possibly miss or make poor contact. On the other hand, with a plate appearance starting with two balls, the batter has the luxury of not swinging at strikes in undesirable locations and waiting for the pitcher to deliver a pitch of his liking.

Given the above discussion, it would seem that the run expectancy on a 2-2 count would be higher if the plate appearance started with two balls than if the pitcher started with a 0-2 count. Let's investigate if there is numerical evidence to actually support this conjecture.

We begin by taking the subset of plays from the 2016 season that went through a 2-2 count and calculating their mean change in expected run value.

```
count22 <- pbp16rc %>%
  filter(c22 == 1)
count22 %>%
  summarize(N = n(), mean_run_value = mean(RUNS.VALUE))

# A tibble: 1 x 2
      N mean_run_value
  <int>          <dbl>
1 48697        -0.0253
```

Using the `case_when()` function, we create a new variable `after2` denoting the ball/strike count after two pitches and calculate the mean run value for each of the three possible levels of `after2`.

```
count22 %>%
  mutate(after2 = case_when(
    c20 == 1 ~ "2-0",
    c02 == 1 ~ "0-2",
    c11 == 1 ~ "1-1",
    TRUE ~ "other")) %>%
  group_by(after2) %>%
```

[8]This is analogous to the *memoryless property* referred to in Section 9.2.1.

```
  summarize(N = n(), mean_run_value = mean(RUNS.VALUE))

# A tibble: 3 x 3
  after2     N mean_run_value
  <chr>  <int>          <dbl>
1 0-2     9837       -0.0311
2 1-1    30332       -0.0323
3 2-0     8528        0.00606
```

The above results appear to imply that plate appearances going through a 2-2 count after having started with two strikes are more favorable to the pitcher than those beginning with two balls.

This result should be considered in light of multiple types of potential selection bias. Many plate appearances starting with two strikes end without ever reaching the 2-2 count, in most cases with an unfavorable outcome for the batter.[9] The plate appearances that survive a 0-2 count reaching 2-2 are hardly a random sample of all the plate appearances. Hard-to-strike-out batters are likely over-represented in such a sample, as well as pitchers with good fastball command (to get ahead 0-2), but weak secondary pitches (to finish off batters).

Similarly, comparing the paths leading to 1-1 counts yields results in line with common sense.

```
count11 <- pbp16rc %>%
  filter(c11 == 1)
count11 %>%
  mutate(after2 = ifelse(c10 == 1, "1-0", "0-1")) %>%
  group_by(after2) %>%
  summarize(N = n(), mean_run_value = mean(RUNS.VALUE))

# A tibble: 2 x 3
  after2     N mean_run_value
  <chr>  <int>          <dbl>
1 0-1    51708       -0.0452
2 1-0    54084        0.00745
```

The numbers above suggest that after reaching a 1-1 count, the batter is expected to perform slightly better if the first pitch was a ball than if it was a strike.

6.3 Behaviors by Count

In this section we explore how the roles of three individuals in the pitcher-batter duel are affected by the ball/strike count. How does a batter alter his

[9] In 2016, 85% of plate appearances beginning with two strikes and not reaching the 2-2 count ended with the batter making an out.

swing when ahead or behind in the count? How does a pitcher vary his mix of pitches according to the count? Does an umpire (consciously or unconsciously), shrink or expand his strike zone depending on the pitch count?

We provide an R data file (a file with extension `.Rdata`) containing all the datasets used in this section. Once the file is loaded into R, the data frames `cabrera`, `sanchez`, `umpires`, and `verlander` are visible by use of the `ls()` function.

```
load("data/balls_strikes_count.RData")
ls()

[1] "cabrera"   "sanchez"   "umpires"   "verlander"
```

These datasets contain pitch-by-pitch data, including the location of pitches as recorded by Sportvision's PITCHf/x system. The `cabrera` data frame contains four years of batting data for 2012 American League Triple Crown winner Miguel Cabrera. The data frame `umpires` has information about every pitch thrown in 2012 where the home plate umpire had to judge whether it crossed the strike zone. The `verlander` data frame has four years of pitching data for 2016 Cy Young Award and MVP recipient Justin Verlander.

6.3.1 Swinging tendencies by count

We saw in Section 6.2 that batters perform worse when falling behind in the count. For example, when there are two strikes in the count, the batter may be forced to swing at pitches he would normally let pass by to avoid being called out on strikes. Using PITCHf/x data, we explore how a very good batter like Miguel Cabrera alters his swinging tendencies according to the ball/strike count.

6.3.1.1 Propensity to swing by location

In this section we focus on the relationships between the variables `balls` and `strikes` indicating the count on the batter, the variables `px` and `pz` identifying the pitch location as it crosses the front of the plate, and the `swung` binary variable, denoting whether or not the batter attempted a swing on the pitch.

We show a scatterplot of Miguel Cabrera's swinging tendency in Figure 6.3.

```
cabrera_sample <- cabrera %>%
  sample_n(500)
k_zone_plot <- ggplot(cabrera_sample, aes(x = px, y = pz)) +
  geom_rect(xmin = -0.947, xmax = 0.947, ymin = 1.5,
            ymax = 3.6, fill = "lightgray", alpha = 0.01) +
  coord_equal() +
  scale_x_continuous("Horizontal location (ft.)",
```

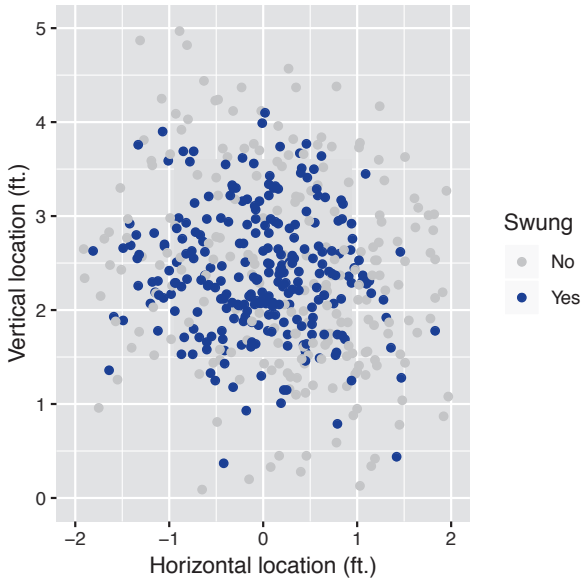


FIGURE 6.3 Scatterplot of Miguel Cabrera's swinging tendency by location. Sample of 500 pitches. View from the catcher's perspective.

```
                    limits = c(-2, 2)) +
  scale_y_continuous("Vertical location (ft.)",
                    limits = c(0, 5))
k_zone_plot +
  geom_point(aes(color = factor(swung))) +
  scale_color_manual("Swung", values = c("gray70", crcblue),
                    labels = c("No", "Yes"))
```

Rather than plot all 6265 pitches, we simplify matters by taking a random sample of 500 pitches. This reduces overlapping in the scatterplot without introducing bias. From Figure 6.3, one can see that Cabrera is less likely to swing at pitches delivered farther away from the strike zone (the black box). However, it is difficult to determine Cabrera's preferred pitch location from this figure.

A contour plot is an effective alternative method to visualize batters' swinging preferences. This type of plot is used to visualize three-dimensional data in two dimensions. Widely used in cartography and meteorology, the contour plot usually features spatial coordinates as the first two variables, while the third variable (which can be, for example, elevation in cartography or barometric pressure in meteorology) is plotted as a contour line, also called an

isopleth. The contour line is a curve joining points sharing equal values of the third variable.

