Notes for Grad Student Orientation 2018

Charles J. Geyer

Contents

Chapter 1

Introduction

These are some notes for grad student orientation in the School of Statistics, University of Minnesota, Fall 2018.

1.1 License

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License (http://creativecommons.org/licenses/by-sa/4.0/).

1.2 R

- The version of R used to make this document is 3.5.1.
- The version of the `bookdown` package used to make this document is 0.7.
- The version of the `rmarkdown` package used to make this document is 1.10.
- The version of the `knitr` package used to make this document is 1.20.
- The version of the `ggplot2` package used to make this document is 3.0.0.

1.3 Other Learning Materials

1.3.1 An Introduction to R

By far the best book on R is free, written by the R core team, always up to date with the current version, and always correct. It is called *An Introduction to R* and can be found in your R distribution. Do

```
help.start()
```

to start browser help and click on the "An Introduction to R" link.

If can also be found on-line at CRAN.

PDF and e-book versions are also available at CRAN.

On the web page `help.start()` gives you or at the link just above you see there are also five other manuals that come with R, but they are for experts. You don't want to read them yet.

1.3.2 An R Short Course

Your humble author was one of five instructors for a two-day short course on R. Here are the notes for it.

1.3.3 An R Course (Undergraduate)

Your humble author taught Stat 3701 (undergraduate statistical computing). Here is the web site for that.

Of particular interest are the

- reproducible research examples.
- course notes.

1.3.4 An R Course (PhD Level)

Your humble author taught Stat 8054 (PhD level statistical computing). Here is the web site for that.

That site is a bit out of date. I will redo it when I teach that course again in Spring 2019, including many topics from my Stat 3701 notes linked above, like web scraping, JSON, and SQL databases. And I will update many topics, like parallel computing.

Chapter 2

R Markdown

2.1 Source

The source for this file is https://raw.githubusercontent.com/cjgeyer/Orientation2018/master/01-rmarkdown.Rmd.

To fully understand it you have to compare what you see here (output) to the the source. So open the source in another tab in your browser.

2.2 What is It, and Why do I Want It?

2.2.1 What is It?

R markdown is the latest in a long line of R packages that provide

- literate programming

and

- reproducible research

using R.

It allows you to mix R code that is executed in the production of the document with a document. Of course plain code with comments goes a little way to explaining code, but literate programming is much better.

R markdown can be converted to output formats other than HTML. Among these are PDF, Microsoft Word, and e-book formats. Other output formats are explained in the Rmarkdown documentation.

2.2.2 Newbie Data Analysis

The way most newbies use R or any other statistical package is to dive right in

- typing commands into R,
- typing commands into a file and cut-and-pasting them into R, or
- using RStudio.

None of these actually document what was done because commands get edited a lot.

If you are in the habit of saving your workspace when you leave R or RStudio, can you explain *exactly* how every R object in there was created? *Starting from raw data?* Probably not.

If you work this way, you are never an "expert" even if you have 50 years experience. You are also **not** doing reproducible research.

2.2.3 Expert Data Analysis

The way experts use plain R is to type commands into a file, say `foo.R` and use

```
R CMD BATCH --vanilla foo.R
```

to run R (from the operating system command line) to do the analysis.

There are several ways experts use literate programming with R. Type commands with explanations into an R Markdown file, and render it in a clean R environment (empty global environment). Either start R with a clean global environment via

```
R --vanilla
```

and do

```
library(rmarkdown)
render("foo.Rmd")
```

or start RStudio with a clean global environment (on the "Tools" menu, select "Global Options" and uncheck "Restore .RData into workspace at startup", then close and restart) load the R Markdown file and click "Knit".

Or use an older competitor of R Markdown, such as R function `Sweave` or R package `knitr`.

The important thing is using a clean R environment so all calculations are fully reproducible. Same results every time the analysis is rerun by you or by anybody, anywhere, on any computer that has R.

That's (the computing part of) reproducible research!

2.2.4 No Snarf and Barf

Snarf and barf is a colorful hacker term for cut and paste.

When doing reproducible research you must **never snarf and barf**. It will inevitably get out of date so snarf-and-barfed output does not match the code in the document that purportedly produces it.

In short, snarf and barf inevitably leads to lies (inadvertent, but still lies).

So don't.

With R Markdown (or `Sweave` or `knitr`) you never need to.

2.3 Getting Started

You don't need RStudio to use R Markdown (despite it being created by people who are now all RStudio employees).

If you have an R Markdown file `baz.Rmd` then

```
Rscript -e 'rmarkdown::render("baz.Rmd", "all")'
```

run from the operating system command line renders it into whatever output formats it says it does and

```
Rscript -e 'rmarkdown::render("baz.Rmd", "html_document")'
```

does a specific format.

But for today, we'll use RStudio.

- Start RStudio.
- On the File menu
  - select "New File"
  - then select "R Markdown"
    - and in the dialog that pops up fill in a title and author
    - and click the "OK" button

You now should have a toy R Markdown document in the upper left panel of the RStudio app. Now

- click the button labeled "knit" having a yarn and needles icon
- and in the dialog that pops up give it a file name and location for where to save the file

  Caution! The extension of the file must be `Rmd` or `rmd` because RStudio refuses to process it otherwise.
- and then the rendered document should show up in the lower right panel or in a pop up

We're in business! We have done R Markdown!

As we go along we can try things.

2.4 Syntax not Involving R

Section 2.3 of the R Markdown book recommends the RStudio cheat sheets.

Section 2.5 of the R Markdown book has its take on the markdown part of R Markdown.

Things to try in your toy document

- italics
- bold face
- monospace font (for code) (uses backticks)
- hyperlinks
- section headings
- lists
- math (if you already know LaTeX or just want to copy the examples in the R Markdown book — we aren't going to try to teach LaTeX here)

2.5 Syntax Involving R

2.5.1 Code Chunks

In your toy document RStudio already gives you two code chunks (R that is executed when the document is rendered and the output stuffed in the document). One does a summary and the other a plot.

