Crash course on reproducible computing in R

Julia Wrobel

Overview

Today, we cover:

- Course overview
- Basic principles of reproducible computing
  - Coding style
  - Using the command line
  - Project organization
- Lab with practice questions, if time allows

About me

Research

- Functional data analysis
- Spatial proteomics
- Application areas: cannabis impairment, wearables/sensor data, ovarian cancer

Announcements

- Discussion board post due tomorrow (1/15) at 10:00AM
- Homework 1 due Wednesday, 1/21 at 10:00AM

Course objectives

My goal for this course is for you to learn practical skills you need to become a statistical methods researcher. There are two main areas we will focus on:

1. Computing
    - R focused
2. Algorithms
    - Look under the hood at algorithms for commonly used methods like logistic regression, LASSO, mixed effects models

There is a new course website where all materials (other than HW) will be shared.

My teaching philosophy

- Ask questions early and often
  - It helps you stay engaged and others are likely to benefit from your question as well
- What you learn is proportional to the effort you put in
  - Try to think about this less as a class where you try to get a grade and more as a place where you are learning the skills to be a good researcher and help you write a **great** dissertation
  - Use ChatGPT, but wisely
- Feel free to work together

Emails vs. Discussion board

Discussion board

- All questions about course content should go on the discussion board!
  - This includes homework questions
  - This allows all students to benefit from questions asked about course content
  - **Posting** and **answering** questions posted on the discussion board will count towards your participation grade

Emails

- Email should only be used for schedule purposes and personal matters

Office hours and homework due dates

Office hours

- Tuesdays at 11:00AM (Weijia)
- Thursdays at 1:00PM (Me)

Homework policies

- Homeworks will be due on Wednesdays at 10am
- Late assignments will receive a maximum of half credit. Assignments more than 3 days late will not be accepted.
  - Let me know (if possible, in advance) if you have known conflicts with the due dates or a special circumstance (conference travel, family emergency)

Grading

- Homework (50%)
  - 7 assignments
- Discussion board posts (10%)
  - 4-5 posts
- Participation (10%)
  - Participation can mean asking questions in class, posting on the discussion board, or attending office hours
  - To quantify, 5+ posts on discussion board besides official posts
  - I encourage you to post computing tricks, cool algorithms, etc
- Final project (30%)
  - Related to your research, if possible

Syllabus

- Course content will be hosted on Course Website
- Homework will be posted on Canvas
- Any other questions?

Coding style

Principles for scientific coding

In this order:

1. Code that works.
2. Code that is reproducible.
3. Code that is readable.
4. Code that is generalizable.
5. Code that is efficient.

A **minimal standard** for scientific computing is 1-3.

Advice for beginner coders

Test code before relying on it.

It's OK to **copy/paste code** from ChatGPT or Stack Overflow, but make sure you **understand how it works**.

- Run line by line and see what each does.
- Change the code and see if it behaves as expected.

Stakes for copy/paste can be high!

- Incorrect analyses.
- Expensive (inadvertent) cloud computing.

For high stakes analyses, ask a colleague for a **code review**.

Advice for more advanced coders

Getting code correct AND readable is **most important**.

- Make your code more efficient later.
- After a paper is submitted for review?

Remember: you don't get bonus points for code that "looks impressive".

Think before you code

Before you start writing code, think about what you want the code to do.

Plain English $\rightarrow$ pseudo-code $\rightarrow$ actual code

This careful thought process can ultimately lead to **more efficient** and **more robust** code development.

Don't repeat yourself

Don't repeat yourself (DRY) is a fundamental concept in programming.

- *Ruthlessly eliminate duplication*, Wilson et al

For example, variables `score1=1`, `score2=2`, `score3=3` → `score=list(1,2,3)`.

If you write the same code more than once, it should be a function.

- Break large tasks into smaller calls to functions.
- Give functions (everything, really) meaningful names.
  - self-documenting code

Functions in R

If you don't know how functions work in R, learn about them now.

- Software carpentry: Creating R Functions
- R for Data Science: Functions
- DataCamp: A Tutorial on Using Functions in R!

