An Introduction to Reproducible Research in RStudio

Will Massengill

Program in Statistics and Methodology
Department of Political Science
Ohio State University

October 28, 2015

Reproducible Research

- Disclaimer: This presentation is largely based on the sources cited at the end.
- Computational research is replicable if independent researchers can easily use replicate the results with the available data and code (Peng, 2011)
- Benefits for science
  - Results are transparent
  - Reduces frustration and effort involved in replication
- Benefits for you
  - Improve your work habits
  - Facilitate future changes to your work
  - Broaden your research impact

Conducting Reproducible Research

- What the product looks like:
  - Your data
  - Code for statistical analyses
  - Presentation of your results
- How to produce the output:
  - Statistical language for collecting, wrangling, and analyzing data, and producing graphics: R
  - Markup language to present results: $\LaTeX$
  - Reproducible research publisher for literate programming: **knitr**
  - Environment that integrates these programs: RStudio
  - Version control software for tracking changes over time: Git and GitHub
  - Command line tools for managing files and running Git

What **knitr** Does

- Parses the source document to identify computer code
  - Using regular expressions (Friedl, 2006)
  - Based on format of **knitr** document
  - Our focus is on .Rnw format, since you'll use this to generate .tex (and .pdf) files
- Evaluates the code
  - Using **evaluate** (Wickham, 2013) and `base::eval()`
- Renders output based on the format of the **knitr** document
  - Using output hooks, which we'll discuss later
- Can also extract R code with `purl("file.Rnw")`

Getting Started with **knitr**

- Set **knitr** as your preferred program for weaving R and LaTeX:

  `Preferences ▷ Sweave ▷ knitr`

- Open a new **knitr** document:

  `File ▷ New File ▷ R Sweave`

- The .Rnw file is a plain text file that **knitr** reads to knit your code into a .tex document, which will ultimately be used to generate a .pdf.
- If it is not installed, `install.packages("knitr")`

Code Input

- R code is inserted in code chunks, which have the format:

```
<<label, option = value>>=
code
@
```

  - Label must be unique to code chunk
  - Options specified like options in R, where values are logical or character
  - Options can even accept conditional statements (e.g., `if () else`)
  - `<<>>=` initiates code chunk, and `@` ends it
- You can also execute inline code with:

```
\Sexpr{code}
```

Working with Text Output

- `evaluate` evaluates source code and returns a list with 6 classes of output: character, source, message, warning, error, and recordedplot
- Output hooks tell **knitr** how to format this output so that it will be appropriate for a .tex file
  - But you can use output hooks to customize your output
- You can also set options globally

Default Code Chunk Options

- `eval = TRUE` instructs **knitr** to evaluate code in this chunk
- `tidy = TRUE` improves readability of the code with spacing and assignment character using **formatR** (Xie, 2012)
- `highlight = TRUE` highlights elements in your code based on their type
- `prompt = FALSE` does not print the prompt character in the code output
- `comment = '##'` prints this comment character in front of results
- `echo = TRUE` prints the source code
- `results = 'markup'` marks up the results based on **knitr** document (also see `'asis'` and `'hide'`)
- `warning/error/message = TRUE` prints these messages
- `split = FALSE` does not redirect output to a different file
- `include = TRUE` includes chunk in your document

An Example: Input

```
<<"ex1">>=
set.seed(50)
x <- sample.int(10, 7, replace = TRUE)
x; diff(x)
identical(diff(x), x[-1] - x[-length(x)])
@
```