Here's a trivial code chunk

```
2 + 2
```

```
## [1] 4
```

Note that the result is not in the document source. Every time the document is rendered, R executes the code producing new results. If you change the code, the results also change (unlike what happens if you snarf-and-barf code and results).

2.5.2 Plots

The toy document RStudio provides shows a plot using R base graphics, here is another using R package `ggplot2`.

```
# set.seed(42) # uncomment to always get the same plot
# for ggplot all data must be in data frame
mydata <- data.frame(x = rnorm(1000))
library(ggplot2)
ggplot(mydata, aes(x)) +
    geom_histogram(aes(y = ..density..), binwidth = 0.5,
        fill = "cornsilk", color = "black") +
    stat_function(fun = dnorm, color = "maroon")
```

The analog with base graphics doesn't work as well because it is two commands that don't talk to each other.

```
# for base graphics we don't need data in data frames
x <- mydata$x
hist(x, probability = TRUE)
curve(dnorm(x), add = TRUE)
```

2.5.3 Tables

The no-snarf-and-barf rule means we have to use R to construct all our tables. To see how to do that, first we need a table.

Make up some regression data.

```
n <- 50
x <- seq(1, n)
a.true <- 3
b.true <- 1.5
y.true <- a.true + b.true * x
s.true <- 17.3
y <- y.true + s.true * rnorm(n)
```

And fit some models to it.

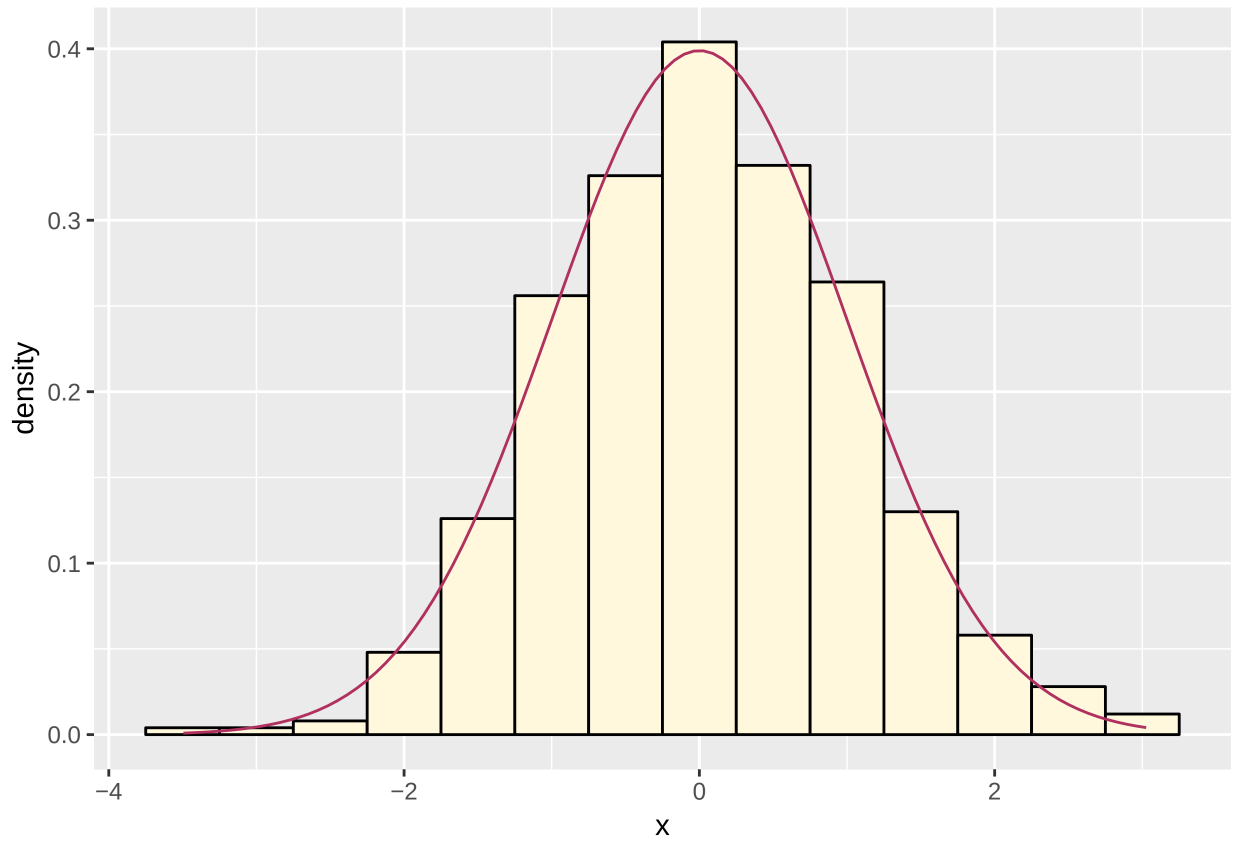


Figure 2.1: Histogram with probability density function.

