Writing reproducible reports

KnitR with R Markdown

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

biostat.wisc.edu/~kbroman
github.com/kbroman
@kwbroman
Course web: bit.ly/tools4rr

Statisticians write a lot of reports, describing the results of data analyses. It's best if such reports are fully reproducible: that the data and code are available, and that there's a clear and automatic path from data and code to the final report.

Knitr is ideal for this effort. It's a system for combining code and text into a single document. Process the document, and the code is replaced with the results and figures that it generates.

I've found it most efficient to produce informal analysis reports as web pages. Markdown is a system for writing simple, readable text, with the sort of marks that you might use in an email message, that gets converted to nicely formated html-based web pages.

My goal in this lecture is to show you how to use Knitr with R Markdown (a variant of Markdown) to make such reproducible reports, and to convince you that this is the way that you should be constructing such analysis reports.

I'd originally planned to also cover KnitR with AsciiDoc, but I decided to drop it; it's best to focus on Markdown.

KnitR in a Knutshell

bit.ly/knitrknutshell

I wrote a short tutorial on knitr, covering a bit more than I'll cover in this lecture.

I'd be glad for suggestions, corrections, or questions.

Data analysis reports

- Figures/tables + email
- Static LaTeX or Word document
- Knitr/Sweave + LaTeX $\rightarrow$ PDF
- Knitr + Markdown $\rightarrow$ Web page

Statisticians write a lot of reports. You do a bunch of analyses, create a bunch of figures and tables, and you want to describe what you've done to a collaborator.

When I was first starting out, I'd create a bunch of figures and tables and email them to my collaborator with a description of the findings in the body of the email. That was cumbersome for me and for the collaborator. ("Which figure are we talking about, again?")

I moved towards writing formal reports in LaTeX and sending my collaborator a PDF. But that was a lot of work, and if I later wanted to re-run things (e.g., if additional data were added), it was a real hassle.

Sweave + LaTeX was a big help, but it's a pain to deal with page breaks.

Web pages, produced with knitr and Markdown, are ideal. You can make super-tall multi-panel figures that show the full details, without worrying page breaks. And hyperlinks are more convenient, too.

What if the data change?

What if you used the wrong version of the data?

If data are added, will it be easy to go back and re-do your analyses, or is there a lot of copying-and-pasting and editing to be done?

I usually start an analysis report with a summary of the experiment, scientific questions, and the data. Recently, a collaborator noticed that I'd used an old version of the data. (I'd cited sample sizes, and so he could see that I didn't have the full set.)

He said, "I'm really sorry you did all that work on the incomplete dataset."

But actually, it didn't take long to find the right file, and the revised analysis was derived instantaneously, as I'd used KnitR.

KnitR code chunks

Input to knitr:

```
We see that this is an intercross with `r nind(sug)`
individuals. There are `r nphe(sug)` phenotypes, and genotype
data at `r totmar(sug)` markers across the `r nchr(sug)`
autosomes.  The genotype data is quite complete.

```{r summary_plot, fig.height=8}
plot(sug)
```
```

Output from knitr:

```
We see that this is an intercross with 163
individuals. There are 6 phenotypes, and genotype
data at 93 markers across the 19
autosomes.  The genotype data is quite complete.

```r
plot(sug)
```

![plot of chunk summary_plot](RmdFigs/summary_plot.png)
```

The basic idea in knitr is that your regular text document will be interrupted by chunks of code delimited in a special way.

This example is with R Markdown.

There are in-line bits of code indicated with backticks. When the document is processed by knitr, they'll be evaluated and replaced by the result.

Larger code chunks with three backticks. This one will produce a plot. When processed by knitr, an image file will be created and a link to the image will be inserted at that location.

In knitr, different types of text have different ways of delimiting code chunks, because it's basically going to do a search-and-replace and depending on the form of text, different patterns will be easier to find.

html

```
<!DOCTYPE html>
<html lang="em">
<head>
  <meta charset=utf-8"/>
  <title>Example html file</title>
</head>

<body>
<h1>Markdown example</h1>

<p>Use a bit of <strong>bold</strong> or <em>italics</em>. Use
backticks to indicate <code>code</code> that will be rendered
in monospace.</p>

<ul>
<li>This is part of a list</li>
<li>another item</li>
</ul>

</body>
</html>
```

[Example]

It's helpful to know a bit of html, which is the markup language that web pages are written in. html really isn't that hard; it's just cumbersome.