As a first step in producing a contour plot, we fit a smooth polynomial surface to the response variable `swung` as a function of the horizontal and vertical locations `px` and `pz` using the `loess()` function. The output of this fit is stored in the object `miggy_loess`.

```
miggy_loess <- loess(swung ~ px + pz, data = cabrera,
  control = loess.control(surface = "direct"))
```

After the surface has been fit, we are interested in predicting the likelihood of a swing by Cabrera at various pitch locations. Using the `expand.grid()` function, we build a data frame consisting of combinations of horizontal locations from −2 (two feet to the left of the middle of home plate) to +2 (two feet to the right of the middle of the plate) and vertical locations from the ground (value of zero) to six feet of height, using subintervals of 0.1 feet. Using the `predict()` function, we obtain the likelihood of Miguel's swinging at every location in the data frame. Note that because `predict()` returns a `matrix`, we use the `as.numeric()` function to convert the fitted values into a numeric vector.

```
pred_area <- expand.grid(px = seq(-2, 2, by = 0.1),
                         pz = seq(0, 6, by = 0.1))
pred_area_fit <- pred_area %>%
  mutate(fit = as.numeric(predict(miggy_loess,
                                  newdata = .)))
```

To spot check, we examine in the data frame `pred_area_fit` the likelihood that Miguel will swing for three different hand-picked locations—a pitch down the middle and two and a half feet from the ground ("down Broadway"), a ball that hits the ground in the middle of the plate ("ball in the dirt"), and another one delivered at mid-height (2.5 feet from the ground) but way outside (two feet from the middle of the plate). In each case, we use the `filter()` function to take a subset of the prediction data frame `pred_area_fit` with specific values of the horizontal and vertical locations `px` and `pz`.

```
pred_area_fit %>%
  filter(px == 0 & pz == 2.5)  # down Broadway

  px  pz   fit
1  0 2.5 0.844

pred_area_fit %>%
  filter(px == 0 & pz == 0)    # ball in the dirt

  px pz   fit
1  0  0 0.154
```

```
pred_area_fit %>%
  filter(px == 2 & pz == 2.5)  # way outside

  px  pz    fit
1  2 2.5 0.0783
```

The results are quite consistent with what one would expect: the pitch right in the heart of the strike zone induces Cabrera to swing more than 80 percent of the time, while the ball in the dirt and the ball outside generate a swing at 15 percent and 8 percent rates, respectively.

We construct a contour plot of the likelihood of the swing as a function of the horizontal and vertical locations of the pitch using the `geom_contour()` function in the `ggplot2` package. For logical consistency, we `filter()` for only those contours corresponding to swing probabilities between 0 and 1. The `direct.label()` function from the `directlabels` package provides helpful labeling. Figure 6.4 shows the resulting contour plot.

```
cabrera_plot <- k_zone_plot %+%
  filter(pred_area_fit, fit >= 0, fit <= 1) +
  stat_contour(aes(z = fit, color = stat(level)),
               binwidth = 0.2) +
  scale_color_gradient(low = "white", high = crcblue)

cabrera_plot <- cabrera_plot %>%
  directlabels::direct.label(method = "bottom.pieces")
cabrera_plot
```

As expected, the likelihood of a swing decreases the further the ball is delivered from the middle of the strike zone. The plot also shows that Cabrera has a tendency to swing at pitches on the inside part of the plate.

6.3.1.2 Effect of the ball/strike count

Figure 6.4 reports Miguel's swinging tendency over all pitch counts. Can we visualize how Cabrera varies his approach according to the ball/strike count? Specifically, does Cabrera become more selective when he is ahead and can afford to wait for a pitch of his liking and, conversely, does he "expand his zone" when there are two strikes and he cannot allow another called strike go by? We described the process of calculating the swing propensity by location in Section 6.3.1.1. Here, we generalize that procedure and iterate it over all counts.

In this case, we restrict our interest to 0-0, 0-2, and 2-0 counts. The vector `counts` contains these values. Next, we split the `cabrera` data frame into a `list` with three elements: one data frame for each of the chosen counts. We accomplish this by `filter()`-ing for those `counts` and using the `split()`

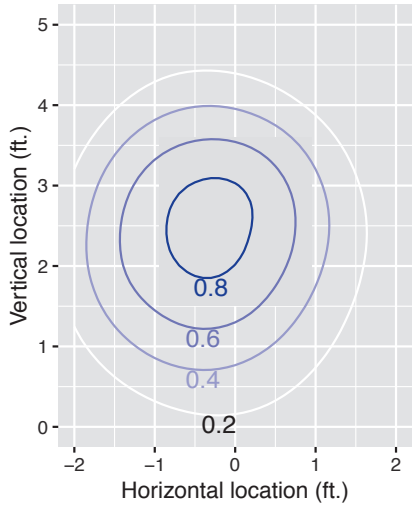


FIGURE 6.4 Contour plot of Miguel Cabrera's swinging tendency by location, where the view is from the catcher's perspective. The contour lines are labeled by the probability of swinging at the pitch.

function to do the splitting. Note that the resulting `count_dfs` object is a `list` of three `data.frame`s.

```
counts <- c("0-0", "0-2", "2-0")
count_dfs <- cabrera %>%
  mutate(count = paste(balls, strikes, sep = "-")) %>%
  filter(count %in% counts) %>%
  split(pull(., count))
```

Next, we use the `map()` function repeatedly to iterate our analysis over the elements of `count_dfs`. First, we compute the LOESS fits for a given set of data specific to the count. Second, we compute three sets of predictions—one for each of the three counts. Third, we convert the numeric matrices returned by `predict()` to numeric vectors called `fit`, and make those into one column `data.frame`s. Fourth, we append those data frames to the `pred_area` data frame. Finally, we stitch all three data frames together using `bind_rows()`, and add variables for the count, number of balls, and number of strikes.

```
count_fits <- count_dfs %>%
  map(~loess(swung ~ px + pz, data = .,
             control = loess.control(surface = "direct"))) %>%
  map(predict, newdata = pred_area) %>%
  map(~data.frame(fit = as.numeric(.))) %>%
```

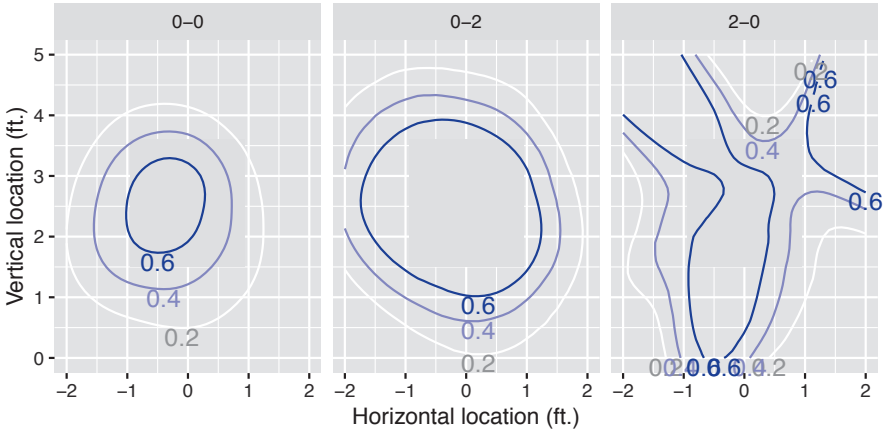


FIGURE 6.5 Contour plots of Miguel Cabrera's swinging tendency by selected ball/strike counts, viewed from the catcher's perspective. The contour lines are labeled by the probability of swinging at the pitch.

```
  map_df(bind_cols, pred_area, .id = "count") %>%
  mutate(balls = str_sub(count, 1, 1),
         strikes = str_sub(count, 3, 3))
```

This process performs the same tasks that we did before: it fits a LOESS model to the pitch location data, then uses that model to generate swing probability predictions across the entire area and returns a `data.frame` with the associated ball and srike count.

We can then use a `facet_wrap()` to show the contour plots on separate panels to compare Cabrera's swinging tendencies by pitch count (Figure 6.5.) To improve legibility, we only show the 20%, 40%, and 60% contours.

```
cabrera_plot %+%
  filter(count_fits, fit > 0.1, fit < 0.7) +
  facet_wrap( ~ count)
```

As expected, Cabrera expands his swing zone when behind 0-2 (his 40% contour line on 0-2 counts has an area comparable to his 20% contour line on 0-0 counts). The third panel in Figure 6.5 is comprised of a relatively low sample size and probably does not tell us much.