Generalize... some

Write code **a bit more general** than your data or specific task.

- Don't assume particular dimensions.
- Don't forget about missing values (even if *your* data have none).

But don't necessarily try to handle every case.

- Try to anticipate what you might be asked for, but don't prepare for every possibility.
- If you think of **extreme cases** where code breaks, **document them**.

Generalize... some

Use **function arguments** to handle different cases.

- Don't assume particular file names.
- Don't assume particular tuning parameters.
- Don't assume particular regression formulas.

How general to make your code depends on its purpose!!

No magic numbers

There are **many decision points** in an analysis. Give them a name!

- How many **bootstrap samples**?
  - `nboot = 1e3`
- What **tolerance for convergence** of an algorithm?
  - `tol_covergence = 1e-3`
- What **threshold** for excluding missing data?
  - `tol_missingness = 3`

No magic numbers

Even better, include them as an **argument to a function** (with default values and documentation, as needed)!

- `get_bootstrap_ci <- function(..., nboot = 1e3)`
- `my_algorithm <- function(..., tol_convergence = 1e-3)`

Other guidelines

- **Indent**!
  - 2 or 4 spaces (join the debate)
  - Tabs can get nasty across systems.
- Use **white space**!
  - After commas, operators
  - after headers in Rmarkdown (following #)
- Use `{}` and `()` to **avoid ambiguity**.
- Keep **lines short**!
  - Rule of thumb is 72 or 80 characters.
  - Most text editors have settings to help with this.

Other guidelines

Bad!

```
# move values above/below quantiles to those quantiles
winsorize<-function(vec,q=0.006){
lohi<-quantile(vec,c(q,1-q),na.rm=TRUE)
if(diff(lohi)<0)lohi<-rev(lohi)
vec[!is.na(vec)&vec<lohi[1]]<-lohi[1]
vec[!is.na(vec)&vec>lohi[2]]<-lohi[2]
vec}
```

Better!

```
# move values above/below quantiles to those quantiles
winsorize <- function(vec, q = 0.006){
  lohi <- quantile(vec, c(q, 1 - q), na.rm = TRUE)
  if(diff(lohi) < 0){
    lohi <- rev(lohi)
  }
  vec[ !is.na(vec) & (vec < lohi[1]) ] <- lohi[1]
  vec[ !is.na(vec) & (vec > lohi[2]) ] <- lohi[2]
  return(vec)
}
```

Naming objects

Give objects informative names:

- Names should be:
  1. most importantly, descriptive
  2. as concise as possible while still being descriptive
- Avoid `tmp1`, `tmp2`, ...
  - ...as `tmp`ting as it may be.
- Functions as verbs, objects as nouns

Naming objects

Have **consistency** in your naming systems:

- E.g., `markers` vs. `mnames`; `nft` vs. `n_ft`

Commit to a case:

- `camelCase` vs. `pothole_case`

Don't confuse yourself:

- `total` vs. `totals`
- `result` vs. `rslt`
- `X` vs. `x`
- `z` vs. `zz`

Self-documenting code

Many of these recommendations are to make code self-documenting.

Comments should be used mostly for **why**, not **what**.

- The **what** should be inherent from the code itself.

This is really **hard to do**.

- Sometimes its harder than its worth, but it is worth trying!

Self-documenting code

- Functions as verbs, variables as nouns
- Make names encode shape, type, and meaning (not implementation)

```
clean_data <- function(raw_df) { ... }

users_df  <- readr::read_csv("users.csv")
is_valid  <- validate_users(users_df)
```

Self-documenting code

Patterns that prevent bugs include

- Singular vs. plural for “one vs. many”
  - `user` vs. `users`, `model` vs. `models`
- Type suffixes when it matters: `_df`, `_vec`, `_lst`, `_mat`
- Units/scales in names: `_sec`, `_ms`, `_km`
- Booleans start with `is_`, `has_`, `can_`, `should_`

```
users_df     <- read_users()
region_vec   <- unique(users_df$region)
timeout_sec  <- 30
revenue_usd  <- 125000
has_dupes    <- anyDuplicated(users_df$id) > 0
```

