POIR 613: Computational Social Science

Pablo Barberá

School of International Relations
University of Southern California
`pablobarbera.com`

Course website:

pablobarbera.com/POIR613/

Good (enough) practices in scientific computing

Based on Nagler (1995) “Coding Style and Good Computing Practices” (PS) and Wilson *et al* (2017) “Good Enough Practices in Scientific Computing” (PLOS Comput Biol)

Good practices in scientific computing

Why should I waste my time?

- Replication is a key part of science:
  - Keep good records of what you did so that others can understand it
- “Yourself from 3 months ago doesn’t answer emails”
  - More efficient research: avoid retracing own steps
  - Your future self will be grateful

General principles:

1. Good documentation: README and comments
2. Modularity with structure
3. Parsimony (without being too smart)
4. Track changes

Summary of good practices

1. Safe and efficient data management
2. Well-documented code
3. Organized collaboration
4. One project = one folder
5. Track changes
6. Manuscripts as part of the analysis

1. Data management

- Save raw data as originally generated
- Create the data you wish to see in the world:
  - Open, non-proprietary formats: e.g. `.csv`
  - Informative variable names that indicate direction: `is_political` instead of `topic` or `V322`; `voted` vs `turnout`
  - Recode missing values to `NA`
  - File names that contain metadata: e.g. `05-alaska.csv` instead of `state5.csv`
- Record all steps used to process data and store intermediate data files if computationally intensive (easier to rerun parts of a data analysis pipeline)
- Separate data manipulation from data analysis
- Prepare README with codebook of all variables
- Periodic backups (or Dropbox, Google Drive, etc.)
- Sanity checks: summary statistics after data manipulation

2.Well-documented code

- Number scripts based on execution order:
  → e.g. `01-clean-data.r`, `02-recode-variables.r`, `03-run-regression.r`, `04-produce-figures.R`...
- Write an explanatory note at the start of each script:
  → Author, date of last update, purpose, inputs and outputs, other relevant notes
- Rules of thumb for modular code:
  1. Any task you run more than once should be a function (with a meaningful name!)
  2. Functions should not be more than 20 lines long
  3. Separate functions from execution (e.g. in `functions.r` file and then use `source(functions.r)` to load functions to current environment
  4. Errors should be corrected when/where they occur
- Keep it simple and don't get too clever
- Add informative comments before blocks of code

3. Organized collaboration

- Create a `README` file with an overview of the project: title, brief description, contact information, structure of folder
- Shared to-do list with tasks and deadlines
- Choose one person as corresponding author / point of contact / note taker
- Split code into multiple scripts to avoid simultaneous edits
- ShareLatex, Overleaf, Google Docs to collaborate in writing of manuscript

4. One project = one folder

Logical and consistent folder structure:

- `code` or `src` for all scripts
- `data` for raw data
- `temp` for temporary data files
- `output` or `results` for final data files and tables
- `figures` or `plots` for figures produced by scripts
- `manuscript` for text of paper
- `docs` for any additional documentation

5 & 6. Track changes; producing manuscript

- Ideally: use version control (e.g. GitHub)
- Manual approach: keep dates versions of code & manuscript, and a `CHANGELOG` file with list of changes
- Dropbox also has some basic version control built-in
- Avoid typos and copy&paste errors: tables and figures are produced in scripts and compiled directly into manuscript with $\LaTeX$

Examples

Replication materials for some of my published articles:

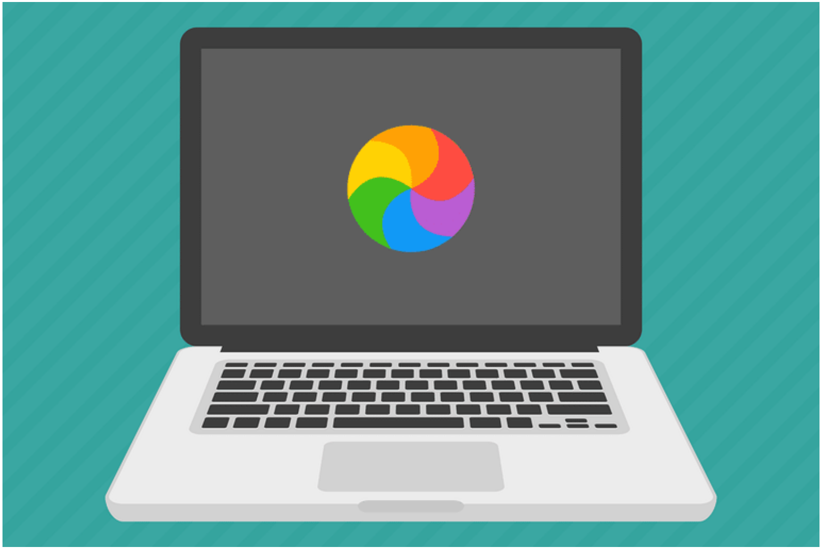
- 2019 APSR
- 2017 ISQ

John Myles White's ProjectTemplate R package.

Replication materials for Leeper 2017:

- Code and data

Efficient data analysis with R



Myths about R as programming language

1. R is an interpreted language, so it must be slow
   - Interpreted = executes code directly without compiling
   - Compiled code = code executed natively on CPU (fast!)
   - BUT: many functions are written in C and C++ and thus run in fast machine code
   - Slow code can be written more efficiently
2. All objects in R are stored in memory
   - You cannot open datasets larger than RAM
   - BUT: most laptops now have 8+ GB of RAM (+virtual mem)
   - `bigmemory` package: work with files on disk
   - Easy to work with large databases in the cloud
3. R only uses one core of your CPU
   - Unlike STATA, no multi-core computing out of the box
   - BUT: many functions and packages now take advantage of multi-core computers
   - Easy to write your own code to do parallel computing

My data is too big! My code is too slow!

What to do?

1. Buy a better computer or expand RAM memory
2. Write more efficient code
3. Use parallel computing
4. Move your code/data to the cloud
5. Use out-of-memory storage: SQL databases, bigmemory package, Hadoop...

Writing efficient R code (Part I)

- Conventional wisdom: avoid for loops at all costs!
- But simply rewriting loops will not make code faster
- Key: use vectorized functions instead of loops
- What is slowing our code down?
  - Additional function calls: `for`, `:`, `[`, `<-`
  - `sapply` hides explicit loop, but loop is still there, and implemented in R code
- Why was + so fast? Implements vectorization by vector filtering
  - Takes vector as input and return vector as output
  - Loop is done in machine native code
  - Other vectorized functions: `ifelse()`, `which()`, `rowSums()`, `colSums()`, `sum()`, `any()`, `rnorm()`...

Writing efficient R code (Part II)

- A common bottleneck is memory re-allocation, e.g.:

  ```
  result <- c()
  for (i in 1:n){
     result[i] <- x[i] + y[i]
  }
  ```

- In iteration, `R` re-sizes the vector and re-allocates memory
- For large operations (e.g. data frames), this can make your code really slow
- Solution: pre-allocate vector size:

  ```
  result <- rep(NA, n)
  for (i in 1:n){
     result[i] <- x[i] + y[i]
  }
  ```