Programming Practices for Research in Economics

Introduction & Motivation

Lachlan Deer Adrian Etter Julian Langer Max Winkler

Department of Economics, Univeristy of Zurich

Fall 2017

Welcome!

Figure 1:

Introductions: Who We Are

- 3 PhD students
  - Lachlan
  - Julian
  - Max
- 1 software engineer
  - Adrian

Introductions: Who are You

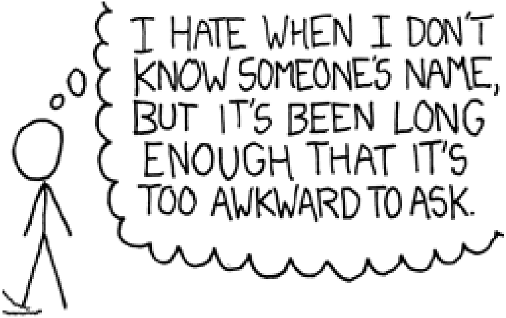

Figure 2:

Logistics: Basic Information

- Group is a mix of "for credit" and audit students
  - Want credit - enrol using sheet we will pass around in class
  - Grading: Pass/Fail based on final assignment
- Sessions are designed to be interactive
  - *Not going to be lecture based*
  - Style: mix of *live coding*, *small challenges*, *longer exercises*
  - We want to get you comfortable using your computing environment to solve problems
    - Bring your laptop!
    - Complete the installation guide prior to a session
    - Ask questions!

Logistics: Structure of each day

- Session 1: 9.30-12.30
- Session 2: 14.00 - 17.00
- Expect coffee breaks in each session
  - Exactly when depends on the leader of a session, and the material
- We expect you have completed the installation guide and have all software installed.
- 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/2017-uzh`
- Installation Guide:
  - `pp4rs.github.io/installation-guide`
- Course Chatter:
  - `pp4rs.slack.com/`
- GitHub repositories:
  - `github.com/pp4rs`
- Course (re-) Registration:
  - `Google Form`
- Pre-course Survey:
  - `Google Form`

Logistics: Assignment

- One final assignment
- Can be submitted in pairs
- Use what you learn in this course to solve a non-trivial economic problem
  - Pick a problem that is of interest to you ...
  - ... but must be understandable by us - (potential) non-experts in your field
- Solution must be submitted using GitHub
- Code and result write-up must be executable using a single line of code
- Propose to us an idea in a private `SlackChat` before you start
- Due 4 weeks *after* the last class

Logistics: Social Event

- Join us for casual drinks
- When: September 8th, 18.00
- Location: TBA

Logistics: Some Self-Promotion

- Keep talking about computational stuff after the course:

1. Discussion Group on Economics and Computing
   - Informal workshop
   - Meets once/twice per month, day and time TBC
   - One or two short demonstrations on how to use some software / package
   - Previously called 'Doing Cool Stuff on the Computer'
   - Relies on active input by all to be successful
2. Using Science Cloud for Research
   - Run by S3IT, hosted by us
   - Probably in late 2017 or early 2018
3. Other short workshops?
   - Announced on the Discussion Group email list, will depend on interest

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"

There are many non-believers

"If I wanted to be a computer scientist, I would have picked a different major in undergrad."

"Students are not interested in (learning) programming"

Even if we have ‘believers’

“What do I drop to make room for teaching more computing?”

- Have to fit in around the curriculum until we achieve critical mass

Most would rather fail than change.

- Most economists treat programming like the current US President treats research on climate change.

Figure 3:

The old view is unsustainable: Open Science Rising

- Pressure to “open up” the research process
  - 4Rs denote reproduction, replication, robustness and revelation
  - (Pagan and Torgler, Nature 2015)
- EU policy for all publicly funded research being *open* by 2020
- 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**

Replication vs Reproducibility: Replication

- Replication:
  - Running the same code on the same machine, do I get the same result?
  - Minimal requirement for qualifying as science
    - Unlikely satisfied by many papers/theses in economics

Replication vs Reproducibility: Replication

- Why haven't we cared about replication in the past?
  - Making a result replicable doesn't add to the knowledge base, perceived as time intensive
  - Is the increase in trustworthiness of original findings valued by the profession?

