Programming Practices for Research in Economics

Introduction & Motivation

Ulrich Bergmann Ursina Schaede Dora Simon Carlo Zanella Christian Zünd

Department of Economics, Univeristy of Zurich

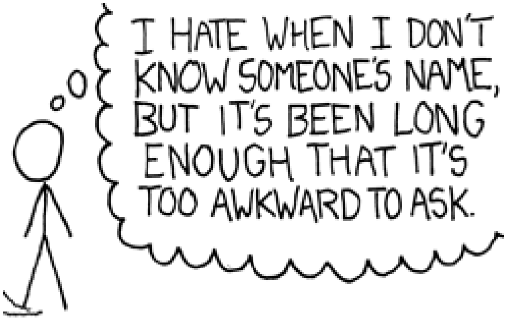
Fall 2018

Welcome!

Introductions: Who We Are

- 5 PhD students
    - Uli
    - Ursina
    - Dora
    - Carlo
    - Christian

Introductions: Who are You



Logistics: Basic Information

- group is a mix of "for credit" and audit students
  - want credit
    - enrol using sheet we will pass around in the last week
    - register for course on `UZH module booking`
    - hand in assignment 4 weeks after course ends
    - grading: pass/fail based on final assignment
- sessions are designed to be interactive
  - mix of *live coding*, *small challenges*, *longer exercises*
  - We want to get you comfortable using your computing environment to solve problems
    - Bring your laptop!
    - We expect you have completed the installation guide and have all software installed.
    - Ask questions!

Logistics: Structure of each day

- Session 1: 9.00-12.00
- Session 2: 13.00 - 16.00
- Expect coffee breaks in each session
  - Exactly when depends on the leader of a session, and the material
- No scheduled office hours
  - Talk to us during the day
  - Email for appointment after class if want to discuss assignment

Logistics: Where to Find Information

Course website:

- `pp4rs.github.io/2018-uzh`

Installation Guide:

- `pp4rs.github.io/installation-guide` Course Chatter:

- `pp4rs.slack.com/`, #pp4rs-2018

GitHub repositories:

- `github.com/pp4rs`

Logistics: Assignment

The details

- Register in class & UZH module booking
- One final assignment
- Can be submitted in groups of 1-3 people
- Due 4 weeks *after* last class
- Propose to us an idea before you start
- grading pass/fail

Use what you learn in this course to solve a non-trivial economic problem

- Code must be in split into meaningful sub-files
- Solution must be submitted using GitHub
- Solution must be executable using a single line of code, e.g. using`Snakemake`, a `Shell` script, `R Markdown`

Logistics: Social Event

- Join us for casual drinks
- When: This Friday (August 31st), after class
- Location: TBA

Motivation

Where we are as a profession

- We spend more and more time building and using software . . .
  - . . . to solve increasingly interesting/complex questions
- Most of us are primarily self-taught
- Hard to measure how well we do things
- Anecdotal evidence suggests “not very”

A new challenge: Open Science

EU policy for all publicly funded research being *open* by 2020 & 4Rs (Pagan and Torgler, Nature 2015)

- Reproduction: Can others reproduce your results using your data?
- Replication: Can others replicate your results using new data?
- Robustness: Do your results depend on the assumptions you made?
- Revelation: Do you communicate the reasoning for your conclusions transparently?

Our generation must adapt to the challenge of "open science"

- But we are lacking the skillset to do so
- Lack of proper training opportunites (particularly with a Social Science focus)

We hope this is a first step in filling this gap

Broad Goals for the Course

1. Improve computing skills, so you can do things you could not do before
2. Show how can do a given set of things with less effort
3. Increase the confidence in results that are produced this way (both yours and others' results)

What We Teach

Core topics:

1. Unix shell
   - Text based interface to computing
   - Automate repetitive tasks
2. Git
   - Track/control and share work
3. Python/R
   - Build modular code to solve typical economics problems
4. Snakemake
   - Automate the execution of your research project