Self-documenting code

Here's an example of bad code:

```
tmp1 <- 9.81
tmp2 <- 5
tmp3 <- 0.5 * tmp1 * tmp2^2
```

Here's an example of documented bad code:

```
# gravitational constant
tmp1 <- 9.81
# time object is falling
tmp2 <- 5
# displacement of the object
tmp3 <- 0.5 * tmp1 * tmp2^2
```

Self-documenting code

Here's self-documenting code:

```
gravitational_force <- 9.81
time_in_seconds <- 5
displacement <- 1/2 * gravitational_force * time_in_seconds^2
```

Now let's add a comment explaining the **why**:

```
# compute displacement of falling object with Newton's equation
gravitational_force <- 9.81
time_in_seconds <- 5
displacement <- 1/2 * gravitational_force * time_in_seconds^2
```

Self-documenting code

Even better, make it a function!

```
# for falling objects based on Newton's equation
compute_displacement <- function(time_in_seconds){
  gravitational_force <- 9.81
  displacement <- 1/2 * gravitational_force * time_in_seconds^2

  return(displacement)
}
```

Recap

Why spend a lecture on coding style if everyone knows how to code?

- Clear code is more likely to be **correct**.
- Clear code is **easier to use**.
- Clear code is **easier to revisit** six months from now.
- Software based on clear code is **easier to maintain**.
- Clear code is **easier to extend**.

My (non-exhaustive) list of coding pet peeves

- dots (“.”) in variable names
- upper case in variable names
- Not putting a vertical line of white space after a section header in an `.Rmd` or Quarto document
- Unnecessarily short and unreadable variable names
- Calling libraries, loading data, or sourcing files in the middle of a script
  - Always do it at the beginning!!

Command line

Operating systems

- Windows
  - Not always programmer friendly
- Mac OSX
  - Better for programming
  - Under the hood, is just Unix
- Unix-based OS (Linux, Solaris, etc...)
  - Best for programming

We'll learn to interact with our computer like it's a Unix OS.

- Best practices for programming
- Needed for cluster computing and AWS (later)

Some terminology

Shell

- user interface for interacting with a computer
- the "outermost" layer of the operating system

Graphical user interface (GUI)

- visual interface (icons, menus, etc...) for interaction
- "Point-and-click"

Command line interface (CLI)

- text-based interface for interacting with computer
- e.g., `bash`, `sh`, `tcsh`, `zsh`, ...

Terminal

If Windows, use Ubuntu for Windows

- Or other Linux distribution (e.g., Debian)
- Biggest difference is how software installed

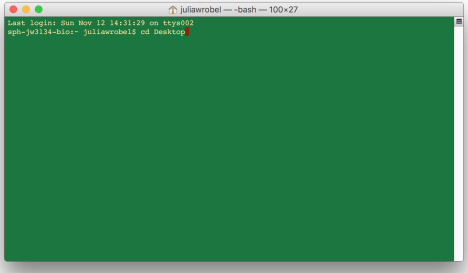
If Mac, use Terminal

- Or iTerm2 – more features

If Linux, whatever terminal emulator comes with your distribution.

Terminal

Your open terminal will look something like this:



- You'll see a *prompt*, which is an alphanumeric string that (usually) ends in "$". Commands are typed after the "$".