Replication vs Reproducibility: Reproducibility

- Scientific Reproducibility: independent verification of results
  - Do results depend on assumptions? data? software / other tools?
  - **Adds to the knowledge base**...
  - Isn't perceived as 'enough of a contribution'

Why are we Programming at all?

- As researchers we:
  - Create tools to be used by others
  - Apply tools developed by others
- Examples:
  - Developing a new estimator means that no-one has implementing the corresponding code before
  - Structural econometrics / macro: Model specifics and solution strategy are closely intertwined
  - Many lab experiments are unique in design and cannot easily port existing work
- Makes it hard to rely on generic programs
  - Code sharing would help, slowly improving (Barnes, 2010)

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

1. *nix bash 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. SQL
   - Manage larger data sets
5. Snakemake
   - Automate the execution of your research project

Advertise the tool, teach the thinking

How (Computational) Research Works?

How I wish Research Works

1. Design a Research Question
   - What do I want to do?
2. Design
   - How can I do what I want to do?
   - Plan out where I am going to go next
3. Implement
   - Do what I planned in design stage
4. Look at and use my results
   - Interpret, and produce output
5. End
   - Project done, will publish!

I have never managed to actually work this way

How it really works?

1. Analysis
2. Design & Implement in tandem
3. Look at results, and iterate on analysis step
4. Redesign, Re-implement
    - Some times go forward, sometimes revert backwards
5. ...

My Workflow has a Name!

- **Agile development**
  - Short iteration cycles
  - Feedback from different levels
- Important: it influences how we want to code, and (indirectly) what we teach (preach?)
  - as do 'coding' principles ...

Agile Feedback Loops

- Key feature is short iterations: single day to two weeks
  - One iteration: what to build next, design it, build it, tests it, and deliver it
  - We often don't know what we want until we (almost) see it
    - Short cycles avoid building things we don't actually want
  - Easy to organize a few days of work, hard to plan a full month
    - Reduce proportion of time spent on (re-)coordination

Working Agile

- Every day starts with a `stand-up meeting` where everyone reports:
  - What they did the day before
  - What they're planning to do that day
  - What's blocking them (if anything)
- Encourages us to break work down into tasks that are at most one day long
- "still working on X" several days in a row
  - Not doing it right, or reluctant to evalaute what we doing
- Many times your just meeting yourself, still useful concept
  - Report back to advisor key parts of these "meeting's minutes"

Making Agile Work

Works best when:

1. Requirements are constantly changing
2. Can communicate continuously, or at worst daily or weekly.
3. The team is small, so that everyone can take part in a single stand-up meeting.
4. Disciplined enough not to use “agile” as an excuse for cowboy coding.
5. You and your colleagues like being empowered.

What Limits Agile Development?

- Cowboy coding:
  - "I need to produce daily results, of course my code cant't be clean"
- Reluctance to be evaluated and make decisions
  - Don't be defensive
  - Your entire (research) career is full of evaluation and decision making

What Agile Development Doesn't Limit

- Making plans
  - Planning *still* important, maybe more so
  - Should write more plans, but shorter (daily) ones
- Documenting code
  - Things can change quickly, want short but descriptive documentation
    1. What this piece of code does
    2. How it relates to other pieces of code/output in your project
    3. How it might be extended next
- All that documentation shouldn't be in the same place...

Guiding Principles

Rule 1: Write Programs for People Not Computers

- Code that a computer can understand $\neq$ code a human understands
- Does that 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
- Write short, readable functions, each taking only a few parameters
  - **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.
- *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?

Rule 1b: Use meaningful variable names

- `p` less useful for short term memory than `price`
- `i`, `j` are (almost) OK for indices in small scopes
  - `ixx` and `jyy` might be better
  - `iDescription` is even 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
- Good ideas:
  - Keep each line of code within 80 characters
  - Consistent naming convention
  - Whitespace is your friend

Rule 2: 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 2a: Let the computer repeat and execute tasks

- 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 2b: Save recent commands when developing new code

- Most text-based interfaces do this automatically
  - Repeat recent operations using history
  - "Reproducibility in the small"
- Saving history supports "reproducibility in the large"
  - An accurate record of how a result was produced
- Have a 'sandbox' where you explore interactively and save your commands
  - Move it across to your main scripts once the pilot excercise works

Rule 2c: Use a build tool to automate workflows

- Build tools originally developed for compiling programs
  - Adapted for managing code dependencies
- Workflow becomes explicit
- Will become your 'magic button'

Rule 3: Make Incremental Changes

- Most researchers don't have "requirements" when making changes to code
  - 'change it until it works how I want it to', then save it
- Code evolves in tandem with research
  - potentially in lumpy increments
- *Agile development*. . .