Applications:

6. Machine Learning
7. Web scrapping
8. Geo-spatial Data

Guiding Principles

Rule 1: Write Programs for People Not Computers

- Code that a computer can understand $\neq$ code a human understands
- Does your difficult-to-understand code do what it's suppose to?
- Makes it hard for other collaborators and researchers to use your code
  - Future you *is an* **other** *collaborator*

Rule 1a: Write Many Short Scripts / Code

- Short-term memory can hold 7 ± 2 items

Functions:

- short, readable, take only a few inputs each
  - **Rule of thumb**: code looking very complex = you're probably doing it wrong.
  - Think of a function/script like a paragraph: limit it to one idea.
  - e.g. transform() transforms x to x'
  - do_stuff() works with x'

Rule 1a: Write Many Short Scripts / Code

Scripts:

- *What not to do:* 5000 line scripts that execute an entire project
  - What does the variable on line 4100 mean?
  - How does it relate to it's initial defintion on line 1350?
- *What to do:* Split into meaningful smaller tasks, e.g.
  - clean-data.R cleans the data
  - create-vars.R creates auxiliary variables
  - regressions.R does the regressions
  - tables.R makes the tables
  - plots.R makes the plots

Rule 1b: Use meaningful variable names

- `p` less useful for long term memory than `price`
- `i`, `j` are (almost) OK for indices in small scopes
  - `i_subject` and `i_trial` might be better
- Be careful with the use of ambiguous names like `temp`

Rule 1c: Make code style and formatting consistent

Which rules don't matter — having rules does

- Brain assumes all differences are significant
  - Inconsistency slows comprehension
  - annoying not to remember what your function x was called

Good ideas:

- Consistent naming convention
  - Camel Case: cleanData()
  - Snake Case: clean_data()
  - Kebab Case: clean-data()
- Keep each line of code within 80 characters
- Whitespace, identation & comments are your friend

Rule 1d: Document your code well

Document Design and Purpose, not Mechanics

- Makes the next person's life easier
- Focus on what the code doesn't say
- Or doesn't say clearly
  - E.g., file formats
  - An example is worth a thousand words. . .

Rule 2: Use a version control system

- Tracks changes
- Allows them to be undone
- Keeps folders clean
- Supports independent parallel development
- Essential for collaboration

  Email is not version control!

Rule 2: Use a version control system

How it's **not** done:

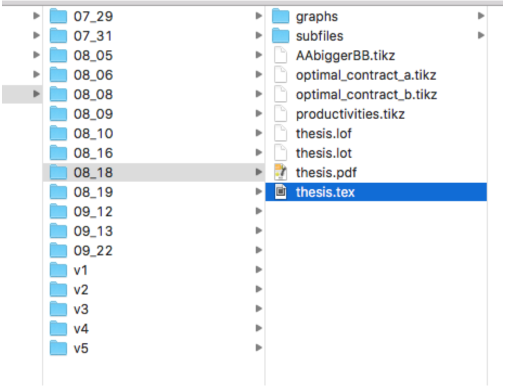


Figure 1: Where was the previous version of paragraph 4 again?!

