Authoring A Book with R Markdown

Yihui Xie

2016-03-15

Contents

Chapter 1

Introduction

This book and the package **bookdown** are still under active development, and should not be considered stable at the moment. You are welcome to experiment with this package, and feedback may be sent to https://github.com/rstudio/bookdown/issues or yihui@rstudio.com.

This book is a guide to authoring books with R Markdown (Allaire et al., 2016) and the R package **bookdown** (Xie, 2016a). It focuses on the features specific to writing books, long-form articles, or reports, such as

- How to typeset figures and tables, and cross-reference them;
- How to generate multiple output formats such as HTML, PDF, and e-books for a single book;
- How to customize the book templates and style different elements in a book;
- The editor support (in particular, the RStudio IDE);
- How to publish a book;

It is not a comprehensive introduction to R Markdown or the **knitr** package (Xie, 2016b), on top of which **bookdown** was built. To learn more about R Markdown, please check out the online documentation http://rmarkdown.rstudio.com. For **knitr**, please see Xie (2015b). You do not have to be an expert of the R language (R Core Team, 2015) to read this book, but you are expected to have some basic knowledge about R Markdown and **knitr**. For beginners, you may get started with the cheatsheets at https://www.rstudio.com/resources/cheatsheets/. To be able to customize the book templates and themes, you should be familiar with LaTeX, HTML and CSS.

1.1 Motivation

Markdown is a wonderful language to write relatively simple documents that contain elements like sections, paragraphs, lists, links, and images, etc. Pandoc (http://pandoc.org) has greatly extended the original Markdown syntax, and added quite a few useful new features, such as footnotes, citations, and tables. More importantly, Pandoc makes it possible to generate output documents of a large variety of formats from Markdown, including HTML, LaTeX/PDF, Word, and slides.

To write a relatively complicated document like a book, there are still a few useful features missing in Pandoc's Markdown at the moment, such as automatic numbering of figures and tables in the HTML output, cross-references of figures and tables, and fine control of the appearance of figures (e.g., currently it is impossible

to specify the alignment of images using the Markdown syntax). These are some of the problems that we have addressed in the **bookdown** package.

Under the constraint that we want to produce the book in multiple output formats, it is nearly impossible to cover all possible features specific to these output formats. For example, it may be difficult to reinvent a certain complicated LaTeX environment in the HTML output using the (R) Markdown syntax. Our main goal is not to replace *everything* with Markdown, but to cover *most* common functionalities required to write a relatively complicated document, and make the syntax of such functionalities consistent across all output formats, so that you only need to learn one thing and it works for all output formats.

Another goal of this project is to make it easy to produce books that look visually pleasant. Some nice existing examples include Gitbook (https://www.gitbook.com), Tufte CSS (http://edwardtufte.github.io/tufte-css/), and Tufte-LaTeX (https://tufte-latex.github.io/tufte-latex/). We hope to integrate these themes and styles into **bookdown**, so authors do not have to dive into the details of how to use a certain LaTeX class or how to configure CSS for HTML output.

1.2 Get started

The easiest way for beginners to get started with writing a book with R Markdown and **bookdown** is through the demo `bookdown-demo` on GitHub:

1. Fork or clone the GitHub repository https://github.com/rstudio/bookdown-demo if you are familiar with GIT and GitHub, or just download it as a Zip file then unzip it locally;
2. Install the RStudio IDE (http://www.rstudio.com) if you have not done so;
3. Open the `bookdown-demo` repository you cloned or downloaded in RStudio by clicking `bookdown-demo.Rproj`;
4. Install the R package **bookdown**:

   ```
   devtools::install_github("rstudio/bookdown")
   ```

5. Open the R Markdown file `index.Rmd` and click the button `Knit` on the toolbar of RStudio;

Now you should see the index page of this book demo in the RStudio Viewer. You may add or change the R Markdown files, come back to `index.Rmd`, and hit the `Knit` button again to preview the book. If you prefer not to use RStudio, you may also compile the book through command line. See the next section for details.

1.3 Usage

Normally, a book contains multiple chapters, and one chapter lives in one R Markdown file, with the filename extension `.Rmd`. Each R Markdown file must start immediately with the chapter title. All R Markdown files must be encoded in UTF-8. Here is an example (the bullets are the filenames, followed by the file content):

- 01-intro.Rmd

  ```
  # Introduction

  This chapter is an overview of the methods that
  we propose to solve an **important problem**.
  ```

- 02-literature.Rmd

```
# Literature

Here is a review of existing methods.
```

- 03-method.Rmd

```
# Methods

We describe our methods in this chapter.
```

- 04-application.Rmd

```
# Applications

Some _significant_ applications are demonstrated
in this chapter.

## Example one

## Example two
```

- 05-summary.Rmd

```
# Final Words

We have finished a nice book.
```

By default, **bookdown** merges all Rmd files by the order of filenames, e.g., `01-intro.Rmd` will appear before `02-literature.Rmd`. Filenames that start with an underscore `_` are skipped. If there exists an Rmd file named `index.Rmd`, it will always be treated as the first file when merging all Rmd files. The reason for this special treatment is that the HTML file `index.html` to be generated from `index.Rmd` is usually the default index file when you view a website, e.g., you are actually browsing http://yihui.name/index.html when you open http://yihui.name/.

You can override the above behavior by including a configuration file named `_bookdown.yml` in the book directory. It is a YAML file (https://en.wikipedia.org/wiki/YAML), and R Markdown users should be familiar with this format since it is also used to write the metadata in the beginning of R Markdown documents. You can use a field named `rmd_files` to define your own list and order of Rmd files for the book. For example,

```
rmd_files: ["index.Rmd", "abstract.Rmd", "intro.Rmd"]
```

In this case, **bookdown** will just use whatever you defined in this YAML field without any special treatments of `index.Rmd` or underscores. If you want both HTML and LaTeX/PDF output from the book, and use different Rmd files for HTML and LaTeX output, you may specify these files for the two output formats separately, e.g.,

```
rmd_files:
  html: ["index.Rmd", "abstract.Rmd", "intro.Rmd"]
  latex: ["abstract.Rmd", "intro.Rmd"]
```

Because **knitr** does not allow duplicate chunk labels in a source document, you need to make sure there are no duplicate labels in your book chapters, otherwise **knitr** will signal an error when knitting the merged Rmd file.

Although we have been talking about R Markdown files, the chapter files do not actually have to be R Markdown. They can be plain Markdown files (`.md`), and do not have to contain R code chunks at all. You can certainly use **bookdown** to compose novels or poems!

At the moment, there are three output formats that you may use: `bookdown::pdf_book`, `bookdown::gitbook`, and `bookdown::html_chapters`. There is a `bookdown::render_book()` function similar to `rmarkdown::render()`, but it was designed to render *multiple* Rmd documents into a book using the output format functions. You may either call this function from command line, or use it in the RStudio IDE. Here are some command line examples:

```
bookdown::render_book("foo.Rmd", "bookdown::gitbook")
bookdown::render_book("foo.Rmd", "bookdown::pdf_book")
bookdown::render_book("foo.Rmd", bookdown::gitbook(lib_dir = "book_assets"))
bookdown::render_book("foo.Rmd", bookdown::pdf_book(keep_tex = TRUE))
```

To use `render_book` and the output format functions in the RStudio IDE, you can define a YAML field named `knit` that takes the value `bookdown::render_book`, and the output format functions can be used in the `output` field, e.g.,

```
---
knit: "bookdown::render_book"
output:
  bookdown::gitbook:
    lib_dir: "book_assets"
  bookdown::pdf_book:
    keep_tex: yes
---
```

Then you can click the `Knit` button in RStudio to compile the Rmd files into a book.

1.4 Configuration

We have mentioned `rmd_files`, and there are more things you can configure for a book in `_bookdown.yml`:

- `book_filename`: the filename of the main Rmd file, i.e., the Rmd file that is merged from all chapters; by default, it is named `_main.Rmd`.
- `chapter_name`: (for HTML output only) either a character string to be prepended to the chapter number in the chapter title (e.g., `'Chapter '`), or an R function that takes the chapter number as the input and returns a string as the new chapter number (e.g., `!expr function(i) paste('Chapter', i)`).
- `before_chapter_script`: one or multiple R scripts to be executed before each chapter, e.g., you may want to clear the workspace before compiling each chapter, in which case you can use `rm(list = ls(all = TRUE))` in the R script.
- `after_chapter_script`: similar to `before_chapter_script`, and the R script is executed after each chapter.

- `edit`: a link that collaborators can click to edit the Rmd source document of the current page; this was designed primarily for Github repositories, since it is easy to edit arbitrary plain-text files on Github even in other people's repositories (if you do not have write access to the repository, Github will automatically fork it and let you submit a pull request after you finish editing the file). This link is should have `%s` in it, which will be substituted by the actual Rmd filename for each page.
  - Optionally, you can have a `text` field under the `edit` field to specify the text to which the link is attached.
- `output_dir`: the output directory of the book; this setting is read and used by `render_book()`.
- `clean`: a vector of files and directories to be cleaned by the `clean_book()` function.

Here is a sample `_bookdown.yml`:

```
book_filename: "my-book.Rmd"
chapter_name: "CHAPTER "
before_chapter_script: ["script1.R", "script2.R"]
after_chapter_script: "script3.R"
edit:
  link: https://github.com/rstudio/bookdown/edit/master/inst/examples/%s
  text: "Edit"
output_dir: "book-output"
clean: ["my-book.bbl", "R-packages.bib"]
```

Besides the configurations in `_bookdown.yml`, you can also specify some Pandoc-related configurations in the YAML metadata of the *first* Rmd file of the book, such as the title, author, and date of the book, etc. For example:

```
---
title: "Authoring A Book with R Markdown"
author: "Yihui Xie"
date: "2016-03-15"
knit: "bookdown::render_book"
output:
  bookdown::gitbook: default
documentclass: book
bibliography: ["book.bib", "packages.bib"]
biblio-style: apalike
link-citations: yes
---
```

1.5 Two rendering approaches

Merging all chapters into one Rmd file and knitting it is one way to render the book in **bookdown**. There is actually another way: you may knit each chapter in a *separate* R session, and **bookdown** will merge the Markdown output of all chapters to render the book. We call these two approaches "Merge and Knit" (MK) and "Knit and Merge" (KM), respectively. The differences between them may seem subtle, but can be fairly important depending on your use cases.

- The most significant difference is that MK runs *all* code chunks in all chapters in the same R session, whereas KM uses separate R sessions for individual chapters. For MK, the state of the R session from previous chapters is carried over to later chapters (e.g., objects created in previous chapters are available to later chapters, unless you deliberately deleted them); for KM, all chapters are isolated

from each other[1]. If you want each chapter to compile from a clean state, use the KM approach. It can be very tricky and difficult to restore a running R session to a completely clean state if you use the MK approach. For example, even you detach/unload packages loaded in a previous chapter, R will not clean up the S3 methods registered by these packages.

- One advantage of KM is that Rmd files that have not been updated since the last time the book was rendered will not be recompiled by default, unless you force all chapters to be recompiled via `render_book(force_knit = TRUE)`. This may save some time, but the speedup may not be very significant, since the major time is normally consumed by running code chunks. If time-consumed chunks are cached, the compilation time for MK and KM may be about the same.
- The KM approach will generate more files under the directory of Rmd files: each Rmd file will generate a Markdown output file (`.md`), and possibly a figure directory and a cache directory (e.g. `01-intro_files/` and `01-intro_cache/`). The MK approach only renders one Rmd file, so it only has one set of output files.
- For KM, whenever you change the output format (e.g., from HTML to PDF), you must recompile all chapters (`render_book(force_knit = TRUE)`), because the Markdown output files for one format may not work for another format. There is no such issue with the MK approach.

The default approach in **bookdown** is MK. To switch to KM, you either use the argument `new_session = TRUE` when calling `render_book()`, or set `new_session: yes` in the configuration file `_bookdown.yml`.

You can still configure `book_filename` in `_bookdown.yml` for the KM approach, but it should be a Markdown filename, e.g., `_main.md`, although the filename extension does not really matter, and you can even leave out the extension, e.g., just set `book_filename: _main`. All other configurations work for both MK and KM.

1.6 Some tips

Typesetting under the paging constraint (e.g., for LaTeX/PDF output) can be an extremely tedious and time-consuming job. I'd recommend you not to look at your PDF output frequently, since most of the time you are very unlikely to be satisfied: text may overflow into the page margin, figures may float too far away, and so on. Do not try to make things look right *immediately*, because you may be disappointed over and over again as you keep on revising the book, and things may be messed up again even if you only did some minor changes (see http://bit.ly/tbrLtx for a nice illustration).

