Introduction to Statistical Programming with R

Aileen Brown Cuevas
Thomas Wiemann
University of Chicago

July 29, 2021

Outline

Statistical Programming

Statistical programming may be defined as the implementation of statistical procedures and statistical analyses on a computer.

To do so efficiently, the choice of the right tool (i.e., programming language) for the problem at hand is crucial. Four distinctions are particularly relevant:

- high-level languages (more human readable, less control) vs low-level languages (less human readable, more control)
- domain languages (optimized for specific tasks) vs general-purpose languages (suitable for large variety of tasks)
- availability of packages
- open-source/free vs proprietary/commercial

For statistical programming, this leaves us with C/C++ for low-level high-performance code, and R, Python, and Julia for higher level code.

Why use R

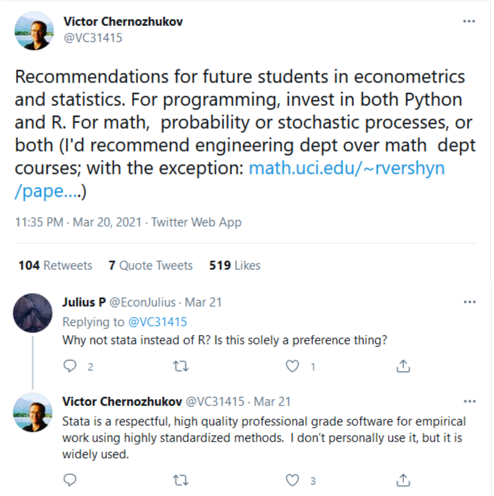
Pros:

- It's free, open source, and available on every major platform.
- Widely used and sought-after language (e.g., see IEEE Spectrum)
- Large and incredibly helpful community (e.g., see Chadley Wickham)
- Excellent availability of data manipulation and visualization packages (e.g., `dplyr`, `ggplot2`).
- Possible to connect to low-level programming languages such as C++ (e.g., `Rcpp`).
- *Thousands* of packages designed and maintained by statisticians. Lot's of cutting-edge methods implemented in R.

Cons:

- Thousands of packages designed and maintained by *statisticians*. We're not professional programmers – code often messy.
- Easy to write, difficult to perfect. Naive code can turn out slow.
- Can be very memory intensive.
- Advanced numerical optimization methods largely missing.

Programming Pluralism



Victor Chernozhukov
@VC31415

Recommendations for future students in econometrics and statistics. For programming, invest in both Python and R. For math, probability or stochastic processes, or both (I'd recommend engineering dept over math dept courses; with the exception: math.uci.edu/~rvershyn /pape....)

11:35 PM · Mar 20, 2021 · Twitter Web App

104 Retweets **7** Quote Tweets **519** Likes

Julius P @EconJulius · Mar 21
Replying to @VC31415
Why not stata instead of R? Is this solely a preference thing?

2 1

Victor Chernozhukov @VC31415 · Mar 21
Stata is a respectful, high quality professional grade software for empirical work using highly standardized methods. I don't personally use it, but it is widely used.

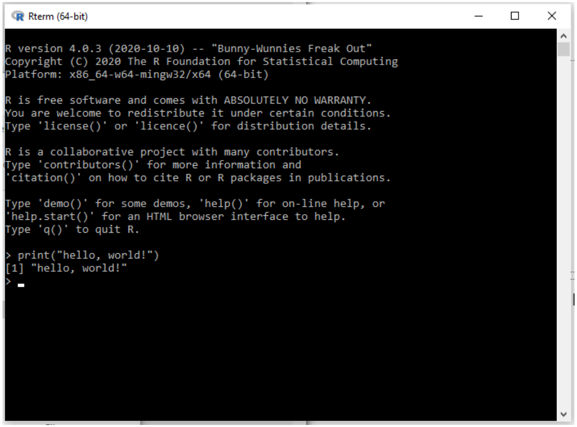
3

Source: `https://twitter.com/VC31415/status/1373402880280047617`

Takeaway: use the most convenient tool for the project at hand.

Getting started: Base R

Download & install R: `https://cran.rstudio.com/`



```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("hello, world!")
[1] "hello, world!"
> 
```

Rough! Better use a integrated development environment (IDE).

Getting started: IDEs

Most popular IDE for R is R Studio:

- Designed specifically for R.
- Project management and integrated version control.
- Debugging support.
- Integrated R Markdown.

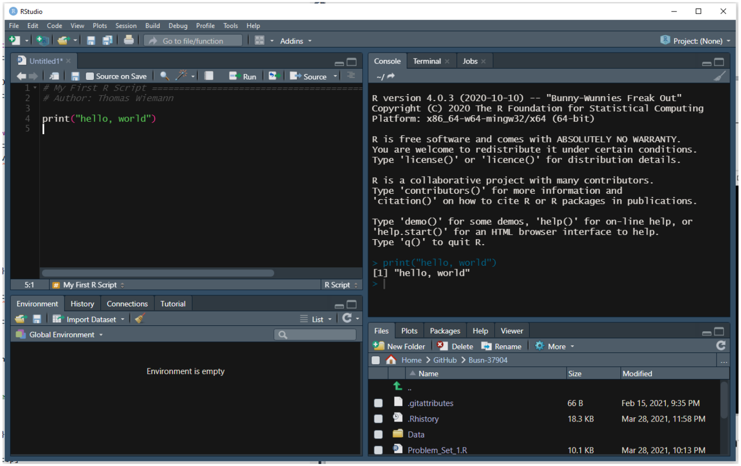
A sometimes convenient alternative to R Markdown is a Jupyter Notebook, especially if you're familiar with Jupyter Notebooks through working with Python or Julia. (Note: Ju-Py-[t]-R)

Personally, I do 99% of R coding in R Studio and 1% in Jupyter Notebooks.