6.3.2 Pitch selection by count

We now move to the other side of the pitcher/batter duel in our investigation of the effect of the count. Pitchers generally possess arsenals of two to five different pitch types. All pitchers have a fastball at their disposal, which is generally a pitch that is easy to throw to a desired location. So-called secondary pitches, such as curve balls or sliders, while often effective (especially when hitters are not expecting them), are harder to control and rarely used by pitchers behind in the count. In this section we look at one pitcher (arguably one of the best in MLB at the time) and explore how he chooses from his pitch repertoire according to the ball/strike count.

The `verlander` data frame, consisting of over 15 thousand observations, consists of pitch data for Justin Verlander for four seasons. Using the `group_by()` and `summarize()` commands, we obtain a frequency table of the types of pitches Verlander threw from 2009–2012. In this case, we compute the pitch type proportions in addition to their frequencies.

```
verlander %>%
  group_by(pitch_type) %>%
  summarize(N = n()) %>%
  mutate(pct = N / nrow(verlander)) %>%
  arrange(desc(pct))

# A tibble: 5 x 3
  pitch_type     N    pct
  <fct>      <int>  <dbl>
1 FF          6756 0.441
2 CU          2716 0.177
3 CH          2550 0.167
4 FT          2021 0.132
5 SL          1264 0.0826
```

As is the case with most major league pitchers, Verlander throws his fastball most frequently. He uses two variations of a fastball: a four-seamer (FF) and a two-seamer (FT). He complements his fastballs with a curve ball (CU), a change-up (CH), and a slider (SL).

We see in the table that 44% of Verlander's pitches during this five-season period were four-seam fastballs.

Before moving to exploring pitch selection by ball/strike count, we compute a frequency table to explore the pitch selection by batter handedness. The `spread()` function helps us display the results in a wide rather than long format.

```
verlander %>%
  group_by(batter_hand, pitch_type) %>%
  summarize(N = n()) %>%
```

```
  spread(key = batter_hand, value = N) %>%
  mutate(L_pct = L / sum(L), R_pct = R / sum(R))

# A tibble: 5 x 5
  pitch_type     L     R  L_pct  R_pct
  <fct>      <int> <int>  <dbl>  <dbl>
1 CH          2024   526 0.228  0.0817
2 CU          1529  1187 0.172  0.184
3 FF          3832  2924 0.432  0.454
4 FT          1303   718 0.147  0.111
5 SL           178  1086 0.0201 0.169
```

Note that Verlander's pitch selection is quite different depending on the handedness of the opposing batter. In particular, the right-handed Verlander uses his changeup nearly a quarter of the time against left-handed hitters, but only eight percent of the time against right-handed hitters. Conversely the slider is nearly absent from his repertoire when he faces lefties, while he uses it close to one out of six times against righties.

Batter-hand differences in pitch selection are common among major league pitchers and they exist because the effectiveness of a given pitch depends on the handedness of the pitcher and the batter. The slider and change-up comparison is a typical example, a slider is very effective against batters of the same handedness while a change-up can be successful when facing opposite-handed batters.

We can also explore Verlander's pitch selection by pitch count as well as batter handedness. In the following code, the `filter()` function is used to select Verlander's pitches delivered to right-handed batters. The rest of the code constructs a table of frequencies by count and pitch type. The `mutate_if()` function divides each numeric variable by the total number of pitches.

```
verlander %>%
  filter(batter_hand == "R") %>%
  group_by(balls, strikes, pitch_type) %>%
  summarize(N = n()) %>%
  spread(key = pitch_type, value = N, fill = 0) %>%
  mutate(num_pitches = CH + CU + FF + FT + SL) %>%
  mutate_if(is.numeric, funs(. / num_pitches)) %>%
  select(-num_pitches)

# A tibble: 12 x 7
# Groups:   balls, strikes [12]
   balls strikes     CH     CU    FF     FT     SL
   <int>   <int>  <dbl>  <dbl> <dbl>  <dbl>  <dbl>
 1     0       0 0.0692 0.115  0.526 0.156  0.134
 2     0       1 0.0624 0.242  0.405 0.101  0.189
```

```
 3     0       2 0.158  0.282  0.274 0.0629 0.223
 4     1       0 0.0493 0.107  0.523 0.112  0.209
 5     1       1 0.0805 0.238  0.398 0.0960 0.187
 6     1       2 0.143  0.327  0.278 0.0617 0.191
 7     2       0 0.0174 0.0174 0.703 0.145  0.116
 8     2       1 0.0623 0.0796 0.512 0.142  0.204
 9     2       2 0.102  0.294  0.357 0.0869 0.161
10     3       0 0.0833 0      0.812 0.104  0
11     3       1 0.0196 0      0.784 0.118  0.0784
12     3       2 0.0429 0.0429 0.693 0.116  0.106
```

The effect of the ball/strike count on the choice of pitches is apparent when comparing pitcher's counts and hitter's counts. When behind 2-0, Verlander uses his four-seamer seven times out of ten; the percentage goes up to 78% when trailing 3-1 and 81% on 3-0 counts. Conversely, when Justin has the chance to strike the batter out, the use of the four-seamer diminishes. In fact he throws it less than 30 percent of the time both on 0-2 and 1-2 counts. On a full count, Verlander's propensity to throw his fastball is similar to those of hitters' counts—this is consistent with the numbers in Figure 6.2 that indicate the 3-2 count being slightly favorable to the hitter. One can explore Verlander's choices by count when facing a left-handed hitter by simply changing `R` to `L` in the code above.

6.3.3 Umpires' behavior by count

Hardball Times author John Walsh wrote a 2010 article titled *The Compassionate Umpire* in which he showed that home plate umpires tend to modify their ball/strike calling behavior by slightly favoring the player who is behind in the count [Walsh, 2010]. In other words, umpires tend to enlarge their strike zone in hitter's counts and to shrink it when pitchers are ahead. In this section we visually explore Walsh's finding by plotting contour lines for three different ball/strike counts.

The `umpires` data frame is similar to those of `verlander` and `cabrera`. A sample of its contents—obtained using the `sample_n()` function—is shown in Table 6.2.

```
sample_n(umpires, 20)
```

The data consist of every pitch of the 2012 season for which the home plate umpire had to judge whether it crossed the strike zone. Additional columns not present in either the `verlander` or the `cabrera` data frames identify the name of the umpire (variable `umpire`) and whether the pitch was called for a strike (variable `called_strike`).

TABLE 6.2 A twenty row sample of the `umpires` dataset.

season	umpire	batter_hand	pitch_type	balls	strikes	px	pz	called_strike
2012	Paul Schrieber	L	CU	1	0	-1.69	2.90	0
2012	Wally Bell	L	SI	1	1	0.72	1.96	1
2012	Tim McClelland	R	FF	1	0	0.32	1.39	0
2012	Mark Carlson	R	SL	1	1	0.68	2.67	1
2012	Kerwin Danley	R	SI	1	1	0.02	1.63	0
2012	Derryl Cousins	R	SL	0	0	-0.06	3.25	1
2012	Mike Estabrook	R	CU	1	0	-0.82	1.51	0
2012	Cory Blaser	R	CU	2	2	1.56	0.97	0
2012	Mike Everitt	R	CU	1	1	-0.59	3.08	1
2012	Tim Timmons	L	FF	1	0	-0.65	1.98	1
2012	Lance Barksdale	R	FT	0	0	1.25	4.57	0
2012	Chris Conroy	R	CH	1	2	-1.65	2.37	0
2012	Gary Cederstrom	R	FC	2	2	1.21	2.22	0
2012	Larry Vanover	R	CH	1	0	-0.12	1.50	0
2012	Marty Foster	L	CU	0	0	-1.71	1.63	0
2012	Jim Reynolds	R	FF	3	0	-0.49	2.04	1
2012	Jim Wolf	L	SL	0	2	-0.42	0.59	0
2012	Brian O'Nora	R	FF	3	2	-0.87	2.10	1
2012	Chris Guccione	R	FT	0	1	-0.55	0.65	0
2012	Manny Gonzalez	L	CU	0	2	-1.74	0.54	0

