R and R Markdown for Reproducible Research

John M. Drake

June 11, 2015

Introduction

This exercise is an introduction to producing *reproducible research* in R using R Markdown. This handout is itself written in R Markdown. Before we write our first codes using R Markdown, we need to think a little about how to organize our work. A useful convention is to create a *work directory*, perhaps on your computer desktop or in your home directory. This directory will be used to store codes, data, figures, and any additional computer files needed or generated along the way. To get started:

1. Create a work directory
2. Open R Studio
3. Choose "New File" -> "R Markdown" from the File drop down menu.
4. Navigate to your work directory and save this file with a name of your choosing, ending in the extension `.Rmd`. This is your practice markdown document.
5. Choose "Set Working Directory" -> "To Source File Location" from the Session drop down menu.

Getting started

A literate program is one that intersperses text and code. Actually, there are two kinds of text. The first is the natural language description of the code. We will call this the *commentary*. The second concerns little snippets of natural language *embedded in code chunks* that help the programmer to navigate within the program. These are *comments*. An R Markdown document consists of a single header – set off at the top of the document by three hyphens – and as many sections of commentary and code chunks as desired. A code chunk is indicated by starting on a new line (usually with a hard return in between for clarity) with three back ticks. Typically, we will also want to specfiy how the code chunk will be evaluated (*i.e.*, we may pass in some *options*), in which case the back ticks are followed by a small number of commands in curly braces. The first of these is the letter "r", followed by a space, then a name for the code chunk. Consult the file you created in the section above for an example. Now would be a good time to compile the document (using the "knit html" button in R Studio) to see how commentary and code chunks together produce a document. Also, see how the command `echo=FALSE` was used to suppress printing of the code that was used to produce the plot. This may be useful in creating final documents, but for documents produced during work it is often advisable to print the code associated with each component of analysis.

By modifying this simple example, you are well on your way to producing documents for reproducible research. For instance, you can delete the R code on line 13 and replace it with a command for reading some data of your choice.

Exercise 1. Find some data on the Internet and download to your working directory. Modify your first code chunk to read and summarize this data. (Hint: If you don't want to use your own data, you may download data on cumulative Ebola virus cases in West Africa from https://ds-ec2.scraperwiki.com/g7nnqgn/ckm9nsfssakeuor/cgi-bin/csv/ebola_data_db_format.csv. You can read the file directly from the internet with the code `ebola <- read.csv('http://tinyurl.com/qhrmkht')`.)

Now, to continue developing your reproducible research project, simply create as many code chunks and interspersed commentary are you require. Practically anything that you'd like R to evaluate can be included in these code chunks.

Formatting text

For readability, you may wish to format your test. For instance, this document contains section headers, bold and italic text, code snippets, hyperlinks, and other typographic conventions. These are all readily introduced into R Markdown documents using some simple commands. Your working document illustrates some of these. For example, the R Markdown URL is offset in angle brackets. This is one way of telling R Markdown this is a URL. Another way (which allows a URL to be attached to some other text) is to use a combination of square backets (for the text) and parentheses (for the URL), *i.e.*, `[This is the URL for R Markdown](http://rmarkdown.rstudio.com)`. Your working document also shows a bold text, indicated by double asterisks around the text to be emphasized, and a code snippet (which isn't evaluated), indicated by single back ticks. Other common formatting is accomplished as follows:

- To italicize text, wrap it in single asterisks.
- Block quotes are indicated by right angle bracket (mathematical notation for "greater than").
- Equations (using LaTeXmarkup language) are wrapped in dollar signs, for instance `$\frac{d}{dx}\left( \int_{0}^{x} f(u)\,du\right)=f(x)$` yields $\frac{d}{dx}\left(\int_{0}^{x} f(u)\,du\right)=f(x)$. Latex is a very extensive language for typesetting mathematical equations. It is outside of the scope of this workshop to teach, but if you are familiar with it already you may find it useful to integrate into your markdown documents.
- A new line is generated by a double hard return.

Section titles start on the second new line after some text and are preceded by a hash `#` indicating the section/sub-section level. For example the code

```
# Header 1
## Header 2
### Header 3
```

produces

Header 1

Header 2

Header 3

** Exercise 2.** Further develop your practice markdown document with commentary on the source and structure of your data. Consider some questions you might ask and elaborate on these. Set up document sections to address these questions in turn.