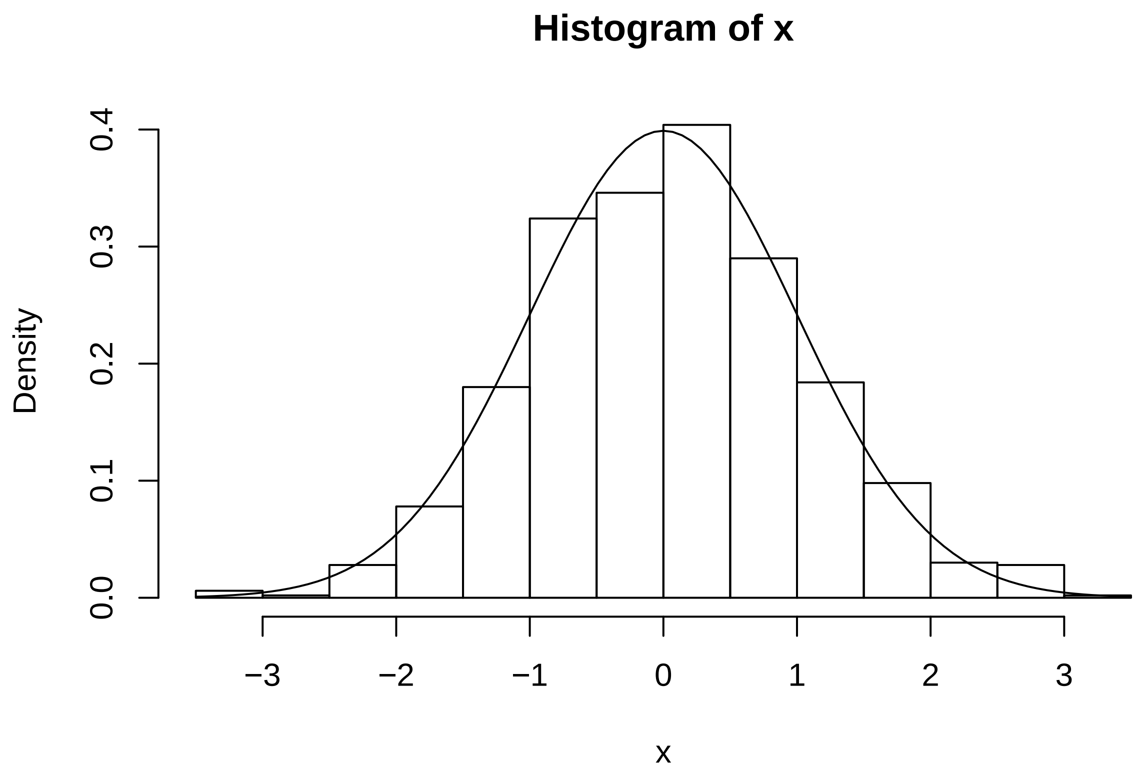


Figure 2.2: Histogram with probability density function (base graphics).

```
out1 <- lm(y ~ x)
out2 <- lm(y ~ x + I(x^2))
out3 <- lm(y ~ x + I(x^2) + I(x^3))
anova(out1, out2, out3)
```

```
## Analysis of Variance Table
##
## Model 1: y ~ x
## Model 2: y ~ x + I(x^2)
## Model 3: y ~ x + I(x^2) + I(x^3)
##   Res.Df   RSS Df Sum of Sq      F Pr(>F)
## 1     48 14199
## 2     47 13975  1    223.35 0.7776 0.3825
## 3     46 13212  1    762.93 2.6562 0.1100
```

That is a table of sorts.

In order to make that a table nicely formatted for the document we are making, first we have to figure out what the output of R function `anova` is and capture it so we can use it.

```
foo <- anova(out1, out2, out3)
class(foo)
```

```
## [1] "anova"      "data.frame"
```

It is a data frame, which is the easiest thing to turn into a table, and the simplest way to do that seems to be the `kable` option on our R chunk

Table 2.1: ANOVA Table

Res.Df	RSS	Df	Sum of Sq	F	Pr(>F)
48	14198.6				
47	13975.3	1	223.35	0.778	0.382
46	13212.3	1	762.93	2.656	0.110

2.5.4 In-Line R

You can also execute R not in a code chunk but just in-line in your writing. Here is an example of that. First let's get some computation we want to quote.

```
summary(out3)
```

```
##
## Call:
## lm(formula = y ~ x + I(x^2) + I(x^3))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -27.655 -12.556  -1.212   9.960  49.313
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.271053  10.353557   0.026   0.9792
## x            3.450615   1.740776   1.982   0.0534 .
## I(x^2)      -0.115500   0.078893  -1.464   0.1500
## I(x^3)       0.001658   0.001017   1.630   0.1100
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.95 on 46 degrees of freedom
## Multiple R-squared:  0.6234, Adjusted R-squared:  0.5988
## F-statistic: 25.38 on 3 and 46 DF,  p-value: 7.747e-10
```

The coefficients for x^2 and x^3 are not statistically significant, as shown by the ANOVA table. What they actually were was -0.1155002 and 0.0016582. Magic!

Note that these are coefficients for a regression on random data that is different every time the document is run (although wouldn't be if we uncommented the `set.seed` command). By not using snarf-and-barf we always have the right numbers.

Also note that R Markdown (in its wisdom or unwisdom) forces you to know a little bit of LaTeX. When the number needs "scientific notation" R Markdown emits LaTeX here so the number must be enclosed in dollar signs (the LaTeX math marker for in-line math). So if these weren't enclosed in dollar signs nonsense would be printed.

2.5.5 R Chunk Options

We have already illustrated a few R chunk options.

Another useful one is `echo = FALSE`. In this paragraph there is such a code chunk, but you don't see it.

That is, you don't see it unless you look at the source.

But we can use it in another code chunk.

```
hide
```

```
## [1] 6.283185
```

Any knitr chunk option found at https://yihui.name/knitr/options/ can be used as an R Markdown chunk option (knitr underlies rmarkdown).

Another useful option is `cache = TRUE`. That will be illustrated in the next chapter.

Chapter 3

Parallel Computing in R

3.1 R Markdown Source

https://raw.githubusercontent.com/cjgeyer/Orientation2018/master/02-parallel.Rmd

3.2 Introduction

The example that we will use throughout this document is simulating the sampling distribution of the MLE for Normal(θ, θ^2) data.

A lot of this may make make no sense. This is what you are going to graduate school in statistics to learn. But we want a non-toy example.

3.3 Set-Up

```
# sample size
n <- 10
# simulation sample size
nsim <- 1e4
# true unknown parameter value
# of course in the simulation it is known, but we pretend we don't
# know it and estimate it
theta <- 1

doit <- function(estimator, seed = 42) {
    set.seed(seed)
    result <- double(nsim)
    for (i in 1:nsim) {
        x <- rnorm(n, theta, abs(theta))
        result[i] <- estimator(x)
    }
    return(result)
}

mlogl <- function(theta, x) sum(- dnorm(x, theta, abs(theta), log = TRUE))
```

```
mle <- function(x) {
    theta.start <- sign(mean(x)) * sd(x)
    if (all(x == 0) || theta.start == 0)
        return(0)
    nout <- nlm(mlogl, theta.start, iterlim = 1000, x = x)
    if (nout$code > 3)
        return(NaN)
    return(nout$estimate)
}
```

- R function `doit` simulates `nsim` datasets, applies an estimator supplied as an argument to the function to each, and returns the vector of results.