Rule 2: Use a version control system

How it's done:

```
464 LaTeX/research-plan.tex                                                    View
@@ -1,266 +1,238 @@
1   \section{Summary of the research plan}
2
3 - Almost all economic research assumes and relies on stable and rational preferences, which are necessary for the existence of a utility function -- the most basic and uncontroversial building block of our profession. Observed choice data has to necessarily satisfy the Weak Axiom of Revelead Preferences (WARP), which implies that a choice option can never become \emph{more} popular if the set of choices increases. Similarly, models of the random utility family, our workhorse models to analyze discrete choice data empirically, like the Logit, Probit, Multinomial Logit, Mixed Logit etc, assume that the choice frequency of a given option can \emph{never} be higher in a larger choice set (cite Marshak). In fact no data can be generated by a random utility process which violates this regularity assumption (cite Natenzon 2017). These models also perform poorly when analyzing such data.
4
5 - Experimental researcher are able to generate such a violation for more than 30 years. Starting with the discovery of the attraction effect by (cite 1982) and the compromise effect by Simonson (cite 1989), more than 100 studies were able to replicate this anomaly. To illustrate the attraction effect, let us start with a choice set with two items: a cashmere sweater which costs \$100 and a synthetic one with costs \$20. The attraction occurs when we add a new choice option, a cotton sweater which costs \$110, to the original choice set. This new option, the decoy, is dominated in both dimensions by the cashmere sweater as it is more expensive and of a lesser material in comparison to the cashmere sweater. At the same time it is better only in the material dimension but more expensive than the synthetic sweater. While a price conscious consumer might prefer the cheap, synthetic option in the original choice set, experimental and field evidence shows that the addition of the decoy will reverse this preference for many consumers. This effect is especially intriguing to the marketing profession. As the dominated option is never chosen by consumers and increases profits without changing prices or product attributes of the original choice options, it has gained a lot of attention in the marketing literature and business practice. We know that the Economist uses attraction effect pricing to nudge consumers into buying the priciest 'print + online' subscription (cite Ariely predictably irrational). (cite blabla) also found in a field setting that consumers of a supermarket can be nudged into buying the pricier option of canned beens when a dominated option is added to the menu.

1   \section{Summary of the research plan}
2
3 + Almost all economic research assumes and relies on stable and rational preferences, which are necessary for the existence of a utility function -- the most basic and uncontroversial building block of our profession. Observed choice data has to necessarily satisfy the Weak Axiom of Revelead Preferences (WARP), which implies that a choice option can never become \emph{more} popular if the set of choices increases. Similarly, models of the random utility family, our workhorse models to analyze discrete choice data empirically, like the Logit, Probit, Multinomial Logit, Mixed Logit etc, assume that the choice frequency of a given option can \emph{never} be higher in a larger choice set (cite Marshak). In fact no data can be generated by a random utility process which violates this regularity assumption Consequently, random utility models perform poorly in the analysis of such data giving estimate that do not reflect the regularity violation (cite Natenzon).
4
5 + Experimental researcher are able to generate such violations for more than 30 years with what is now know as the decoy effect. The decoy effect can be divided into two related effect, the attraction effect which was discovered in 1982 by (cite 1982) and the compromise effect which was discovered by Simonson in 1989 (cite 1989). Since then, more than 100 studies were able to replicate the anomalies. To illustrate the \emph{attraction effect}, let us start with a choice set which contains two items: the target, a high quality cashmere sweater which costs \$100 and the competitor, a low quality synthetic one with costs \$20. A price conscious consumer might prefer the cheaper option in this original choice set. The attraction occurs when a third option, the decoy, which is dominated in price and quality only by the target, is added to the choice set. A typical decoy in this example is a cotton sweater which costs \$110. Both survey and field data show that the introduction of the decoy will make the high quality target look like a good deal in comparison to the low quality competitor and move consumers into purchasing the target more frequently. This effect is especially intriguing to the marketing profession. As the dominated option is never chosen by consumers and increases profits without changing prices or product attributes of the original choice options, it has gained a lot of attention in the marketing literature and business practice. We know that the Economist uses attraction effect pricing to nudge consumers into buying the priciest 'print + online' subscription (cite Ariely predictably irrational). (cite blabla) also found in a field setting that consumers of a supermarket can be nudged into buying the pricier option of canned beens when a dominated option is added to the menu.
```

Figure 2: GitHub - There it is!

