Advanced Macroeconomics II

Handout 1 - Course Intro, Version Control, Best Practices

Sergio Ocampo

Western University

September 16, 2020

A bit about me

- I'm new! (kind off... sort off...)
  - First time teaching a PhD course (sorry)
- I am from Colombia (undergrad, masters and worked at central bank)
- Spent a year at the Inter-American Development Bank
- PhD at Minnesota
- Work on "modern macro" (Minnesota style)
  - We (minnesotans) have a very broad definition of macro...

A bit about my work

- Papers on many topics:
  - Tasks and occupations, wealth taxes, concentration, self-employment
- One unifying theme: **Heterogeneity**
- Modern macro is all about (cross-sectional) heterogeneity
  - Workers vary in skills, investors in rate of return, entrepreneurs in productivity, markets in concentration, consumers in wealth and income
  - List goes on: age, marital status, health, race, gender, human capital
- The line between modern macro and micro is blurry:
  - Macro models need to capture a lot of micro-behavior
  - Empirical backing for model assumptions from data

A bit about my work: Wealth taxation (1/4)

- Should capital be taxed? What is the optimal value of τ_k?

$$c + a' = a + (1 - \tau_k)\, ra + wn$$

  - People used to think answer was no: $\tau_k = 0$ (Chamley-Judd)

- That answer is wrong:
  - Theoretically (Straub & Werning, 2020)
  - Quantitatively - Here is where heterogeneity plays a role

- $\tau_k > 0$ optimal if agents face **idiosyncratic labor income risk** (Aiyagari, 1995; Imrohoroglu, 1998; Boar & Midrigan, 2020)
  - Result maintained after adding life cycle and other taxes (Conesa, Kitao & Krueger, 2009; Many others)

A bit about my work: Wealth taxation (2/4)

Use it or lose it: efficiency gains from wealth taxation
with Fatih Guvenen, Burhan Kuruscu, Gueorgui Kambourov and Daphne Chen

- How is taxing capital income different from taxing wealth?

$$c + a' = \tau_a a + (1 - \tau_k) ra + wn$$

- They are equivalent! Replace τ_k for $\tau_a = r\tau_k$
  - This is true even if agents are heterogeneous: labor income, life cycle, retirement, mortality risk, bequest motives
- Equivalence breaks if agents have **heterogeneous returns**!
  - Wealth taxes favors agents with high r (leading to efficiency gains...)

A bit about my work: Wealth taxation (3/4)

Lesson:

- Different forms of heterogeneity have different effects
- Ask: what is the relevant form of heterogeneity
  - relevant theoretically
  - relevant empirically

Why heterogeneous returns?

- Theoretically interesting (break equivalence of taxes)
- Empirically relevant
  - Necessary to capture fat tail of income/wealth distribution (Work of Benhabib, Bisin and coauthors; Akira-Toda, 2019)
  - Direct empirical evidence: Norway (Fagereng, Guiso, Malacrino & Pistaferri, 2020) US (Smith, Yagan, Zidar & Swick, 2020)

A bit about my work: Wealth taxation (4/4)

Two tasks:

1. Establish conceptual result (this time it was easy)
2. Show result is quantitatively relevant

Quantitative methods in modern macro

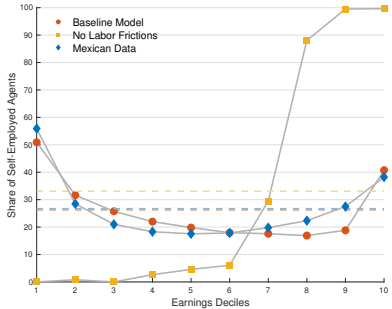
- Heterogeneous agent models required for validation
- In my paper I include and match moments for:
  - Life cycle: work, retirement, mortality risk, bequests
  - Source of income (entrepreneurial activity, savings, labor)
    - Labor income risk
    - Heterogeneous returns
- Individual problem has 6 states variables (11 million combinations)

A bit about my work: A pattern

Quantitative validation of "macro results" requires heterogeneity

- What is behind the (aggregate) trend of concentration in the U.S?
  - Look at competition in individual markets
- Why is self-employment so much higher in developing countries?
  - Who are the self-employed? Why does it matter?
  - Look at the self-employed across the earnings distribution

1. Look at pattern in micro-data
2. Contrast model result
3. Understand mechanisms



Course objectives

What you get out of the course:

1. Quickly implement and test research ideas
   - Most ideas are bad... you need a way to check
2. Workhorse heterogeneous agent model
   - Key to understand literature
3. Coding: methods, tools, practice

What I get out of the course:

1. I am going to learn Julia with you
2. Hopefully convert some of you to the true faith

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model

- Most tools can be learned here
- Starting point for many models

2.1 Value function iteration (and how to speed it)
2.2 Continuous choice / First order conditions
2.3 The endogenous grid method
2.4 Shocks and expectations