- R function `mlogl` is minus the log likelihood of the model in question. We could easily change the code to do another model by changing only this function. (When the code mimics the math, the design is usually good.)

- R function `mle` calculates the estimator by calling R function `nlm` to minimize it. The starting value, either `sign(mean(x)) * sd(x)` is a reasonable estimator because `mean(x)` is a consistent estimator of θ and `sd(x)` is a consistent estimator of $|\theta|$.

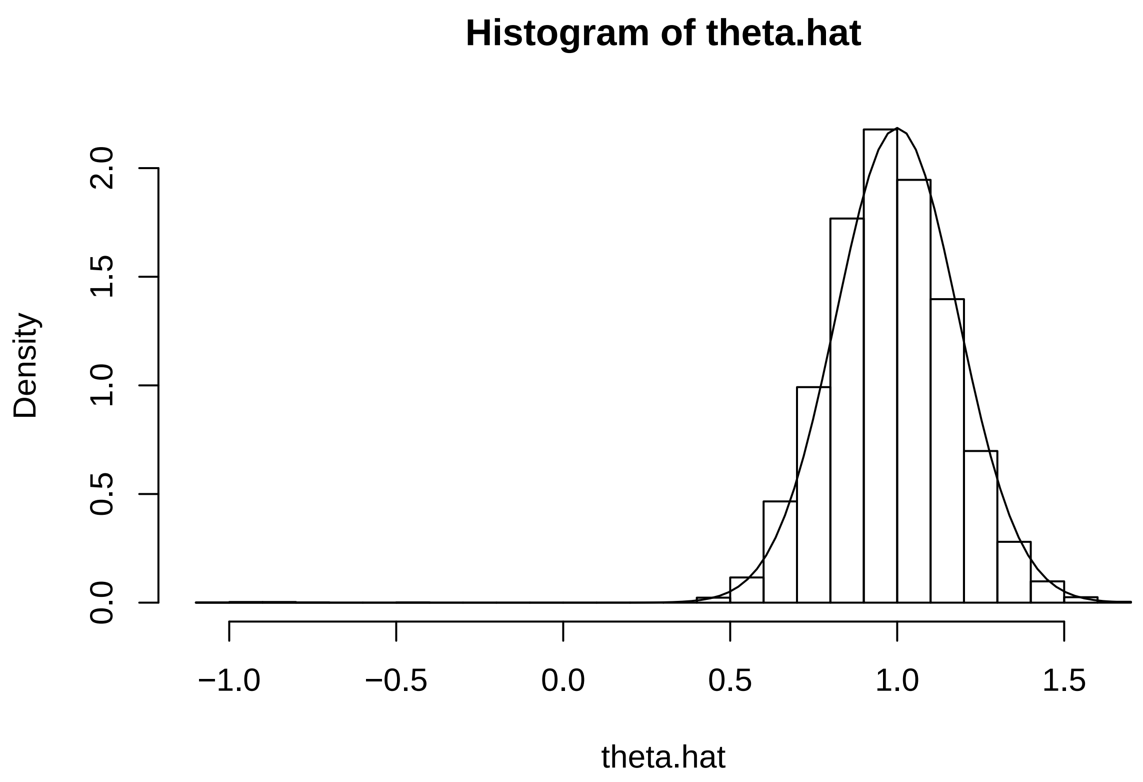
3.4 Doing the Simulation without Parallelization

3.4.1 Try It

```
theta.hat <- doit(mle)
```

3.4.2 Check It

```
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



The curve is the PDF of the asymptotic normal distribution of the MLE, which uses the formula

$$I_n(\theta) = \frac{3n}{\theta^2}$$

which you will learn how to calculate when you take a theory course (if you don't already know).

Looks pretty good. The large negative estimates are probably not a mistake. The parameter is allowed to be negative, so sometimes the estimates come out negative even though the truth is positive. And not just a little negative because $|\theta|$ is also the standard deviation, so it cannot be small and the model fit the data.

```r
sum(is.na(theta.hat))
```

```
## [1] 0
```

```r
mean(is.na(theta.hat))
```

```
## [1] 0
```

```r
sum(theta.hat < 0, na.rm = TRUE)
```

```
## [1] 9
```

```r
mean(theta.hat < 0, na.rm = TRUE)
```

```
## [1] 9e-04
```

3.4.3 Time It

Now for something new. We will time it.

```r
time1 <- system.time(theta.hat.mle <- doit(mle))
time1
```

```
##    user  system elapsed
##   0.987   0.000   0.989
```

3.4.4 Time It More Accurately

That's too short a time for accurate timing. Also we should probably average over several IID iterations to get a good average. Try again.

```
nsim <- 1e5
nrep <- 7
time1 <- NULL
for (irep in 1:nrep)
    time1 <- rbind(time1, system.time(theta.hat.mle <- doit(mle)))
time1
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     9.863        0   9.863          0         0
## [2,]     9.693        0   9.694          0         0
## [3,]     9.630        0   9.630          0         0
## [4,]     9.719        0   9.719          0         0
## [5,]     9.716        0   9.716          0         0
## [6,]     9.760        0   9.760          0         0
## [7,]     9.982        0   9.982          0         0
```

```
apply(time1, 2, mean)
```

```
##  user.self   sys.self    elapsed user.child  sys.child
##   9.766143   0.000000   9.766286   0.000000   0.000000
```

```
apply(time1, 2, sd) / sqrt(nrep)
```

```
##  user.self   sys.self    elapsed user.child  sys.child
## 0.04489004 0.00000000 0.04485146 0.00000000 0.00000000
```

3.5 Parallel Computing With Unix Fork and Exec

3.5.1 Introduction

This method is by far the simplest but

- it only works on one computer (using however many simultaneous processes the computer can do), and
- it does not work on Windows.

3.5.2 Toy Problem

First a toy problem that does nothing except show that we are actually using different processes.

