Research Computing Bootcamp, day 2

Bulat Gafarov

University of California, Davis

September 24, 2019

Plan for today

- Why do we need to learn software engineering techniques?
- Debugging the code
- Prototype vs readable and reusable code
- Modularity principle
- Testing your code
- Commenting your code
- Optimizing your code
- Where to learn more?

Why do we need to learn coding techniques?

- Homeworks vs research projects
- There are universal principles of good code (applicable to LaTex, R, Stata, Matlab etc)
- Good coding saves time and mental power!
- Your coauthors, advisors, readers, and *future you* will appreciate that

Why do we need to learn coding techniques?

- Homeworks vs research projects
- There are universal principles of good code (applicable to LaTex, R, Stata, Matlab etc)
- Good coding saves time and mental power!
- Your coauthors, advisors, readers, and *future you* will appreciate that

Why do we need to learn coding techniques?

- Homeworks vs research projects
- There are universal principles of good code (applicable to LaTex, R, Stata, Matlab etc)
- Good coding saves time and mental power!
- Your coauthors, advisors, readers, and *future you* will appreciate that

Why do we need to learn coding techniques?

- Homeworks vs research projects
- There are universal principles of good code (applicable to LaTex, R, Stata, Matlab etc)
- Good coding saves time and mental power!
- Your coauthors, advisors, readers, and *future you* will appreciate that

Some features of good code

1. Works as intended
2. Easy to read by others and *future yourself*
3. Easy to find errors
4. Can be reused in other projects easily
5. Can be modified and extended easily

Some features of good code

1. Works as intended
2. Easy to read by others and *future yourself*
3. Easy to find errors
4. Can be reused in other projects easily
5. Can be modified and extended easily

Some features of good code

1. Works as intended
2. Easy to read by others and *future yourself*
3. Easy to find errors
4. Can be reused in other projects easily
5. Can be modified and extended easily

Some features of good code

1. Works as intended
2. Easy to read by others and *future yourself*
3. Easy to find errors
4. Can be reused in other projects easily
5. Can be modified and extended easily

Some features of good code

1. Works as intended
2. Easy to read by others and *future yourself*
3. Easy to find errors
4. Can be reused in other projects easily
5. Can be modified and extended easily

Example: LaTeX

- Making your own presentation/article style
- Adding computer-generated figures/tables
- Reorganizing your article / slides
- Changing notation using macros
- Making cross-references

Example: Stata do files

- Choosing names for variables
- Use the same data cleaning procedure with new data set
- Running multiple robustness checks

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Looking for errors in code

- Big chunks of code are hard to understand! Start with small chunks
- Execute code line by line
- Use breakpoints to skip parts that work
- Print out intermediate results
- Split big formulas into substeps and use multiple lines (one operation per line)
- Avoid reusing variables for different purpose in the code (like tmp variable)

Prototyping vs reusable code

- First version of the code may be poorly written
- Some code is not going to be reused. You can sacrifice quality to save your time
- Other code will be read and used many many times, over multiple years, and by many readers (huge time savings in long run)
- Such code requires extra effort

Evils of duplication

- It is bad practice to copy and paste parts of code!
- Errors get transferred from one part of the code to another
- If code changes in one place, it need to be changed in another place
- Duplication makes code longer

Modularity

- Small chunks are great:
  - easy to keep in mind
  - easy to debug
  - easier to reuse and re-purpose
- Split code into "modules"
  - Create your own functions
  - Split scripts into small subscripts that perform a single task
  - Easier to reuse and re-purpose without duplication!
- Group related modules in folders with appropriate names

Levels of abstraction

- It is hard to switch between low and high level reasoning
- It makes errors hard to catch!
- Make level of abstraction consistent within a "module"
- Make modules no longer than one page of a code, ideally 5-10 lines

Example: LaTeX

- Put preamble in a separate file (document class, style, macros etc)
- Put sections in separate files and use input command (great for reorganizing document, version control specific to a section, reusing slides)
- Bibliography is in a separate file (.bib)
- Tables and figures are in separate files (generated using Stata or R)
- Diagrams (tikz etc)

Example: Stata

- Split long do file into separate files that do a single task
- Main do file would consist of calls of do subfiles
- You could run sections separately (debugging), reuse some of the in other projects

Choice of names

- Variables names should be self explanatory and you can read
- Use names you can search and replace (avoid single letters)
- Avoid encoding meaning / using acronyms
- Module names should reflects the content, functions should reflect what it does
- Avoid using fun(x) , g(x) etc (need to understand the full code to use it)
- Good naming conventions help to avoid extensive commenting
- Use long names in high level code like scripts, and short names in low level code inside modules

Testing your code

- When changing a code, run tests
- *Unit tests*
  - Small scripts that run individual modules and compare results with a known answer
- *Integration tests*
  - Scripts that test how modules work together
- Most languages (including Julia and Matlab) have dedicated software for automatic testing
- Robustness checks in your paper are unit tests of your project, while comparison with literature is your integration test!

Commenting the code

- Comments are duplication of your code that needs to be updated!
- Try to make code that does not require a lot of commenting
- Using good names, good modular structure helps
- Introducing extra variables for intermediate results helps as well

Your code works! What's next?

- Does it run fast enough?
  - If not, try Profiling
- Are you planning to use it again in the future?
  - If yes, consider spending extra time to replace naming conventions, eliminate duplication, make modules more reusable
- Do not optimize the code too early. Always start with a readable code, not fast code
- It is often hard to guess bottlenecks in the code
- Only optimize parts that profiler told you
- Your time is more valuable than computer's time!

Reading materials

- *Clean code: a handbook of agile software craftsmanship* by Martin, Robert C. Pearson Education, 2009.
- *The Pragmatic Programmer: From Journeyman to Master*, by Andrew Hunt, David Thomas, and Ward Cunningham. Addison-Wesley Professional, 2000.