Using the terminal through RStudio

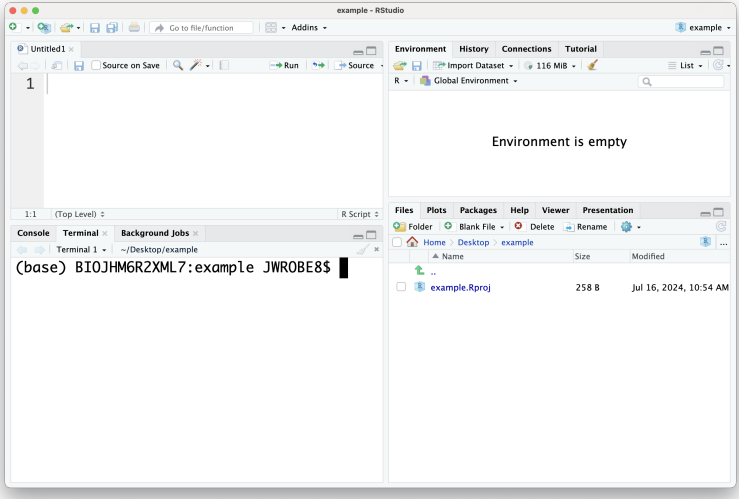
You can also use the terminal directly in Rstudio!

- Rstudio introduced the terminal tab in 2017
- For our course, this may be the simplest approach

Display the Terminal tab: If the tab isn't visible, you can display it by going to Tools > Terminal > Move Focus to Terminal.

- Can also use the keyboard shortcut Shift+Alt+M or Shift+Option+M on Mac.

RStudio terminal



Working directories

What is a working directory?

Moving around directories

Folders within you computer are called **directories**. You can navigate around to different directories, remove or create directories, remove or create files, move files around, and list their contents all from the terminal.

These next sections may seem fairly basic but these are also the commands I use the most often, so I'm going to spend some time on them.

Moving around directories

Command	Action
`pwd`	print working directory
`cd`	change directory
`ls`	list files in directory

- **cd**: Takes you to the home directory
  * `cd ..`: Moves up one directory
  * `cd ../..`: Moves up two directories
- **ls -a**: list **hidden** files as well as other files
  * Hidden files are often part of the instructions for the OS or a particular application
  * Usually invisible when searching through folders
  * Examples: .git, .gitignore, .Rhistory

Absolute vs. relative file paths

Absolute paths

- `/Users/juliawrobel`
- `~/Documents`
- `/`

Relative paths

- `./Documents`
- `../Documents`
- `../../`

Absolute vs. relative file paths

- Absolute paths include the **whole path** for a directory
- Relative paths depend on the working directory that they are executed in
  - The `./` means "in the current directory"
  - The `../` means "in one directory up from the current directory".

Adding/removing files

Command	Action
`mkdir`	make a new directory
`rm`	delete a file or directory
`mv`	move a file or directory
`cp`	copy a file or directory

Structure of a bash command

```
command [options] [arguments]
```

1. `command`: the bash function you want to run, e.g. `ls`, `cd`, `echo`, etc.

2. `options`: also called "flags", these are additional parameters to modify the behavior of the command, e.g.,

   - `ls -R` lists all directories and contents recursively
   - `ls -aR` recursively lists all files and hidden files

3. `arguments`: inputs to the command, such as file names or other data that tell the command what to operate on

   - `rm` what? `cp` what?
   - `ls \Documents` lists all files in the documents folder

Solving computing problems

- `man [command]` (bash version of `?command_name`)
- Google (with `site:stackoverflow.com`?)
- ChatGPT
- Ask friends/classmates
- Try stuff!

Wild cards

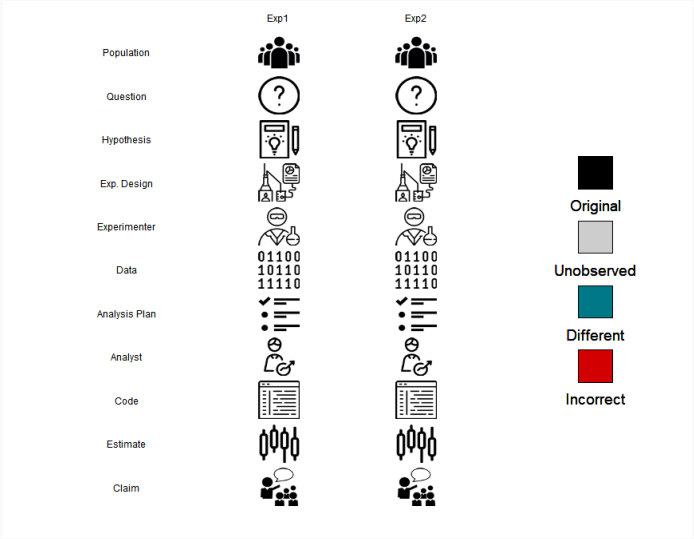
Command	Action
*	match anything
?, ??, ...	match a single character
[...]	match a range of characters

```
# files in cwd with .txt extension
ls -l *.txt
# all files in cwd named a_file with three character extension
ls a_file.???
# .txt files in cwd name a_file, b_file, ..., e_file
ls [a-e]_file.txt
```