```
library(parallel)
ncores <- detectCores()
mclapply(1:ncores, function(x) Sys.getpid(), mc.cores = ncores)
```

```
## [[1]]
## [1] 31733
##
```

```
## [[2]]
## [1] 31734
##
## [[3]]
## [1] 31735
##
## [[4]]
## [1] 31736
##
## [[5]]
## [1] 31737
##
## [[6]]
## [1] 31738
##
## [[7]]
## [1] 31739
##
## [[8]]
## [1] 31740
```

3.5.3 Parallel Streams of Random Numbers

To get random numbers in parallel, we need to use a special random number generator (RNG) designed for parallelization.

```
RNGkind("L'Ecuyer-CMRG")
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[2]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[3]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[4]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[5]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[6]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[7]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[8]]
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
```

```
set.seed(42)
mclapply(1:ncores, function(x) rnorm(5), mc.cores = ncores)
```

```
## [[1]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
##
## [[2]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[3]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[4]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[5]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[6]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[7]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
##
## [[8]]
## [1]  1.3502819 -1.0055894 -0.4591798 -0.0628527 -0.2706805
```

Just right! We have different random numbers in all our jobs. And it is reproducible.

But this may not work like you may think it does. If we do it again we get exactly the same results. Running `mclapply` does not change `.Random.seed` in the parent process (the R process you are typing into). It only changes it in the child processes (that do the work). But there is no communication from child to parent *except* the list of results returned by `mclapply`.

This is a fundamental problem with `mclapply` and the fork-exec method of parallelization. And it has no real solution. You just have to be aware of it.

If you want to do exactly the same random thing with `mclapply` and get different random results, then you must change `.Random.seed` in the parent process, either with `set.seed` or by otherwise using random numbers *in the parent process*.

3.5.4 The Example

We need to rewrite our `doit` function

- to only do `1 / ncores` of the work in each child process,
- to not set the random number generator seed, and
- to take an argument in some list we provide.

```
doit <- function(nsim, estimator) {
    result <- double(nsim)
    for (i in 1:nsim) {
        x <- rnorm(n, theta, abs(theta))
        result[i] <- estimator(x)
    }
    return(result)
}
```

3.5.5 Try It

```
mout <- mclapply(rep(nsim / ncores, ncores), doit,
    estimator = mle, mc.cores = ncores)
lapply(mout, head)
```

```
## [[1]]
## [1] 0.9051972 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320
##
## [[2]]
## [1] 0.8317815 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521
##
## [[3]]
## [1] 0.8627829 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396
##
## [[4]]
## [1] 1.0422013 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716
##
## [[5]]
## [1] 0.8057316 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027
##
## [[6]]
## [1] 1.0156983 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221
##
## [[7]]
## [1] 1.2287013 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714
##
## [[8]]
## [1] 0.7768910 1.0376265 0.8830854 0.8911714 1.0288567 1.1609360
```

3.5.6 Check It

Seems to have worked.

```
length(mout)
```

```
## [1] 8
```

```
sapply(mout, length)
```

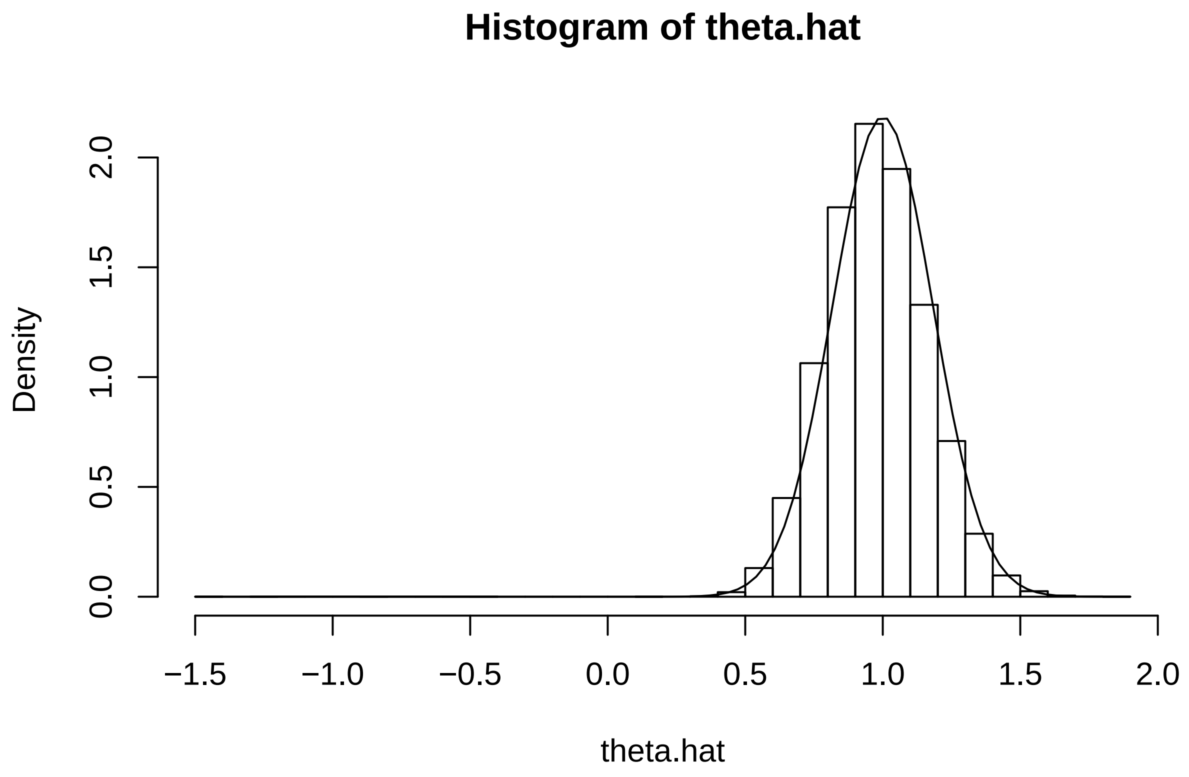
```
## [1] 12500 12500 12500 12500 12500 12500 12500 12500
```

```
lapply(mout, head)
```

```
## [[1]]
## [1] 0.9051972 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320
##
## [[2]]
## [1] 0.8317815 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521
##
## [[3]]
## [1] 0.8627829 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396
##
## [[4]]
## [1] 1.0422013 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716
##
## [[5]]
## [1] 0.8057316 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027
##
## [[6]]
## [1] 1.0156983 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221
##
## [[7]]
## [1] 1.2287013 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714
##
## [[8]]
## [1] 0.7768910 1.0376265 0.8830854 0.8911714 1.0288567 1.1609360
```

Plot it.