Rule 2a: Put everything that has been created manually in version control

Add all inputs

- things generated by you / others
- scripts, data (if not too large), images from other sources

Leave out outputs

- things generated by the computer
- Use build tools to reproduce those instead
- Unless they take a very long time to create

Rule 2b: Work and track in small steps

this allows to move back to exactly this point in time and also understand the progress of your project better

- create snapshots regularly
- creates snapshots for logical steps, e.g.
  - “added slide for clean code”
  - “added slides about GIT”

Rule 2c: Use an issue tracking tool

- A shared to-do list
- Items can be assigned to people
- Supports comments, links to code and papers, etc.
- "Version control is where we've been, the issue tracker is where we're going"
- **Email is not an issue tracker!**
    - Although my advisor would seemingly disagree ;)

Rule 2d: Use pre-merge code reviews

- Develop on different branches or forks
- Review changes before merging in version control
  - Significantly reduces errors

Rule 3: Let the Computer Do the Work

- Computers exist to repeat things quickly and accurately
- 99% accuracy vs 63% percent chance of error in *simple* tasks

Rule 3a: Let the computer repeat and execute tasks

Functions

- Rule of 3:
  - If you copy-paste code 3 times, write a function instead
  - This reduces error rates and work

Projects

- Write little programs for everything
  - Even if they're called scripts, macros, or aliases
- Easy to do with text-based programming compared to GUIs
- We will search for the 'magic button'
  - One command that will execute your entire project
  - . . . after you have written ordered instructions

Rule 3b: Use a build tool to automate workflows

- Build tools originally developed for compiling programs
- Workflow becomes explicit
- Will become your ‘magic button’

Rule 4: Define things once, and only once

- Every input must have a single authoritative representation in the system.
- Define something *exactly once*
  - Make calls to that input each time you need to reference it
  - Example: Define important parameters in a dictionary, import into each script

Rule 5: Optimize Software Only After It Works Correctly

Experts find it hard to predict performance bottlenecks

- Get it right, then make it fast
- Small changes can have dramatic impact on performance
- Don't be scared to ask questions about how you could further improve

Use a profiler to identify bottlenecks

- Reports how much time is spent on each line of code

Rule 6: Plan for Mistakes

- No single practice catches everything
- turn bugs into test cases
- We can only try to prevent many
- Practice makes perfect

Rule 6a: Add assertions to programs to check their operation

"This must be true here or there is an error"

- No point proceeding if the program is broken...
- Error messages are implemented and expected by other user and programmers
- ... they also help you to make less mistakes and find errors faster

Can you summarize all of that?!

1. Use text-based interfaces
2. Turn history into scripts
3. Put everything in version control
4. Use test-driven development

A Warning

Where your brain may end up

A Warning

- 15 days $\times$ 6 hours/day = 90 hours of content
  - That's a lot! ... and fast
- You **will be tired** at various points
  - But don't confuse that with questioning the point of the course
- We can't transform your practices overmight ...
  - but persistance will make your programming life much (much!) more efficient
  - Think of us as a 'kick in the arse' to get you started

Let's Get Started!

Acknowledgements

This module is based on the 2016 and 2017 versions of the course:

- `Programming Practices For Economists`, by Lachlan Deer, Adrian Etter, Julian Langer & Max Winkler

It is designed after and borrows a lot from:

- `Effective Programming Practices for Economists`, a course by Hans-Martin von Gaudecker
- Software Carpentry's `Managing Software Research Projects` lesson

`Guiding Principles` borrows a lot from the paper

- Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) `Best Practices for Scientific Computing`. PLoS Biol 12(1): e1001745.

License

Suggested Citation:

- Ulrich Bergmann, Lachlan Deer, Adrian Etter, Julian Langer, Ursina Schaede, Dora Simon, Max Winkler, Carlo Zanella & Christian Zund, 2018, `Introduction and Motivation`, Programming Practices for Research in Economics, University of Zurich

Programming Practices Team

Programming Practices for Research in Economics was created by

- Lachlan Deer
- Adrian Etter
- Julian Langer
- Max Winkler

at the Department of Economics, University of Zurich in 2016.
These slides are from the 2018 edition, conducted by

- Ulrich Bergmann
- Ursina Schaede
- Dora Simon
- Carlo Zanella
- Christian Zünd