Getting started: R Studio

Download & install R Studio:

`https://rstudio.com/products/rstudio/download`



```r
# My First R Script ========================================
# Author: Thomas Wiemann

print("hello, world")
```

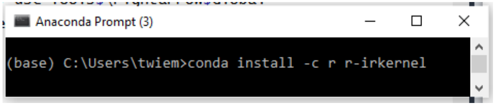
```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("hello, world")
[1] "hello, world"
>
```

Name	Size	Modified
..		
.gitattributes	66 B	Feb 15, 2021, 9:35 PM
.Rhistory	18.3 KB	Mar 28, 2021, 11:58 PM
Data		
Problem_Set_1.R	10.1 KB	Mar 28, 2021, 10:13 PM

To change visual appearance, use Tools⟶Global Options⟶Appearance and change the editor theme (here: *Chaos*).

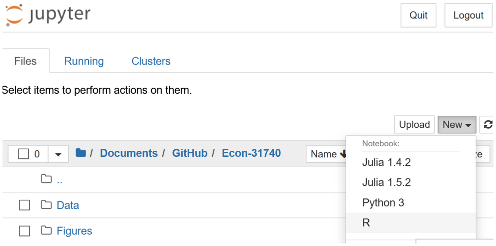
Getting started: Jupyter Notebooks for R

- Download & install Anaconda: `https://www.anaconda.com/products/individual`
- Install R kernel via Anaconda prompt



```
(base) C:\Users\twiem>conda install -c r r-irkernel
```

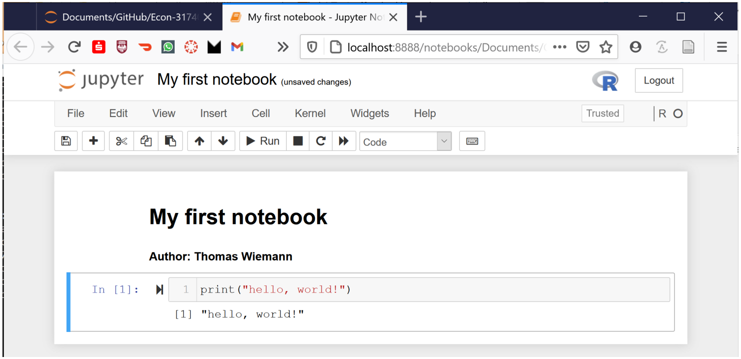
- Open Jupyter Notebook and select the R kernel



Select items to perform actions on them.

Getting started: Jupyter Notebooks for R (Contd.)

Jupyter Notebooks are a browser-based application and so will open in Chrome, Firefox, etc.


Documents/GitHub/Econ-3174
My first notebook - Jupyter No
localhost:8888/notebooks/Documents/
jupyter
My first notebook (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
R
Run
Code

My first notebook

Author: Thomas Wiemann

```
In [1]:  print("hello, world!")
[1] "hello, world!"
```

They are particularly useful for sharing output scripts that include brief sections of code and lots of explanation.

Version Control: Overview

A key feature of any editor is the UNDO operator (e.g., `ctrl + z`). Intuitively, a version control system is a sophisticated UNDO operator.

Working with version control allows you to

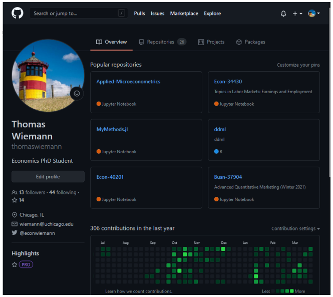
- track changes in your code;
- compare different versions of your code;
- rebuild your code as it existed at any prior point.

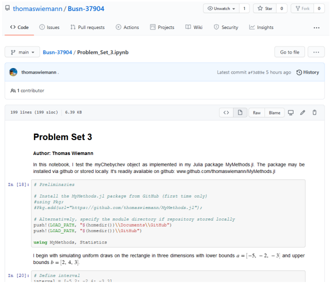
Invaluable for debugging and collaboration!

A particularly popular version control system is Git. It's free, available on all major operating systems, and widely supported by free hosting websites (e.g., GitHub). R Studio also has a Git integration.

Version Control: Git with GitHub

GitHub is a free hosting website for code. Excellent for sharing code (like problem sets). Lots of software packages are hosted on GitHub.





Problem Set 3

Author: Thomas Wiemann

In this notebook, I test the myChebychev object as implemented in my Julia package MyMethods.jl. The package may be installed via github or stored locally. It's readily available on github: www.github.com/thomaswiemann/MyMethods.jl

```
In [18]: # Preliminaries

         # Install the MyMethods.jl package from GitHub (first time only)
         #using Pkg;
         #Pkg.add(url="https://github.com/thomaswiemann/MyMethods.jl");

         # Alternatively, specify the module directory if repository stored locally
         push!(LOAD_PATH, "$(homedir())\\Documents\\GitHub")
         push!(LOAD_PATH, "$(homedir())\\GitHub")

         using MyMethods, Statistics
```

I begin with simulating uniform draws on the rectangle in three dimensions with lower bounds $a = [-5, -2, -3]$ and upper bounds $b = [2, 4, 3]$.

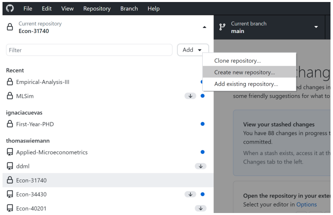
```
In [20]: # Define interval
         interval = [-5 2; -2 4; -3 3]
```

Installation:

1. Sign-up for GitHub: `www.github.com`
2. Download the GitHub desktop app: `https://desktop.github.com`
3. After installation, open GitHub app and sign-in with your account.

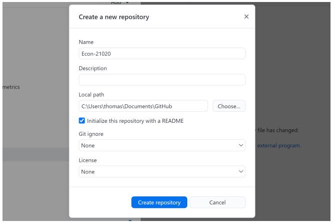
We're now ready to start our first repository!

Sample problem set repository



View your stashed changes
You have 88 changes in progress t
committed.
When a stash exists, access it at th
Changes tab to the left.
Open the repository in your exte
Select your editor in Options

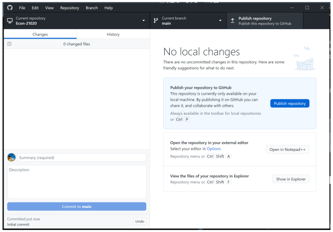
(a) File→Add→Create new...