An Example: Output

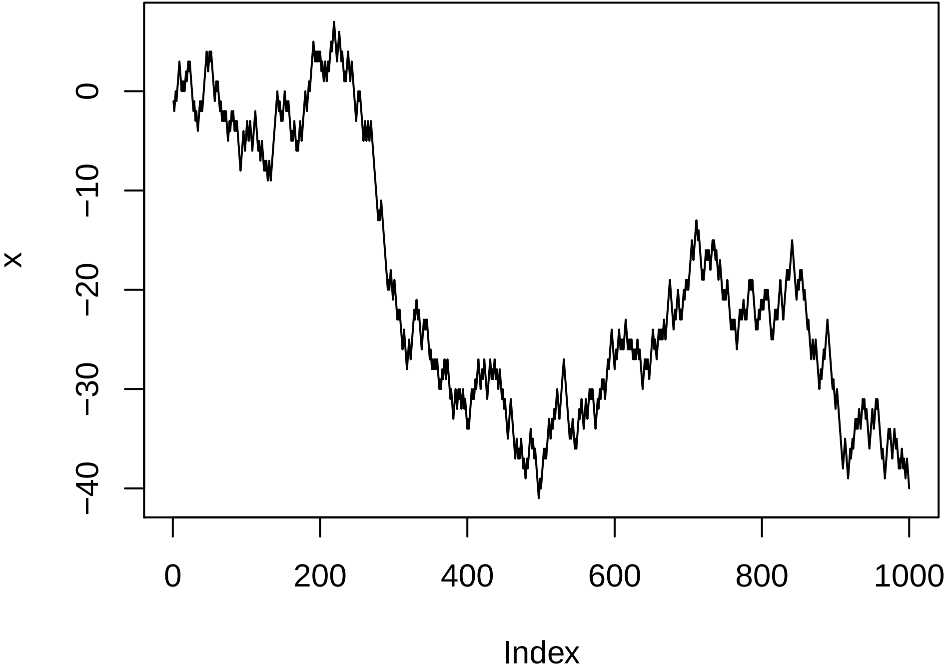
```
set.seed(50)
x <- sample.int(10, 7, replace = TRUE)
x; diff(x)

## [1] 8 5 3 8 6 1 7
## [1] -3 -2  5 -2 -5  6

identical(diff(x), x[-1] - x[-length(x)])

## [1] TRUE
```

Working with Graphical Output

- It will be useful to load the `caption` and `subfig` packages in your LaTeX header
- Plot first recorded as plot object by **evaluate**
  - Plots recorded on per expression basis (see second example below)
  - `fig.keep = 'last'` only keeps last plot created by *high-level* plotting command (e.g., `plot()`)
- Then, plot replayed in graphical device
  - `dev = 'pdf'` uses `grDevices::pdf` to generate plot
- Options
  - `fig.show = 'asis'` inserts plots where they were created in chunk
  - `fig.width = 7` (`fig.height = 7`) generates a 7" by 7" plot in the graphical device
  - `out.width` (`out.height`) modify the size of the plot in the presentation document
  - Control `figure` environment in LaTeX: `fig.env`, `fig.pos`, and `fig.scap`

Plotting a Random Walk

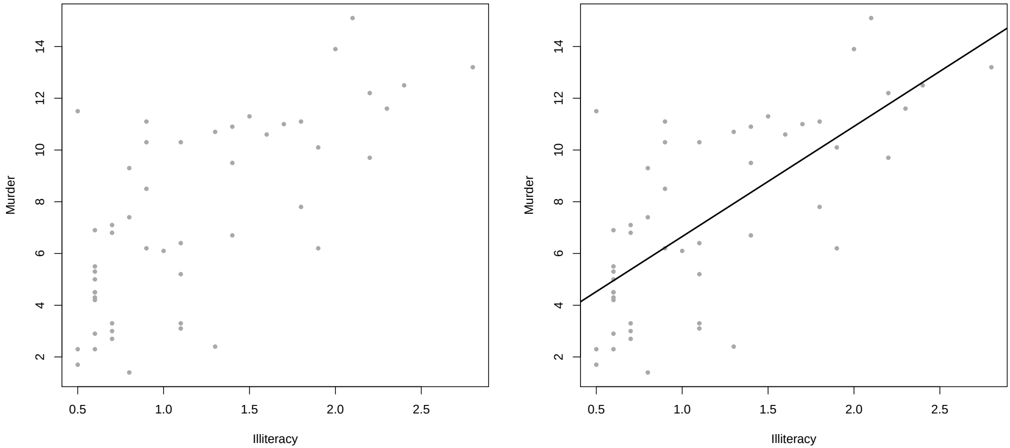
```
<<"ex2", fig.height = 3.5, fig.width = 5, fig.align = 'center', echo = FALSE>>=
set.seed(1)
n <- 1000
x <- cumsum(sample(c(-1, 1), n, TRUE))
par(mar = c(4, 4, 0.1, 0.3))
plot(x, type = "l", lwd = 0.5)
@
```