If you want to preview the book, preview the HTML output. Work on the PDF book after you have finished the content of the book, and are very sure no major revisions will be required.

If certain code chunks in your R Markdown documents are time-consuming to run, you may cache them by adding the chunk option `cache = TRUE` in the chunk header, and you are recommended to label such code chunks as well, e.g.,

````
```{r important-computing, cache=TRUE}
````

We will talk about how to quickly preview books as you keep on editing it in Chapter 5. In short, you can use the `preview_chapter()` function to render a single chapter instead of the whole book. The function `serve_book()` makes it easy to live-preview HTML book pages: whenever you modify an Rmd file, the book can be recompiled and the browser can be automatically refreshed accordingly.

[1] Of course, no one can stop you from writing out some files in one chapter, and reading them in another chapter.
````

Chapter 2

Components

In this chapter, we show the syntax of some basic components of a book, including R code, figures, tables, citations, and so on. First we start with the syntax of Pandoc's Markdown.

2.1 Markdown syntax

We give a very brief introduction to Pandoc's Markdown in this section. Readers who are familiar with Markdown can skip this section. The comprehensive syntax of Pandoc's Markdown can be found on the Pandoc website http://pandoc.org.

You can make text *italic* by surrounding it with underscores or asterisks, e.g., `_text_` or `*text*`. For **bold** text, use two underscores (`__text__`) or asterisks (`**text**`). Text surrounded by `~` will be converted to a subscript (e.g., `H~2~SO~4~` renders H_2SO_4), and similarly, two carets like `^` produces a superscript (e.g., `ClO^-^` renders ClO^-). To mark text as `inline code`, use a pair of backticks, e.g., `` `code` ``[1]. Small caps can be produced by the HTML tag `span`, e.g., `<span style="font-variant:small-caps;">Small caps</span>` renders Small caps. Links are created using `[text](link)`, e.g., `[RStudio](http://www.rstudio.com)`, and 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.]`. We will talk about citations in Section 2.7. Section headers can be written after a number of pound signs, e.g.,

```
# First-level header

## Second-level header

### Third-level header
```

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

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

[1] To include literal backticks, just use more backticks outside, e.g., you can use two backticks to preserve one backtick inside: `` `code` ``.

The output is:

- one item
- one item
- one item
    - one item
    - one item

Ordered list items start with numbers (the rule for nested lists is the same as above), e.g.,

```
1. the first item
2. the second item
3. the third 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

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:

> "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
````

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:

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

To number and refer to equations, put them in the equation environments and assign labels to them, e.g.,

```
\begin{equation}
  f\left(k\right) = \binom{n}{k} p^k\left(1-p\right)^{n-k}
  \label{eq:binom}
\end{equation}
```

It renders the equation below:

$$f\left(k\right) = \binom{n}{k} p^k \left(1-p\right)^{n-k} \tag{2.1}$$

You may refer to it using `\@ref(eq:binom)`, e.g., see Equation 2.1. Due to some technical constraints, you must label your equations with the prefix `eq:` in **bookdown**.

2.2 R code

There are two types of R code in R Markdown/**knitr** documents: R code chunks, and inline R code. The syntax for the latter is `` `r R_CODE` ``, and it can be embedded inline with other document elements. R code chunks look like plain code blocks, but has `{r}` after the three backticks and (optionally) chunk options inside `{}`, e.g.,

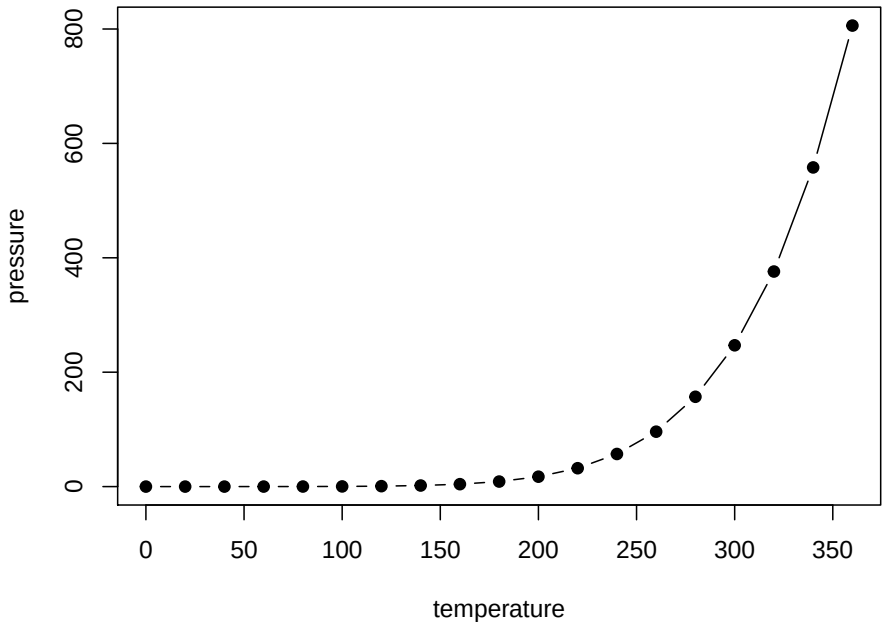
````
```{r chunk-label, echo = FALSE, fig.cap = 'A figure caption.'}
1 + 1
rnorm(10)  # 10 random numbers
plot(dist ~ speed, cars)  # a scatterplot
```
````

To learn more about **knitr** chunk options, see Xie (2015b) or http://yihui.name/knitr/options. For books, additional R code can be executed before/after each chapter; see `before_chapter_script` and `after_chapter_script` in Section 1.4.

2.3 Figures

By default, figures have no captions in the output generated by **knitr**, which means they will just be placed wherever they were generated in the R code. Below is such an example.

```
par(mar = c(4, 4, 0.1, 0.1))
plot(pressure, pch = 19, type = "b")
```



The disadvantage of typesetting figures in this way is that when there is not enough space on the current page to place a figure, it may either reach the bottom of the page (hence exceeds the page margin), or be pushed to the next page, leaving a large white margin at the bottom of the current page. That is basically why there are "floating environments" in LaTeX: elements that cannot be split over multiple pages (like figures) are put in floating environments, so they can float to a page that has enough space to hold them. There is also a disadvantage of floating things forward or backward, though. That is, readers may have to jump to a different page to find the figure mentioned on the current page. This is simply a natural consequence of having to typeset things on multiple pages of fixed sizes. This issue does not exist in HTML, however, since everything can be placed continuously on one single page (presumably with infinite height), and there is no need to split anything across multiple pages of the same page size.

If we assign a figure caption to a code chunk via the chunk option `fig.cap`, R plots will be put into figure environments, which will be automatically labeled and numbered, and can also be cross-referenced. The label of a figure environment is generated from the label of the code chunk, e.g., if the chunk label is `foo`, the figure label will be `fig:foo` (the prefix `fig:` is added before `foo`). To reference a figure, use the syntax `\@ref(label)`[2], where `label` is the figure label, e.g., `fig:foo`.

If you want to cross-reference figures or tables generated from a code chunk, please make sure the chunk label only contains alphanumeric characters (a-z, A-Z, 0-9) and dashes (-). Other characters do not qualify.

The chunk option `fig.asp` can be used to set the aspect ratio of plots, i.e., the ratio of figure height/width. If the figure width is 6 inches (`fig.width = 6`) and `fig.asp = 0.7`, the figure height will be automatically calculated from `fig.width * fig.asp = 6 * 0.7 = 4.2`. Figure 2.1 is an example using the chunk options `fig.asp = 0.7`, `fig.width = 6`, and `fig.align = 'center'`, generated from the code below:

```
par(mar = c(4, 4, 0.1, 0.1))
plot(pressure, pch = 19, type = "b")
```

[2]Do not forget the leading backslash!

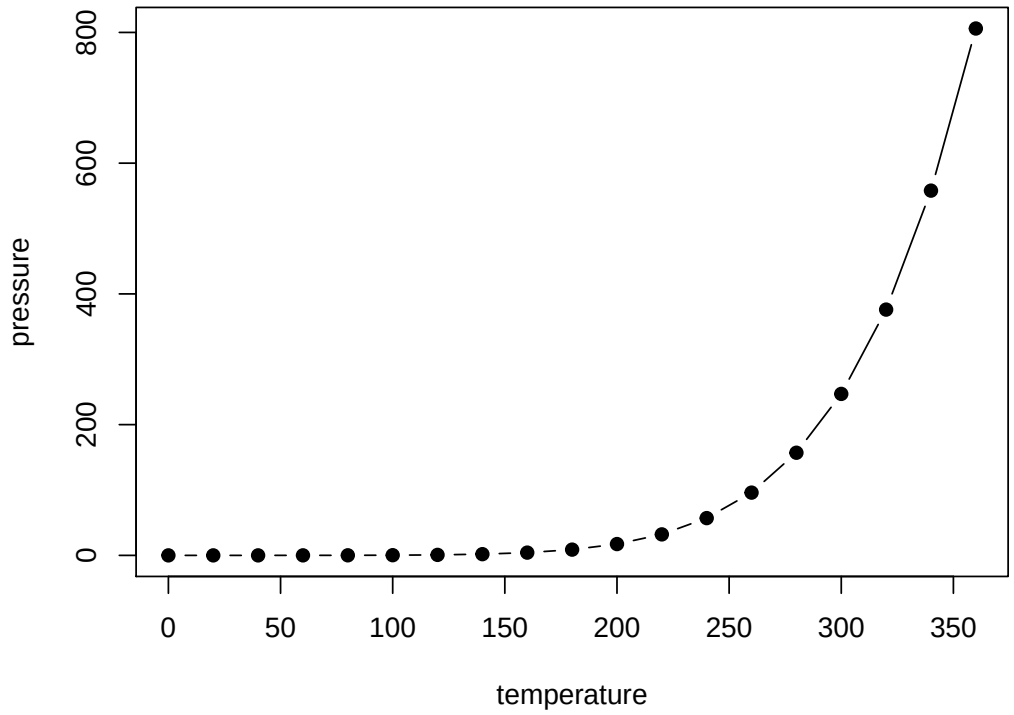


Figure 2.1: A figure example with the specified aspect ratio, width, and alignment.

The actual size of a plot is determined by the chunk options `fig.width` and `fig.height` (the size of the plot generated from a graphical device), and we can specify the output size of plots via the chunk options `out.width` and `out.height`. The possible value of these two options depends on the output format of the document. For example, `out.width = '30%'` is a valid value for HTML output, but not for LaTeX/PDF output. However, **knitr** will automatically convert a percentage value for `out.width` of the form `x%` to `(x / 100) \linewidth`, e.g., `out.width = '70%'` will be treated as `.7\linewidth` when the output format is LaTeX. This makes it possible to specify a relative width of a plot in a consistent manner. Figure 2.2 is an example of `out.width = 70%`.

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

If you want to put multiple plots in one figure environment, you must use the chunk option `fig.show = 'hold'` to hold multiple plots from a code chunk and include them in one environment. You can also place plots side by side if the sum of the width of all plots is smaller than or equal to the current line width. For example, if two plots have the same width `50%`, they will be placed side by side. Similarly, you can specify `out.width = '33%'` to arrange three plots on one line. Figure 2.3 is an example of two plots, each with a width `50%`.

```
par(mar = c(4, 4, 0.1, 0.1))
plot(pressure, pch = 19, type = "b")
plot(cars, pch = 19)
```

Sometimes you may have certain images that are not generated from R code, and you can include them them in R Markdown via the function `knitr::include_graphics()`. Figure 2.4 is an example of three **knitr** logos included in a figure environment. You may pass one or multiple image paths to the `include_graphics()` function, and all chunk options that apply to normal R plots also apply to these images, e.g., you can use `out.width = '33%'` to set the widths of these images in the output document.

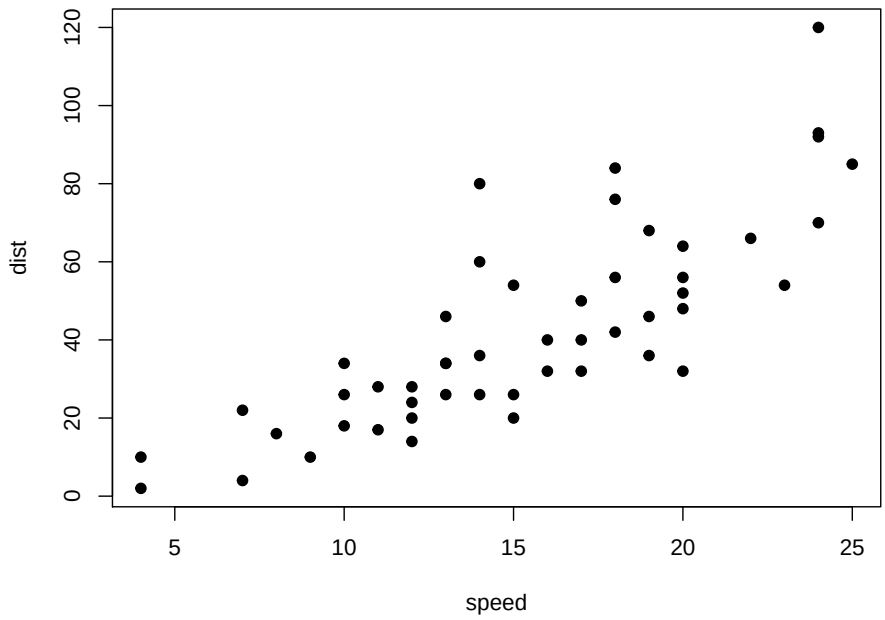


Figure 2.2: A figure example with a relative width 70%.