Name
Econ-21020
Description

Local path
C:\Users\thomas\Documents\GitHub
☑ Initialize this repository with a README
Git ignore
None
License
None

(b) Name appropriately



0 changed files

No local changes
There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.

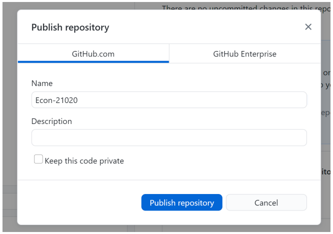
Publish your repository to GitHub
This repository is currently only available on your local machine. By publishing it on GitHub you can share it, and collaborate with others.
Always available in the toolbar for local repositories or Ctrl P

Open the repository in your external editor
Select your editor in Options
Repository menu or Ctrl Shift A

View the files of your repository in Explorer
Repository menu or Ctrl Shift F

Committed just now
Initial commit

(c) "publish" repository



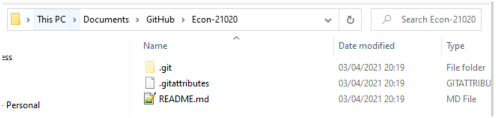
Name
Econ-21020
Description

☐ Keep this code private

(d) Decide whether to keep private

Sample problem set repository (Contd.)

After initialization, the repository contains only the bare essentials.



This PC › Documents › GitHub › Econ-21020

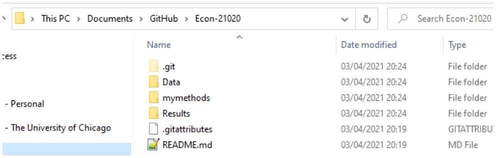
Search Econ-21020

ess

Personal

Name	Date modified	Type
.git	03/04/2021 20:19	File folder
.gitattributes	03/04/2021 20:19	GITATTRIBU
README.md	03/04/2021 20:19	MD File

Before adding code, it's useful to organize the repository for a clean workflow. For a problem set repository, it's often convenient to have separate folders for data, results, and complementary code (here: `mymethods`).



This PC › Documents › GitHub › Econ-21020

Search Econ-21020

cess

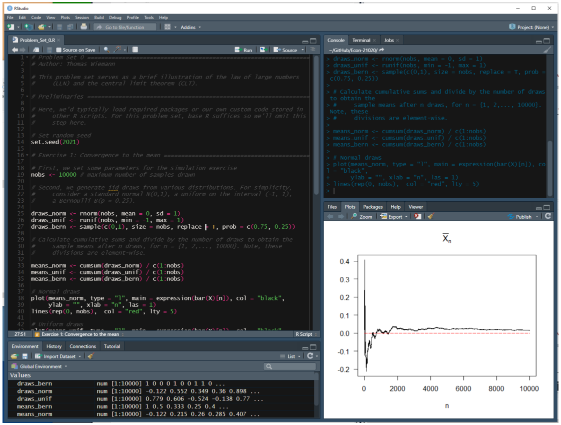
- Personal

- The University of Chicago

Name	Date modified	Type
.git	03/04/2021 20:24	File folder
Data	03/04/2021 20:24	File folder
mymethods	03/04/2021 20:24	File folder
Results	03/04/2021 20:24	File folder
.gitattributes	03/04/2021 20:19	GITATTRIBU
README.md	03/04/2021 20:19	MD File

Sample problem set repository (Contd.)

Let's now add our first R script. As an example for a problem set, `Problem_Set_0.R` provides a basic simulation exercises for the law of large numbers and the central limit theorem.



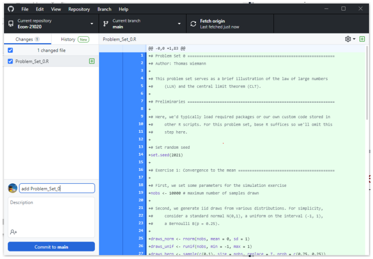
```r
# Problem Set 0 ================================================
# Author: Thomas Wiemann

# This problem set serves as a brief illustration of the law of large numbers
#     (LLN) and the central limit theorem (CLT).

# Preliminaries ================================================

# Here, we'd typically load required packages or our own custom code stored in
#     other R scripts. For this problem set, base R suffices so we'll omit this
#     step here.

# Set random seed
set.seed(2021)

# Exercise 1: Convergence to the mean ==========================

# First, we set some parameters for the simulation exercise
nobs <- 10000 # maximum number of samples drawn

# Second, we generate iid draws from various distributions. For simplicity,
#     consider a standard normal N(0,1), a uniform on the interval (-1, 1),
#     a Bernoulli B(p = 0.25).

draws_norm <- rnorm(nobs, mean = 0, sd = 1)
draws_unif <- runif(nobs, min = -1, max = 1)
draws_bern <- sample(c(0,1), size = nobs, replace = T, prob = c(0.75, 0.25))

# Calculate cumulative sums and divide by the number of draws to obtain the
#     sample means after n draws, for n = {1, 2,..., 10000}. Note, these
#     divisions are element-wise.

means_norm <- cumsum(draws_norm) / c(1:nobs)
means_unif <- cumsum(draws_unif) / c(1:nobs)
means_bern <- cumsum(draws_bern) / c(1:nobs)

# Normal draws
plot(means_norm, type = "l", main = expression(bar(X)[n]), col = "black",
     ylab = "", xlab = "n", las = 1)
lines(rep(0, nobs),  col = "red", lty = 5)

# Uniform draws
```

```
~/GitHub/Econ-21020/
> draws_norm <- rnorm(nobs, mean = 0, sd = 1)
> draws_unif <- runif(nobs, min = -1, max = 1)
> draws_bern <- sample(c(0,1), size = nobs, replace = T, prob =
 c(0.75, 0.25))
>
> # Calculate cumulative sums and divide by the number of draws
 to obtain the
> #     sample means after n draws, for n = {1, 2,..., 10000}.
 Note, these
> #     divisions are element-wise.
>
> means_norm <- cumsum(draws_norm) / c(1:nobs)
> means_unif <- cumsum(draws_unif) / c(1:nobs)
> means_bern <- cumsum(draws_bern) / c(1:nobs)
>
> # Normal draws
> plot(means_norm, type = "l", main = expression(bar(X)[n]), co
l = "black",
+      ylab = "", xlab = "n", las = 1)
> lines(rep(0, nobs),  col = "red", lty = 5)
> |
```

```
Values
  draws_bern          num [1:10000] 1 0 0 0 1 0 0 1 1 0 ...
  draws_norm          num [1:10000] -0.122 0.552 0.349 0.36 0.898 ...
  draws_unif          num [1:10000] 0.779 0.606 -0.524 -0.138 0.77 ...
  means_bern          num [1:10000] 1 0.5 0.333 0.25 0.4 ...
  means_norm          num [1:10000] -0.122 0.215 0.26 0.285 0.407 ...
```