Rule 3a: Work in small steps

- Frequent feedback and ability to correct the course when things go awry.
- People can concentrate for 45-90 minutes without a break
  - So size each burst of work to fit that
  - within a burst, save history of changes as you make (small) progress
- Longer cycle should be a week or two
- Design Issues and To Dos with these timelines in mind

Rule 3b: Use a version control system

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

Email is not version control!

Rule 3c: Put everything that has been created manually in version control

- Not just software: papers, raw images, ...
  - Not gigabytes...
  - ...but metadata about those gigabytes
- Leave out things generated by the computer
  - Use build tools to reproduce those instead
  - Unless they take a very long time to create

Rule 4: Don't Repeat Yourself (or Others)

- Anything repeated in two or more places will eventually be wrong in at least one
- If it's faster to 're-create and duplicate' than to discover or understand, fix what you're doing
  - Usual solution: re-modularize and import the "replicated" part

Rule 4a: 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 4b: Modularize code rather than copying and pasting

- Reducing code cloning reduces error rates
  - Decreases testing needed to ensure your code does the right thing
  - . . . and increases comprehension
- a parameter dictionary is an example of modularization

Rule 4c: Re-use (good) code instead of rewriting it

- Takes years to build high-quality numerical or statistical software
- Your time is better spent doing research on top of that, not trying to rewrite it to suit your needs,
  - Or avoid the errors we are getting from the software
- Code your advisor sends you may not be *good code*
  - Sometimes you can be better off rewriting, than comprehending
    - Recall in these situations we wanted to fix the problem
  - There is a generation gap

Rule 5: Plan for Mistakes

- No single practice catches everything
- Practice defense in depth
- *Note: improving quality increases productivity*

Rule 5a: Add assertions to programs to check their operation

- "This must be true here or there is an error"
  - Like diagnostic circuits in hardware
- No point proceeding if the program is broken...
  - ...and they serve as executable documentation
- Error messages are implemented and expected by other programmers
  - Indicators to improve your code

Aside: Testing is Hard

- "If I knew what the right answer was, I'd have published by now."
- How to test your code works? Compare to
  1. Experimental or synthetic data
  2. Analytic solutions of simple problems
  3. Old (trusted) programs
- Documents what errors are acceptable

Rule 5c: Turn bugs into test cases

- Write a test that fails when the bug is present
  - Work on the code until that test passes...
  - ...and no others are failing
- Write tests parallel to code
  - Otherwise you won't write them

Rule 6: Optimize Software Only After It Works Correctly

- Experts find it hard to predict performance bottlenecks
  - Small changes can have dramatic impact on performance
- Get it right, then make it fast
- Don't be scared to ask questions about how you could further improve
  - e.g. the engineering team

Rule 6a: Use a profiler to identify bottlenecks

- Reports how much time is spent on each line of code
- Re-check on new computers or when switching libraries
- Summarize across multiple tests

Rule 6b: Write code in the highest-level language possible

- People write the same number of lines of code per hour regardless of language
- Use the most expressive language available to get the "right" version...
  - (Potentially) Rewrite core pieces in lower-level language to get the "fast" version
  - General trade-off: expressiveness vs speed
  - We don't explore: High level, fast langauge - Julia

Rule 7: Document Design and Purpose, not Mechanics

- Documentation: make 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 7a: Document interfaces and reasons, not implementations

- Interfaces and reasons change more slowly than implementation details,
  - sDocumenting them is more efficient
- Most people care about using code more than understanding it

Rule 7b: Refactor code instead of explaining

- Good code can be understood when read aloud
- Good programmers build libraries so that solving their problem is straightforward
- Too many comments?
  - You probably violated rule 1b
- No comments?
  - You *will* forget
- Commenting the comments?
  - Your lost!