We proceed similarly to the analysis of Section 6.3.1.2, using the `loess()` function to estimate the umpires' likelihood of calling a strike, based on the location of the pitch. Here we limit the analysis to plate appearances featuring right-handed batters, as it has been shown that umpires tend to call pitches slightly differently depending on the handedness of the batter.

```
umpires_rhb <- umpires %>%
  filter(batter_hand == "R",
         balls == 0 & strikes == 0 |
           balls == 3 & strikes == 0 |
           balls == 0 & strikes == 2)
```

By slightly modifying the code above, the reader can easily repeat the process for other counts. In this section we compare the 0-0 count to the most extreme batter and pitcher counts, 3-0 and 0-2 counts, respectively.

To do this, we can re-purpose our `map()` pipeline from above, incorporating the `pred_area` data frame. Note that the response variable in the LOESS model here is `called_strike`. In addition, the `loess()` smoother is applied on a subset of 3000 randomly selected pitches, to reduce computation time.

```
ump_count_fits <- umpires_rhb %>%
  mutate(count = paste(balls, strikes, sep = "-")) %>%
  split(pull(., count)) %>%
  map(sample_n, 3000) %>%
  map(~loess(called_strike ~ px + pz, data = .,
             control = loess.control(surface = "direct"))) %>%
  map(predict, newdata = pred_area) %>%
  map(~data.frame(fit = as.numeric(.))) %>%
  map_df(bind_cols, pred_area, .id = "count") %>%
  mutate(balls = str_sub(count, 1, 1),
         strikes = str_sub(count, 3, 3))
```

Figure 6.6 shows that the umpire's strike zone shrinks considerably in a 0-2 pitch count, and slightly expanded in a 3-0 count. To isolate the 0.5 contour lines, we `filter()` the fitted values for those near 0.5, and then use `geom_contour()` to set the width of the bins to be small.

```
k_zone_plot %+% filter(ump_count_fits, fit < 0.6 & fit > 0.4) +
  geom_contour(aes(z = fit, color = count, linetype = count),
               binwidth = 0.1) +
  scale_color_manual(values = crc_3)
```

6.4 Further Reading

Palmer [1983] is possibly one of the first examinations of the balls/strikes count

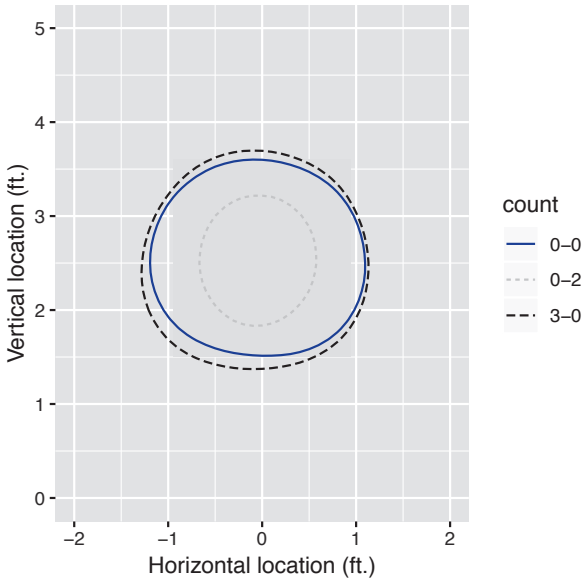


FIGURE 6.6 Umpires' 50/50 strike calling zone in different balls/strikes counts viewed from the catcher's perspective.

effect on the outcome of plate appearances: it is based on data from World Series games from 1974 to 1977 and features a table resembling Figures 6.1 and 6.2. Walsh [2008] calculates the run value of a ball and of a strike at every count and uses the results for ranking baseball's best fastballs, sliders, curveballs, and change-ups. Walsh [2010] shows how umpires are (perhaps unconsciously) affected by the balls/strikes count when judging pitches. In particular, he presents a scatterplot showing a very high correlation between the strike zone area and the count run value (see Figure 6.2). Allen [2009a,b] and Marchi [2010] illustrate so-called platoon splits (i.e. the different effectiveness against same-handed versus opposite-handed batters) for various pitch types.

6.5 Exercises

1. (**Run Value of Individual Pitches**)

   (a) Calculate the run value of a ball and of a strike at any count. For 3-ball and 2-strike counts you need the value of a walk and a strikeout respectively (you can calculate them as done for other events in Chapter 5).

   (b) Compare your values to the ones proposed by John Walsh in the

article `www.hardballtimes.com/main/article/searching-for-the-games-best-pitch/`.

2. (**Length of Plate Appearances**)

   (a) Calculate the length, in term of pitches, of the average plate appearance by batting position using Retrosheet data for the 2016 season.

   (b) Does the eighth batter in the National League behave differently than his counterpart in the American League?

   (c) Repeat the calculations in (a) and (b) for the 1991 and 2016 seasons and comment on any differences between the seasons that you find.

3. (**Pickoff Attempts**)

   Identify the baserunners who, in the 2016 season, drew the highest number of pickoff attempts when standing at first base with second base unoccupied.

4. (**Umpire's Strike Zone**)

   By drawing a contour plot, compare the umpire's strike zone for left-handed and right-handed batters. Use only the rows of the data frame where the pitch type is a four-seam fastball.

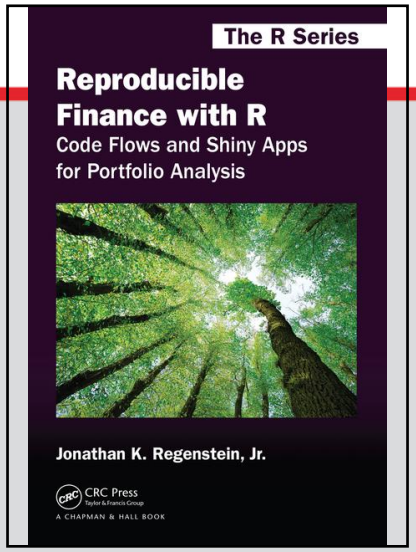
5. (**Umpire's Strike Zone, Continued**)

   By drawing one or more contour plots, compare the umpire's strike zone by pitch type. For example, compare the 50/50 contour lines of four-seam fastballs and curveballs when a right-handed batter is at the plate.

CHAPTER

6

SHARPE RATIO



This chapter is excerpted from

Reproducible Finance with R

by Jonathan K. Regenstein, Jr.

Learn more

7

Sharpe Ratio

The Sharpe Ratio is defined as the mean of the excess monthly portfolio returns above the risk-free rate, divided by the standard deviation of the excess monthly portfolio returns above the risk-free rate. This is the formulation of the Sharpe Ratio as of 1994; if we wished to use the original formulation from 1966 the denominator would be the standard deviation of all the monthly portfolio returns.

The Sharpe Ratio measures excess returns per unit of risk, where we again take the standard deviation to represent portfolio risk. The Sharpe Ratio was brought to us by Bill Sharpe - arguably the most important economist for modern investment management as the creator of the Sharpe Ratio, CAPM (which we will cover later) and Financial Engines, a forerunner of today's robo-advisor movement.

The Sharpe Ratio equation is as follows:

$$Sharpe\ Ratio = (\overline{R_p - R_f})/\sigma_{excess}$$

The numerator is the mean excess return above the risk-free rate and the denominator is the standard deviation of those excess returns. In other words, it is the ratio of return to risk and so a higher Sharpe Ratio indicates a 'better' portfolio.