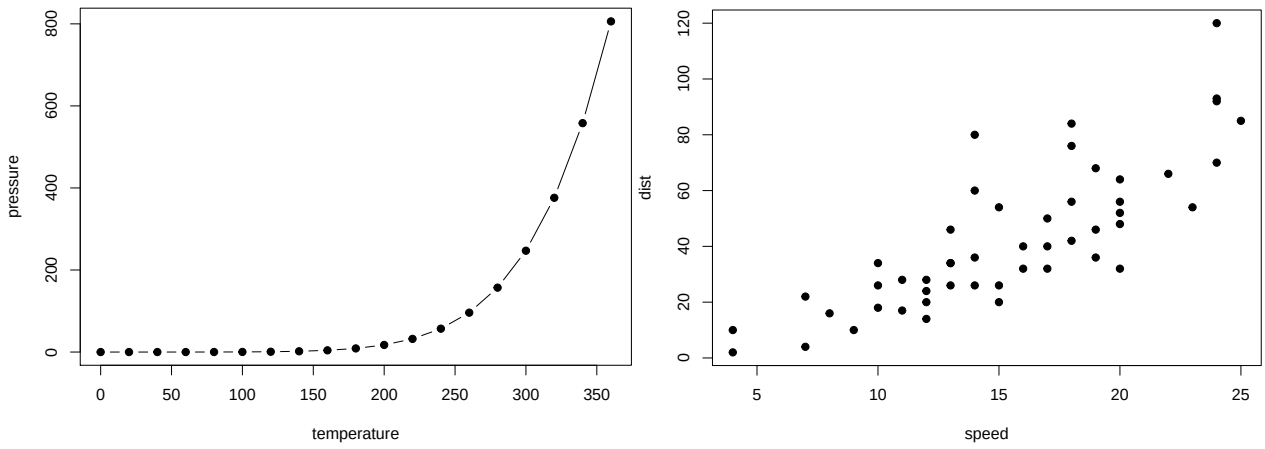


Figure 2.3: Two plots placed side by side.

Figure 2.4: Three knitr logos included in the document from an external PNG image file.

```
knitr::include_graphics(rep("images/knit-logo.png", 3))
```

There are a few advantages of using `include_graphics()`:

1. You do not need to worry about the document output format, e.g., when the output format is LaTeX, you may have to use the LaTeX command `\includegraphics{}` to include an image, and when the output format is Markdown, you have to use `![]()`. The function `include_graphics()` in **knitr** takes care of these details automatically.
2. The syntax for controlling the image attributes is the same as when images are generated from R code, e.g., chunk options `fig.cap`, `out.width`, and `fig.show` still have the same meanings.
3. `include_graphics()` is smart enough to use PDF graphics automatically when the output format is LaTeX and the PDF graphics files exist, e.g., an image path `foo/bar.png` can be automatically replaced with `foo/bar.pdf` if the latter exists. PDF images often have better qualities than raster images in LaTeX/PDF output. Of course, you can disable this feature by `include_graphics(auto_pdf = FALSE)` if you do not like it.

2.4 Tables

For now, the most convenient way to generate a table is the function `knitr::kable()`, because there are some internal tricks in **knitr** to make it work with **bookdown** and users do not have to know anything about these implementation details. We will explain how to use other packages and functions later in this section.

Like figures, tables with captions will also be numbered and can be referenced. The `kable()` function will automatically generate a label for a table environment, which is the prefix `tab:` plus the chunk label. For example, the table label for a code chunk with the label `foo` will be `tab:foo`, and we can still use the syntax `\@ref(label)` to reference the table. Table 2.1 is a simple example.

```
knitr::kable(
  head(mtcars, 10), booktabs = TRUE,
  caption = 'A table of the first 10 rows of the mtcars data.'
)
```

If you want to put multiple tables in a single table environment, just wrap the data objects (usually data frames in R) into a list. See Table 2.2 for an example.

```
knitr::kable(
  list(
    head(iris[,1:2],3),
```

Table 2.1: A table of the first 10 rows of the mtcars data.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

Table 2.2: A Tale of Two Tables.

Sepal.Length	Sepal.Width
5.1	3.5
4.9	3.0
4.7	3.2

	mpg	cyl	disp
Mazda RX4	21.0	6	160
Mazda RX4 Wag	21.0	6	160
Datsun 710	22.8	4	108
Hornet 4 Drive	21.4	6	258
Hornet Sportabout	18.7	8	360

```
    head(mtcars[,1:3],5)
  ),
  caption = 'A Tale of Two Tables.', booktabs = TRUE
)
```

When you do not want a table to float in PDF, you may use the LaTeX package **longtable**, which can break a table across multiple pages. To use **longtable**, just pass `longtable = TRUE` to `kable()`, and make sure to include `\usepackage{longtable}` in the LaTeX preamble (see Section 4.1 for how to customize the LaTeX preamble). Of course, this is irrelevant to HTML output, since tables in HTML do not need to float.

```
knitr::kable(
  iris[1:100, ], longtable = TRUE, booktabs = TRUE,
  caption = 'A table generated by the longtable package.'
)
```

Table 2.3: A table generated by the longtable package.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa

5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa
5.4	3.4	1.7	0.2	setosa
5.1	3.7	1.5	0.4	setosa
4.6	3.6	1.0	0.2	setosa
5.1	3.3	1.7	0.5	setosa
4.8	3.4	1.9	0.2	setosa
5.0	3.0	1.6	0.2	setosa
5.0	3.4	1.6	0.4	setosa
5.2	3.5	1.5	0.2	setosa
5.2	3.4	1.4	0.2	setosa
4.7	3.2	1.6	0.2	setosa
4.8	3.1	1.6	0.2	setosa
5.4	3.4	1.5	0.4	setosa
5.2	4.1	1.5	0.1	setosa
5.5	4.2	1.4	0.2	setosa
4.9	3.1	1.5	0.2	setosa
5.0	3.2	1.2	0.2	setosa
5.5	3.5	1.3	0.2	setosa
4.9	3.6	1.4	0.1	setosa
4.4	3.0	1.3	0.2	setosa
5.1	3.4	1.5	0.2	setosa
5.0	3.5	1.3	0.3	setosa
4.5	2.3	1.3	0.3	setosa
4.4	3.2	1.3	0.2	setosa
5.0	3.5	1.6	0.6	setosa
5.1	3.8	1.9	0.4	setosa
4.8	3.0	1.4	0.3	setosa
5.1	3.8	1.6	0.2	setosa
4.6	3.2	1.4	0.2	setosa
5.3	3.7	1.5	0.2	setosa
5.0	3.3	1.4	0.2	setosa
7.0	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor
6.9	3.1	4.9	1.5	versicolor
5.5	2.3	4.0	1.3	versicolor
6.5	2.8	4.6	1.5	versicolor
5.7	2.8	4.5	1.3	versicolor
6.3	3.3	4.7	1.6	versicolor

4.9	2.4	3.3	1.0	versicolor
6.6	2.9	4.6	1.3	versicolor
5.2	2.7	3.9	1.4	versicolor
5.0	2.0	3.5	1.0	versicolor
5.9	3.0	4.2	1.5	versicolor
6.0	2.2	4.0	1.0	versicolor
6.1	2.9	4.7	1.4	versicolor
5.6	2.9	3.6	1.3	versicolor
6.7	3.1	4.4	1.4	versicolor
5.6	3.0	4.5	1.5	versicolor
5.8	2.7	4.1	1.0	versicolor
6.2	2.2	4.5	1.5	versicolor
5.6	2.5	3.9	1.1	versicolor
5.9	3.2	4.8	1.8	versicolor
6.1	2.8	4.0	1.3	versicolor
6.3	2.5	4.9	1.5	versicolor
6.1	2.8	4.7	1.2	versicolor
6.4	2.9	4.3	1.3	versicolor
6.6	3.0	4.4	1.4	versicolor
6.8	2.8	4.8	1.4	versicolor
6.7	3.0	5.0	1.7	versicolor
6.0	2.9	4.5	1.5	versicolor
5.7	2.6	3.5	1.0	versicolor
5.5	2.4	3.8	1.1	versicolor
5.5	2.4	3.7	1.0	versicolor
5.8	2.7	3.9	1.2	versicolor
6.0	2.7	5.1	1.6	versicolor
5.4	3.0	4.5	1.5	versicolor
6.0	3.4	4.5	1.6	versicolor
6.7	3.1	4.7	1.5	versicolor
6.3	2.3	4.4	1.3	versicolor
5.6	3.0	4.1	1.3	versicolor
5.5	2.5	4.0	1.3	versicolor
5.5	2.6	4.4	1.2	versicolor
6.1	3.0	4.6	1.4	versicolor
5.8	2.6	4.0	1.2	versicolor
5.0	2.3	3.3	1.0	versicolor
5.6	2.7	4.2	1.3	versicolor
5.7	3.0	4.2	1.2	versicolor
5.7	2.9	4.2	1.3	versicolor
6.2	2.9	4.3	1.3	versicolor
5.1	2.5	3.0	1.1	versicolor
5.7	2.8	4.1	1.3	versicolor

If you decide to use other packages to generate tables, you have to make sure the label for the table environment appears in the beginning of the table caption in the form `(\#label)`, where `label` must have the prefix `tab:`. You have to be very careful about the *portability* of the table generating function: it should work for both HTML and LaTeX output automatically, so it must consider the output format internally (check `knitr::opts_knit$get('pandoc.to')`). When writing out an HTML table, the caption must be written in the `<caption></caption>` tag. For simple tables, `kable()` should suffice. If you have to create

complicated tables (e.g., with certain cells spanning across multiple columns/rows), you will have to take the aforementioned issues into consideration.

2.5 Cross-references

We have explained how cross-references work for figures (Section 2.3) and tables (Section 2.4). In fact, you can also reference sections using the same syntax `\@ref(label)`, where `label` is the section ID. By default, Pandoc will generate an ID for all section headers, e.g., a section `# Hello World` will have an ID `hello-world`. We recommend you to manually assign an ID to a section header to make sure you do not forget to update the reference label after you change the section header. To assign an ID to a section header, simply add `{#id}` to the end of the section header.

When a referenced label cannot be found, you will see two question marks like **??**, as well as a warning message in the R console when rendering the book.

2.6 Custom blocks

You can generate custom blocks using the `block` engine in **knitr**, i.e., the chunk option `engine = 'block'`, or the more compact syntax ```` ```{block} ````. This engine should be used in conjunction with the chunk option `type`, which takes a character string. When the `block` engine is used, it generates a `<div>` to wrap the chunk content if the output format is HTML, and a LaTeX environment if the output is LaTeX. The `type` option specifies the class of the `<div>` and the name of the LaTeX environment. For example, the HTML output of this chunk

````
```{block, type='FOO'}
Some text for this block.
```
````

will be this:

```
<div class="FOO">
Some text for this block.
</div>
```

and the LaTeX output will be this:

```
\begin{FOO}
Some text for this block.
\end{FOO}
```

It is up to the book author how to define the style of the block. You can define the style of the `<div>` in CSS and include it in the output via the `includes` option in the YAML metadata. Similarly, you may define the LaTeX environment via `\newenvironment` and include the definition in the LaTeX output via the `includes` option. For example, we may save the following style in a CSS file, say, `style.css`:

```
div.FOO {
  font-weight: bold;
  color: red;
}
```

And the YAML metadata of the R Markdown document can be:

```
---
output:
  bookdown::html_chapters:
    includes:
      in_header: style.css
---
```

We have defined a few types of blocks for this book to show notes, tips, and warnings, etc. Below are some examples:

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License versions 2 or 3. For more information about these matters see http://www.gnu.org/licenses/.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License versions 2 or 3. For more information about these matters see http://www.gnu.org/licenses/.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License versions 2 or 3. For more information about these matters see http://www.gnu.org/licenses/.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License versions 2 or 3. For more information about these matters see http://www.gnu.org/licenses/.