0
−10
−20
−30
−40
x
0
200
400
600
800
1000
Index

Another Graphics Example

```
<<"ex3", fig.cap = 'Illiteracy and Murder Rate', fig.subcap = c('Points Only',
 'The Regression'), out.width = '.49\\linewidth', echo = FALSE, fig.keep = 'all'>>=
attach(data.frame(state.x77))
fit <- lm(Murder ~ Illiteracy)
plot(Illiteracy, Murder,  pch = 20, col = 'darkgrey')
abline(fit, lwd = 2)
@
```



(a) Points Only (b) The Regression

Figure 1: Illiteracy and Murder Rate

Discussing Results

- `As Figure~\{fig:ex3} indicates, the slope of the regression is \Sexpr{round(coef(fit)[2], 2)}.`

  As Figure 1 indicates, the slope of the regression is 4.26.

- Alternatively, we can print results from the code chunk:

```
<<"printEq", results = 'asis'>>=
cat("The linear regression", sprintf("$Murder = %.02f + %.02f Illiteracy$...",
                              coef(fit)[1], coef(fit)[2]))
@
```

The linear regression $Murder = 2.40 + 4.26 Illiteracy$...

Formatting Results in a Table

(Make sure \usepackage{dcolumn} is in your LaTeX header)

```
<<"ex4", include = FALSE, warning = FALSE>>=
library(apsrtable)
fit2 <- lm(Murder ~ Illiteracy + log(Income))
fit3 <- lm(Murder ~ Illiteracy + log(Income) + log(Area))
@
```

```
\begin{table}
\caption{Regressions of Murder Rate}
\label{tab:tab1}
\begin{center}
```

```
<<"table", echo = FALSE, results = 'asis'>>=
apsrtable(fit, fit2, fit3, Sweave = TRUE, stars = "default")
@
```

```
\end{center}
\end{table}
```

Table 1: Regressions of Murder Rate

	Model 1	Model 2	Model 3
(Intercept)	2.40**	−17.94	−29.40
	(0.82)	(26.42)	(24.60)
Illiteracy	4.26***	4.52***	4.53***
	(0.62)	(0.71)	(0.66)
log(Income)		2.39	2.58
		(3.10)	(2.85)
log(Area)			0.92**
			(0.30)
N	50	50	50
R^2	0.49	0.50	0.59
adj. R^2	0.48	0.48	0.56
Resid. sd	2.65	2.66	2.45

Standard errors in parentheses

† significant at $p < .10$; $^{*}p < .05$; $^{**}p < .01$; $^{***}p < .001$

Final Details, Part 1

- When chunk option `cache = TRUE`, results from the chunk will be loaded lazily only if:
  - The code chunk has not been changed since its last execution
  - Any (cached) chunks on which it depends have not been changed since its last execution
  - Chunk dependencies can be specified manually (`dependson = 'chunklabel'`) or automatically (see Xie, 2014)
- You can also use chunks within other chunks
  - Embed code chunks with: `<<label>>`
  - To reuse chunks `A` and `B`, use the option `ref.label = c('A', 'B')`

Final Details, Part 2

- We can extend chunk options with chunk hooks
- Can create an option for plot margins

```
<<"hook1", include = FALSE, warning = FALSE>>=
knit_hooks$set(margin = function(before, options, envir) {
    if (before) # Only run before chunk is executed
        par(mar = c(4, 4, 0.1, .1)) else NULL
})
@
```