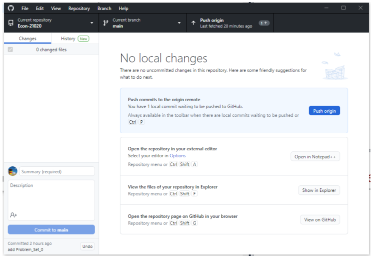
The file should be stored directly in the repository (i.e., it's filepath should look something like: "`~/GitHub/Econ-21020/Problem_Set_0.R`").

Sample problem set repository (Contd.)



```
@@ -0,0 +1,83 @@
+# Problem Set 0 ===============================================================
+# Author: Thomas Wiemann
+
+# This problem set serves as a brief illustration of the law of large numbers
+#     (LLN) and the central limit theorem (CLT).
+
+# Preliminaries ===============================================================
+
+# Here, we'd typically load required packages or our own custom code stored in
+#     other R scripts. For this problem set, base R suffices so we'll omit this
+#     step here.
+
+# Set random seed
+set.seed(2021)
+
+# Exercise 1: Convergence to the mean ==========================================
+
+# First, we set some parameters for the simulation exercise
+nobs <- 10000 # maximum number of samples drawn
+
+# Second, we generate iid draws from various distributions. For simplicity,
+#     consider a standard normal N(0,1), a uniform on the interval (-1, 1),
+#     a Bernoulli B(p = 0.25).
+
+draws_norm <- rnorm(nobs, mean = 0, sd = 1)
+draws_unif <- runif(nobs, min = -1, max = 1)
+draws_bern <- sample(c(0,1), size = nobs, replace = T, prob = c(0.75, 0.25))
```

(a) Add brief description and commit



No local changes

There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.

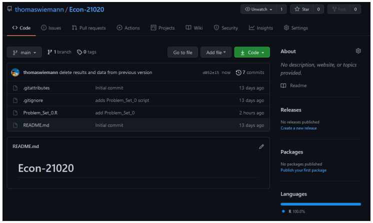
Push commits to the origin remote
You have 1 local commit waiting to be pushed to GitHub.
Always available in the toolbar when there are local commits waiting to be pushed or Ctrl P

Open the repository in your external editor
Select your editor in Options
Repository menu or Ctrl Shift A

View the files of your repository in Explorer
Repository menu or Ctrl Shift F

Open the repository page on GitHub in your browser
Repository menu or Ctrl Shift G

(b) "Push" to upload to GitHub



thomaswiemann delete results and data from previous version d052e15 now 7 commits

.gitattributes	Initial commit	13 days ago
.gitignore	adds Problem_Set_0 script	13 days ago
Problem_Set_0.R	add Problem_Set_0	2 hours ago
README.md	Initial commit	13 days ago

README.md

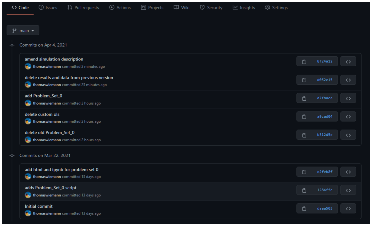
Econ-21020

No description, website, or topics provided.

No releases published
Create a new release

No packages published
Publish your first package

(c) Repository immediately updated...



Commits on Apr 4, 2021

amend simulation description thomaswiemann committed 2 minutes ago	8f24a12
delete results and data from previous version thomaswiemann committed 23 minutes ago	d052e15
add Problem_Set_0 thomaswiemann committed 2 hours ago	d7fbaea
delete custom ols thomaswiemann committed 2 hours ago	a9cad04
delete old Problem_Set_0 thomaswiemann committed 2 hours ago	b312d5e

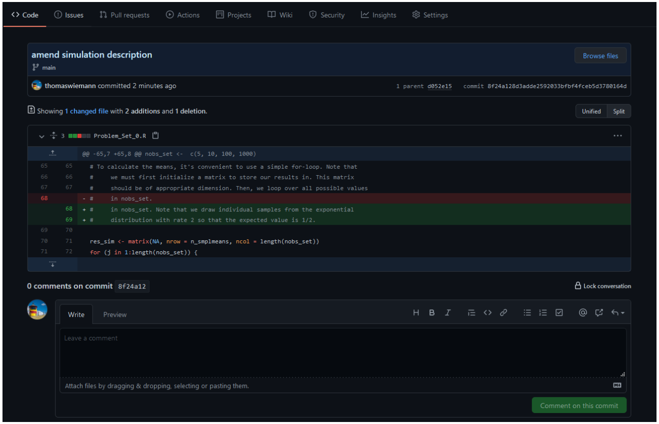
Commits on Mar 22, 2021

add html and ipynb for problem set 0 thomaswiemann committed 13 days ago	e2feb8f
adds Problem_Set_0 script thomaswiemann committed 13 days ago	1284ffe
Initial commit thomaswiemann committed 13 days ago	daaa503

(d) ... but all versions backed-up

Sample problem set repository (Contd.)

Particularly useful: line-by-line comparisons between different versions of the repository.



amend simulation description

main

thomaswiemann committed 2 minutes ago

1 parent d052e15 commit 8f24a128d3adde2592033bfbf4fceb5d3780164d

Showing 1 changed file with 2 additions and 1 deletion.