R is free software and comes with ABSOLUTELY NO WARRANTY. You are welcome to redistribute it under the terms of the GNU General Public License versions 2 or 3. For more information about these matters see http://www.gnu.org/licenses/.

2.7 Citations

Although Pandoc supports multiple ways of writing citations, we recommend you to use BibTeX databases because they work best with LaTeX/PDF output. Pandoc can process other types of bibliography databases

with the utility `pandoc-citeproc` (https://github.com/jgm/pandoc-citeproc), but it may not render certain bibliography items correctly (especially in case of multiple authors). With BibTeX databases, you will be able to define the bibiliography style if it is required by a certain publisher or journal.

A BibTeX database is a plain-text file (with the conventional filename extension `.bib`) that consists of bibliography entries like this:

```
@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 = {2015},
  url = {https://www.R-project.org/},
}
```

A bibliography entry starts with `@type{`, where `type` may be `article`, `book`, `manual`, and so on. Then there is a citation key, like `R-base` in the above example. To cite an entry, use `@key` or `[@key]` (the latter puts the citation in braces), e.g., `@R-base` is rendered to R Core Team (2015), and `[@R-base]` generates "(R Core Team, 2015)". If you are familiar with the **natbib** package in LaTeX, `@key` is basically `\citet{key}`, and `[@key]` is equivalent to `\citep{key}`.

There are a number of fields in a bibliography entry, such as `title`, `author`, and `year`, etc. You may see https://en.wikipedia.org/wiki/BibTeX for possible types of entries and fields in BibTeX.

There is a helper function `write_bib()` in **knitr** to generate BibTeX entries automatically for R packages. Note it only generates one BibTeX entry for the package itself at the moment, whereas a package may contain multiple entries in the `CITATION` file, and some entries are about the publications related to the package. These entries are ignored by `write_bib()`.

```
# the second argument can be a .bib file
knitr::write_bib(c("knitr", "stringr"), "")
```

```
@Manual{R-knitr,
  title = {knitr: A General-Purpose Package for Dynamic Report Generation in R},
  author = {Yihui Xie},
  year = {2016},
  note = {R package version 1.12.22},
  url = {http://yihui.name/knitr/},
}
@Manual{R-stringr,
  title = {stringr: Simple, Consistent Wrappers for Common String Operations},
  author = {Hadley Wickham},
  year = {2015},
  note = {R package version 1.0.0},
  url = {https://CRAN.R-project.org/package=stringr},
}
```

Once you have got one or multiple `.bib` files, you may use the field `bibliography` in the YAML metadata of your R Markdown document, and you can also specify the bibliopgraphy style via `biblio-style` (this only applies to PDF output), e.g.,

```
---
bibliography: ["one.bib", "another.bib", "yet-another.bib"]
```

```
biblio-style: "apalike"
link-citations: true
---
```

The field `link-citations` can be used to add internal links from the citation text of the author-year style to the bibilography entry in the HTML output.

2.8 HTML widgets

HTML widgets (Vaidyanathan et al., 2016) were originally designed for HTML output only, and they require the availability of JavaScript, so they will not work in non-HTML output formats, such as LaTeX/PDF. Before **knitr** v1.13, you will get an error when you render HTML widgets to an output format that is not HTML. Since **knitr** v1.13, HTML widgets will be rendered automatically as screenshots taken via the **webshot** package (Chang, 2016). Of course, you need to install the **webshot** package. Additionally, you have to install PhantomJS (http://phantomjs.org), since it is what **webshot** uses to capture screenshots. Both **webshot** and PhantomJS can be installed automatically from R:

```
install.packages("webshot")
webshot::install_phantomjs()
```

The function `install_phantomjs()` works for Windows, OS X, and Linux. You may also choose to download and install PhantomJS by yourself, if you are familiar with modifying the system environment variable `PATH`.

When **knitr** detects an HTML widget object in a code chunk, it either renders the widget normally when the current output format is HTML, or save the widget as an HTML page and calls **webshot** to capture the screen of the HTML page when the output format is not HTML. Here is an example of a table created from the **DT** package (Xie, 2015a):

```
DT::datatable(iris)
```

If you are reading this book as web pages now, you should see an interactive table generated from the above code chunk, e.g., you may sort the columns and search in the table. If you are reading a non-HTML version of this book, you should see a screenshot of the table. The screenshot may look a little different with the actual widget rendered in the web browser, due to the difference between a real web browser and PhantomJS' virtual browser.

There are a number of **knitr** chunk options related to screen-capturing. First, if you are not satisfied with the quality of the automatic screenshots, or want a screenshot of the widget of a particular state (e.g., after you click on a certain column of a table), you may capture the screen manually, and provide your own screenshot via the chunk option `screenshot.alt` (alternative screenshots). This option takes the paths of images. If you have multiple widgets in a chunk, you can provide a vector of image paths. When this option is present, **knitr** will no longer call **webshot** to take automatic screenshots.

Second, sometimes you may want to force **knitr** to use static screenshots instead of rendering the actual widgets even on HTML pages. In this case, you can set the chunk option `screenshot.force = TRUE`, and widgets will always be rendered as static images. Note you can still choose to use automatic or custom screenshots.

Third, **webshot** has some options to control the automatic screenshots, and you may specify these options via the chunk option `screenshot.opts`, which takes a list like `list(delay = 2, cliprect = 'viewport')`. See the help page `?webshot::webshot` for the full list of possible options, and the package vignette `vignette('intro', package = 'webshot')` has illustrated the effect of these options. Here the `delay` option can be important for widgets that take long time to render: `delay` specifies the number of

Show 10 entries Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

Showing 1 to 10 of 150 entries

Previous 1 2 3 4 5 ... 15 Next

Figure 2.5: A table widget rendered via the DT package.

seconds to wait before PhantomJS takes the screenshot. If you see an incomplete screenshot, you may want to specify a longer delay (the default is 0.2 seconds).

Fourth, if you feel it is slow to capture the screenshots, or do not want to do it every time the code chunk is executed, you may use the chunk option `cache = TRUE` to cache the chunk. Caching works for both HTML and non-HTML output formats.

Screenshots behave like normal R plots in the sense that many chunk options related to figures also apply to screenshots, including `fig.width`, `fig.height`, `out.width`, `fig.cap`, and so on. So you can specify the size of screenshots in the output document, and assign figure captions to them as well. The image format of the automatic screenshots can be specified via the chunk option `dev`, and possible values are `pdf`, `png`, and `jpeg`. The default for PDF output is `pdf`, and it is `png` for other types of output. Note `pdf` may not work as faithfully as `png`: sometimes there are certain elements on an HTML page that fail to render to the PDF screenshot, so you may want to use `dev = 'png'` even for PDF output. It depends on specific cases of HTML widgets, and you can try both `pdf` and `png` (or `jpeg`) before deciding which format is more desirable.

2.9 Web pages and Shiny apps

Similar to HTML widgets, arbitrary web pages can be embedded in the book. You can use the function `knitr::include_url()` to include a web page through its URL. When the output format is HTML, an `iframe` is used[3]; in other cases, **knitr** tries to take a screenshot of the web page (or use the custom screenshot you provided). All chunk options are the same as those for HTML widgets. One option that may require your special attention is the `delay` option: HTML widgets are rendered locally, so usually they are fast to load for PhantomJS to take screenshots, but an arbitrary URL may take longer to load, so you may want to use a larger `delay` value, e.g., use the chunk option `screenshot.opts = list(delay = 5)`.

A related function is `knitr::include_app()`, which is very similar to `include_url()`, and it was designed for embedding Shiny apps via their URLs in the output. Its only difference with `include_url()` is that it automatically adds a query parameter `?showcase=0` to the URL, if no other query parameters are present in the URL, to disable the Shiny showcase mode, which is unlikely to be useful for screenshots or iframes. If you do want the showcase mode, just use `include_url()` instead of `include_app()`. Below is a Shiny app example (Figure 2.6):

```
knitr::include_app("https://yihui.shinyapps.io/miniUI/", height = "600px")
```

Again, you will see a live app if you are reading an HTML version of this book, and a static screenshot if you are reading other types of formats. The above Shiny app was created using the **miniUI** package (Cheng, 2016), which provides layout functions that are particularly nice for Shiny apps on small screens. If you use normal Shiny layout functions, you are likely to see vertical and/or horizontal scrollbars in the iframes because the page size is too big to fit an iframe. When the default width of the iframe is too small, you may use the chunk option `out.width` to change it. For the height of the iframe, use the `height` argument of `include_url()`/`include_app()`.

Shiny apps may take even longer to load than usual URLs. You may want to use a conservative value for the `delay` option, e.g., 10. Needless to say, `include_url()` and `include_app()` require a working Internet connection, unless you have previously cached the chunk (but web pages inside iframes still will not work without an Internet connection).

[3] An `iframe` is basically a box on one web page to embed another web page.

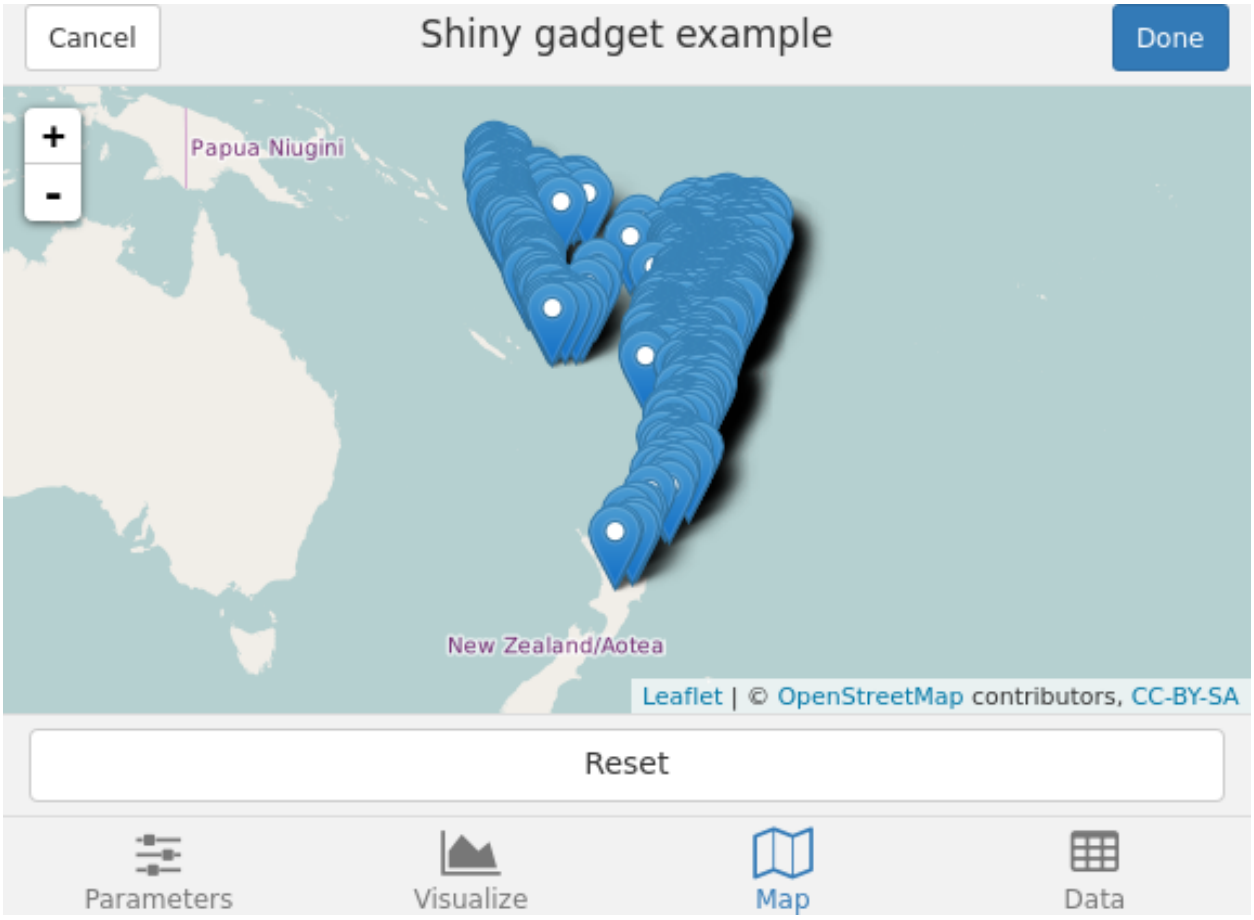


Figure 2.6: A Shiny app created via the miniUI package: https://yihui.shinyapps.io/miniUI/.