- Can be triggered locally in chunk header with `margin = TRUE`
- Can also be triggered globally with `opts_chunk$set(margin = TRUE)`
- `?knitr::knit_hooks`

Managing Your Files

- To make reproducable your research easier, it is best to explicitly tie your files together in a logical way
- To access a file, we must know it's file path
- A file path tells you how your file is hierarchically stored on your hard disk
- These hierarchical lists are called directories, or file trees
  - You can think of directories as folders
- Root directory: the ultimate parent directory
  - Begins with `C:\` in Windows
  - Begins with the first / on Unix-like operating systems
- Subdirectories are directories within the root directory
- Your working directory is the directory "where" you are working
- Note: For Windows, in R you will need to type \\ instead of a single \

Manipulating Files

Table 2: Commands for File Management

Task	R	Unix-like Shell
Present Working Directory	`getwd`	`pwd`
Change Working Directory	`setwd`	`cd`
List Files in Working Directory	`list.files`	`ls`
Make New Directory	`dir.create`	`mkdir`/`sudo mkdir`
Create New File	`file.create`/`cat`	`echo`
Delete File/Directory	`unlink`	`rm`
Rename File	`file.rename`	`mv` (within same directory)
Copy File	`file.copy`	`cp`

Version Control with Git and GitHub

- Keeping track of the changes between files in typical file sequences (`File.v1.txt` $\rightarrow$ `File.v2.txt` $\rightarrow$ `File.v3.txt` $\rightarrow$ `File.v3b.txt` $\rightarrow \ldots$) is daunting, especially with plain text files (.R, .Rnw, or .tex)
- Version control systems (VCSs), like Git, track changes in documents over time
- Because Git is a distributed VCS, collaborators can work modify the same document at the same time, merging their changes together afterward
- Git is a command line program, i.e., it interfaces with your computer's shell through the command line
  - The shell passes commands to your operating system
  - bash is a common shell program
- For many Git commands, you'll use a terminal emulator, such as Terminal on Macs or PowerShell on Windows

What Git Does

- When you create a Git repository for a file (`git init`), Git creates a series of hidden folders (to see: `find -a`) that store data about the file and repository
  - A Git repository is like a directory with a bunch of extra files for Git's operations
- Each time you `commit` a file, Git saves information about the contents of the file in its *object store*, along with file metadata, such as its directories, in its *index*
- The contents are saved with a SHA-1 hash, a unique identifier for the contents of the file. The contents of the file can be reproduced by decrypting the SHA-1 hash (`git cat-file -p <key>`)
- Rather than track each change to each document explicitly, Git can uncover the differences between files using these keys (`git diff`)

The Git Workflow

- Set up a repository with a new file
  - Create a directory for your project
  - Initialize the repository
  - Create a file
  - `add` this file to be staged
  - `commit` the file
- Make changes to a project
  - `branch` off the master project and `checkout` a branch to make changes
  - `clone` the entire repository or `fetch` particular objects
  - If the repository is remote, `pull` objects from the repo, `push` committed files to the repository, and resolve conflicting changes in the "Issues" area on your GitHub repository

First Things First

If you haven't yet, visit
`https://help.github.com/articles/set-up-git/`.

Running Git Locally

- Open your Terminal emulator

```
$ git # Can be a useful reference
$ pwd # Can give you an idea about where you are
$ ls # Also helps you determine where you are
$ mkdir Desktop/test/ # Create new folder on desktop
$ cd Desktop/test/ # Change directory to this folder
$ echo 'Hello world' > hello.txt # Create text file in directory
```

- Let's use Git

```
$ git status # Nothing yet
$ git init # Initialize working directory as repo
$ find . # Lots of new hidden files
$ open .git # Can click through the folder
$ git add hello.txt # All files to be staged
$ git status # Another look
$ git commit hello.txt -m "my first commit" # Commit files
$ git status -s # Short description
```

Running Git Locally

- Let's modify the file, see changes, and recommit

```
$ git echo 'hello again' > hello.txt $ git diff # See differences between files
$ git add hello.txt
$ git commit hello.txt -m "Second try"
$ git diff # No differences to note $ git log # See commit history
```

