Math Camp 2020: Programming (part 1)

Frank Pinter

20 August 2020

Outline

Structure of today's session

1. General programming advice ($\approx$ 30 min)
2. Julia walkthrough ($\approx$ 60 min)
3. Open time for questions ($\approx$ 30 min)
   - Feel free to ask questions throughout!

All materials from today's presentation are on GitHub at
`https://github.com/fpinter/math-camp-coding`

Table of Contents

Basic principles

1. Learn by doing (practice!)
2. Always keep your future self in mind
3. Your time is valuable
4. Don't reinvent the wheel

Learn by doing

- Especially early in grad school, treat learning about programming as an investment
- Practice new skills as often as you can
- Programming needs frequent reinforcement

Always keep your future self in mind

- When you return to a project later on, you should be able to:
  - Figure out what's going on relatively quickly
  - Not screw things up
- Write clear documentation and keep it updated (don't rely on your memory)
- Clearly written code $\gg$ over-commenting
  - Use good variable names
  - Use good function names
  - Use functions to simplify things
- Write comments with a specific audience in mind
  - Typically your future self and your collaborators

Your time is valuable

- Your time is more valuable than the computer's time
- Prioritize organization, readability, and clarity over fast runtime
- Resist the temptation to focus on runtime too early
  - Wait until you've checked for accuracy
  - Wait until you have a clear idea which parts are actually critical
- Runtime is important in parts of the code you'll be running many times as part of your usual workflow (and those parts only)

Your time is valuable

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We* **should** *forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> *Yet we should not pass up our opportunities in that critical 3%.*

–Donald Knuth (1974)

Don't reinvent the wheel

- There might be ways to solve your problem you hadn't thought of
- Talk to your cohort, talk to people you know, ask questions online
- Stay up to date on tools and the technical community
- If something feels like a common problem, spend time looking for a common solution
  - Poll: are you familiar with regular expressions?

Table of Contents

We are more like software engineers than we think

Miklos Koren
@korenmiklos

Replying to @rlmcelreath

A key misunderstanding in scientific computing is that software engineering best practices are only for large-scale production systems. Wrong. These practices shine not when the exact same code runs a zillion times, but when code has to be changed often. Exactly like in science.

4:02 PM · Aug 14, 2020 · Twitter Web App

1 Retweet **17** Likes

(source)

When writing code

- Don't repeat yourself
  - Don't copy and paste; write functions

- Write (and save) formal tests
  - Write tests for functions *when you write the functions*
  - Write checks your data should pass
  - Tip: accumulate a bank of test cases over time

- Know and use the idioms of your language
  - Know *why* your code works the way it does
  - Know the common gotchas
  - Nothing should be magic!

- Understand all unexpected results
  - Check all your results for anything unusual (smell test)
  - Learn how to read your language's error messages

When organizing your project

- Split your code into steps, with a clear order
  - Having a master script is strongly recommended
  - You should be able to clear all your outputs/intermediate files and run the master script
- Aim for full reproducibility, including a detailed readme file instructing a replicator *exactly what to do*
  - Keep this file continuously updated as you work
  - Fewer steps for the replicator = better
- Don't write critical parts of your code under time pressure
  - If you do, go back and clean it up later
- Use version control to track changes over time

Note on choosing programming languages

- Tired: wars between programming languages on Twitter
- Wired: using the right language for the task at hand
- There's no rule saying you have to use the same language for everything (even within a project)
- Questions to ask yourself:
  - What do your coauthors use?
  - In what language do you work most efficiently?
  - What functionality do you need?
    - e.g., data cleaning, web scraping, heavy computation

Table of Contents

First year is different from research

- The work you do in first year is very different from the work you'll do later on, regardless of field
- First year teaches you how methods work conceptually, not how to use them in research
- Advice: treat this as an opportunity to learn by doing
  - Develop good habits
  - Spend some time planning your workflow and your approach
  - Decide what skills you want to learn, and take first year as an opportunity to learn them

First year vs. research

	G1 coursework	Research/real life
Day to day work	Numerical computation with clean or simulated data	Mostly wrangling real-world data (unless you're a theorist)
Maintainability	Submit and you're done	Return to your code many times, sometimes years later
Accuracy	Nice to have	Be obsessive about making sure your results are correct

First year vs. resarch

	G1 coursework	Research/real life
Testing	Smell test + write formal tests for basic debugging	Smell test + write lots of formal tests
Collaboration	Discuss with group, but write code independently	Divide tasks + perhaps do code review
Version control	Nice to have, but optional	Very important

Table of Contents

Further resources

- LJ Ristovska's presentation
- Jesús Fernández-Villaverde's lecture notes
- QuantEcon
- Harvard IQSS training materials

How to get help

1. Check the built-in help in the language
2. Google
   - Often the result will be a Stack Overflow answer – these are often helpful but not always
   - Watch out for out-of-date info (especially for Julia)
3. Ask someone
   - Plug for the econ department Slack
4. Ask a question on Stack Overflow
   - Stack Overflow has guidance on how to ask a good question (varies by language); read that first

Table of Contents

Why Julia today?

- The focus of math camp: skills you'll use in first year
  - Numerical computation, matrix algebra, optimization

- Julia excels at these and its matrix syntax is clean

- Historically the dominant language for first year PhD was Matlab
  - Julia syntax is closely based on Matlab
  - Unlike Matlab, Julia is free and open-source, with a growing community, and many of the advantages of modern languages
  - You can switch back to Matlab anytime if you want

Alternatives to Julia

- Matlab
  - Legacy code + inertia
  - Dynare (for macro)
  - Lacks features of modern languages
  - Expensive outside of academia (or use Octave)

- R
  - De facto standard in statistics
  - Great for work with real data
  - Matrix syntax is less intuitive

- Python
  - De facto standard in the tech industry and, increasingly, physical sciences
  - Great all-purpose language ("Swiss army knife")
  - Matrix syntax has improved but is still less intuitive than Julia's

Pros and cons of Julia

- Pros
  - Clean syntax and fast execution for numerical computation
  - Native support for automatic differentiation makes optimization easy, robust, and quick
- Cons
  - Generally harder to use than R or Python for manipulation of real data
  - The language is unstable (most resources from before 2018/Julia v1.0 are useless)
  - The community is smaller than R and Python

More on Julia vs. other languages

- Why I encourage econ PhD students to learn Julia (Jonathan Dingel, September 2018)
- Scientific Computing Languages (Jesús Fernández-Villaverde, November 2019)

Now...

Time for the demo!