Useful command line shortcuts

Key stroke	Action
↑	move to previous command
↓	move to next command
`tab`	autocomplete command or file
`ctrl+c`	cancel (running) command
`ctrl+z`	suspend command
`ctrl+r`	search for command in history
`ctrl+l`	clear the screen

Reproducible project organization



Why should I care about reproducibility?

- Careful coding means more likely to produce **correct results**.
- In the long run, you will **save (a lot of) time**.
- Higher **impact** of your science.
- **Avoid embarrassment** on a public stage.

Basic principles

- Put everything in one version-controlled directory.
- Develop your own system.
- Be consistent, but look for ways to improve.
  - naming conventions, file structure
- Raw data are sacred. Keep them separate from everything else.
- Separate code and data.
- Use meaningful file names.
- Use YYYY-MM-DD date formatting.
- No absolute paths.

What to organize?

It is probably useful to have a system for organizing:

- data analysis projects;
- methods projects;
- first-author papers;
- talks.

The systems should adhere to the same general principles, but different requirements may necessitate different structures.

Think about organization of a project from the outset!

Example data analysis project

```
YYYY_MM_PI_topic/
  data/
      raw_data.csv
      tidied_data.Rdata
  analysis/
      exploratory_data_analysis.Rmd
      report.Rmd
  source/
      01_clean_raw_data.R
      02_modeling_functions.R
      03_plotting_functions.R
      utils.R
  results/
  literature/
  README.md
```

Example data analysis project, cont'd

I typically have other ancillary files in my root directory as well. These files are important for workflow or reproducibility:

```
YYYY_MM_PI_topic/
   YYYY_MM_PI_topic.Rproj
   sandbox/
   .git/
   .gitignore
```

Organizing data

Raw data are sacred... but may be a mess.

- You'll be surprised (and disheartened) by how many color-coded excel sheets you'll get in your life.

Tempting to edit raw data by hand. Don't!

- Everything scripted!
- Use meta-data files to describe raw and cleaned data.

You should be able to get a new version of the data and easily re-run your analysis. Urge your collaborators to not mess with the structure of the data in between versions!

Exploring data

Write out a set of comments describing what you are try to accomplish and fill in code from there.

- I do this for every coding project.
  - Data analysis, methods coding, package development

Leave a search-able comment tag by code to return to later

- I use e.g., `# TO DO: add math expression to labels; make colors prettier`.

Exploring data

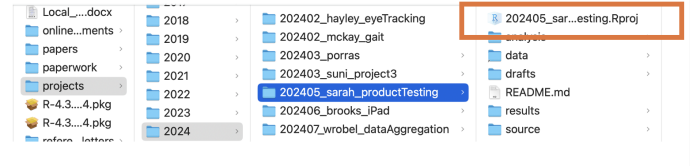
Other helpful ideas for formalizing exploratory data analysis:

- `.Rhistory` files: all the commands used in an R session
- `save` intermediate objects and document what they contain
- Informal `.Rmd` documents.
  - easy way to organize code/comments into readable format

.Rproj files

You may have noticed a file with the extension `.Rproj` in the productTesting folder

- These are called `R projects`
- `projectr::proj_start()` automatically sets up an `.Rproj`.



I'm going to try to convince you that these are the best.

Benefits of using R projects

Project organization:

- Relative file paths: ensures file paths are relative to the project directory, making scripts portable and easier to share.
- Separate workspaces: prevents conflicts between variables and packages across different projects.

