BMI 826-003

Tools for Reproducible Research

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

biostat.wisc.edu/~kbroman
github.com/kbroman
@kwbroman
Course web: bit.ly/tools4rr

This is the introductory lecture for a special topics course at UW–Madison on tools for reproducible research.

A minimal standard for data analysis and other scientific computations is that they be reproducible: that the code and data are assembled in a way so that another group can re-create all of the results (e.g., the figures in a paper). The importance of such reproducibility is now widely recognized, but it is not so widely practiced as it should be, in large part because many computational scientists (and particularly statisticians) have not fully adopted the required tools for reproducible research.

In this course, we will discuss general principles for reproducible research but will focus primarily on the use of relevant tools (particularly `make`, `git`, and `knitr`), with the goal that the students leave the course ready and willing to ensure that all aspects of their computational research (software, data analyses, papers, presentations, posters) are reproducible.

Reproducible

vs.

Replicable

Computational work is reproducible if one can take the data and code and produce the same set of results. Replicable is more stringent: can someone repeat the experiment and get the same results?

Reproducibility is a minimal standard. That something is reproducible doesn't imply that it is correct. The code may have bugs. The methods may be poorly behaved. There could be experimental artifacts.

(But reproducibility is probably correlated with correctness.)

Note that some scientists say replicable for what I call reproducible, and vice versa.

Levels of quality

- Are the tables and figures reproducible from the code and data?
- Does the code actually do what you think it does?
- In addition to what was done, is it clear why it was done?

  (e.g., how were parameter settings chosen?)
- Can the code be used for other data?
- Can you extend the code to do other things?

Reproducibility is not black and white. And the ideal is hard to achieve.

Basic principles

- Everything via code
- Everything automated
  Workflow and dependencies clearly documented
- Get the data in the most-raw form possible
- Get any/all data and meta-data possible
- Keep track of the provenance of all data files
- Be self-sufficient

Pointing and clicking is not reproducible. Ideally, you press just one button.

Make sure you have all of the data and that you know exactly where it came from.

But what is raw data? How far back should you go? Data that I get from collaborators has usually gone through a considerable amount of pre-processing. Should we have captured that, in order for the work to be considered reproducible?

If your collaborator asks, "In what form would you like the data?" you should respond, "In its current form."

Why do we care?

- Avoid embarrassment
- More likely correct
- Save time, in the long run
- Greater potential for extensions; higher impact

Doing things properly (writing clear, documented, well-tested code) is time consuming, but it could save you a ton of aggravation down the road. Ultimately, you'll be more efficient, and your work will have greater impact.

Your code and analyses will be easier to debug, maintain, and extend.

Your closest collaborator is you six months ago, but you don't reply to emails.

I heard this from Paul Wilson, UW-Madison.

What could go wrong?

- "The attached is similar to the code we used."
- "Where did this data file come from?!"
- "Can you repeat the analysis, omitting subject X?"
- "This part of your script is now giving an error."

If you've not heard any of these things, it's just a matter of time.

Need to avoid

- Open a file to extract as CSV
- Open a data file to do even a slight edit
- Paste results into the text of a manuscript
- Copy-paste-edit tables
- Copy-paste-adjust figures

If you do anything "by hand" once, you'll have to do it 100 times.

Basic tools

- Automation with Make
- Unix command line
- Latex and Markdown
- Knitr
- Version control with git
- R packages
- Python (or Ruby or Perl)

These are the basic tools that I think are important for reproducible computational research; they form the core topics for the course.

Make is for automation and for documenting dependencies. For reproducibility, the command line is your best friend. Latex and Markdown allow preparation of beautiful documents without pointing or clicking. Knitr is for combining code and text; knitr and make are the key tools for reproducibility. Version control isn't strictly necessary for reproducibility, but once you get the hang of it, you'll never go back. R's packaging system is among its best features. A scripting language like Python is invaluable for manipulating data files. Many things that are awkward in R are easy in Python.