An html document contains pairs of tags to indicate content, like `<h1>` and `</h1>` to indicate that the enclosed text is a "level one header", or `<em>` and `</em>` to indicate emphasis (generally italics). A web browser will parse the html tags and render the web page, often using a cascading style sheet (CSS) to define the precise style of the different elements.

Note that there are six levels of headers, with tags `<h1>`, `<h2>`, `<h3>`, …, `<h6>`. Think of these as the title, section, subsection, sub-subsection, …

CSS

```
ul,ol {
  margin: 0 0 0 35px;
}

a {
  color: purple;
  text-decoration: none;
  background-color: transparent;
}

a:hover
{
  color: purple;
  background: #CAFFFF;
}
```

[Example]

I don't really want to talk about CSS, but I thought I should at least acknowledge its existence.

CSS is really important for defining how your document will appear. Much of the time, you just want to find someone else's CSS document that is satisfactory to you.

Markdown

````
# Markdown example

Use a bit of **bold** or _italics_. Use backticks to indicate
`code` that will be rendered in monospace.

- This is part of a list
- another item

Include blocks of code using three backticks:

```
x <- rnorm(100)
```

Or indent four spaces:

    mean(x)
    sd(x)

And it's easy to create links, like to
[Markdown](http://daringfireball.net/projects/markdown/).
````

[Example | MD cheat sheet]

Markdown is a system for writing simple, readable text that is easily converted into html. The reason it's useful to know a bit of html is that then you have a better idea how the final product will look. (Plus, if you want to get fancy, you can just insert a bit of html within the Markdown document.)

Markdown is just a system of marks that will get searched-and-replaced to create an html document. A big advantage of the Markdown marks is that the source document is much like what you might write in an email, and so it's much more human-readable.

Github (which we'll talk about next week) automatically renders Markdown files as html, and you can use Markdown for ReadMe files. And the website for this course is mostly in Markdown.

R Markdown

- R Markdown is a variant of Markdown, developed at RStudio.com
- Markdown + KnitR + extras
- A few extra marks
- LaTeX equations
- Bundle images into the final html file

R Markdown is a variant of Markdown developed by the folks at RStudio.

It's Markdown with KnitR code chunks, but there are a number of added features, most importantly the ability to use LaTeX equations.

Code chunks, again

````
```{r knitr_options, include=FALSE}
opts_chunk$set(fig.width=12, fig.height=4, fig.path='Figs/',
               warning=FALSE, message=FALSE)
set.seed(53079239)
```

### Preliminaries

Load the R/qtl package using the `library` function:

```{r load_qtl}
library(qtl)
```

To get help on the read.cross function in R, type one of the
following:

```{r help, eval=FALSE}
help(read.cross)
?read.cross
```
````

[Example]

A couple of additional points about code chunks.

You can (and should) assign names to the code chunks. It will make it easier to fix errors, and figure files will be named based on the name of the chunk that produces them.

Code chunks can also have options, like `include=FALSE` and `eval=FALSE`. And you can define global options, which will apply to all subsequent chunks.

Chunk options

`echo=FALSE`	Don't include the code
`results="hide"`	Don't include the output
`include=FALSE`	Don't show code or output
`eval=FALSE`	Don't evaluate the code at all
`warning=FALSE`	Don't show R warnings
`message=FALSE`	Don't show R messages
`fig.width=#`	Width of figure
`fig.height=#`	Height of figure
`fig.path="Figs/"`	Path for figure files

There are lots of chunk options.

These are the chunk options that I use most, but there are lots more. Each should be valid R code, and can be basically any valid R code, so you can get pretty fancy.

The ending slash in `fig.path` is important, as this is just pasted to the front of the figure file names. If not included, the figures would be in the main directory but with names starting with "`Figs`".

Global chunk options

````
```{r knitr_options, include=FALSE}
opts_chunk$set(fig.width=12, fig.height=4, fig.path='Figs/',
               warning=FALSE, message=FALSE,
               include=FALSE, echo=FALSE)
set.seed(53079239)
```

```{r make_plot, fig.width=8, include=TRUE}
x <- rnorm(100)
y <- 2*x + rnorm(100)
plot(x, y)
```
````

- Use global chunk options rather than repeat the same options over and over.
- You can override the global values in specific chunks.

I'll often use `include=FALSE` and `echo=FALSE` in a report to a collaborator, as they won't want to see the code and raw results. I'll then use `include=TRUE` for the figure chunks.

And I'll set some default choice for figure heights and widths but then adjust them a bit in particular figures.

Package options

```
```{r package_options, include=FALSE}
opts_knit$set(progress = TRUE, verbose = TRUE)
```
```

- It's easy to confuse global chunk options with package options.
- I've not used package options.
- So focus on `opts_chunk$set()` not `opts_knit$set()`.