Chapter 3

Output Formats

The **bookdown** package primarily supports three types of output formats: HTML, LaTeX/PDF, and e-books. In this chapter, we introduce the possible options for these formats. Output formats can be specified either in the YAML metadata of the first Rmd file of the book, or in a separate YAML file named `_output.yml` under the root directory of the book. A brief example of the former (output formats are specified in the `output` field of the YAML metadata):

```
---
title: "An Impressive Book"
author: "Li Lei and Han Meimei"
output:
  bookdown::gitbook:
    lib_dir: assets
    split_by: section
    config:
      toolbar:
        position: static
  bookdown::pdf_book:
    keep_tex: yes
  bookdown::html_chapters:
    css: toc.css
documentclass: book
---
```

An example of `_output.yml`:

```
bookdown::gitbook:
  lib_dir: assets
  split_by: section
  config:
    toolbar:
      position: static
bookdown::pdf_book:
  keep_tex: yes
bookdown::html_chapters:
  css: toc.css
```

In this case, all formats should be at the top level, instead of under an `output` field. You do not need the three dashes `---` in `_output.yml`.

3.1 HTML

The main difference between rendering a book (using **bookdown**) with rendering a single R Markdown document (using **rmarkdown**) to HTML is that a book will generate multiple HTML pages by default — normally one HTML file per chapter. This makes it easier to bookmark a certain chapter or share its URL with others as you read the book, and faster to load a book into the web browser. Currently we have provided a number of different styles for HTML output: the GitBook style, the Bootstrap style, and the Tufte style.

3.1.1 GitBook style

The GitBook style was borrowed GitBook, a project launched by Friendcode, Inc (https://www.gitbook.com) and dedicated to helping authors write books with Markdown. It provides a beautiful style, with a layout consisting of a sidebar showing the table of contents on the left, and the main body of a book on the right. The design is responsive to the window size, e.g., the navigation buttons are displayed on the left/right of the book body when the window is wide enough, and collapsed into the bottom when the window is narrow to give readers more horizontal space to read the book body.

We have made several improvements over the original GitBook project. The most significant one is that we replaced the Markdown engine with R Markdown v2 based on Pandoc, so that there are a lot more features for you to use when writing a book. For instance,

- You can embed R code chunks and inline R expressions in Markdown, and this makes it easy to create reproducible documents and frees you from synchronizing your computation with its actual output (**knitr** will take care of it automatically);
- The Markdown syntax is much richer: you can write anything that Pandoc's Markdown supports, such as LaTeX math expressions and citations;
- You can embed interactive content in the book (for HTML output only), such as HTML widgets and Shiny apps;

We have also added some useful features in the user interface that we will introduce in detail soon. The output format function for the GitBook style in **bookdown** is `gitbook()`. Below are its arguments:

```
gitbook(fig_caption = TRUE, number_sections = TRUE, self_contained = FALSE,
  lib_dir = "libs", ..., split_by = c("chapter", "chapter+number", "section",
    "section+number", "rmd", "none"), config = list())
```

Most arguments are passed to `rmarkdown::html_document()`, including `fig_caption`, `lib_dir`, and `...`. You can check out the help page of `rmarkdown::html_document()` for the full list of possible options. We strongly recommend you to use `fig_caption = TRUE` for two reasons: 1) it is important to explain your figures with captions; 2) enabling figure captions means figures will be placed in floating environments when the output is LaTeX, otherwise you may end up with a lot of white space on certain pages. The format of figure/table numbers depends on if sections are numbered or not: if `number_sections = TRUE`, these numbers will be of the format `X.i`, where `X` is the chapter number, and `i` in an incremental number; if sections are not numbered, all figures/tables will be numbered sequentially through the book from 1, 2, ..., N. Note in either case, figures and tables will be numbered independently.

Among all possible arguments in `...`, you are most likely to use the `css` argument to provide one or more custom CSS files to tweak the default CSS style. There are a few arguments of `html_document()` that have been hard-coded in `gitbook()` and you cannot change them: `toc = TRUE` (there must be a table of contents), `theme = NULL` (not using any Bootstrap themes), and `template` (there will be an internal GitBook template).

Please note if you change `self_contained = TRUE` to make self-contained HTML pages, the total size of all HTML files can be significantly increased since there are many JS and CSS files tha have to be embedded in every single HTML file.

Besides these `html_document()` options, `gitbook()` has two other arguments: `split_by` and `config`. The `split_by` argument specifies how you want to split the HTML output into multiple pages, and its possible values are:

- `rmd`: use the base filenames of the input Rmd files to create the HTML filenames, e.g., generate `chapter3.html` for `chapter3.Rmd`;
- `none`: do not split the HTML file (the book will be a single HTML file);
- `chapter`: split the file by the first-level headers;
- `section`: split the file by the second-level headers;
- `chapter+number` and `section+number`: similar to `chapter` and `section`, but the files will be numbered;

For `chapter` and `section`, the HTML filenames will be determined by the header identifiers, e.g., the filename for the first chapter with a chapter title `# Introduction` will be `introduction.html` by default. For `chapter+number` and `section+number`, the chapter/section numbers will be prepended to the HTML filenames, e.g., `1-introduction.html` and `2-1-literature.html`. The header identifier is automatically generated from the header text by default[1], and you can manually specify an identifier using the syntax `{#your-custom-id}` after the header text, e.g.,

```
# An Introduction {#introduction}

The default identifier is `an-introduction` but we changed
it to `introduction`.
```

There are several sub-options in the `config` option for you to tweak some details in the user interface. Recall that all output format options (not only for `bookdown::gitbook`) can be either passed to the format function if you use the command-line interface `bookdown::render_book()`, or written in the YAML metadata. We display the default sub-options of `config` in the `gitbook` format as YAML metadata below (note they are indented under the `config` option):

```
bookdown::gitbook:
  config:
    toc:
      collapse: subsection
      scroll_highlight: true
      before: null
      after: null
    toolbar:
      position: fixed
    edit:
      link: null
      text: null
    download: null
    search: true
    fontsettings:
      theme: white
      family: sans
      size: 2
    sharing:
      facebook: yes
      twitter: yes
```

[1] To see more details on how an identifier is automatically generated, see the `auto_identifiers` extension in Pandoc's documentation http://pandoc.org/README.html#header-identifiers

```
    google: no
    weibo: no
    instapper: no
    vk: no
    all: ['facebook', 'google', 'twitter', 'weibo', 'instapaper']
```

The `toc` option controls the behavior of the table of contents (TOC). You can collapse some items initially when a page is loaded via the `collapse` option. Its possible values are `subsection`, `section`, `none` (or `null`). This option can be helpful if your TOC is very long and has more than three levels of headings: `subsection` means collapsing all TOC items for subsections (X.X.X), `section` means those items for sections (X.X) so only the top-level headings are displayed initially, and `none` means not collapsing any items in TOC. For those collapsed TOC items, you can toggle their visibility by clicking their parent TOC items. For example, you can click a chapter title in the TOC to show/hide its sections.

The `scroll_highlight` option in `toc` means whether to enable highlighting of TOC items as you scroll the book body (by default this feature is enabled). Whenever a new header comes into the current viewport as you scroll down/up, the corresponding item in TOC on the left will be highlighted.

Since the sidebar has a fixed width, when an item in TOC is truncated because the heeading text is too wide, you can hover the cursor over it to see a tooltip showing the full text.

You may add more items before and after the TOC using the HTML tag `<li>`. These items will be separated from TOC using a horizontal divider. You can use the pipe character `|` so that you do not need to escape any characters in these items following the YAML syntax, e.g.,

```
toc:
  before: |
    <li><a href="...">My Awesome Book</a></li>
    <li><a href="...">John Smith</a></li>
  after: |
    <li><a href="https://github.com/rstudio/bookdown">
    Proudly published with bookdown</a></li>
```

As you navigate through different HTML pages, we will try to preserve the scroll position of TOC. Normally you will see the scrollbar in TOC at a fixed position even if you navigate to the next page. However, if the TOC item for the current chapter/section is not visible when the page is loaded, we will automatically scroll TOC to make it visible to you.

The GitBook style has a toolbar at the top of each page that allows you to dynamically change the book settings. The `toolbar` option has a sub-option `position`, which can take values `fixed` or `static`. The default is that the toolbar will be fixed at the top of the page, so even if you scroll down the page, the toolbar is still visible there. If it is `static`, the toolbar will not scroll with the page, i.e., once you scroll away, you will no longer see it.

The first button on the toolbar can toggle the visibility of the sidebar. You can also hit the `S` key on your keyboard to do the same thing. The GitBook style can remember the visibility status of the sidebar, e.g., if you closed the sidebar, it will remain closed the next time you open the book. In fact, the GitBook style remembers many other settings as well, such as the search keyword and the font settings.

The second button on the toolbar is the search button. Its keyboard shortcut is `F` (Find). When the button is clicked, you will see a search box at the top of the sidebar. As you type in the box, the TOC will be filtered to display the sections that match the search keyword. Now you can use the arrow keys `Up`/`Down` to highlight the next keyword on the current page. When you click the search button again (or hit `F` outside the search box), the search keyword will be emptied and the search box will be hidden. To disable searching, set the option `search: no` in `config`.

The third button is for font/theme settings. You can change the font size (bigger or smaller), the font family (serif or sans serif), and the theme (`White`, `Sepia`, or `Night`). These settings can be changed via the `fontsettings` option.

The `edit` option is the same as the option that we mentioned in Section 1.4. If it is not empty, an edit button will be added to the toolbar. This was designed for potential contributors of the book to contribute to the book by editing the book on GitHub after clicking the button and send pull requests.

If your book has other output formats for readers to download, you may provide the `download` option so that a download button can be added to the toolbar. This option takes either a character vector, or a list of character vectors with the length of each vector being 2. When it is a character vector, it should be either a vector of filenames, or filename extensions, e.g., both of the following settings are okay:

```
download: ["book.pdf", "book.epub"]
download: ["pdf", "epub", "mobi"]
```

When you only provide the filename extensions, the filename is derived from the book filename of the configuration file `_bookdown.yml` (Section 1.4). When `download` is `null`, `gitbook()` will look for PDF, EPUB, and MOBI files in the book output directory, and automatically add them to the `download` option. If you just want to suppress the download button, use `download: no`. All files for readers to download will be displayed in a dropdown menu, and the filename extensions are used as the menu text. When the only available format for readers to download is PDF, the download button will be a single PDF button instead of a drop-down menu.

An alternative form for the value of the `download` option is a list of length-2 vectors, e.g.,

```
download: [["book.pdf", "PDF"], ["book.epub", "EPUB"]]
```

You can also write it as:

```
download:
  - ["book.pdf", "PDF"]
  - ["book.epub", "EPUB"]
```

Each vector in the list consists of the filename and the text to be displayed in the menu. Compared to the first form, this form allows you to customize the menu text, e.g., you may have two different copies of PDF for readers to download and you will need to make the menu items different.

On the right of the toolbar, there are some buttons to share the link on social network websites such as Twitter, Facebook, and Google+. You can use the `sharing` option to decide which buttons to enable. If you want to get rid of these buttons entirely, just use `sharing: null` (or `no`).

Finally, there are a few more top-level options in the YAML metadata that can be passed to the GitBook HTML template via Pandoc. They may not have clear visible effects on the HTML output, but they may be useful when you deploy the HTML output as a website. These options include:

- `description`: A character string to be written to the `content` attribute of the tag `<meta name="description" content="">` in the HTML head (if missing, the title of the book will be used). This can be useful for the purpose of search engine optimization (SEO);
- `apple-touch-icon`: A path to an icon (e.g., a PNG image). This is for iOS only: when the website is added to the Home screen, the link is represented by this icon.
- `apple-touch-icon-size`: The size of the icon (by default, 152 x 152 pixels).
- `favicon`: A path to the "favorite icon". Typically this icon is displayed in the browser's address bar, or in front of the page title on the tab if the browser support tabs.

Below we show some sample YAML metadata (again, please note these are *top-level* options):

```
---
title: "An Awesome Book"
author: "John Smith"
description: "This book introduces the ABC theory, and ..."
apple-touch-icon: "touch-icon.png"
apple-touch-icon-size: 120
favicon: "favicon.ico"
---
```

3.1.2 Bootstrap style

If you have used R Markdown before, you should be familiar with the Bootstrap style (http://getbootstrap.com), which is the default style of the HTML output of R Markdown. The output format function in **rmarkdown** is `html_document()`, and we have a corresponding format `html_book()` in **bookdown** using `html_document()` as the base format. In fact, there is a more general format `html_chapters()` in **bookdown** and `html_book()` is just its special case:

```
html_chapters(toc = TRUE, number_sections = TRUE, fig_caption = TRUE, lib_dir = "libs",
  template = bookdown_file("templates/default.html"), ..., base_format = rmarkdown::html_document,
  page_builder = build_chapter, split_by = c("section+number", "section",
    "chapter+number", "chapter", "rmd", "none"))
```

Note it has a `base_format` argument that takes a base output format function, and `html_book()` is basically `html_chapters(base_format = rmarkdown::html_document)`. All arguments of `html_book()` are passed to `html_chapters()`:

```
html_book(...)
```

That means you can use most arguments of `rmarkdown::html_document`, such as `toc` (whether to show the table of contents), `number_sections` (whether to number section headings), and so on. Again, check the help page of `rmarkdown::html_document` to see the full list of possible options. Note the argument `self_contained` is hard-coded to `FALSE` internally, so you cannot change the value of this argument. We have explained the argument `split_by` in the previous section.