3 Problem_Set_0.R

```
@@ -65,7 +65,8 @@ nobs_set <-  c(5, 10, 100, 1000)
  # To calculate the means, it's convenient to use a simple for-loop. Note that
  #     we must first initialize a matrix to store our results in. This matrix
  #     should be of appropriate dimension. Then, we loop over all possible values
- #     in nobs_set.
+ #     in nobs_set. Note that we draw individual samples from the exponential
+ #     distribution with rate 2 so that the expected value is 1/2.

  res_sim <- matrix(NA, nrow = n_smplmeans, ncol = length(nobs_set))
  for (j in 1:length(nobs_set)) {
```

0 comments on commit 8f24a12

Best Practices

Getting used to version control and GitHub can take some time. A few best practices may help you get started. (Note that these are not universal guidelines and depend on the project, but it's a good set of rules for homework repositories.)

- Commit recent changes after completing a logical step (e.g., finishing an exercise). This should be done relatively frequently.
- Always add a brief description to your commit.
- Upload to GitHub each time you stop working on your project.
- Avoid uploading large files. Small data sets are fine, large data sets may need to be stored locally.
- When collaborating, "push" and "pull" frequently.

As a student at UChicago, you qualify for GitHub pro. Comes with a few benefits, including unlimited collaborators on private repositories. Link: `https://education.github.com/benefits?type=student`.

Optional Homework

Exercise: setup a problem set repository for this course.

To do so, complete the following steps:

1. Download and install R.
2. Download and install R Studio.
3. Create a GitHub account.
4. Download and install the GitHub desktop application.
5. Create a new **public** repository with the name `Econ-21020` and publish it to GitHub.
6. Add a single R script called `hello_world.R`. The file should contain the following code: `print(‘hello, world!’)`.
7. Commit your changes to the repository and ‘push’ to GitHub.
8. Copy the link of the repository and include it in your next problem set. The link should look something like this (but of course with your username): `www.github.com/thomaswiemann/Econ-21020`.

All necessary instructions can be found in this slide deck.

Introduction to Statistical Programming with R (Part. II)

Aileen Brown Cuevas
Thomas Wiemann
University of Chicago

July 29, 2021

Outline

Data Types

The simplest kind of variables in R are *atomic types*. These do not have an internal structure. Examples are:

- `integer: 1L, 2L, 42L, ...`
- `numeric: pi, 0.25, Inf, ...`
- `logical: TRUE, FALSE.`
- `character: "lorem ipsum", "a", "", ...`

More complex data types are combinations of atomic types. It's useful to organize the most common data types based on their dimensionality and whether they are homogeneous (i.e., whether all of it's contents must be of the same type).

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Data Types: Atomic vectors

Atomic vectors are usually created with `c()`, short for combine. Even when nesting, atomic vectors are always flat.

R does not have any scalar variables. Single numbers or characters are vectors of length 1. Vectors of length 0 also exist (and are a common cause for bugs).

Example: Atomic vectors

```
c(1L, c(2L, c(3L, 4L))) # integer
#> [1] 1 2 3 4

c(1, 2, 3, 4) # numeric
#> [1] 1 2 3 4

c(T, F) # logical
#> [1] TRUE FALSE

c("these␣are", "some␣strings") # character
#> [1] "these are" "some strings"
```

Data Types: Factors

Another fundamental data type that is essentially a vector with attributes is `factor`. These are used to store categorical and ordinal variables.

Factor variables can easily be initialized from an existing vector by using `factor()`.

Example: Factors

```
factor(c("red", "blue", "green", "red", "green"))
#> [1] red blue green red green
#> Levels: blue green red

factor(c(1, 2, 1, 2, 3))
#> [1] 1 2 1 2 3
#> Levels: 1 2 3
```

Data Types: Matrices

A two-dimensional collection of atomic vectors is a `matrix`. They are convenient for storing and manipulating data with linear algebra.

Matrices are easily initialized with `matrix()`. Note that R uses column-wise initialization of matrices. Vectors (or matrices) of appropriate dimension may also be combined via row- or column-concatenation using `rbind()` or `cbind()`.

Example: Matrices

```
matrix(c(1, 2, 3, 4), ncol = 2, nrow = 2)
#>        [, 1] [, 2]
#> [1, ]     1     3
#> [2, ]     2     4

cbind(c(1, 2), c(3, 4))
#>        [, 1] [, 2]
#> [1, ]     1     3
#> [2, ]     2     4
```

Data Types: Arrays

The n-dimensional generalization of `matrix` is called `array`. They are much less common but it's good to be aware of them.

Use `array()` for initialization.

Example: Arrays

```
array(c(1:8), dim = c(2, 2, 2))
#> , , 1
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
#>
#> , , 2
#>      [,1] [,2]
#> [1,]    5    7
#> [2,]    6    8
```

Data Types: Lists

A `list` is a 1-dimensional data type with elements of arbitrary type. Note that this is their key difference to atomic vectors.

Use `list()` for initialization (instead of `c()`).

Example: Lists

```
list(1:3, "a", TRUE, 1L)
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]
#> [1] 1
```

Data Types: Data frames

Data frames are the most common way of storing data in R. Essentially, a data frame is a list of equal-length vectors. Given this construction, it shares some properties of both the matrix and the list, which can be very convenient for data analysis.

Use `data.frame()` for initialization. Multiple data frames (of appropriate dimension) may be concatenated via `rbind()` or `cbind()`.

Example: Data frames

```
data.frame(x = 2:4, y = c("a", "b", "c"))
#>   x y
#> 1 2 a
#> 2 3 b
#> 3 4 c
```

Data Types: Which one to choose?

Sometimes the data gives you little choice in the data structure. More often than not, however, you have to make the decision. Doing so incorrectly can have noticeable consequences. Poor choices can make your code needlessly complicated, slow, and memory-intensive.

As a simple guideline for the choice between data frames and matrices, consider how linear algebra intensive your code is. If you're using linear algebra operations frequently, use a `matrix` instead of a data frame.

For advanced projects (outside the scope of this course), also consider other data structures provided by R packages (e.g., `data.table`, sparse matrices).