```
theta.hat <- unlist(mout)
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



3.5.7 Time It

```
time4 <- NULL
for (irep in 1:nrep)
    time4 <- rbind(time4, system.time(theta.hat.mle <-
        unlist(mclapply(rep(nsim / ncores, ncores), doit,
            estimator = mle, mc.cores = ncores))))
time4
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     0.007    0.021   2.076     13.991     0.225
## [2,]     0.006    0.021   2.842     19.136     0.247
## [3,]     0.000    0.026   3.013     20.365     0.260
## [4,]     0.000    0.024   3.146     20.880     0.304
## [5,]     0.002    0.025   4.328     24.477     0.382
## [6,]     0.003    0.031   3.636     22.148     0.372
## [7,]     0.000    0.036   3.240     20.928     0.304
```

```
apply(time4, 2, mean)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
##  0.002571429  0.026285714  3.183000000 20.275000000  0.299142857
```

```
apply(time4, 2, sd) / sqrt(nrep)
```

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 0.001109636 0.002066908 0.262382944 1.222326470 0.022870681
```

We got the desired speedup. The elapsed time averages

```
apply(time4, 2, mean)["elapsed"]
```

```
## elapsed
##   3.183
```

with parallelization and

```
apply(time1, 2, mean)["elapsed"]
```

```
##  elapsed
## 9.766286
```

without parallelization. But we did not get an 8-fold speedup with 8 cores. There is a cost to starting and stopping the child processes. And some time needs to be taken from this number crunching to run the rest of the computer. However, we did get a 3.1-fold speedup. If we had more cores, we could do even better.

3.6 The Example With a Cluster

3.6.1 Introduction

This method is more complicated but

- it works on clusters like the ones at LATIS (College of Liberal Arts Technologies and Innovation Services or at the Minnesota Supercomputing Institute.
- according to the documentation, it does work on Windows.

3.6.2 Toy Problem

First a toy problem that does nothing except show that we are actually using different processes.

```
library(parallel)
ncores <- detectCores()
cl <- makePSOCKcluster(ncores)
parLapply(cl, 1:ncores, function(x) Sys.getpid())
```

```
## [[1]]
## [1] 31766
##
## [[2]]
## [1] 31775
##
## [[3]]
## [1] 31784
##
## [[4]]
## [1] 31793
##
## [[5]]
## [1] 31802
##
## [[6]]
## [1] 31811
##
## [[7]]
## [1] 31820
##
## [[8]]
## [1] 31829
```

```
stopCluster(cl)
```

This is more complicated in that

- first you you set up a cluster, here with `makePSOCKcluster` but not everywhere — there are a variety of different commands to make clusters and the command would be different at LATIS or MSI — and
- at the end you tear down the cluster with `stopCluster`.

Of course, you do not need to tear down the cluster before you are done with it. You can execute multiple `parLapply` commands on the same cluster.

There are also a lot of other commands other than `parLapply` that can be used on the cluster. We will see some of them below.

3.6.3 Parallel Streams of Random Numbers

```
cl <- makePSOCKcluster(ncores)
clusterSetRNGStream(cl, 42)
parLapply(cl, 1:ncores, function(x) rnorm(5))
```

```
## [[1]]
## [1] -0.93907708 -0.04167943  0.82941349 -0.43935820 -0.31403543
##
## [[2]]
## [1]  1.11932846 -0.07617141 -0.35021912 -0.33491161 -1.73311280
```

```
##
## [[3]]
## [1] -0.2084809 -1.0341493 -0.2629060  0.3880115  0.8331067
##
## [[4]]
## [1]  0.001100034  1.763058291 -0.166377859 -0.311947389  0.694879494
##
## [[5]]
## [1]  0.2262605 -0.4827515  1.7637105 -0.1887217 -0.7998982
##
## [[6]]
## [1]  0.8584220 -0.3851236  1.0817530  0.2851169  0.1799325
##
## [[7]]
## [1] -1.1378621 -1.5197576 -0.9198612  1.0303683 -0.9458347
##
## [[8]]
## [1] -0.04649149  3.38053730 -0.35705061  0.17722940 -0.39716405
```

```
parLapply(cl, 1:ncores, function(x) rnorm(5))
```

```
## [[1]]
## [1] -2.1290236  2.5069224 -1.1273128  0.1660827  0.5767232
##
## [[2]]
## [1] 0.03628534 0.29647473 1.07128138 0.72844380 0.12458507
##
## [[3]]
## [1] -0.1652167 -0.3262253 -0.2657667  0.1878883  1.4916193
##
## [[4]]
## [1]  0.3541931 -0.6820627 -1.0762411 -0.9595483  0.0982342
##
## [[5]]
## [1]  0.5441483  1.0852866  1.6011037 -0.5018903 -0.2709106
##
## [[6]]
## [1] -0.57445721 -0.86440961 -0.77401840  0.54423137 -0.01006838
##
## [[7]]
## [1] -1.3057289  0.5911102  0.8416164  1.7477622 -0.7824792
##
## [[8]]
## [1]  0.9071634  0.2518615 -0.4905999  0.4900700  0.7970189
```

We see that clusters do not have the same problem with continuing random number streams that the fork-exec mechanism has.

- Using fork-exec there is a *parent* process and *child* processes (all running on the same computer) and the *child* processes end when their work is done (when `mclapply` finishes).

- Using clusters there is a *master* process and *slave* processes (possibly running on many different computers) and the *slave* processes end when the cluster is torn down (with `stopCluster`).

So the slave processes continue and remember where they are in the random number stream.

3.6.4 The Example on a Cluster

3.6.4.1 Set Up

Another complication of using clusters is that the slave processes are completely independent of the master. Any information they have must be explicitly passed to them.

This is very unlike the fork-exec model in which all of the child processes are copies of the parent process inheriting all of its memory (and thus knowing about any and all R objects it created).

So in order for our example to work we must explicitly distribute stuff to the cluster.