We will start with the built-in function from the `xts` world and will look at the by-hand equation as part of the tidyverse.

7.1 Sharpe Ratio in the xts world

For any work with the Sharpe Ratio, we first must choose a risk-free rate (hereafter RFR) and will use .3%.

```
rfr <- .0003
```

From there, calculating the Sharpe Ratio in the `xts` world is almost depressingly convenient. We call `SharpeRatio(portfolio_returns_xts, Rf = rfr)`, passing our portfolio returns and risk-free rate to the built-in function from `PerformanceAnalytics`.

```
sharpe_xts <-
  SharpeRatio(portfolio_returns_xts_rebalanced_monthly,
              Rf = rfr,
              FUN = "StdDev") %>%
  `colnames<-`("sharpe_xts")

sharpe_xts
```

```
                                  sharpe_xts
StdDev Sharpe (Rf=0%, p=95%):     0.2749
```

7.2 Sharpe Ratio in the tidyverse

For our tidyverse example, we will implement the Sharpe Ratio equation via pipes and `dplyr`.

We start with our object `portfolio_returns_dplyr_byhand` and then run `summarise(ratio = mean(returns - rfr)/sd(returns - rfr))`, which maps to the equation for the Sharpe Ratio.

```
sharpe_tidyverse_byhand <-
  portfolio_returns_dplyr_byhand %>%
  summarise(sharpe_dplyr = mean(returns - rfr)/
              sd(returns - rfr))

sharpe_tidyverse_byhand
```

```
# A tibble: 1 x 1
  sharpe_dplyr
         <dbl>
1        0.275
```

7.3 Sharpe Ratio in the tidyquant world

`tidyquant` allows us to wrap the `SharpeRatio()` function inside the `tq_performance()` function.

```
sharpe_tq <-
  portfolio_returns_tq_rebalanced_monthly %>%
  tq_performance(Ra = returns,
                 performance_fun = SharpeRatio,
                 Rf = rfr,
                 FUN = "StdDev") %>%
  `colnames<-`("sharpe_tq")
```

Let's compare our 3 Sharpe objects.

```
sharpe_tq %>%
  mutate(tidy_sharpe = sharpe_tidyverse_byhand$sharpe_dplyr,
         xts_sharpe = sharpe_xts)
```

```
# A tibble: 1 x 3
  sharpe_tq tidy_sharpe xts_sharpe
      <dbl>       <dbl>      <dbl>
1     0.275       0.275      0.275
```

We have consistent results from `xts`, `tidyquant` and our by-hand piped calculation. Next, we compare to the Sharpe Ratio of the S&P500 in the same time period.

```
market_returns_xts <-
    getSymbols("SPY",
               src = 'yahoo',
               from = "2012-12-31",
               to = "2017-12-31",
               auto.assign = TRUE,
               warnings = FALSE) %>%
    map(~Ad(get(.))) %>%
    reduce(merge) %>%
    `colnames<-`("SPY") %>%
    to.monthly(indexAt = "lastof",
               OHLC = FALSE)

market_sharpe <-
```

```
  market_returns_xts %>%
  tk_tbl(preserve_index = TRUE,
         rename_index = "date") %>%
    mutate(returns =
             (log(SPY) - log(lag(SPY)))) %>%
    na.omit() %>%
    summarise(ratio =
                mean(returns - rfr)/sd(returns - rfr))

market_sharpe$ratio
```

```
[1] 0.4348
```

Our portfolio has *underperformed* the market during our chosen time period. Welcome to the challenges of portfolio construction during a raging bull market.

7.4 Visualizing Sharpe Ratio

Before visualizing the actual Sharpe, we will get a sense for what proportion of our portfolio returns exceeded the RFR.

When we originally calculated Sharpe by-hand in the tidyverse, we used `summarise` to create one new cell for our end result. The code was `summarise(ratio = mean(returns - rfr)/sd(returns - rfr))`.

Now, we will make two additions to assist in our data visualization. We will add a column for returns that fall below the risk-free rate with `mutate(returns_below_rfr = ifelse(returns < rfr, returns, NA))` and add a column for returns above the risk-free rate with `mutate(returns_above_rfr = ifelse(returns > rfr, returns, NA))`.

This is not necessary for calculating the Sharpe Ratio, but we will see how it illustrates a benefit of doing things by-hand with `dplyr`: if we want to extract or create certain data transformations, we can add it to the piped code flow.

```
sharpe_byhand_with_return_columns <-
  portfolio_returns_tq_rebalanced_monthly %>%
  mutate(ratio =
           mean(returns - rfr)/sd(returns - rfr)) %>%
  mutate(returns_below_rfr =
           if_else(returns < rfr, returns, as.numeric(NA))) %>%
```

```
  mutate(returns_above_rfr =
           if_else(returns > rfr, returns, as.numeric(NA))) %>%
  mutate_if(is.numeric, funs(round(.,4)))

sharpe_byhand_with_return_columns %>%
  head(3)
```

```
# A tibble: 3 x 5
  date         returns ratio returns_below_rfr
  <date>         <dbl> <dbl>             <dbl>
1 2013-01-31  0.0308   0.275         NA
2 2013-02-28 -0.000900 0.275         -0.000900
3 2013-03-31  0.0187   0.275         NA
# ... with 1 more variable: returns_above_rfr <dbl>
```

Now we can create a scatter plot in order to quickly grasp how many of our returns are above the RFR and how many are below the RFR.

We will create green points for returns above RFR with `geom_point(aes(y = returns_above_RFR), colour = "green")` and red points for returns below RFR with `geom_point(aes(y = returns_below_rfr), colour = "red")`.

We also add a blue vertical line at November of 2016 for the election and a horizontal purple dotted line at the RFR.

```
sharpe_byhand_with_return_columns %>%
  ggplot(aes(x = date)) +
  geom_point(aes(y = returns_below_rfr),
             colour = "red") +
  geom_point(aes(y = returns_above_rfr),
             colour = "green") +
  geom_vline(xintercept =
               as.numeric(as.Date("2016-11-30")),
             color = "blue") +
  geom_hline(yintercept = rfr,
             color = "purple",
             linetype = "dotted") +
  annotate(geom = "text",
           x = as.Date("2016-11-30"),
           y = -.04,
           label = "Election",
           fontface = "plain",
           angle = 90,
           alpha = .5,
           vjust =  1.5) +
```

```
  ylab("percent monthly returns") +
  scale_y_continuous(breaks = pretty_breaks(n = 10)) +
  scale_x_date(breaks = pretty_breaks( n = 8))
```

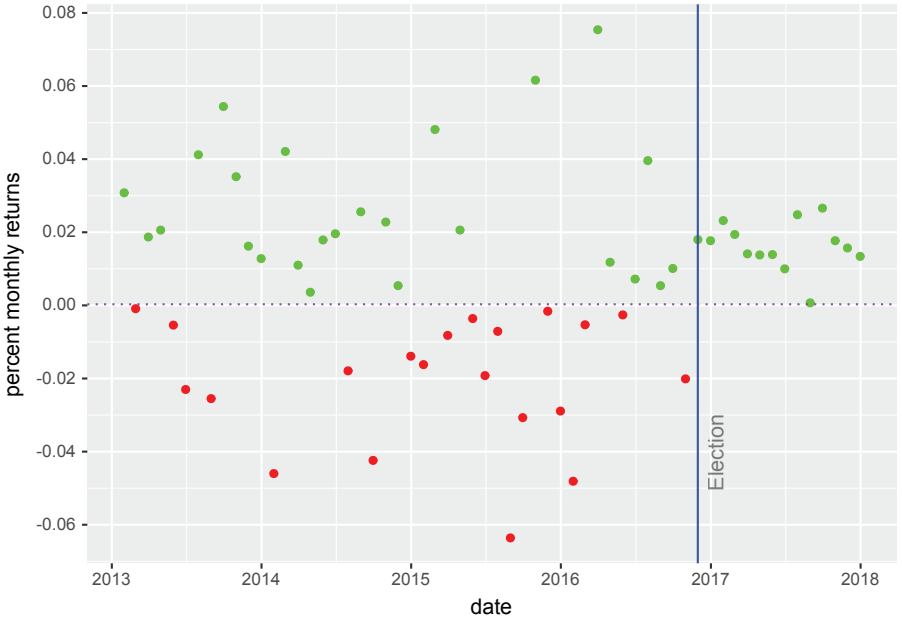


FIGURE 7.1: Scatter Returns Around Risk Free Rate

Have a look at Figure 7.1 and notice that there are zero returns below the RFR after the election in 2016.