Git, continued

- Checkouts
  - To change you working directory to a file, commit, or branch, use `git checkout ---`, where `---` is the object reference
  - To avoid referencing most recent commit with its SHA-1 hash, tag it and then check it out

```
$ git tag -a v1 -m "Version1"
$ git checkout v1
```

- Branches
  - If you want to keep modifying a project in one direction, without changing the master file, you can create a switch branches
  - To show your current branch, `git branch`
  - To create a branch called Test1, `git branch Test1`
  - To switch to a branch, use `git checkout`
  - To create and switch to a branch called Test1, use `git checkout -b Test1`
  - To merge master and Test branches, use `git merge Test1`

Getting Started with GitHub

- GitHub is an online host for Git repositories
- This is a great place to store replication files for your research project
- Also a great place to find good code
- Provides a graphical user interface for projects with Git

Repositories on GitHub

- Push our existing repository to GitHub
  - Once you've logged in, create a new repository (+▼ in the top right corner)
  - Name your repository
  - Add remote repository (called `origin`) and push master branch of local repository to it

```
$ git remote add origin git@github.com:massengillw/NewTest.git # Can use url
$ git push -u origin master # Push local repository to remote repository
```

- Work with new repository on GitHub
  - Once you've logged in, create a new repository
  - Name your repository
  - Add README file
  - Should also add .gitignore file (tells Git which files not to track)

Using GitHub

- On the repository's page, you can
- Create and commit a new file with `RepoName/+`
- Create and resolve issues (Issues tab on right side of page)
- Create and navigate branches
  - `Branch:master` ▼
  - Can create and commit new files in the new branch
- Generate pull request
  - Request for help on the project
  - Others can `pull` the project and try to improve the code

Incorporating RStudio

- Like for **knitr**, RStudio also integrates Git into your workflow
- Configure RStudio for Git
  - `Tools` ▷ `Global Options` ▷ `Git/SVN`. Under Git Executable:
  - For Macs, browse and find like `/usr/bin/git`
  - For Windows, browse and find git.exe, probably in your Program Files
- Create a new version-controlled project
  - `File` ▷ `New Project` ▷ `Empty Project`
  - Check box to create Git repository for project
  - Also note that you can clone an existing repository into a new project
- Initialize a current project
  - Open existing project, then: `Tools` ▷ `Project Options` ▷ `Git/SVN` ▷ `Git`

A Final Example

- Create a new repository on GitHub
- Create a new project in RStudio
  - `File` ▷ `Version Control` ▷ `Git`
  - Paste repository's URL
  - Name your project (the same name as GitHub repo)
  - Tell RStudio where to save it
- Create a new file
- Stage, or add, the file
- Commit the file, adding a comment for your commit
- Push your commit to your remote repository
- Modify the file, stage, commit, push
- Explore the file changes in your remote repository (see Blame and History)

Have Fun!

If you have any questions, see the references, search Google, or email me at
massengill.8@osu.edu

References

Chacon, Scott, and Ben Straub. 2014. *Pro-Git*. New York: Apress.

Gandrud, Christopher. 2014. *Reproducible Research with R and RStudio*. Boca Raton, FL: CRC Press.

Loeliger, Jon, and Matthew McCullough. 2012. *Version Control with Git*. Cambridge: O'Reilly Media.

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334:1226–1227.

R Core Team. 2014. *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.

Shotts Jr., William, E. 2012. *The Linux Command Line: A Complete Introduction*. San Fransicso: No Starch Press.

Wickham, Hadley. 2015. "evaluate: Parsing and Evaluation Tools that Provide More Details than the Default." R package version 0.7.2. http://CRAN.R-project.org/package=evaluate.

Xie, Yihui. 2014. *Dynamic Documents with R and knitr*. Boca Raton, FL: CRC Press.

Xie, Yihui. 2015. "knitr: A General-Purpose Package for Dynamic Report Generation in R." R package version 1.10.5.