```
clusterExport(cl, c("doit", "mle", "mlogl", "n", "nsim", "theta"))
```

Now all of the slaves have those R objects, as copied from the master process right now. If we change them in the master (pedantically if we change the R objects those *names* refer to) the slaves won't know about it. They only would make changes if code were executed on them to do so.

3.6.4.2 Try It

So now we are set up to try our example.

```
pout <- parLapply(cl, rep(nsim / ncores, ncores), doit, estimator = mle)
```

3.6.4.3 Check It

Seems to have worked.

```
length(pout)
```

```
## [1] 8
```

```
sapply(pout, length)
```

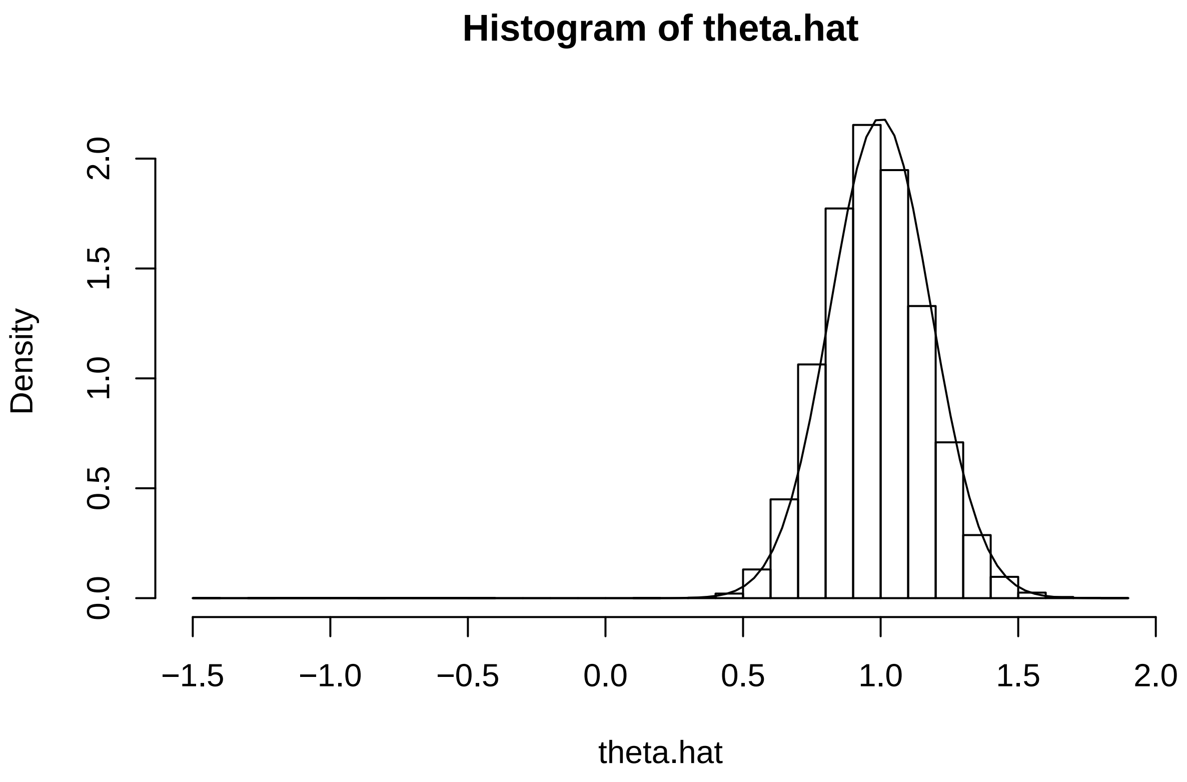
```
## [1] 12500 12500 12500 12500 12500 12500 12500 12500
```

```
lapply(pout, head)
```

```
## [[1]]
## [1] 1.0079313 0.7316543 0.4958322 0.7705943 0.7734226 0.6158992
##
## [[2]]
## [1] 0.9589889 0.9799828 1.1347548 0.9090886 0.9821320 1.0032531
##
## [[3]]
## [1] 1.3432331 0.7821308 1.2010078 0.9792244 1.1148521 0.9269000
##
## [[4]]
## [1] 0.9790400 1.1787975 0.7852431 1.2942963 1.0768396 0.7546295
##
## [[5]]
## [1] 0.9166641 0.8326720 1.1864809 0.9609456 1.3137716 0.9832663
##
## [[6]]
## [1] 0.9488173 1.0792078 0.9774531 0.8106612 0.8403027 1.1296857
##
## [[7]]
## [1] 1.0077599 0.9867766 1.1643493 0.9478923 1.1770221 1.2789464
##
## [[8]]
## [1] 1.0046353 0.9560784 1.0354414 0.9045423 0.9455714 1.0312553
```

Plot it.