Scientific documents often contain additional objects, such as lists, tables, and figures. Lists are easy to produce in R Markdown. Unordered lists (bullet lists) are started on the second line after some text and indicated by single asterisks. Sublists are indicated by indentation with four spaces. Thus,

```
* List item
* List item
    * Sub-list item
    * Sub-list item
```

produces

- List item
- List item
    - Sub-list item
    - Sub-list item

Numbered lists are similarly generated. Unnumbered and numbered lists may even be mixed. Thus,

```
1. List item
2. List item
    1. Sub-list item
    * Sub-list item
```

produces

1. List item
2. List item
    1. Sub-list item

    - Sub-list item

Exercise 3. Add some lists to your practice document, perhaps to enumerate research questions or proposed methods of analysis.

Images may be included by referencing the file name set off with an exclamation mark, *i.e.* `![alt text](figures/img.png)`. Tables are encoded with a combination of hyphens and pipes. Consult the authoring basics webpage for details.

Figures and code chunk arguments

The example markdown document that you generated initially includes a scatterplot showing the relationship between speed and distance in the `cars` database. In general, your research documents may contain many figures. These are easily produced in R and are automatically included in your compiled document. Note that a separate code chunk should be created for each figure.

Exercise 4. Plot some aspect of your data and embed in your compiled practice document.

What do you do if the default figure dimensions aren't quite right? How do you instruct R to change the size when creating the graphical device? This can be done by passing additional arguments to the code chunk in the code chunk header. For instance, in the example document, we may add the arguments `fig.width=3` and `fig.height=8` to the header at the top of the code chunk (the line beginning with three back ticks).

Exercise 5. Modify the dimensions of your plot to find a size that neatly displays the important information. How does the figure change when the plot dimensions are changed while keeping the ratio constant?

Other code chunk options you may find useful are

- `eval` (whether or not to evaluate the code chunk)
- `cache` (whether to cache the results of the code chunk, for instance if the compute time is large)
- `echo` (whether or not to include the source code in your compiled document)
- `tidy` (whether or not the R code should be automatically tidied for presentation)
- `fig.path` (where to store figures)

A complete list of code chunk options is available at http://yihui.name/knitr/options/. In general, R Studio may be very helpful in identifying what arguments are allowed for code chunks. When typing the argument list, R Studio will pop up a menu of possible arguments. Once the argument has been typed, R Studio will prompt with allowable values.

Exercise 6. Find some numerical operation that requires a long time to compute (say finding the maximum of 1e9 random numbers). Embed this operation in your document and evaluate with cache turned both on and off.

Inline code evaluation

So far, our use of R Markdown has been in the "traditional" literate programming mode, alternating between commentary and code chunks. In some cases, it is useful to evaluate a piece of code *within* the commentary. This is easily done by setting off the code-to-be-evaluated in single back ticks with the letter `r` and a space after the first back tick. This comes in particularly handy in the next section.

Exercise 7. Perform some desciptive analysis of your data. Describe mean, standard deviation, minimum, maximum and possibly other statistics. Compute these inline as part of your commentary.

References

Often you will want to reference previously published document in your report. There are many ways of handling bibliographies and citations in R Markdown by specifying biobliogreaphic information in the header. A list of the allowable formats is online at http://rmarkdown.rstudio.com/authoring_bibliographies_and_citations.html. Possibly the easiest approach is to use the `knitcitations` package (http://cran.r-project.org/web/packages/knitcitations/index.html). This package allows one to cite exiting documents using its digital object identifier (DOI), web URL, or bibtex key.