Operators

Operators are convenient shortcuts to many of the key operations that frequently arise in data analysis. The key groups of operators are

- Assignment operators
- Arithmetic operators
- Relational operators
- Logical operators
- Subsetting operators

We'll discuss each briefly and conclude with a note on operator preference.

(There are also two linear algebra operators. We'll turn to them later.)

Operators: Assignment

Assignment associates an internal object (e.g., a vector of integers) with a variable in your code. The key assignment operator in R is "<-". It's not the only one – but it's the only one you should use.

There are other assignment operators you may sometimes see in other's code. "=" is identical to "<-" and both deal with local assignment. The global assignment operator in R is "<<-".

Assignment: Assignment Operators

```
x <- c(1, 2)
x # print(x)
#> [1] 1 2

y <- cbind(x, x)
y # print(y)
#>        [, 1] [, 2]
#> [1, ]     1     1
#> [2, ]     2     2
```

Operators: Arithmetic

The familiar arithmetic operators are implemented with their usual symbols. In particular, "+" is addition, "−" is subtraction, "∗" is multiplication, "/" is division, and "^" is the exponent operator. There are two other useful operators: "%%" calculates the modulus and "%/%" calculates integer division.

Note that these operations are element-wise! R is vectorized.

Example: Arithmetic Operators

```
c(1, 2, 3, 4) + 1
#> [1] 2 3 4 5

c(1, 2, 3, 4) * c(1, 2, 3, 4)
#> [1] 1 4 9 16

c(3, 4, 5, 6) %% 3
#> [1] 0 1 2 0
```

Operators: Relational

Relational operators are very convenient for generation of logical variables. In R, the key relational operators are the familiar “greater than” and “smaller than” operators – “>” and “<” – as well as their weak extensions – “>=” and “<=”. Equality is indicated by a double equality sign: “==”. Not-equal is using the negation operator: “`!=`”.

As before: these operations are element-wise!

Example: Relational Operators

```
x <- 1
y <- x + 1
c(y < x, y <= x, y != x)
#> [1] FALSE FALSE TRUE

(c(3, 4, 5, 6) %% 3) == 0
#> [1] TRUE FALSE FALSE TRUE
```

Operators: Logical

Logical operators are convenient for aggregating logical variables. Here, "&" is element-wise *logical-and*, and "|" is element-wise *logical-or*. The negation operator is "!".

In addition to the element-wise logical operators, there are also non-element-wise alternatives – "&&" and "||". Both only evaluate the first logical variable. Be cautious when using them.

Example: Logical Operators

```
!(c(TRUE, FALSE) & TRUE)
#> [1] FALSE TRUE

x <- 1
y <- x + 1
(y < x - 1) | (y > x)
#> [1] TRUE

(c(1, 2, 3, 4) == c(1, 5, 5, 5)) && (x == 1)
#> [1] TRUE
```

Operators: Subsetting

Subsetting allows you to retrieve and manipulate parts of your data structure. This is crucial for succinctly expressing complex operations.

R has three subsetting operators:

- [for vectors, matrices, arrarys, and data frames
- [[for lists
- $ for named lists and data frames

Example: Subsetting Operators

```
df <- data.frame(x = 2:4, y = c("a", "b", "c"))
df[, 1]
#> [1] 2 3 4

df[, 1] == df[, "x"] & df[, 1] == df$x
#> [1] TRUE TRUE TRUE
```

Operators: Precedence

Precedence rules for operators can be complicated. Rather than memorize these rules and risk confusion, use parentheses to avoid ambiguity.

Example: Operator Precedence

```
TRUE || FALSE == FALSE || FALSE
#> [1] TRUE

(TRUE || FALSE) == (FALSE || FALSE)
#> [1] FALSE

TRUE || (FALSE == FALSE) || FALSE
#> [1] TRUE
```

Functions

Functions are a fundamental building block of R: Anything that "happens" is the result of a function call.

Much like it's mathematical equivalent, a function takes some inputs (called *arguments*) and returns an output. In R, a function can only return a single object (e.g., a vector or a list).

Writing your own functions is – thankfully – incredibly easy. If you catch yourself copy-pasting lines of code, consider writing your own function and calling it instead.

Example: Custom Functions

```
myplus <- function(x, y) {
  res <- x + y
  return(res)
}#MYPLUS

myplus(1, 2) == (1 + 2)
#> [1] TRUE
```

Functions

Base R implements many key functions for data analysis. Being familiar with them is important for efficient programming.

Some of the key function groups are:

- Distributional functions
- Statistical functions
- Logical functions
- Linear Algebra
- Importing and Exporting Data

We'll briefly go over each now.

Functions: Distributions

The four key distributional functions we need are

- the probability density (mass) function,
- the cumulative density (mass) function,
- the quantile function,
- the sampling function for generating random draws.

R implements these for all standard distributions with the same pattern:

```
(d, p, q, r) * (binom, chisq, exp, f, norm, t, unif)
```

where `d` denotes the pdf (pmf), `p` denotes the CDF (CMF), `q` denotes the quantile function, and `r` denotes the sampling function.

`binom` denotes the binomial distribution, `chisq` denotes the χ^2-distribution, `exp` denotes the exponential distribution, `f` denotes the F-distribution, `norm` denotes the normal distribution, `t` denotes the t-distribution, and `unif` denotes the uniform distribution.

(Note: There are many other distributions implemented in R!)

Functions: Sample Statistics

Sample statistics are functions of the observed data. They are useful for summarizing key features of the data.

Particularly useful functions are:

- `sum`, `mean`, `sd`, `var`
- `cov`, `cor`
- `quantile`
- `table`

Example: Sample Statistics

```
x <- rexp(n = 100000, rate = 2) # E[X] = 1/2
mean(x)
#> [1] 0.501

x <- rnorm(n = 10000, mean = 0, sd = 1)
quantile(x = x, probs = c(0.025, 0.975))
#>    2.5%  97.5%
#> -1.967  1.947
```

Functions: Sample Statistics (Contd.)