Next we will build a histogram of the distribution of returns with `geom_histogram(alpha = 0.25, binwidth = .01, fill = "cornflowerblue")` and add a vertical line at the RFR.

```
sharpe_byhand_with_return_columns %>%
  ggplot(aes(x = returns)) +
  geom_histogram(alpha = 0.45,
                 binwidth = .01,
                 fill = "cornflowerblue") +
  geom_vline(xintercept = rfr,
             color = "green") +
  annotate(geom = "text",
           x = rfr,
```

```
           y = 13,
           label = "rfr",
           fontface = "plain",
           angle = 90,
           alpha = .5,
           vjust =  1)
```

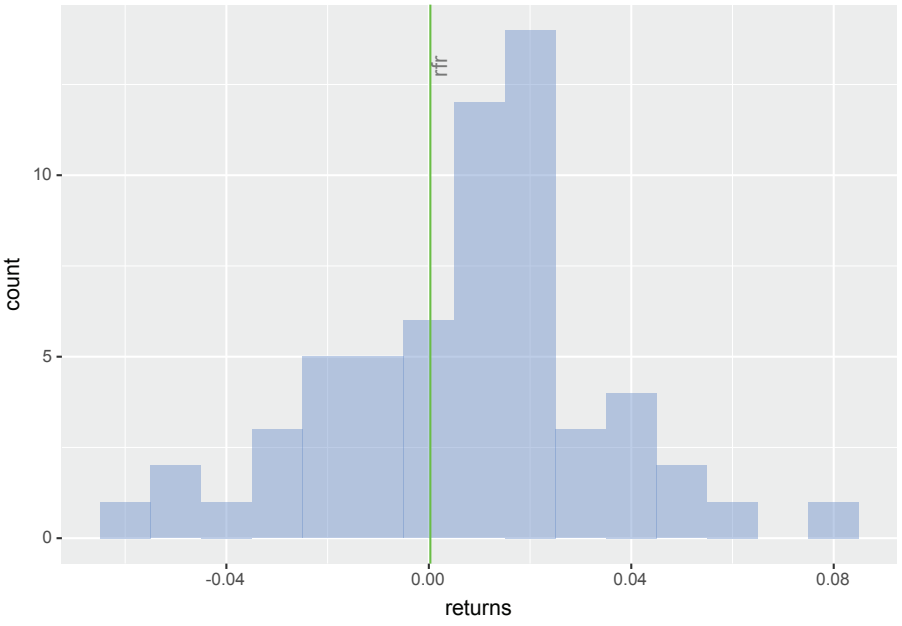


FIGURE 7.2: Returns Histogram with Risk-Free Rate ggplot

Figure 7.2 shows the distribution of returns in comparison to the risk-free rate, but we have not visualized the actual Sharpe Ratio yet.

Similar to standard deviation, skewness and kurtosis of returns, it helps to visualize the Sharpe Ratio of our portfolio in comparison to other assets.

```
asset_returns_long %>%
  summarise(stand_dev = sd(returns),
            sharpe = mean(returns - rfr)/
              sd(returns - rfr))%>%
  add_row(asset = "Portfolio",
    stand_dev =
      portfolio_sd_xts_builtin[1],
```

```
    sharpe =
      sharpe_tq$sharpe_tq) %>%
  ggplot(aes(x = stand_dev,
             y = sharpe,
             color = asset)) +
  geom_point(size = 2) +
  geom_text(
   aes(x =
    sd(portfolio_returns_tq_rebalanced_monthly$returns),
     y =
    sharpe_tq$sharpe_tq + .02,
        label = "Portfolio")) +
  ylab("Sharpe Ratio") +
  xlab("standard deviation") +
  ggtitle("Sharpe Ratio versus Standard Deviation") +
  # The next line centers the title
  theme_update(plot.title = element_text(hjust = 0.5))
```

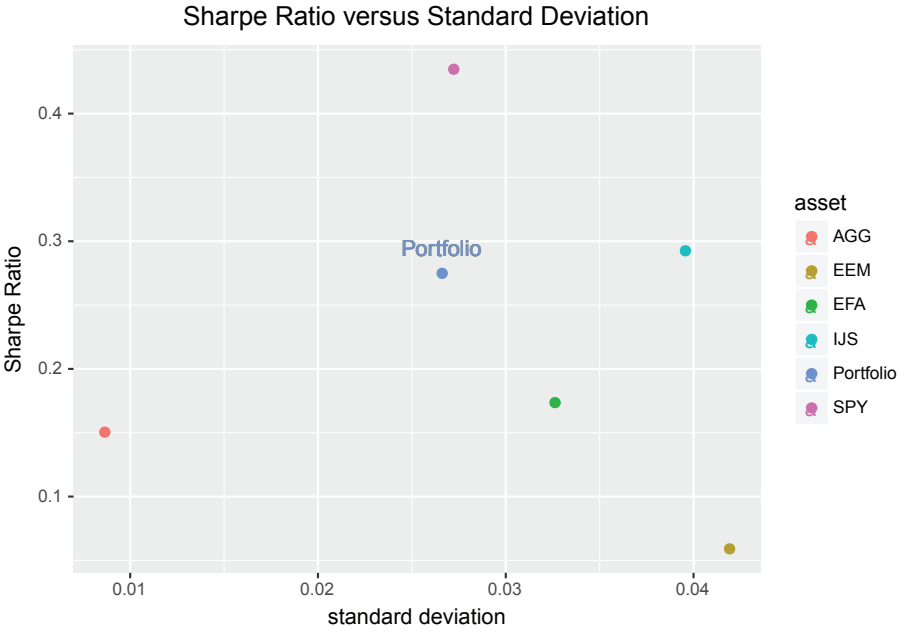


FIGURE 7.3: Sharpe versus Standard Deviation

Figure 7.3 indicates that the S&P500 again seems to dominate our portfolio, though it does have slightly more risk.

That's interesting to observe but, just as with standard deviation, skewness and kurtosis, these overall numbers might obscure important periods of fluctuation in our data. We can solve that by working with the rolling Sharpe Ratio.

7.5 Rolling Sharpe Ratio in the xts world

Very similar to how we calculated rolling standard deviation, skewness and kurtosis, our `xts` work starts with `rollapply()`.

Note that we use a more wordy function format here because we need to pass in the argument `FUN = "StdDev"`. Try running the code without that argument and review the error.

```
window <- 24

rolling_sharpe_xts <-
  rollapply(portfolio_returns_xts_rebalanced_monthly,
            window,
            function(x)
            SharpeRatio(x,
                        Rf = rfr,
                        FUN = "StdDev")) %>%
  na.omit() %>%
  `colnames<-`("xts")
```

7.6 Rolling Sharpe Ratio with the tidyverse and tibbletime

We can combine the tidyverse and `tibbletime` to calculate the rolling Sharpe Ratio calculation starting from a `tibble`.

We first write our own function by combining `rollify()` and `ratio = mean(returns - rfr)/sd(returns - rfr)`.