To install the development version, we first run the following lines:

```r
library(devtools)
```

```
##
## Attaching package: 'devtools'
##
## The following objects are masked from 'package:utils':
##
##     ?, help
##
## The following object is masked from 'package:base':
##
##     system.file
```

```r
install_github("cboettig/knitcitations")
```

```
## Installing github repo knitcitations/master from cboettig
## Downloading master.zip from https://github.com/cboettig/knitcitations/archive/master.zip
## Installing package from /tmp/Rtmpng7e2X/master.zip
## arguments 'minimized' and 'invisible' are for Windows only
## Installing knitcitations
## '/usr/lib/R/bin/R' --vanilla CMD build  \
##   '/tmp/Rtmpng7e2X/devtools6c6b2a223595/knitcitations-master'  \
##   --no-manual --no-resave-data
##
## '/usr/lib/R/bin/R' --vanilla CMD INSTALL  \
##   '/tmp/Rtmpng7e2X/knitcitations_1.0.7.1.tar.gz'  \
##   --library='/home/jdrake/R' --install-tests
```

Next we load the library, clean up the bibliographic environment, and set the options so the bibliography compiles.

```r
library(knitcitations)
cleanbib()
options("citation_format" = "pandoc")
```

Now, the functions `citet` and `citep` may be used to include references with the bibliographic information pulled directly from the web. Particularly, all peer reviewed journal articles are assigned a permanent DOI. If you can identify this (for instance from the journal's website for the article), you cite it directly. To do this, we recognize that the functions `citet` and `citep` are actually R functions. So, we use the inline code evaluation from the last section, specifically a back tick, followed by the function, calling the DOI as a text string. Using the DOI "10.1371/journal.pbio.1002056" and `citet` yields

Drake et al. (2015)

Using the same DOI and `citep` instead generates

(Drake et al. 2015).

Sometimes we will want to cite documents that do not have a DOI, such as the URL for the `knitcitations` package. In this case, the URL (as a character string) is passed to the function `citet` or `citep`.

Additionally, you do need to include the code in a chunk at the end of your document to actually generate the bibliography file as well as the simple command `bibliography: "references.bib"` in the header.

Exercise 8. Find some relevant peer-reviewed background literature on the web and refer to it in the commentary of your working example.

Postscript on best practices

This exercise was designed primarily to introduce the practical aspects of producing reproducible research. It neglects various habits and methods for organizing work, although these practices are well worth considering. One practice concerns the principle of dataflow: when a data point is changed, all downstream analysis should change as well. But, what does it mean to *change* a data point? If data points are changing, how can research be reproduced?

One approach is to adopt the practice to *never change the raw data*. That is, once collected, cleaned, and organized, the raw data table for an analysis may be archived and the archived version considered to be fixed? But, what if an error is discovered? Or, more commonly, what if analysis requires some subset of the data or transformation of the data? The practice of literate statistical programming allows these operations to be included in the algorithm. That is, they are considered to be part of the work flow. If a data point needs to be changed, this can be done programmatically, in a *pre-processing* step that precedes the actual data analysis. Besides ensuring the stability of the underlying information, this approach also enables documentation of the rationale for any such changes to the data.

Two other considerations also must be made when preparing a reproducible research project for storage. The first is how will the organization of the various files, programs, documents, etc. be communicated to future users. The convention is to create a text file, `readme.txt`, containing the necessary information. Minimally, this file should describe:

- Date of creation
- Author
- Means for contacting the author (i.e., email address)
- List of filenames included in the package, their formats, and their function
- Change log

This system will work well enough for tidy projects that are done in a fairly short period of time and then archived in a more or less fixed condition. Larger projects or dynamic projects that are ongoing require some kind of *version control* such as Git/Github. A version control system records all the changes to a set of files so that the state of the project at any time in its history can be readily reconstructed.

Another question that arises is what kind of documents should one produce in a reproducible research project? In my experience, any project typically involves multiple documents. The first "working document" is really a transcript of the research process – perhaps lightly edited to remove various dead ends. A second "report" or "manuscript" is much more heavily edited – removing exploratory analyses, diagnostics, "double-checks", and other unnecessary material – and styled in the usual fashion (Introduction -> Methods -> Results -> Discussion). This second document, despite being more cursory, is nonetheless reproducible in the sense that it recapitulates an entire workflow and enables data flow.

Bibliography

Drake, John M., RajReni B. Kaul, Laura W. Alexander, Suzanne M. O'Regan, Andrew M. Kramer, J. Tomlin Pulliam, Matthew J. Ferrari, and Andrew W. Park. 2015. "Ebola Cases and Health System Demand in Liberia." Edited by Steven Riley. *PLOS Biology* 13 (1). Public Library of Science (PLoS): e1002056. doi:10.1371/journal.pbio.1002056.