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model
3. Adding distortions
   - 3.1 (k,K) models
   - 3.2 Sovereign default models

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model
3. Adding distortions
4. Heterogeneity
   - 4.1 The Bewley/Hugget/Aiyagari/Imrohoroglu model
   - 4.2 The stationary distribution
   - 4.3 The life cycle heterogeneous agent model

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model
3. Adding distortions
4. Heterogeneity
5. Extensions
   - 5.1 Discrete choice (Occupational choice problems/Retirement)
   - 5.2 Transition out of steady state

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model
3. Adding distortions
4. Heterogeneity
5. Extensions
6. Some topics
   - 6.1 Cross-Sectional moments for macroeconomics
   - 6.2 Modeling at the frontier

(Tentative) Course outline

1. Basic tools (version control + coding best practices + basic code)
2. The Neo-Classical growth model
3. Adding distortions
4. Heterogeneity
5. Extensions
6. Some topics
7. Search Models (just the basics)
   - 7.1 The McCall model
   - 7.2 The DMP model
   - 7.3 Directed Search

Course mechanics

- Weekly topic covered in live (video) lecture
  - 3 hours with break in the middle
- Weekly problem set
  - Problem sets to be done individually
  - Submit solution via a public (github) repository
  - Readme file, ready-to-execute file, pdf if necessary
- All grade comes from problem sets
  - You get to drop two

Version Control: Git

Slides by Dominic Smith

What is version control?

- Software that keep track of changes to files
- Store history of all changes done to code/figures/output
- The language that lets you keep track of this changes is called **Git**
- We will only deal with the (very) basics of version control
  - Basic Git commands
- All projects benefit from version control. We will use it in all assignments.

Why is version control useful?

- You often need to access previous versions of files
- You always need backups of your code and results
- You often need to share your code with others and collaborate
- Git is better than alternatives:
  - Renaming files or creating new folders

Benefits of Git

- Git only tracks the changes (differences between files)
  - Only saves files when they change (no overhead)
  - Easy to compare versions (only see changes)
- Git lets you know which files were modified for a specific purpose
- Git gives names to the changes to your code (easy identification)
- Git does not clutter your folders with version upon version
- Easy to share and collaborate

When should you use version control?

(Almost) Always!

- Particularly useful in any long project
- Models evolve in versions
  - You often have to go back and check how things work in a smaller version of the model
- Robustness exercises demand several versions of the same code
- When collaborating

Version control for collaboration

- Git can act locally to keep track of the changes in your local files
  - A local repository of files and their changes
- Git repositories can also interact
  - Link your local repository to an online repository (github, bitbucket)
  - Other people can link to the same online repository
  - Collaborate by submitting your local changes to the online repository
- Online repository is also a backup for your code

Version control in this course

- All assignments should be available on a git repository
- You must create the repository and maintain it
- Upload problem sets to the repository
- Problem sets are individual, but if you choose to collaborate do it with Git (and let me know)

Install Git (if you don't have it)

- Go to https://git-scm.com/
  - Can use getting started tutorial there
  - Atlassian also has useful information: https://www.atlassian.com/git/tutorials
  - Github also has tutorials
- GIT comes with a GUI (graphical user interface) and command line
  - Things are faster with command line

Start a git repository for the class

Go to a folder with files you want to track (in the command line)

1. Type **git init**: Creates/Initializes an empty repository in that folder
2. Create a readme file and a .gitignore file
   - 2.1 readme.txt gives some information about what the repository contents
   - 2.2 .gitignore tells git not to track certain types of files
3. Type **git add 'list of files'**: Tells git to track files/folders in the list
4. Type **git commit –m "First Commit"**: Saves the version of the files with a note that this is your first commit
5. Link your repository to an online repository (problem set)

Now you can always go back to this exact version of your files

What to do now?

Assume we just committed files

1. Modify some number of files, potentially adding new ones
2. **git add** any new files
3. **git commit –a -m** “Message to remember what you modified”

Two main reasons to modify files:

1. Added a new feature
2. Fixed a bug

You want it to be clear which code fixed bug and which added feature

- Use messages to inform of what happened
  - **git commit -a - m “added program to fix bug”**
  - **git commit -a - m “changed program to add new feature”**

Help! I'm stuck in VI/VIM

If you don't type **–m** after git commit you get in trouble!

- You get thrown into the default text editor, often VI/VIM
- These are archaic and moody editors...