Other topics

- Organizing projects
- Writing clear code
- Don't Repeat Yourself (DRY)
- Testing and debugging
- Handling big jobs
- Licenses; human subjects data

We'll also cover all of these things.

The organization of the data and code for a project is a major determinant of whether others will be able to make sense of it. Good code is not just correct but is clearly written. Code is easier to maintain and understand if it is modular. Adding good tests will help you to find problems in your code earlier rather than later. But you'll spend a lot of time debugging, so we should talk about debugging strategies. Big computational jobs (particularly big computer simulations) raise additional issues; reproducibility is especially tricky. Finally, code that you distribute should be licensed. And if you're working with data on human subjects, you need to be extra careful.

Don't Repeat Yourself

- In code, in documentation, etc.
- Repeated bits of code are harder to maintain
  - Write a function
- Use documentation systems like Roxygen2
  - Documentation in just one place
- Make use of others' code

DRY is among the more important concepts/techniques.

For example, I organized a meeting in Madison last spring. The program book and website both drew information from a single basic source. Change that one document and both the program and web site are updated.

My R/qtl package is an anti-example.

This course

- Brief intro to various tools and concepts
- Try everything out as we go along
  Ask questions!
- I don't know everything
  Make suggestions!
- Project
  - Write a bit of R code
  - Use version control
  - Make it an R package
  - Write a vignette

About this course: I'm trying to get you started; pointing you in the right direction. But I don't know everything, and I don't always do things in the most efficient way possible: please offer me suggestions!

We won't have time for a comprehensive introduction to the tools. My main goal is to convince you of their importance: to motivate you to adopt them.

Automation with GNU Make

- `Make` is for more than just compiling software
- The essence of what we're trying to do
- Automates a workflow
- Documents the workflow
- Documents the dependencies among data files, code
- Re-runs only the necessary code, based on what has changed

People usually think of Make as a tool for automating the compilation of software, but it can be used much more generally.

To me, Make is the essential tool for reproducible research: automation plus the documentation of dependencies and workflows.

Example `Makefile`

```
# Example Makefile for a paper
mypaper.pdf: mypaper.bib mypaper.tex Figs/fig1.pdf Figs/fig2.pdf
    pdflatex mypaper
    bibtex mypaper
    pdflatex mypaper
    pdflatex mypaper

# cd R has to be on the same line as R CMD BATCH
Figs/fig1.pdf: R/fig1.R
    cd R;R CMD BATCH fig1.R fig1.Rout

Figs/fig2.pdf: R/fig2.R
    cd R;R CMD BATCH fig2.R fig2.Rout
```


You can get really fancy with make, but this example shows you the basics.

Records look like `target: dependencies` and are followed by a set of lines of code for creating the target from the dependencies. Those lines of code must start with a tab character (not spaces), and if you need to change directories, you have to do that on the same line as the command.

If you type `make` (or `make Makefile`), the `mypaper.pdf` file will be created; but first, any dependencies will be updated, if necessary, based on the time the files were last modified.

So, for example, if `fig1.R` had been edited, then the commands to construct `fig1.pdf` would be constructed, followed by the commands to construct `mypaper.pdf`.

Fancier example

```
FIG_DIR = Figs

mypaper.pdf: mypaper.tex ${FIG_DIR}/fig1.pdf ${FIG_DIR}/fig2.pdf
    pdflatex mypaper

# One line for both figures
${FIG_DIR}/%.pdf: R/%.R
    cd R;R CMD BATCH $(<F)

# Use "make clean" to remove the PDFs
clean:
    rm *.pdf Figs/*.pdf
```


As I said, you can get really fancy with GNU Make.

Use variables for directory names or compiler flags. (This example is not a good one.)

Use pattern rules and automatic variables to avoid repeating yourself. With `%`, we have one line covering both `fig1.pdf` and `fig2.pdf`. The `$(<F)` is the file part of the first dependency.

Look at the manual for make and the many online tutorials, such as the one from Software Carpentry.