The arguments `template` and `page_builder` are for advanced users, and you do not need to understand them unless you have strong need to customize the HTML output, and those many options provided by `rmarkdown::html_document()` still do not give you what you want.

If you want to pass a different HTML template to the `template` argument, the template must contain three pairs of HTML comments, and each comment must be on a separate line:

- `<!--bookdown:title:start-->` and `<!--bookdown:title:end-->` to mark the title section of the book. This section will be placed only on the first page of the rendered book;
- `<!--bookdown:toc:start-->` and `<!--bookdown:toc:end-->` to mark the table of contents section, which will be placed on all HTML pages;
- `<!--bookdown:body:start-->` and `<!--bookdown:body:end-->` to mark the HTML body of the book, and the HTML body will be split into multiple separate pages. Recall that we merge all R Markdown or Markdown files, render them into a single HTML file, and split it;

You may open the default HTML template to see where these comments were inserted:

```
bookdown:::bookdown_file("templates/default.html")
# you may use file.edit() to open this file
```

Once you know how **bookdown** works internally to generate multiple-page HTML output, it will be easier to understand the argument `page_builder`, which is a function to compose each individual HTML page using the HTML fragments extracted from the above comment tokens. The default value of `page_builder` is a function `build_chapter` in **bookdown**, and its source code is relatively simple (ignore those internal functions like `button_link()`):

```
build_chapter = function(
  head, toc, chapter, link_prev, link_next, rmd_cur, html_cur, foot
) {
  # add a has-sub class to the <li> items that has sub lists
  toc = gsub('^(<li>)(.+<ul>)$', '<li class="has-sub">\\2', toc)
  paste(c(
    head,
    '<div class="row">',
    '<div class="col-sm-12">',
    toc,
    '</div>',
    '</div>',
    '<div class="row">',
    '<div class="col-sm-12">',
    chapter,
    '<p style="text-align: center;">',
    button_link(link_prev, 'Previous'),
    edit_link(rmd_cur),
    button_link(link_next, 'Next'),
    '</p>',
    '</div>',
    '</div>',
    foot
  ), collapse = '\n')
}
```

Basically, this function takes a number of components like the HTML head, the table of contents, the chapter body, and so on, and it is expected to return a character string which is the HTML source of a complete HTML page. You may manipulate all components in this function using text-processing functions like `gsub()` and `paste()`.

What the default page builder does is to put TOC in the first row, the body in the second row, navigation buttons at the bottom of the body, and concatenate them with the HTML head and foot. Here is a sketch of the HTML source code that may help you understand the output of `build_chapter()`:

```
<html>
  <head>
    <title>A Nice Book</title>
  </head>
  <body>

    <div class="row">TOC</div>

    <div class="row">
```

```
      CHAPTER BODY
      <p>
        <button>PREVIOUS</button>
        <button>NEXT</button>
      </p>
    </div>

  </body>
</html>
```

For all HTML pages, the main difference is the chapter body, and most of the rest of elements are the same. The default output from `html_book()` will include the Bootstrap CSS and JavaScript files in the `<head>` tag.

The TOC is often used for navigation purposes. In the GitBook style, the TOC is displayed in the sidebar. For the Bootstrap style, we did not apply a special style to it, so it is shown as a plain unordered list (in the HTML tag `<ul>`). It is easy to turn this list into a navigation bar with some CSS techniques. We have provided a CSS file `toc.css` in this package that you can use, and you can find it here:

```
bookdown:::bookdown_file("examples/css/toc.css")
```

You may copy this file to the root directory of your book, and apply it to the HTML output via the `css` option, e.g.,

```
---
output:
  bookdown::html_book:
    toc: yes
    css: toc.css
---
```

There are many possible ways to turn `<ul>` lists to navigation menus if you do a little bit searching on the web, and you can choose a menu style that you like. The `toc.css` we just mentioned is a style with white menu texts on a black background, and supports sub-menus (e.g., section titles are displayed as dropdown menus under chapter titles).

As a matter of fact, you can get rid of the Bootstrap style in `html_document()` if you set the `theme` option to `null`, and you are free to apply arbitrary styles to the HTML output using the `css` option (and possibly the `includes` option if you want to include arbitrary content in the HTML head/foot).

3.1.3 Tufte style

Like the Bootstrap style, the Tufte style is provided by an output format `tufte_html_book()`, which is also a special case of `html_chapters()` using `tufte::tufte_html()` as the base format. Please see the **tufte** package (Xie and Allaire, 2016) if you are not familiar with the Tufte style. Basically, it is a layout with a main column on the left and a margin column on the right. The main body is in the main column, and the margin column is used to place footnotes, margin notes, references, and margin figures, and so on.

All arguments of `tufte_html_book()` have exactly the same meanings as `html_book()`, e.g., you can also customize the CSS via the `css` option. There are a few elements that are specific to the Tufte style, though, such as margin notes, margin figures, and fullwidth figures. These elements require special syntax to generate, and please see the documentation of the **tufte** package. Note you do not need to do anything special to footnotes and references (just use the normal Markdown syntax `^[footnote]` and `[@citation]`), since they will be automatically put in the margin. A brief YAML example of the `tufte_html_book` format:

```
---
output:
  bookdown::tufte_html_book:
    toc: yes
    css: toc.css
---
```

3.2 LaTeX/PDF

We strongly recommend you to use an HTML output format instead of LaTeX when you develop a book, since you will not be too distracted by the typesetting details, which can bother you a lot if you constantly look at the PDF output of a book. Leave the job of careful typesetting to the very end (ideally after you have really finished the content of the book).

The LaTeX/PDF output format is provided by `pdf_book()` in **bookdown**. There is not a significant difference between `pdf_book()` and the `pdf_document()` format in **rmarkdown**. The main purpose of `pdf_book()` is to resolve the labels and cross-references written using the syntax described in Sections 2.3, 2.4, and 2.5. If the only output format that you want for a book is LaTeX/PDF, you may use the syntax specific to LaTeX, such as `\label{}` to label figures/tables/sections, and `\ref{}` to cross-reference them via their labels, because Pandoc supports LaTeX commands in Markdown. However, the LaTeX syntax is not portable to other output formats, such as HTML and e-books. That is why we introduced the syntax `(\#label)` for labels and `\@ref(label)` for cross-references.

There are some top-level YAML options that will be applied to the LaTeX output. For a book, you may change the default document class to `book` (the default is `article`), and specify a bibliography style required by your publisher. A brief YAML example:

```
---
documentclass: book
bibliography: [book.bib, packages.bib]
biblio-style: apalike
---
```

The `pdf_book()` format is a general format like `html_chapters`, and it also has a `base_format` argument:

```
pdf_book(toc = TRUE, number_sections = TRUE, fig_caption = TRUE, ..., base_format = rmarkdown::pdf_docu
```

You can change the `base_format` function to other output format functions, and **bookdown** has provided a simple wrapper function `tufte_book2()`, which is basically `pdf_book(base_format = tufte::tufte_book)`, to produce a PDF book using the Tufte PDF style (again, see the **tufte** package).

3.3 E-Books

Currently **bookdown** provides two e-book formats, EPUB and MOBI. Books of these formats can be read on devices like smartphones, tablets, or special e-readers such as Kindle.

3.3.1 EPUB

To create an EPUB book, you can use the `epub_book()` format. It has some options in common with `rmarkdown::html_document()`:

```
epub_book(fig_width = 5, fig_height = 4, dev = "png", fig_caption = TRUE, number_sections = TRUE,
  toc = FALSE, toc_depth = 3, stylesheet = NULL, cover_image = NULL, metadata = NULL,
  chapter_level = 1, epub_version = c("epub3", "epub"), md_extensions = NULL,
  pandoc_args = NULL)
```

The option `toc` is turned off because the e-book reader can often figure out a TOC automatically from the book, so it is not necessary to add a few pages for the TOC. There are a few options specific to EPUB:

- `stylesheet`: It is similar to the `css` option in HTML output formats, and you can customize the appearance of elements using CSS;
- `cover_image`: The path to the cover image of the book;
- `metadata`: The path to an XML file for the metadata of the book (see Pandoc documentation for more details);
- `chapter_level`: Internally an EPUB book is a series of "chapter" files, and this option determines the level by which the book is split into these files. This is similar to the `split_by` argument of HTML output formats we mentioned in Section 3.1, but an EPUB book is a single file, and you will not see these "chapter" files directly. The default level is the first level, and if you set it to 2, it means the book will be organized by section files internally, which may make the reader faster to load the book;
- `epub_version`: Version 3 or 2 of EPUB;

An EPUB book is essentially a collection of HTML pages, e.g., you can apply CSS rules to its elements, embed images, insert math expressions (because MathML is partially supported), and so on. Figure/table captions, cross-references, custom blocks, and citations mentioned in Chapter 2 also work for EPUB. You may compare the EPUB output of this book to the HTML output, and you will see the only major difference is the visual appearance.

There are several EPUB readers available, including Calibre (https://www.calibre-ebook.com), Apple's iBooks, and Google Play Books.

3.3.2 MOBI

MOBI e-books can be read on Amazon's Kindle devices. Pandoc does not support MOBI output natively, but Amazon has provided a tool named KindleGen (https://www.amazon.com/gp/feature.html?docId=1000765211) to create MOBI books from other formats, including EPUB and HTML. We have provided a simple wrapper function `kindlegen()` in **bookdown** to call KindleGen to convert an EPUB book to MOBI. This requires you to download KindleGen first, and make sure the KindleGen executable can be found via the system environment variable `PATH`.

3.4 A single document

Sometimes you may not want to write a book, but just a single long-form article or report instead. Usually what you do is call `rmarkdown::render()` with a certain output format. The main features missing there are the automatic numbering of figure/table captions, and cross-referencing figures/tables/sections. We have factored out these features from **bookdown**, so that you can use them without having to prepare a book of multiple Rmd files.

The functions `html_document2()`, `tufte_html2()`, `pdf_document2()`, `tufte_handout2()`, and `tufte_book2()` are designed for this purpose. If you render an R Markdown document with the output format, say, `bookdown::html_document2`, you will get figure/table numbers and be able to cross-reference them in the single HTML page using the syntax described in Chapter 2.

Although the `gitbook()` format was designed primarily for books, you can actually also apply it to a single R Markdown document. The only difference is that there will be no search button on the single page output, because you can simply use the searching tool of your web browser to find text (e.g., press `Ctrl + F` or `Command + F`). You may also want to set the option `split_by` to `none` to only generate a single output page, in which case there will not be any navigation buttons, since there are no other pages to navigate to. You can still generate multiple-page HTML files if you like. Another option you may want to use is `self_contained = TRUE` when it is only a single output page.

Chapter 4

Customization

As we mentioned in the very beginning of this book, you are expected to have some basic knowledge about R Markdown, and we have been focusing on introducing the **bookdown** features instead of **rmarkdown**. In fact, R Markdown is highly customizable, and there are many options that you can use to customize the output document. Depending on how much you want to customize the output, you may use some simple options in the YAML metadata, or just replace the entire Pandoc template.

4.1 YAML options

For most types of output formats, you can customize the syntax highlighting styles using the `highlight` option of the specific format. Currently, the possible styles are `default`, `tango`, `pygments`, `kate`, `monochrome`, `espresso`, `zenburn`, and `haddock`. For example, you can choose the `tango` style for the `gitbook` format:

```
---
output:
  bookdown::gitbook:
    highlight: tango
---
```

For HTML output formats, you are most likely to use the `css` option to provide your own CSS stylesheets to customize the appearance of HTML elements. There is an option `includes` that applies to more formats, including HTML and LaTeX. The `includes` option allows your to insert arbitrary custom content before and/or after the body of the output. It has three sub options: `in_header`, `before_body`, and `after_body`. You need to know the basic structure of an HTML or LaTeX document to understand these options. The source of an HTML document looks like this:

```
<html>

  <head>
  <!-- head content here, e.g. CSS and JS -->
  </head>

  <body>
  <!-- body content here -->
  </body>

</html>
```

The `in_header` option takes a file path and inserts it into the `<head>` tag. The `before_body` file will be inserted right below the opening `<body>` tag, and `after_body` is inserted before the closing tag `</body>`.