If you are doing something fancy, you may need knitr package options, but I've not used them.

I've gotten confused about them, though: `opts_chunk$set` vs. `opts_knit$set`.

In-line code

```
We see that this is an intercross with `r nind(sug)`
individuals. There are `r nphe(sug)` phenotypes, and genotype
data at `r totmar(sug)` markers across the `r nchr(sug)`
autosomes.  The genotype data is quite complete.
```

- Each bit of in-line code needs to be within one line; they can't span across lines.
- I'll often precede a paragraph with a code chunk with `include=FALSE`, defining various variables, to simplify the in-line code.
- Never hard-code a result or summary statistic again!

In-line code to insert summary statistics and such is a key feature of KnitR.

Even if you wanted the code for your figures or data analysis to be separate, you'd still want to make use of this feature.

Remember my anecdote earlier in this lecture: if I hadn't mentioned sample sizes, my collaborator wouldn't have noticed that I was using an old version of the data.

Rounding

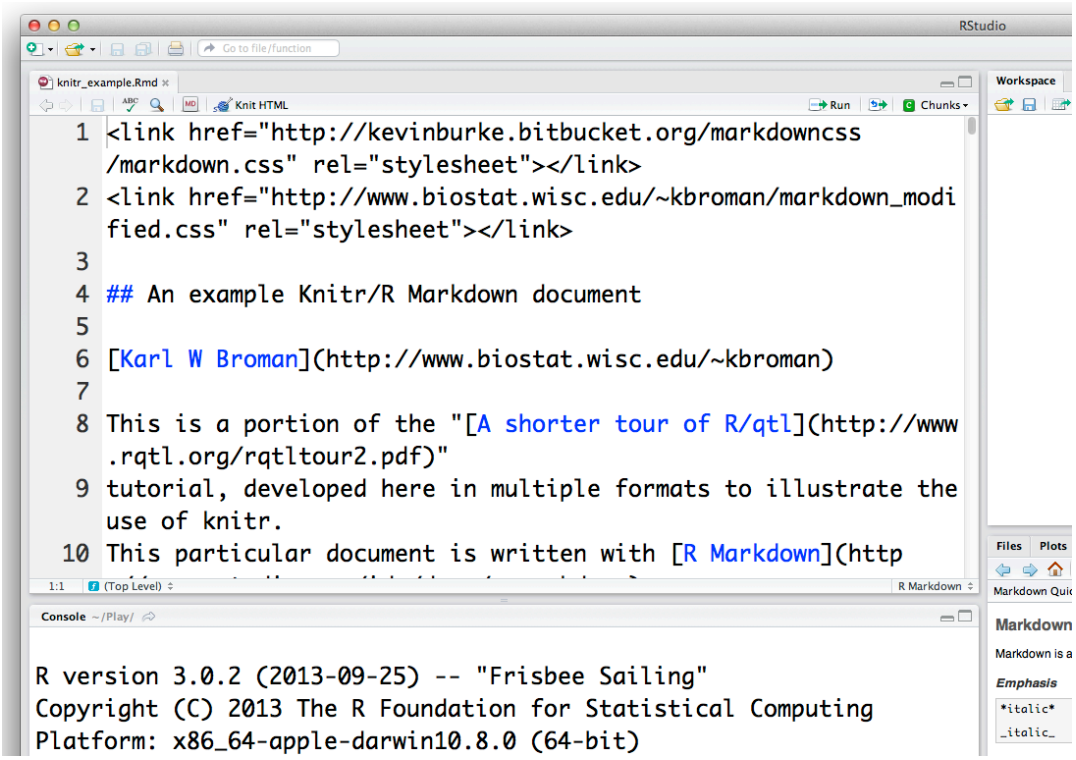
- `cor(x,y)` might produce `0.8992877`, but I want `0.90`.
- `round(cor(x,y), 2)`, would give `0.9`, but I want `0.90`.
- You could use `sprintf("%.2f", cor(x,y))`, but `sprintf("%.2f", -0.001)` gives `-0.00`.
- Use the `myround` function in my R/broman package.
- `myround(cor(x,y), 2)` solves both issues.

I'm very particular about rounding. You should too.

If you're a C programmer, sprintf seems natural. No one else agrees.

The R/broman package is on both github and CRAN.

R Markdown → html, in RStudio



The easiest way to convert an R Markdown file to html is with R Studio.

Open the R Markdown file in R Studio and click the "Knit HTML" button (with the ball of yarn and knitting needle).