Rule 7c: Embed the documentation in the code

- Most languages have specially-formatted comments or strings
  - Likely to be kept up to date
  - More accessible than opening a supplementary file
- General movement towards "code in documentation" rather than vice versa
  - `Jupyter` and `Rmarkdown` are relatively extreme examples

Rule 8: Collaborate

- Computers were invented to calculate
- The web was invented to collaborate
- Research is more fun when it's shared
  - Unfortunately you will likely be obligated to write one solo authored paper during your PhD
  - But *it really isn't a solo effort*

Rule 8a: Use pre-merge code reviews

- Review changes before merging in version control
  - Develop on different branches or forks
- Significantly reduces errors
  - Good way to share knowledge
- It's what makes open source possible

Rule 8b: Use pair programming

- When bringing someone new up to speed and when tackling particularly tricky problems useful to do it together
- Two people, one keyboard, one screen
  - An extreme form of code review
  - Puts you on the same page
  - Rarely done ... probably our fault
- Can get a bit tiring if done tpp often

Rule 8c: 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 ;)

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

Who can benefit from this course?

Research Projects that can benefit

1. The reduced form empirical paper
2. The econometric theory paper
3. The simulation / calibration paper
4. The structural econometric paper
5. (Theory papers?)

The Reduced Form Research Project

- All estimators you ever need implemented in R
- Use it from beginning to end
- Potentially Python for formatting of tables, if `stargazer` or `xtable` does not give you directly what you want
- Consider SQL for data management ...
  - ... if your data do not fit into memory;
  - ... if you need to describe complicated relations between observational units;
  - ... or if they have multiple dimensions

The Econometric Theory paper

- Deriving the theoretical properties of an estimator alone rarely gets published
  - Monte Carlo simulations are the absolute minimum
  - Should provide package which can be used by applied researchers (Koenker and Zeileis, 2009)
  - Especially if you want the estimator to be used
- Develop your code in Python, or R
  - When you're 'done', write the R package
  - Close to trivial if your code is well done in the first place

The Simulation / Calibration paper

- Real data and lots of assumptions on distributions leading to simulated data
- A (potentially) computationally intensive component in the main analysis
- Most problems: Python
- In some cases: a mix of Fortran or C with Python (f2py, Cython, etc.)
  - or Julia
- May want to scale to cloud computing, computational cluster

The Structural Econometric Paper

- Generate predictions about behaviour from economic theory, find the “right” parameter values for the theory
- Data management is more involved, allow for heterogeneity, need standard errors
- Suggestions from simulation scenario apply
- Separate the estimation of parameters, calculation of standard errors, and counterfactual analysis
    - Waiting to proceed seqeuentially wastes computational time

Each paper needs

- Version Control: keep track of what you have done, go back if needed
- Automatic Task Execution: executes code in the correct order each time
- Clean and Documented code: so you and others understand what your code does!

Where to next?

Our Course

- What we cover (in more detail):
  - Unit I. Command Line (Adrian)
  - Unit II. Version Control (Adrian feat. Lachlan)
    - VC with a Local Repository
    - VC with a Remote Repository
    - Project Management with GitLab
  - Unit III. Programming Languages
    - Python (Julian and Lachlan)
    - R (Julian and Max)
    - SQL (Lachlan)

Our Course (cont.)

- What we cover
  - Unit IV. Automation of Task Execution
    - Snakemake (Lachlan)
  - Unit V. Coding Practices (Adrian)

We Cannot Cover Everything

- We miss (important) topics such as:
  - Unit testing
  - Complete documentation of Research Projects
  - High Performance Computing with (. . .)
  - many others ..

A Warning

Figure 4:

Where your brain may end up

Figure 5:

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!

Figure 6:

Acknowledgements

- This module 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.
- The material from above sources is made available under Attribution based Licenses, as is this course's material.

License

- Material is licensed under a CC-BY-NC-SA license. Further information is available at our `course homepage`
- Suggested Citation:
  Deer, Lachlan; Adrian Etter, Julian Langer, Max Winkler (2017), `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 2017 edition, conducted by the original course developers.