A LaTeX source document has a similar structure:

```
\documentclass{book}

% LaTeX preamble
% insert in_header here

\begin{document}
% insert before_body here

% body content here

% insert after_body here
\end{document}
```

The `includes` option is very useful and flexible. For HTML output, it means you can insert arbitrary HTML code to the output. For example, when you have LaTeX math expressions rendered via the MathJax library in the HTML output, and want to number the equations in the `equation` environment, you can create a text file that contains the following code:

```
<script type="text/x-mathjax-config">
MathJax.Hub.Config({
  TeX: { equationNumbers: { autoNumber: "AMS" } }
});
</script>
```

Let's assume the file is named `mathjax-number.html`, and it is in the root directory of your book (the directory that contains all your Rmd files). You can insert this file into the HTML head via the `in_header` option, e.g.,

```
---
output:
  bookdown::gitbook:
    includes:
      in_header: mathjax-number.html
---
```

If you use the HTML output format `html_book` or `gitbook` in **bookdown**, this has already been set up in the HTML templates, so you do not really need to insert such a file in the HTML head. You only need to do it for other HTML output formats.

Similarly, if you are familiar with LaTeX, you can add arbitrary LaTeX code to the preamble. That means you can use any LaTeX packages and set up any package options for your book. For example, this book used the `in_header` option to use a few more LaTeX packages like **booktabs** (for better-looking tables) and **longtable** (for tables that span across multiple pages), and apply a fix to an XeLaTeX problem that links on graphics do not work:

```
\usepackage{booktabs}
\usepackage{longtable}

\ifxetex
```

```
  \usepackage{letltxmacro}
  \setlength{\XeTeXLinkMargin}{1pt}
  \LetLtxMacro\SavedIncludeGraphics\includegraphics
  \def\includegraphics#1#{% #1 catches optional stuff (star/opt. arg.)
    \IncludeGraphicsAux{#1}%
  }%
  \newcommand*{\IncludeGraphicsAux}[2]{%
    \XeTeXLinkBox{%
      \SavedIncludeGraphics#1{#2}%
    }%
  }%
\fi
```

The above LaTeX code is saved in a file `preamble.tex`, and the YAML metadata looks like this:

```
---
output:
  bookdown::pdf_book:
    includes:
      in_header: preamble.tex
---
```

4.2 Theming

Sometimes you may want to change the overall theme of the output, and usually this can be done through the `in_header` option described in the previous section, or the `css` option if the output is HTML. Some output formats have their unique themes, such as `gitbook`, `tufte_html_book`, and `tufte_book2`, and you may not want to customize these themes too much. By comparison, the output formats `html_book()` and `pdf_book()` are not tied to particular themes and more customizable.

As mentioned in Section 3.1.2, the default style for `html_book()` is the Bootstrap style. The Bootstrap style actually has several built-in themes that you can use, including `default`, `cerulean`, `journal`, `flatly`, `readable`, `spacelab`, `united`, `cosmo`, `lumen`, `paper`, `sandstone`, `simplex`, and `yeti`. You can set the theme via the `theme` option, e.g.,

```
---
output:
  bookdown::html_book:
    theme: united
---
```

If you do not like any of these Bootstrap styles, you can set `theme` to `null`, and apply your own CSS through the `css` or `includes` option.

For `pdf_book()`, besides the `in_header` option mentioned in the previous section, another possibility is to change the document class. There are many possible LaTeX classes for books, such as **memoir** (https://www.ctan.org/pkg/memoir), **amsbook** (https://www.ctan.org/pkg/amsbook), KOMA-Script (https://www.ctan.org/pkg/koma-script) and so on. A brief sample of the YAML metadata specifying the `scrbook` class from the KOMA-Script package:

```
---
documentclass: scrbook
output:
```

```
  bookdown::pdf_book:
    template: null
---
```

Some publishers (e.g., Springer and Chapman & Hall/CRC) have their own LaTeX style or class files. You may try to change the `documentclass` option to use their document classes, although typically it is not as simple as that. You may end up with using `in_header`, or even design a custom Pandoc LaTeX template to accommodate these document classes.

4.3 Templates

When Pandoc converts Markdown to another output format, it uses a template under the hood. The template is a plain text file that contains some variables of the form `$variable$`. These variables will be replaced by their values generated by Pandoc. Below is a very brief template for HTML output:

```
<html>
  <head>
    <title>$title$</title>
  </head>

  <body>
  $body$
  </body>
</html>
```

It has two variables `title` and `body`. The value of `title` comes from the `title` field of the YAML metadata, and `body` is the HTML code generated from the body of the Markdown input document. For example, suppose we have a Markdown document:

```
---
title: A Nice Book
---

# Introduction

This is a **nice** book!
```

If we use the above template to generate an HTML document, its source code will be like this:

```
<html>
  <head>
    <title>A Nice Book</title>
  </head>

  <body>

  <h1>Introduction</h1>

  <p>This is a <strong>nice</strong> book!</p>

  </body>
</html>
```

The actual HTML, LaTeX, and EPUB templates are more complicated, but the idea is the same. You just need to know what variables are available: some variables are built-in Pandoc variables, and some can be either defined by users in the YAML metadata, or passed from the command line option `-V` or `--variable`. Some variables only make sense to specific output formats, e.g., the `documentclass` variable is only used in LaTeX output. Please see the documentation of Pandoc to learn more about these variables, and you can find all default Pandoc templates from the GitHub repository https://github.com/jgm/pandoc-templates.

Note that for HTML output, **bookdown** requires some additional comment tokens in the template, and we have explained them in Section 3.1.2.

4.4 Internationalization

If the language of your book is not English, you will need to translate certain English words and phrases into your language, such as the words "Figure" and "Table" when figures/tables are automatically numbered in the HTML output. Internationalization may not be an issue for LaTeX output, since some LaTeX packages can automatically translate these terms into the local language, such as the **ctexcap** package for Chinese.

TODO…

There is one caveat when you write in a language that uses multibyte characters, such as Chinese, Japanese, and Korean (CJK): Pandoc cannot generate identifiers from section headings that are pure CJK characters, so you will not be able to cross-reference sections (they do not have labels), unless you manually assign identifiers to them by appending `{#identifier}` to the section heading, where `identifier` is an arbitrary identifier you choose.

Chapter 5

Editing

In this chapter, we explain how to edit, build, preview, and serve the book locally. You can use any text editors to edit the book, and we will show some tips about the RStudio IDE. We will introduce the underlying R functions for building, previewing, and serving the book before we introduce the editor, so that you really understand what happens behind the scenes when you click a certain button in the RStudio IDE, and can also customize other editors calling these functions.

5.1 Build the book

To build all Rmd files into a book, you can call the `render_book()` function in **bookdown**. Below are the arguments of `render_book()`:

```
render_book(input, output_format = NULL, ..., clean = TRUE, envir = parent.frame(),
  output_dir = "_book", new_session = FALSE, force_knit = FALSE, preview = FALSE)
```

The most important argument is `output_format`, which can take a character string (e.g., `'bookdown::gitbook'`), or an object returned by an output format function (e.g., `bookdown::gitbook(lib_dir = 'assets')`). You can leave this argument empty, and the default output format will be the first output format specified in the YAML metadata of the first Rmd file or a separate YAML file `_output.yml`, as mentioned in Section 1.4. If you plan to generate multiple output formats for a book, you are recommended to specify all formats in `_output.yml`.

Once all formats are specified in `_output.yml`, it is easy to write an R or Shell script or Makefile to compile the book. Below is a simple example of using a Shell script to compile a book to HTML (with the GitBook style) and PDF:

```
#!/usr/bin/env Rscript

bookdown::render_book("index.Rmd", "bookdown::gitbook")
bookdown::render_book("index.Rmd", "bookdown::pdf_book")
```

The Shell script does not work on Windows (not strictly true, though), but hopefully you get the idea.

The argument `...` is passed to the output format function. Arguments `clean` and `envir` are passed to `rmarkdown::render()`, to decide whether to clean up the intermediate files, and specify the environment to evaluate R code, respectively.

The output directory of the book can be specified via the `output_dir` argument. By default, the book is generated to the `_book` directory. This can also be changed via the `output_dir` field in the configuration

file `_bookdown.yml`, so that you do not have to specify it multiple times for rendering a book to multiple output formats. The `new_session` argument has been explained in Section 1.5; `force_knit` applies to `new_session = TRUE` only, and it determines whether to knit all Rmd files even if their output Markdown files are newer than the Rmd source files. When you set `preview = TRUE`, only the Rmd files specified in the `input` argument are rendered, which can be convenient when previewing a certain chapter, since you do not recompile the whole book, but when publishing a book, this argument should certainly be set to `FALSE`.

A number of output files will be generated by `render_book()`. Sometimes you may want to clean up the book directory and start all over again, e.g., remove the figure and cache files that were generated automatically from **knitr**. The function `clean_book()` was designed for this purpose. By default, it tells you which files are possibly output files that you can delete. If you have looked at this list of files, and are sure no files were mistakenly identified as output files (you certainly do not want to delete an input file that you created by hand), you can delete all of them using `bookdown::clean_book(TRUE)`. Since deleting files is a relatively dangerous operation, we would recommend you to maintain your book through version control tools such as GIT, or a service that supports backup and restoration, so you will not lose certain files forever after you delete them by mistake.

5.2 Preview a chapter

Building the whole book can be slow when the size of the book is big. Two things can affect the speed of building a book: the computation in R code chunks, and the conversion from Markdown to other formats via Pandoc. The former can be improved by enabling caching in **knitr** using the chunk option `cache = TRUE`, and there is not much you can do to make the latter faster. However, you can choose to render only one chapter using the function `preview_chapter()` in **bookdown**, and usually this will be much faster than rendering the whole book. Only the Rmd files passed to `preview_chapter()` will be rendered.

Previewing the current chapter is helpful when you are only focusing on this chapter, since you can quickly see the actual output as you add more content or revise the chapter. Although the preview works for all output formats, we recommend you to preview the HTML output.

One downside of previewing a chapter is that the cross-references to other chapters will not work, since **bookdown** knows nothing about other chapters in this case. That is a reasonably small price to pay for the gain in speed.

5.3 Serve the book

Instead of running `render_book()` or `preview_chapter()` over and over again, you can actually live preview the book in the web browser, and the only thing you need to do is save the Rmd file. The function `serve_book()` in **bookdown** can start a local web server to serve the HTML output based on the **servr** package (Xie, 2016c).

```
serve_book(dir = ".", output_dir = "_book", preview = TRUE, in_session = TRUE,
  ...)
```

You just pass the root directory of the book to the `dir` argument, and this function will start a local web server so you can view the book output using the server. The default URL to access the book output is `http://127.0.0.1:4321`. If you run this function in an interactive R session, this URL will be automatically opened in your web browser. If you are in the RStudio IDE, the RStudio Viewer will be used as the default web browser, so you will be able to write the Rmd source files and preview the output in the same environment (e.g, source on the left and output on the right).

The server will listen to changes in the book root directory: whenever you modify any files in the book directory, `serve_book()` can detect the changes, recompile the Rmd files, and refresh the web browser

automatically. If the modified files do not include Rmd files, it just refreshes the browser (e.g., if you only updated a certain CSS file). This means once the server is launched, all you have to do next is simply write the book and save the files. Compilation and preview will take place automatically as you save files.

If it does not really take too much time to recompile the whole book, you may set the argument `preview = FALSE`, so that every time you update the book, the whole book is recompiled, otherwise only the modified chapters are recompiled via `preview_chapter()`.

The `...` arguments are passed to `servr::httw()`, and please see its help page to know all possible options, such as `port` and `daemon`. There are pros and cons using `in_session = TRUE` or `FALSE`:

- For `in_session = TRUE`, you will have access to all objects created in the book in the current R session: if you use a daemonized server (via the argument `daemon = TRUE`), you can check the objects at any time when the current R session is not busy; otherwise you will have to stop the server before you can check the objects. This can be useful when you need to interactively explore the R objects in the book. The downside of `in_session = TRUE` is that the output may be different with the book compiled from a fresh R session, because the state of the current R session may not be clean.
- For `in_sesion = FALSE`, you do not have access to objects in the book from the current R session, but the output is more likely to be reproducible since everything is created from new R sessions. Since this function is only for previewing purposes, the cleanness of the R session may not be a big concern.

You may choose `in_session = TRUE` or `FALSE` depending on your specific use cases. Eventually, you should run `render_book()` from a fresh R session to generate a reliable copy of the book output.