Note the little "MD" button next to that. Click that, and you'll get the "Markdown Quick Reference."

What actually happens: The `knit` function in the knitr package processes all of the code chunks and in-line code and creates a Markdown file and possibly a bunch of figure files.

Then, the `markdownToHTML` function in the markdown package converts the Markdown file to an HTML file, with embedded figures.

RStudio is especially useful when you're first learning R Markdown and KnitR, as it's easy to create and view the corresponding html file, and you have access to that Markdown Quick Reference.

R Markdown → html, in R

```
> library(knitr)
> knit("knitr_example.Rmd")
> library(markdown)
> markdownToHTML("knitr_example.md", "knitr_example.html")
```

```
> library(knitr)
> knit2html("knitr_example.Rmd")
```


When you click the "Knit HTML" button in RStudio, it just runs some R code for you (and then opens the result in a preview window).

But you can do the same thing directly, in R.

R Markdown → html, GNU make

```
knitr_example.html: knitr_example.Rmd
  R -e 'library(knitr);knit2html("knitr_example.Rmd")'
```


I prefer to do this from the command-line, using a Makefile. Then it's more obvious what's happening.

Reproducible knitr documents

- Don't use absolute paths like `~/Data/blah.csv`
- Keep all of the code and data in one directory (and its subdirectories)
- If you must use absolute paths, define the various directories with variables at the top of your document.
- Use `R --vanilla` or perhaps

```
R --no-save --no-restore --no-init-file --no-site-file
```

- Use GNU make to document the construction of the final product (tell future users what to do)
- Include a chunk with `sessionInfo()` at the end
- For simulations, use `set.seed` in your first chunk.

That you've used knitr doesn't mean the work is really reproducible. The source and data need to be available to others, they need to know what packages were used and how to compile it, and then they need to be able to compile it on their system.

The complicated alternative to `R --vanilla` is if you want to still load `~/.Renviron`, for example, to define `R_LIBS`.

If you use `set.seed` at the top of the document, it should be that the random aspects will give exactly the same results. I'll use `runif(1, 0, 10^8)` and then paste that big number within `set.seed()`.

Two anecdotes: The github repository for the Reproducible Research with R and R Studio book uses some absolute paths that basically make it not reproducible.

Earn et al. (2014) Proc Roy Soc B 281(1778):20132570 has a really nice supplement, written with KnitR. But it says, "The source code is available upon request." It's not really reproducible, then.

Controlling figures

```
```{r test_figure, dev.args=list(pointsize=18)}
x <- rnorm(100)
y <- 2*x + rnorm(100)
plot(x,y)
```
```

- The default is for KnitR/R Markdown is to use the `png()` graphics device.
- Use another graphics device with the chunk option `dev`.
- Pass arguments to the graphics device via the chunk option `dev.args`.

Graphics in KnitR are super easy. For the most part, you don't have to do anything! If a code chunk produces a figure, it will be inserted.

But depending on the type of figure, you might want to try different graphics devices. And sometimes you want to pass arguments to the graphics device.

Yesterday (6 Feb 2014), to change the size of axis labels, you couldn't just use the pointsize device argument; you'd also need to use something like `par(cex.lab=1.5)`. But I posted a question about it on StackOverflow, and Yihui Xie responded and then immediately fixed the problem. I used a bit of twitter in there too, to get his attention.

To download and install the development version of knitr, you can use the `install_github` function in Hadley Wickham's devtools package. Use `install.packages("devtools")` if you don't already have it installed. Then `library(devtools)` and `install_github("yihui/knitr")`.

Tables

````
```{r kable, results="asis"}
x <- rnorm(100)
y <- 2*x + rnorm(100)
out <- lm(y ~ x)
kable(summary(out)$coef, format="html",
      digits=2)
```
````

````
```{r xtable, results="asis"}
library(xtable)
tab <- xtable(coef_tab, digits=c(0, 2, 2, 1, 3))
print(tab, type="html")
```
````


Two ways to make tables with R Markdown: the `kable` function in the knitr package, and the `xtable` function in the xtable package.

`kable` is simpler, but has fewer options.

`xtable` gives you more complete control.

Important principles

Modify your desires to match the defaults.

Focus your compulsive behavior on things that matter.

Focus on the text and the figures before worrying too much about fine details of how they appear on the page.

And consider which is more important: a manuscript, web page, blog, grant, course slides, course handout, report to collaborator, scientific poster.

You can spend a ton of time trying to get things to look just right. Ideally, you spend that time trying to construct a general solution. Or you can modify your desires to more closely match what you get without any effort.