```r
theta.hat <- unlist(mout)
hist(theta.hat, probability = TRUE, breaks = 30)
curve(dnorm(x, mean = theta, sd = theta / sqrt(3 * n)), add = TRUE)
```



3.6.4.4 Time It

```r
time5 <- NULL
for (irep in 1:nrep)
    time5 <- rbind(time5, system.time(theta.hat.mle <-
        unlist(parLapply(cl, rep(nsim / ncores, ncores),
            doit, estimator = mle))))
time5
```

```
##      user.self sys.self elapsed user.child sys.child
## [1,]     0.003    0.004   3.460          0         0
## [2,]     0.007    0.000   3.270          0         0
## [3,]     0.006    0.000   3.402          0         0
## [4,]     0.006    0.000   3.425          0         0
## [5,]     0.005    0.000   3.491          0         0
## [6,]     0.007    0.000   3.382          0         0
## [7,]     0.004    0.004   3.696          0         0
```

```
apply(time5, 2, mean)
```

```
##   user.self    sys.self     elapsed  user.child   sys.child
## 0.005428571 0.001142857 3.446571429 0.000000000 0.000000000
```

```
apply(time5, 2, sd) / sqrt(nrep)
```

```
##    user.self     sys.self      elapsed   user.child    sys.child
## 0.0005714286 0.0007377111 0.0493210363 0.0000000000 0.0000000000
```

We got the desired speedup. The elapsed time averages

```
apply(time5, 2, mean)["elapsed"]
```

```
##  elapsed
## 3.446571
```

with parallelization and

```
apply(time1, 2, mean)["elapsed"]
```

```
##  elapsed
## 9.766286
```

without parallelization. But we did not get an 8-fold speedup with 8 cores. There is a cost to starting and stopping the child processes. And some time needs to be taken from this number crunching to run the rest of the computer. However, we did get a 2.8-fold speedup. If we had more cores, we could do even better.

We also that this method isn't as good as the other method. So why do we want it (other than that the other doesn't work on Windows)? Because it scales. You can get clusters with thousands of cores, but you can't get thousands of cores in one computer.

3.6.5 Tear Down

Don't forget to tear down the cluster when you are done.

```
stopCluster(cl)
```

3.7 LATIS and Fork-Exec

We are going to just show how to do this in an interactive session.

which is just like the fork-exec part of this document except for a few minor changes for running on LATIS.

SSH into `compute.cla.umn.edu`

Then

```
qsub -I -l nodes=1:ppn=8
cd tmp/Orientation2018 # or wherever
rm -f 02-fork-exec.R
wget https://raw.githubusercontent.com/cjgeyer/Orientation2018/master/02-fork-exec.R
module load R/3.4.1
# we are assured that a more recent version of R will be available soon
R CMD BATCH --vanilla 02-fork-exec.R
cat 02-fork-exec.Rout
exit
exit
```

Shows that this works.

We are not going to try to explain today how to do this as a batch job.

3.8 LATIS and Clusters

Almost the same thing again

```
qsub -I -l nodes=1:ppn=8
cd tmp/Orientation2018 # or wherever
rm -f 02-cluster.R
wget https://raw.githubusercontent.com/cjgeyer/Orientation2018/master/02-cluster.R
module load R/3.4.1
# we are assured that a more recent version of R will be available soon
R CMD BATCH --vanilla 02-cluster.R
cat 02-cluster.Rout
exit
exit
```

except this doesn't seem to work right now. LATIS seems to have to do some more work on MPI clusters and R.

Chapter 4

Version Control with Git

4.1 Reading

The best book on Git that I know of is *Pro Git*, which can be bought but is also available for free on the web at https://git-scm.com/book/en/v2.

4.2 Git Command Line

There should be instruction on using Git from the command line, but there's no time. The *Pro Git* book, or my notes for the IRSA short course at https://irsaatumn.github.io/RWorkshop18/git-version-control.html will have to do.

4.3 Git from RStudio

These are notes on how to use Git the dumb way using RStudio and GitHub.

In your GitHub account on `github.umn.edu`

- Make a new repo that we can trash later (just for messing around).
  - Log in to GitHub
  - Click on the green button that says "New repository"
  - Give it a name (of course) and maybe a description.
  - Do not add `.gitignore` file or a README file or a license (maybe do that later)
  - Ignore all the helpful advice on the next page (none of it is for RStudio)

In RStudio

- On the File menu
  - select "New project..."
  - then select "Version Control"
  - then select "Git"
    - the "Repository URL" is something like https://github.umn.edu/geyer/junk

        * the “Project directory name” is something like `junk`
        * the “Create project as subdirectory of” is done with file browser
        * I also did “Open in new session”
    - then punch “Create Project”
- We’re in business !!!!!
- Note that RStudio creates 2 files automagically (`.gitignore` and `*.Rproj`) we may have to edit the former to do what we want.
- Before we do any commits we have to tell Git who you are (unless you have used Git previously and have already done this)
    - On the Tools menu
        * select “Terminal”
        * then select “New Terminal”
        * then you can do

          ```
          git config --global user.name "Your Name"

          git config --global user.email "your@email.com"
          ```

          (I don’t have to do this because I did it long ago) where “Your Name” and “your@email.com” are replaced by your actual name and e-mail address.
        * Then click on the “Console” button to get back to the R console
- Open a new R markdown file
    - On the File menu
        * select “New File”
        * then select “R Markdown”
            · and in the window that comes up fill in a title and author
            · and click the “OK” button
- RStudio then gives you a simple example R Markdown file
- click the “Knit” button (in the upper left pane) and it first asks you to save the file (by default in the directory for our project). Do that: give it a name ending in .Rmd, for example, `mess.Rmd`
- Now we are really in business !!!!!
- Now do a `git commit` (either using the command line) or using the “Git” button in the upper right pane in RStudio
- in the window that comes up click to stage at least the *.Rproj* .Rmd and .gitignore files
- and type in a commit message
- and click the “commit” button
- If we go to the command line and do “git status” or “git log” we see that we have indeed done a commit.
- Now we do a push using the “Push” button in the upper right panel (oh garbage! We have to type user name and password every time we do anything! This sucks! RStudio is infinitely inferior to the command line in this respect. See the chapter. Actually, that was unfair. RStudio can help you use SSH keys. On the “Tools” menu, choose “Global Options” and then “Git/SVN”)

Nevertheless, we have done a commit and push. As can be seen by looking on the repo on GitHub.

- Now we can make all the changes we want to the "project" and commit and push whenever we want and if we had collaborators we could also pull from them (I assume). We definitely could if we were using the command line.

More reading. More stuff about using RStudio with Git and GitHub can be found at Hadley Wickham's web site http://r-pkgs.had.co.nz/git.html.