5.4 RStudio IDE

When you compile an R Markdown document in the RStudio IDE, the default function called by RStudio is `rmarkdown::render()`, which is not what we want for books. To call the function `bookdown::render_book()` instead, you can set the `knit` field to be `bookdown::render_book` in the YAML metadata of R Markdown documents, e.g.,

```
---
title: "A Nice Book"
knit: bookdown::render_book
output:
  bookdown::gitbook: default
---
```

Then when you hit the `Knit` button on the RStudio toolbar, RStudio will call `bookdown::render_book()` to render the Rmd files. If you only want to preview the current Rmd file, use `bookdown::preview_chapter` instead, e.g.,

```
---
title: "A Nice Book"
knit: bookdown::preview_chapter
output:
  bookdown::gitbook: default
---
```

If an Rmd file does not have YAML metadata in the beginning, **bookdown** will automatically add `knit: bookdown::preview_chapter` to it the first time you compile the whole book through `bookdown::render_book()`.

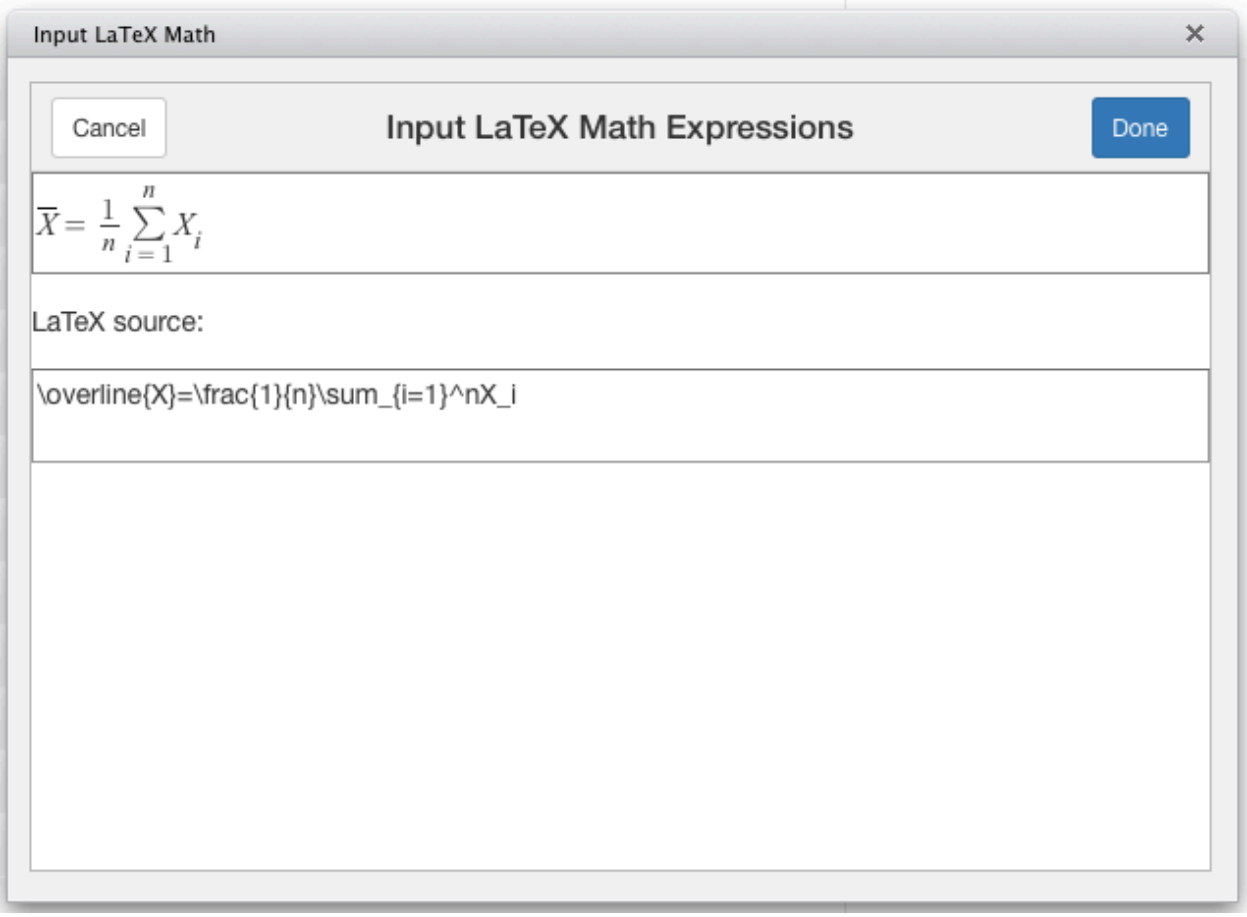


Figure 5.1: The RStudio addin to help input LaTeX math.

The **bookdown** package comes with a few addins for RStudio. If you are not familiar with RStudio addins, you may check out the documentation at http://rstudio.github.io/rstudioaddins/. After you have installed the **bookdown** package and use RStudio v0.99.878 or later, you will see a dropdown menu on the toolbar named "Addins" and menu items like "Preview Book" and "Input LaTeX Math" after you open the menu.

The addin "Preview Book" calls `bookdown::serve_book()` to compile and serve the book. It will block your current R session, i.e., when `serve_book()` is running, you will not be able to do anything in the R console any more. To avoid blocking the R session, you can daemonize the server using `bookdown::serve_book(daemon = TRUE)`. Note this addin must be used when the current document opened in RStudio is under the root directory of your book, otherwise `serve_book()` may not be able to find the book source.

The addin "Input LaTeX Math" is essentially a small Shiny application that provides a text box to help you type LaTeX math expressions (Figure 5.1). As you type, you will see the preview of the math expression and its LaTeX source code. This will make it much less error-prone to type math expressions — when you type a long LaTeX math expression without preview, it is easy to make mistakes such as `X_ij` when you meant `X_{ij}`, or omitting a closing bracket. If you have selected a LaTeX math expression in the RStudio editor before clicking the addin, the expression will be automatically loaded and rendered in the text box. This addin was built on top of the MathQuill library (http://mathquill.com). It is not meant to provide full support to all LaTeX commands for math expressions, but should help you type some common math expressions.

Chapter 6

Publishing

As you develop the book, you may put the draft book in the public to get early feedback from readers, e.g., publish it to a website. After you finish writing the book, you need to think about options to formally publish it as either printed copies or e-books.

6.1 GitHub

You can host your book on GitHub for free via GitHub Pages (https://pages.github.com). GitHub supports Jekyll (http://jekyllrb.com), a static website builder, to build a website from Markdown files. That may be the more common use case of GitHub Pages, but GitHub also supports arbitrary static HTML files, so you can just host the HTML output files of your book on GitHub. To publish to your book to GitHub Pages, you need to create a `gh-pages` branch in your repository, build the book, put the HTML output (including all external resources like images, CSS, and JavaScript files) in this branch, and push the branch to the remote repository.

If your book repository does not have the `gh-pages` branch, you may use the following commands to create one:

```
# assume you have initialized the git repository,
# and are under the directory of the book repository now

# create a branch named gh-pages and clean up everything
git checkout --orphan gh-pages
git rm -rf .

# create a hidden file .nojekyll
touch .nojekyll
git add .nojekyll

git commit -m"Initial commit"
git push origin gh-pages
```

The hidden file `.nojekyll` tells GitHub that your website is not to be built via Jekyll, since the **bookdown** HTML output is already a standalone website. If you are on Windows, you may not have the `touch` command, and you can just create the file in R using `file.create('.nojekyll')`.

After you have set up GIT, the rest of work can be automated via a script (Shell, R, or Makefile, depending on your preference). Basically, you compile the book to HTML, then run git commands to push the files to

GitHub, but you probably do not want to do this over and over again manually and locally. It can be very handy to automate the publishing process completely on the cloud, so once it is set up correctly, all you have to do next is write the book and push the Rmd source files to GitHub, and your book will always be automatically built and published from the server side.

One service that you can utilize is Travis CI (https://travis-ci.org). It is free for public repositories on GitHub, and was designed for continuous integration (CI) of software packages. Travis CI can be connected to GitHub in the sense that whenever you push to GitHub, Travis can be triggered to run certain commands/scripts on the latest version of your repository[1]. These commands are specified in a YAML file named `.travis.yml` in the root directory of your repository, and they are usually for the purpose of testing software, but in fact they are quite open-ended, meaning that you can run arbitrary commands on a Travis (virtual) machine. That means you can certainly run your own scripts to build your book on Travis. Note Travis only supports Ubuntu and Mac OS X at the moment, so you should have some basic knowledge about Linux/Unix commands.

The next question is, how to publish the book built on Travis to GitHub? Basically you have to grant Travis write access to your GitHub repository. This authorization can be done via several ways, and the easiest one to beginners may be a personal access token. Below are a few steps you may follow:

1. Create a personal access token for your account on GitHub: https://help.github.com/articles/creating-an-access-token-for-command-line-use/
2. Encrypt it in the environment variable `GH_TOKEN` via command line `travis encrypt` and store it in `.travis.yml`. If you do not know how to install or use the Travis command-line tool, simply save this environment variable via `https://travis-ci.org/user/repo/settings` where `user` is your GitHub ID, and `repo` is the name of the repository;
3. You can clone this `gh-pages` branch on Travis using your GitHub token, add the HTML output files from R Markdown (do not forget to add figures and CSS style files as well), and push to the remote repository.

Assume you are in the `master` branch right now (where you put the Rmd source files), and have compiled the book to the `_book` directory. What you can do next on Travis is:

```
# clone the repository to the book-output directory
git clone -b gh-pages \
  https://${GH_TOKEN}@github.com/${TRAVIS_REPO_SLUG}.git \
  book-output
cd book-output
cp -r ../_book/* ./
git add *
git commit -m"Update the book"
git push origin gh-pages
```

The variable name `GH_TOKEN` and the directory name `book-output` are arbitrary, and you can use any names you prefer, as long as the names do not conflict with existing environment variable names or directory names. This script, together with the build script we mentioned in Section 5.1, can be put in the `master` branch as Shell scripts, e.g., you can name them as `_build.sh` and `_deploy.sh`. Then your `.travis.yml` may look like this:

```
language: r

env:
  global:
    - secure: A_LONG_ENCRYPTED_STRING
```

[1] You need to authorize the Travis CI service for your repository on GitHub first. See https://docs.travis-ci.com/user/getting-started/ for how to get started with Travis CI.

```
r_github_packages:
  - rstudio/bookdown

script:
  - ./build.sh
  - ./deploy.sh
```

The `language` key tells Travis to use a virtual machine that has R installed. The `secure` key is your encrypted personal access token. If you have already saved the `GH_TOKEN` variable using the web interface on Travis instead of the command-line tool `travis encrypt`, you can leave out this key.

If you use the container-based infrastructure on Travis, you can enable caching by using `sudo: false` in `.travis.yml`. Normally you should cache at least two types of directories: the figure directory (e.g., `_main_files`) and the cache directory (e.g., `_main_cache`). These directory names may also be different if you have specified the **knitr** chunk options `fig.path` and `cache.path`, but I'd strongly recommend you not to change these options. The figure and cache directories are stored under the `_bookdown_files` directory of the book output directory (e.g., `_book/`). A `.travis.yml` file that has enabled caching of **knitr** figure and cache directories may have additional configurations `sudo` and `cache` like this:

```
sudo: false

cache:
  packages: yes
  directories:
    - $TRAVIS_BUILD_DIR/_book/_bookdown_files
```

If your book is very time-consuming to build, you may use the above configurations on Travis to save time. Note `packages: yes` means the R packages installed on Travis are also cached.

GitHub and Travis CI are certainly not the only choices to build and publish your book. You are free to store and publish the book on your own server.

6.2 Publishers

6.3 Licensing

6.4 Self-publishing

Bibliography

Allaire, J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., Wickham, H., Atkins, A., and Hyndman, R. (2016). *rmarkdown: Dynamic Documents for R*. R package version 0.9.5.1.

Chang, W. (2016). *webshot: Take Screenshots of Web Pages*. R package version 0.3.

Cheng, J. (2016). *miniUI: Shiny UI Widgets for Small Screens*. R package version 0.1.1.

R Core Team (2015). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., and Russell, K. (2016). *htmlwidgets: HTML Widgets for R*. R package version 0.6.

Xie, Y. (2015a). *DT: A Wrapper of the JavaScript Library 'DataTables'*. R package version 0.1.45.

Xie, Y. (2015b). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016a). *bookdown: Authoring Books with R Markdown*. R package version 0.0.50.

Xie, Y. (2016b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.12.22.

Xie, Y. (2016c). *servr: A Simple HTTP Server to Serve Static Files or Dynamic Documents*. R package version 0.3.1.

Xie, Y. and Allaire, J. (2016). *tufte: Tufte's Styles for R Markdown Documents*. R package version 0.2.2.