Reproducibility

- Can hand off entire directory to someone else and have them rerun your analysis
- Works great with the `here` package

Double clicking the `202405_sarah_productTesting.Rproj` opens up an R Studio session and automatically sets your working directory to the `202405_sarah_productTesting` folder.

The `here` package

No absolute paths.

- Absolute paths are the enemy of project reproducibility.

For `R` projects, the `here` package provides a simple way to use relative file paths.

- Read Jenny Bryan and James Hester's chapter on project-oriented work-flows.

The use of `here` is simple and best illustrated by example.

The `here` package

Consider this simple project structure.

```
my_project/
   my_project.Rproj
   data/
      my_data.csv
   output/
   R/
      my_analysis.R
   Rmd/
       my_report.Rmd
```

Here, the folder `my_project` is the **root directory**.

- Where `.Rproj` lives
- All file paths should be relative to `my_project`!

The `here` package

Makes it easy to load data using a relative file path that works across different operating systems:

```
library(here)
# relative path using here()
here_path = here("data", "file_i_want.csv")
my_data = read.csv(here_path)
```

In contrast to:

```
# absolute path
ugly_path = "/Users/JWROB/projects/my_project/data/file_i_want.csv"
my_data = read.csv(ugly_path)
```

In contrast to:

```
# relative path using NOT using here()
relative_path = "./data/file_i_want.csv"
my_data = read.csv(relative_path)
```

The `here` package

`here` works, regardless of where the associated source file lives inside your project

- If you have an `.Rproj` file in your root directory of your project, here will set the location of the `.Rproj` to be the top-level directory
  - This is the behavior we want!
- These paths will "just work" during interactive development, without incessant fiddling with the working directory of your IDE's R process.
- I am oversimplifying the heuristics, feel free to read more.

The `here` package

What if I want to load data in a document that lives in a subfolder such as `my_project\analysis\code.Rmd`?

- Doesn't matter! You can use the same code within the `.Rmd` document to load the data

```
library(here)
path_to_data = here("data", "file_i_want.csv")
my_data = read.csv(path_to_data)
```

What if my data I want to access is nested in a subfolder of data, such as `my_project\data\raw_data\raw_file.csv`?

```
library(here)
path_to_data = here("data", "raw_data", "raw_file.csv")
my_data = read.csv(path_to_data)
```

Automated project initiation using `projectr`

Hosted on my GitHub, heavily adapted from Jeff Goldsmith

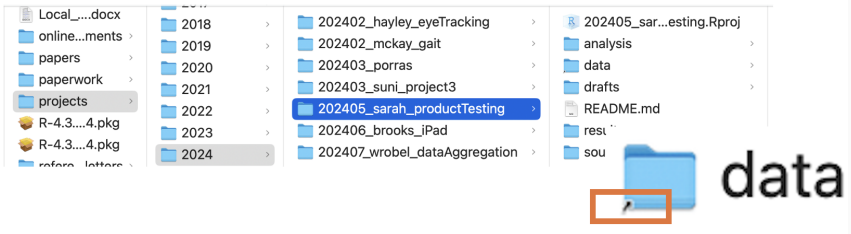
```
devtools::install_github("julia-wrobel/projectr")

projectr::proj_start(proj_dir = "~/projects/2024/202407_PERSON_PROJECT",
                     data_dir = "~/Data/202407_data")
```

- **proj_dir**: where on your computer you want the project to live
- **data_dir**: where you want the data to live, if not within your project folder
    - Sets up a symbolic link from the project directory to this folder

Symbolic links

There are great reasons NOT to put raw data in the project folder

- iOS uploads many folders automatically to the cloud
- data has PHS so you can't put it on GitHub
- data is stored on OneDrive/Box/AWS and you can't download a local copy



Setting up a symbolic link in the terminal

Pseudo code for setting up a symbolic link:

```
ln -s /path/to/target /path/to/symlink
```

The `projectr` package

Right now only works if you have a Mac (sorry).

In lab we will build our own code with similar functionality.