In the example below, we test the fairness of an unfair coin given a sample of $n = 100$ observations. Recall that a binomial random variable with 1 trial is simply a Bernoulli random variable.

Example: Test fairness of a coin flip

```
# Simulate data from Bernoulli s.t. E[X] = 0.6.
#     That is, the coin is unfair!
n <- 100
x <- rbinom(n = n, size = 1, prob = 0.6)

# Calculate test statistic for H_0: E[X] = 0.5.
z <- sqrt(n) * (mean(x) - 0.5) / sd(x)

# Conduct two sided test using the CLT.
2 * pnorm(q = -abs(z), mean = 0, sd = 1)
#> [1] 0.002
```

Functions: Linear Algebra

Linear algebra is incredibly useful for econometrics and statistics – both for theory and for implementation.

The key functions are

- the transpose function: `t`
- the inverse function: `solve`
- matrix multiplication: %*%
- out product: %o%

Implementing linear regression from scratch has never been so easy! Recall $\hat{\beta}_{ols} = \left(X^\top X\right)^{-1} X^\top y$.

Example: Linear Algebra

```
myols <- function(y, X) {
  beta <- solve(t(X) %*% X) %*% (t(X) %*% y)
  return(beta)
}#MYOLS
```

Functions: Logical functions

The key logical functions are

- `any`
- `all`
- `which`

Two others that are sometimes convenient are `which.min` and `which.max`.

Example: Logical Functions

```
x <- c(TRUE, FALSE, TRUE)
any(x)
#> [1] TRUE

all(x)
#> [1] FALSE

which(x)
#> [1] 1 3
```

Functions: Importing and Exporting Data

Here we discuss importing and exporting comma separated values-files (i.e., csv-files). This can easily be done with `read.csv` and `write.csv`.

There are many R packages for importing other file formats. See, e.g., `readstata13`, `haven`, etc.

Example: Importing and Exporting Data

```
# Simulate some data
n <- 100
df <- data.frame(x = rnorm(n, 0, 1),
                 y = rbinom(n, 1, 0.6))

# Export to .csv
write.csv(df, file = "my_simulated_data.csv",
          row.names = FALSE)

# Import from .csv
df_2 <- read.csv(file = "my_simulated_data.csv")
head(df_2$y) # prints first 5 entries
#> [1] 0 1 0 1 1
```

Functions: Packages

Base R is great (and you should use it) but there are excellent extensions available in the form of packages.

Official packages are hosted on CRAN. These packages are well maintained and have to adhere to certain quality standards. Examples are `ggplot2` for plotting or `dyplr` for data manipulation.

In-official packages (or development versions) are often hosted on GitHub. Using the package `devtools`, these can be directly installed as well.

After installation, packages need to be activated each time you want to use them. This is done via `library`.

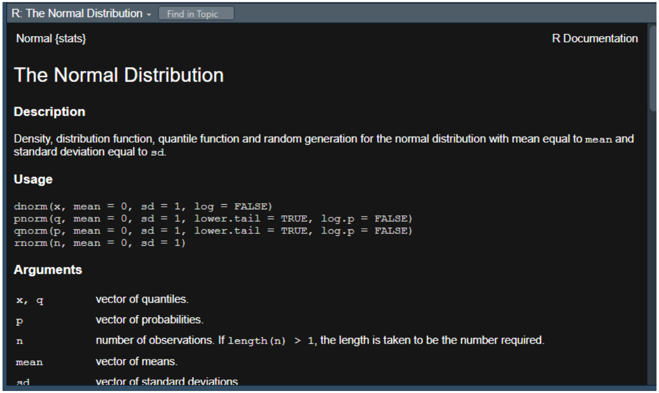
Example: Installing Packages

```r
# CRAN packages
install.packages(c("ggplot2", "dyplr", "devtools"))

# GitHub package
library(devtools)
install_github("jkcshea/l1svr") # needs devtools
```

Documentation

Before calling a function, it's important you familiarize yourself with the syntax and default arguments. The perfect resource for this are the R help files. These can be conveniently accessed via the "?" operator.

The following is a screenshot from the output after calling `?rnorm`.



Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations

Other helpful resources are Google, Stack Overflow, Stack Exchange, etc.

Style-Guide

"*Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read.*" – Wickham (2019)

Adopting (and adapting) a so-called style guide is the better alternative to re-inventing punctuation from scratch. The following gives a few highlights from the `tidyverse` style guide (`https://style.tidyverse.org/`).

We focus on:

- Naming
- Spacing
- Comments
- Sections

Style-Guide: Naming

Coming up with good variable names can be challenging!

- Variable and function names should be in lowercase.
- Use an underscore to separate words withing a name.
- Variable names should be nouns and function names should be verbs.
- Strive for names that are clear and concise.

Example: Variable names

```r
# Good
sample_mean_x <- mean(x)
avg_x <- mean(x)

# Bad
SampleMean_1 <- mean(x)
avgx <- mean(x)
```

Style-Guide: Spacing

- Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.). The exception is "`:`", which should be used without spaces.
- Always put a space after a comma, and never before.
- Place spaces before left parentheses, except in a function call.
- Don't add space within parentheses.

Example: Spacing

```
# Good
x <- 1:10
if (1 %in% x) print("Wow!")
x <- c(x, x)

# Bad
x <- 1 : 10
if(1%in%x)print("Wow!")
x <- c ( x , x )
```

Style-Guide: Comments

- Use # to add a comment.
- Comment frequently but not excessively.
- For multi-line comments, add *four* spaces at the beginning of each line after the first one.

Important: Limit your code to 80 characters per line. Many IDEs add a vertical line to help with this.