Notice we still pass in our `rfr` and `window` variables from previous code chunks.

```
# Creat rolling function.
sharpe_roll_24 <-
  rollify(function(returns) {
    ratio = mean(returns - rfr)/sd(returns - rfr)
    },
window = window)
```

Next we pass our portfolio data object to that rolling function, via `mutate()`.

```
rolling_sharpe_tidy_tibbletime <-
  portfolio_returns_dplyr_byhand %>%
  as_tbl_time(index = date) %>%
  mutate(tbltime_sharpe = sharpe_roll_24(returns)) %>%
  na.omit() %>%
  select(-returns)
```

7.7 Rolling Sharpe Ratio with tidyquant

To calculate the rolling Sharpe Ratio with `tidyquant`, we first build a custom function where we can specify the RFR and an argument to the `SharpeRatio()` function. Again, our rolling Sharpe Ratio work is more complex than previous rolling calculations.

```
sharpe_tq_roll <- function(df){
  SharpeRatio(df,
              Rf = rfr,
              FUN = "StdDev")
}
```

It is necessary to build that custom function because we would not be able to specify `FUN = "StdDev"` otherwise.

Now we use `tq_mutate()` to wrap `rollapply()` and our custom function, and apply them to `portfolio_returns_tq_rebalanced_monthly`.

```
rolling_sharpe_tq <-
portfolio_returns_tq_rebalanced_monthly %>%
tq_mutate(
        select     = returns,
```

```
        mutate_fun = rollapply,
        width      = window,
        align      = "right",
        FUN        = sharpe_tq_roll,
        col_rename = "tq_sharpe"
    ) %>%
  na.omit()
```

Now we can compare our 3 rolling Sharpe Ratio objects and confirm consistency.

```
rolling_sharpe_tidy_tibbletime %>%
 mutate(xts_sharpe = coredata(rolling_sharpe_xts),
        tq_sharpe = rolling_sharpe_tq$tq_sharpe ) %>%
  head(3)
```

```
# A time tibble: 3 x 4
# Index: date
  date       tbltime_sharpe xts_sharpe tq_sharpe
  <date>              <dbl>      <dbl>     <dbl>
1 2014-12-31          0.312      0.312     0.312
2 2015-01-31          0.237      0.237     0.237
3 2015-02-28          0.300      0.300     0.300
```

7.8 Visualizing the Rolling Sharpe Ratio

Finally, we can start to visualize the Sharpe Ratio across the history of the portfolio.

We start with `highcharter` and `xts`.

```
highchart(type = "stock") %>%
  hc_title(text = "Rolling  24-Month Sharpe") %>%
  hc_add_series(rolling_sharpe_xts,
                name = "sharpe",
                color = "blue") %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE) %>%
  hc_add_theme(hc_theme_flat()) %>%
  hc_exporting(enabled = TRUE)
```

FIGURE 7.4: Rolling Sharpe highcharter

Figure 7.4 is confirming a trend that we noticed previously, that this portfolio has done quite well since November of 2016.

If we wish to visualize rolling Sharpe with `ggplot`, we can convert that xts object to a data frame and then pipe it, or we can start with one of our tidy `tibble` objects. The flow below starts with `xts` and converts to `tibble` with `tk_tbl()` so that we can get familiar with a new function.

```
rolling_sharpe_xts %>%
  tk_tbl(preserve_index = TRUE,
         rename_index = "date") %>%
  rename(rolling_sharpe = xts) %>%
  ggplot(aes(x = date,
             y = rolling_sharpe)) +
  geom_line(color = "cornflowerblue") +
  ggtitle("Rolling 24-Month Sharpe Ratio") +
  labs(y = "rolling sharpe ratio") +
```

```
  scale_x_date(breaks = pretty_breaks(n = 8)) +
   theme(plot.title = element_text(hjust = 0.5))
```

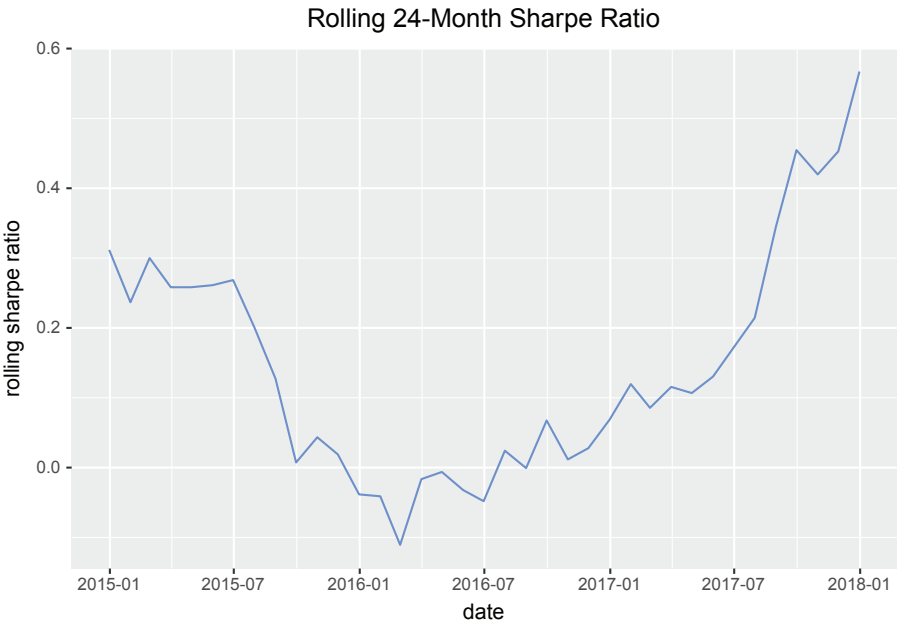


FIGURE 7.5: Rolling Sharpe ggplot

Figure 7.5 is showing the same data as Figure 7.4 but on a slightly more compressed scale. Would the scale variation lead us or an end user to think differently about this portfolio?

Those rolling charts allows us to see how our portfolio Sharpe Ratio decreased steadily into 2016, bottomed out, and then started to grind higher.

Let's take all this work and make it accessible via Shiny!

7.9 Shiny App Sharpe Ratio

The Sharpe Ratio Shiny app structure should feel familiar but have a quick look at the final app in Figure 7.6 and notice a few differences from our usual:

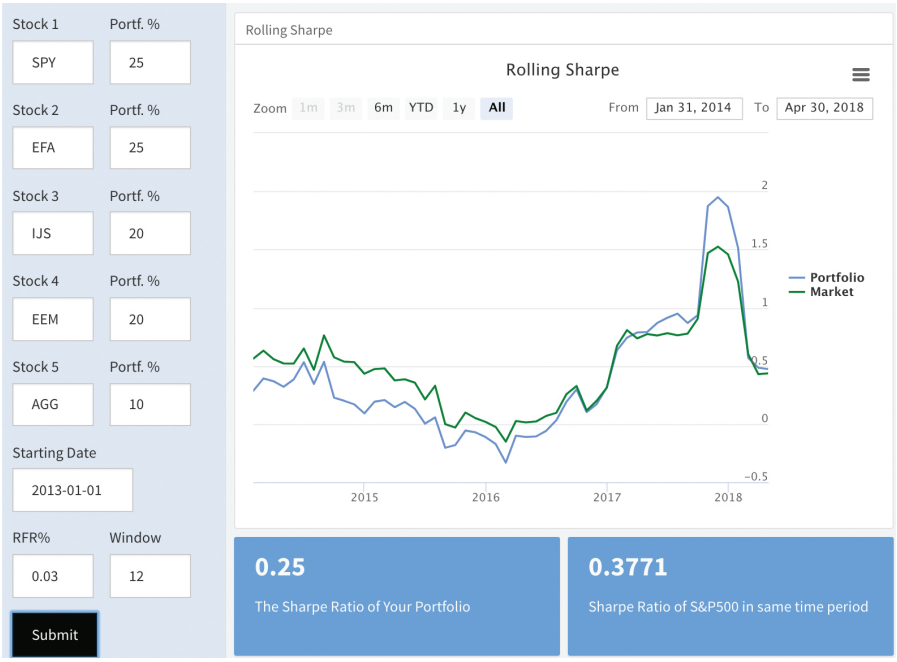


FIGURE 7.6: www.reproduciblefinance.com/shiny/sharpe-ratio/

Because the Sharpe Ratio is best understood by comparison we chart the rolling Sharpe Ratio of our portfolio alongside that of the S&P500, plus we have added two blue value boxes. That means we need to calculate the rolling and overall Sharpe for the S&P500 based on whatever starting date the user selects.