Here is what to do:

1. Type ESC then :qw and hit return
2. You'll need to commit again
3. Alternatively you can learn VIM, but that is a lot of work

“Advanced” commands

- **git clone**: Useful to start a new project using an old repository, also good to link to an online repository
- **git pull/push**: Useful to communicate with your online repository, pull a new version from it, push a new version to it
- **git branch**: Creates a copy of your repository and tracks changes to it separately
  - Type **git branch Branch_Name** to create branch
  - Type **git checkout Branch_Name** to move to the branch
  - This is the most useful command to keep track of alternative versions of your code
- **git merge**: Merges two branches, useful when done experimenting
- **git reset –hard HEAD**: Returns your repository to its last commit, useful for undoing catastrophic mistakes

Best Practices

1. Breaking up code
2. Readme files
3. Start small
4. Time your code
5. No parallelization

Breaking up code (1/5)

Having all your code in the same script is a bad idea

- Worse: It looks like a good idea at the time
- Your future self will regret it
- Your future projects will suffer from it

Why break scripts up?

- Easy to edit (less lines/script, know what is in script)
- Easy to track changes (most scripts left untouched)
- Easy to reuse (across projects, across model versions)

Breaking up code: Three ways to do it (2/5)

1. Load lines of code from another script
2. Functions
3. Modules

Breaking up code: Three ways to do it (2/5)

1. Load lines of code from another script
   - Useful for portions of code that are repeated often
   - Also useful for separating portions of code that are different
     - Solving model vs Graphing solution vs Saving results
   - No need to pass variables
   - Uses same workspace as the "main" script
2. Functions
3. Modules

Breaking up code: Three ways to do it (2/5)

1. Load lines of code from another script
2. Functions
   - Useful for portions of code that (kind of) repeat themselves... Perform the same tasks but use different variables
   - Need to have (more) defined inputs/outputs (private scope)
3. Modules

Breaking up code: Three ways to do it (2/5)

1. Load lines of code from another script
2. Functions
3. Modules
    - Basically groups of functions

Breaking up code: Breaking up is so hard to do (3/5)

- Most code starts as a simple problem (no need to break up)
- But they grow so fast!
  - My wealth taxation code has 7 modules
  - Module on model solution has 7500+ lines, 51 functions
- Not ex-ante clear where one module should end or another start
- Not always clear where to place functions

Breaking up code: Too much of a good thing (4/5)

Careful with repetition, breaking up code incorporates overhead

- Assign consumption for (say) 20 million agents.
- Two options
  1. c[i] = Y(a[i],n[i]), where Y(x,z) = (1+r)*x+w*z is a function
  2. c[i] = (1+r)*a[i]+w*n[i]
- First option makes it easy to change income
  - Only have to do it in one place (function definition)
- Second option avoids calling function Y millions of times...

Breaking up code: Rules of Thumb (5/5)

- Have a "main" script. Keep it as simple as possible.
  - Include flags (run everything, run some parts, load results)
- Modules for:
  - Toolbox (multi-project)
  - Parameter values
  - Initialization (set up grids, transition matrices, etc)
  - Model Solution
  - Model Simulation
  - Model Results (compute stats, save results)
  - Graphs
- Functions for:
  - Everything that you write three times!

Readme files

- Always have one!
  - I learned this one the hard way... So much code I have no idea what it does
- Easy to do:
  - "The code in this folder solves X model."

Start small

- Always start from the simplest version of the model
- Key is to understand the mechanism you want
- Mechanism should work without added features
- You will always face the question: what is really driving your results?

Bonus: You often know the answer in smaller models

Time your code

- Only way to know what is working and what is not
- Valuable information for scaling up code
  - Estimation
  - Simulation
- Poor man's timing:
  - Matlab's tic-toc or Julia's @time or package "TimerOutputs"
  - Use often, "no" overhead, fast iteration
- Rich man's timing:
  - Profile (both Matlab and Julia)
  - Use sparingly, less manageable as code grows

Hold off on parallelization

- Parallelization is not a substitute for good code
- Easy to be lazy... just add more threads...
- Parallelization often introduces new errors
  - You need to have a working benchmark you trust

Message applies to other forms of code optimization:

- First have your code working, then make it fast

Julia/Matlab

Julia vs Matlab

- I am convinced Julia is the future of scientific computing
- Matlab is easy to do.. but Julia seems as easy
- Julia is much more versatile
- You can use the program you prefer (but I want you to use Julia)

Key: You won't get to choose Matlab if you are doing large scale models

Installation

Matlab

- https://wts.uwo.ca/sitelicense/matlab/

Julia

- https://juliacomputing.com/products/juliapro.html
  - Choose current stable release
  - This should be bundled with Atom, the editor we will use
- Install plots package: import Pkg; Pkg.add("Plots")

Resources

- Julia's manual (actually very readable):
  https://docs.julialang.org/en/v1/
- Best allies: Google + StackOverflow + JuliaDiscourse
- QuantEcon: https://julia.quantecon.org
- Share what you find!

Appendix

Appendix Slides

- Nothing yet...