Example: Multi-line comment

```
# Commenting code is important. Be concise. When
#     necessary, use indentation to increase
#     readability.
```

Example: Simple comment

```
x <- 1
x <- x + 1 # This works well for brief comments
```

Style-Guide: Sections

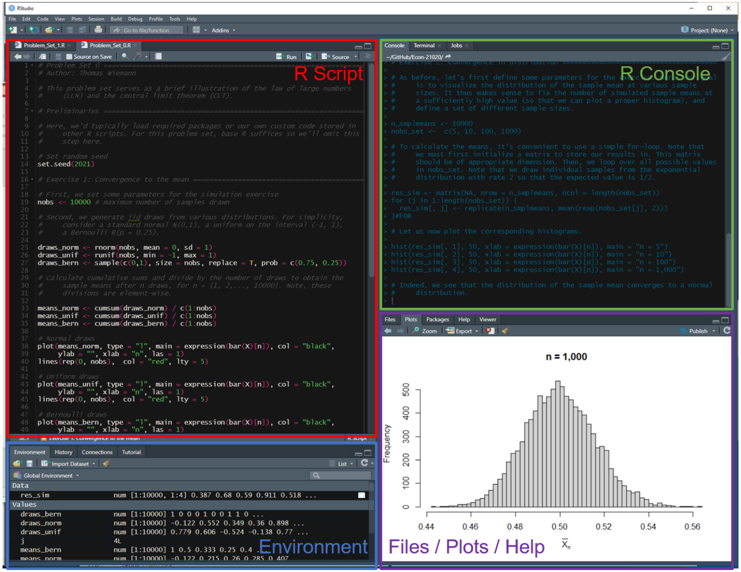
Use comments to break up a single R script into multiple sections. For example, each section dedicated to a single exercise of a problem set.

Using "=" does the job nicely:

Example: Sectioning an R Script

```
# Section name ===================================
# Adding a brief explanation here may be good.

x <- 1 # Some code
x <- x + 1

# Another section name ============================
# Etc...
```

Coding in R Studio



```r
# Problem Set 0 ================================================
# Author: Thomas Wiemann

# This problem set serves as a brief illustration of the law of large numbers
#     (LLN) and the central limit theorem (CLT).

# Preliminaries ================================================

# Here, we'd typically load required packages or our own custom code stored in
#     other R scripts. For this problem set, base R suffices so we'll omit this
#     step here.

# Set random seed
set.seed(2021)

# Exercise 1: Convergence to the mean ==========================

# First, we set some parameters for the simulation exercise
nobs <- 10000 # maximum number of samples drawn

# Second, we generate iid draws from various distributions. For simplicity,
#     consider a standard normal N(0,1), a uniform on the interval (-1, 1),
#     a Bernoulli B(p = 0.25).

draws_norm <- rnorm(nobs, mean = 0, sd = 1)
draws_unif <- runif(nobs, min = -1, max = 1)
draws_bern <- sample(c(0,1), size = nobs, replace = T, prob = c(0.75, 0.25))

# Calculate cumulative sums and divide by the number of draws to obtain the
#     sample means after n draws, for n = {1, 2,..., 10000}. Note, these
#     divisions are element-wise.

means_norm <- cumsum(draws_norm) / c(1:nobs)
means_unif <- cumsum(draws_unif) / c(1:nobs)
means_bern <- cumsum(draws_bern) / c(1:nobs)

# Normal draws
plot(means_norm, type = "l", main = expression(bar(X)[n]), col = "black",
     ylab = "", xlab = "n", las = 1)
lines(rep(0, nobs),  col = "red", lty = 5)

# Uniform draws
plot(means_unif, type = "l", main = expression(bar(X)[n]), col = "black",
     ylab = "", xlab = "n", las = 1)
lines(rep(0, nobs),  col = "red", lty = 5)

# Bernoulli draws
plot(means_bern, type = "l", main = expression(bar(X)[n]), col = "black",
     ylab = "", xlab = "n", las = 1)
```

```
> # As before, let's first define some parameters for the exercise. Here, our goal
> #     is to visualize the distribution of the sample mean at various sample
> #     sizes. It thus makes sense to fix the number of simulated sample means at
> #     a sufficiently high value (so that we can plot a proper histogram), and
> #     define a set of different sample sizes.
>
> n_smplmeans <- 10000
> nobs_set <-  c(5, 10, 100, 1000)
>
> # To calculate the means, it's convenient to use a simple for-loop. Note that
> #     we must first initialize a matrix to store our results in. This matrix
> #     should be of appropriate dimension. Then, we loop over all possible values
> #     in nobs_set. Note that we draw individual samples from the exponential
> #     distribution with rate 2 so that the expected value is 1/2.
>
> res_sim <- matrix(NA, nrow = n_smplmeans, ncol = length(nobs_set))
> for (j in 1:length(nobs_set)) {
+   res_sim[, j] <- replicate(n_smplmeans, mean(rexp(nobs_set[j], 2)))
+ }#FOR
>
> # Let us now plot the corresponding histograms.
>
> hist(res_sim[, 1], 50, xlab = expression(bar(X)[n]), main = "n = 5")
> hist(res_sim[, 2], 50, xlab = expression(bar(X)[n]), main = "n = 10")
> hist(res_sim[, 3], 50, xlab = expression(bar(X)[n]), main = "n = 100")
> hist(res_sim[, 4], 50, xlab = expression(bar(X)[n]), main = "n = 1,000")
>
> # Indeed, we see that the distribution of the sample mean converges to a normal
> #     distribution.
> |
```

```
Data
  res_sim        num [1:10000, 1:4] 0.387 0.68 0.59 0.911 0.518 ...
Values
  draws_bern     num [1:10000] 1 0 0 0 1 0 0 1 1 0 ...
  draws_norm     num [1:10000] -0.122 0.552 0.349 0.36 0.898 ...
  draws_unif     num [1:10000] 0.779 0.606 -0.524 -0.138 0.77 ...
  j              4L
  means_bern     num [1:10000] 1 0.5 0.333 0.25 0.4 ...
  means_norm     num [1:10000] -0.122 0.215 0.26 0.285 0.407
```

Coding in R Studio (Contd.)

There are essentially three options for executing code in R Studio:

- Entering code directly in the R console. Note that this makes reproduction (practically) impossible.
- Selecting a single line in the R script and hitting `enter`. This runs the line of code and advances to the next line.
- Block-selecting multiple lines of code and hitting `enter`. This runs all selected lines in sequence.

Frequently run your code to check whether everything works as intended.

References

Wickham, H. (2019). *Advanced R*. CRC press.