There are several calculations for this app and I divide them into market calculations and portfolio calculations.

In the chunks below, we run our market Sharpe Ratio equations, relying on the user-selected RFR, rolling window and starting date. The code flow runs through three reactives: `market_returns()`, which is used to find the `market_sharpe()` and the `market_rolling_sharpe()`.

First, we get the RFR, rolling window and market returns.

```
# market calculations

# Get the RFR from the end user
rfr <- eventReactive(input$go, {input$rfr/100})
```

```
# Get the rolling window from the end users
window <- eventReactive(input$go, {input$window})

# Calculate market returns based on starting date
market_returns <- eventReactive(input$go, {

    getSymbols("SPY", src = 'yahoo',
            from = input$date,
             auto.assign = TRUE,
             warnings = FALSE) %>%
    map(~Ad(get(.))) %>%
    reduce(merge) %>%
    `colnames<-`("SPY") %>%
    to.monthly(indexAt = "lastof",
               OHLC = FALSE) %>%
    Return.calculate(method = "log") %>%
    na.omit()
})
```

We now have a reactive object called `market_returns()`. Next, we calculate the overall and rolling market Sharpe Ratio of that object.

```
# Calculate market Sharpe Ratio
market_sharpe <- eventReactive(input$go, {

  SharpeRatio(market_returns(),
              Rf = rfr(),
              FUN = "StdDev")
})


# Calculate rolling market Sharpe Ratio
market_rolling_sharpe <- eventReactive(input$go, {

  rollapply(market_returns(),
            window(),
            function(x)
            SharpeRatio(x,
                        Rf = rfr(),
                        FUN = "StdDev")) %>%
  na.omit()
})
```

We will use two of those reactives in the main part of the app.

`market_sharpe()` appears in a blue value box and `market_rolling_sharpe()` appears on the `highcharter` chart.

Next, we calculate our *portfolio* Sharpe Ratio. The code flow is very similar to the previous, except we start by building portfolio returns.

```
# Run portfolio returns calculations

portfolio_returns <- eventReactive(input$go, {

  symbols <- c(input$stock1, input$stock2,
               input$stock3, input$stock4,
               input$stock5)

  validate(need(input$w1 + input$w2 +
                input$w3 + input$w4 +
                input$w5 == 100,
          "The portfolio weights must sum to 100%!"))

  w <- c(input$w1/100, input$w2/100,
         input$w3/100, input$w4/100,
         input$w5/100)

  getSymbols(symbols, src = 'yahoo', from = input$date,
             auto.assign = TRUE, warnings = FALSE) %>%
  map(~Ad(get(.))) %>%
  reduce(merge) %>%
  `colnames<-`(symbols) %>%
  to.monthly(indexAt = "lastof",
             OHLC = FALSE) %>%
  Return.calculate(method = "log") %>%
  na.omit() %>%
  Return.portfolio(weights = w)

})
```

We now have a reactive object called `portfolio_returns()`. Next, we calculate the overall and rolling market Sharpe Ratio of that object.

```
# Calculate portfolio Sharpe Ratio
portfolio_sharpe <- eventReactive(input$go, {

  validate(need(input$w1 + input$w2 + input$w3 +
                  input$w4 + input$w5 == 100,
                "------"))
```

```
  SharpeRatio(portfolio_returns(),
              Rf = rfr(),
              FUN = "StdDev")

})

# Calculate portfolio rolling Sharpe Ratio
portfolio_rolling_sharpe <- eventReactive(input$go, {

  rollapply(portfolio_returns(),
            window(),
            function(x) SharpeRatio(x,
                          Rf = rfr(),
                          FUN = "StdDev")) %>%
  na.omit()
})
```

We will use two objects from that code chunk in the main part of the app: `portfolio_sharpe()` and `portfolio_rolling_sharpe()`.

Note one crucial line in the above chunk: `validate(need(input$w1 + input$w2 + input$w3 + input$w4 + input$w5 == 100, ...))`. This is where we ensure that the weights sum to 100. If they do not, the user will see an error message that reads "The portfolio weights must sum to 100%!"

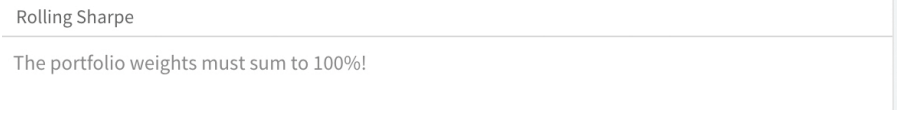


FIGURE 7.7: Weights Error Message

We have not included that error message shown in Figure 7.7 in any of our previous apps because we were introducing new concepts and did not want to clutter the code. It is a good idea to include messages like this as guideposts for our users.

Next, we pass our rolling market and rolling portfolio Sharpe Ratios to `renderHighchart()`, and add a legend to the end of the code flow so that the user can see which line is which.

```
# build one highchart that displays rolling Sharpe of both
# the portfolio and the market
renderHighchart({
```

```
  validate(need(input$go, "Please choose your portfolio assets,
                weights, rfr, rolling window and start date
                and click submit."))

  highchart(type = "stock") %>%
  hc_title(text = "Rolling Sharpe") %>%
  hc_add_series(portfolio_rolling_sharpe(),
                name = "Portfolio",
                color = "cornflowerblue") %>%
  hc_add_series(market_rolling_sharpe(),
                name = "Market",
                color = "green") %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE) %>%
  hc_exporting(enabled = TRUE) %>%
  # Add a legend
  hc_legend(enabled = TRUE,
            align = "right",
            verticalAlign = "middle",
            layout = "vertical")
})
```

Now we build and display the overall Sharpe Ratios of the portfolio and the market with two blue `valueBox()` aesthetics.

```
# value box for portfolio Sharpe Ratio
renderValueBox({
  valueBox(value = tags$p(round(portfolio_sharpe(), 4),
                          style = "font-size: 70%;"),
           color = "primary")
})
```

```
# value box for market Sharpe Ratio
renderValueBox({
  valueBox(value = tags$p(round(market_sharpe(), 4),
                          style = "font-size: 70%;"),
           color = "primary")
})
```

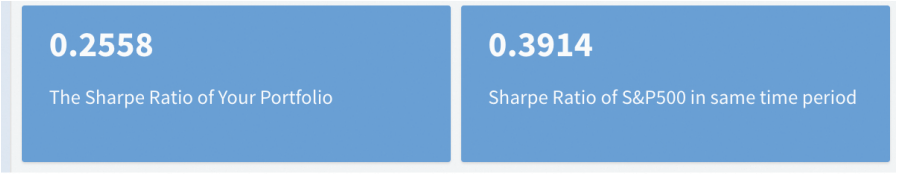


FIGURE 7.8: Sharpe Ratio Value Boxes

Figure 7.8 displays the value boxes. Maybe we love them and, more importantly, maybe our end users love them or despise them. The only way to know is to test and iterate, and that raises an important point about Shiny apps. Shiny involves experimentation because it depends on how end users experience the world. That is not a natural way to think about our work as a data scientist or quant. For example, as a data scientist or a quant, we might be perfectly satisfied to know that our code runs, calculates the correct rolling Sharpe Ratio and builds a nice data visualization. As a Shiny app builder, we must be concerned with whether an end user likes our work enough to interact with it, ideally more than once, and derive value from it. Considering the user experience is an exciting part of the Shiny challenge, indeed!

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