Organizing Data for Economic Research

Part 1: Managing Workflow

Brendan M. Price
UC Davis

October 31, 2019

Three-part series

Part 1: Managing Workflow
- Principles
- Project folders
- Maintenance
- Computing
- Collaboration

Part 2: Handling Data (Thurs. 11/14)
- Coding practices
- Data cleaning, validation, & analysis

Part 3: Sharing Your Work (Thurs. 12/05)
- Figures & tables
- Slides & drafts
- Replication packages

Basic game plan

General principles + Stata specifics

- Ecumenical—but Stata's the devil I know

Afterwards: will post accompanying materials

- These slide decks
- Sample codebase
- Stata scheme for `slick` graphics
- Beamer template for `sleek` slides

Recommended reading

Workflow & coding:

1. Gentzkow & Shapiro, “Code & Data for the Social Sciences”
2. Long, *The Workflow of Data Analysis Using Stata*
3. Paarsch & Golyaev, *A Gentle Intro to Effective Computing*
4. Cameron & Trivedi, *Microeconometrics Using Stata*
5. Brooks, *The Mythical Man-Month*

Aesthetics:

5. Tufte, *The Visual Display of Quantitative Information*
6. Schwabish, *Better Presentations*
7. Oetiker et al., *The Not So Short Introduction to LaTeX 2_ε*
8. Bringhurst, *The Elements of Typographic Style*
   (not about data—but a fun read for the design enthusiast)

Not comprehensive—suggestions welcome

Notably absent

Some things I won't be covering:

- Data acquisition
- Web-scraping
- Shell scripts
- Pre-analysis plans
- (Not) *p*-hacking
- Any econometrics

Only a minimal treatment of version control

- Perpetually on my to-learn list
- For now: "poor [hu]man's version control"

Plenty of Stata tips, but not a Stata tutorial basic premise:

there are big returns
to better data organization
& better data presentation

Benefits $\gg$ costs

Non-trivial costs:

- Frontloaded time investments
- “Productive procrastination”

Big behind-the-scenes benefits:

- Net time savings
- Better grasp of the data
- Happier coauthors*
- Fewer headaches, lower blood pressure**

Big public-facing benefits:

- Fewer mistakes
- Reproducibility
- Professional credibility

*Results may vary. **Not medical advice.

principles

Caveat emptor

Before we get to do's & don'ts:

> Break any of these rules
> sooner than ~~say~~ *do* anything outright barbarous.
>
> *George Orwell, ''Politics and the English Language''*

Don't take this as gospel

- No claim of optimality or universality

Main takeaway: think about workflow

- Adopt what you like
- Adapt to your project
- Ditch the rest

Rule #0

Cardinal rule: “real” work happens in scripts

- Work interactively to play with data
- Work interactively to tinker with code
- But that’s about it

Store “official” data processing in code files

- Codify everything you can, even “one-off” tasks
- Coding creates a paper trail; “one-offs” often repeated
- See Gentzkow & Shapiro for horror stories

I am likely preaching to the choir, and I won’t belabor this point

Desiderata

1. Make data work reproducible & system-independent.
2. Make project folders carefully organized & self-contained.
3. Use consistent naming conventions & coding style.
4. All else equal: make codebase maximally lean.
5. All else equal: make codebase maximally fast.
6. Occupy no more disk space than needed.
7. Write readable code.
8. Document more than you think necessary.
9. Make project executable in one fell click.
   Corollary: no circular dependencies between files.
10. Make it obvious which input file made which output.

Mid-project rules of thumb

1. Code as if the end [of the project] is near.
2. Code as if your advisor is watching.
3. Optimize, streamline, & recycle code.
   Gentzkow & Shapiro: "profile slow code relentlessly".
4. Declutter unsentimentally, but not overzealously.
5. Keep all files in good working order.
6. Minimize the use of intermediate files.
7. Update, archive, or delete deprecated files.
8. Rethink workflow as projects evolve.
9. If collaborating: open communication & uniform conventions.
10. Don't neglect cosmetics: groom daily.

project folders

One project ≡ one folder

Give each project its own hermetically sealed project folder

- Avoid cross-project spillovers
- Control permissions on a project-by-project basis
- Reduce cost of making replication packages

Exception: incubating new projects within old ones

- Use case: projects relying on similar data/codebase
- Okay to do temporarily while probing viability
- If new project sticks, give it a proper home

But usually better to seed a new folder with old files

- See discussion of `/dta/trf` folder below

Organizing the project folder

Think carefully about folder organization

- Role of each sub[sub. . .]folder
- Folder- & file-naming conventions

Ask yourself:

- Will it scale up nicely?
- Will it port well across projects?
- Will coauthors buy in?

Best to organize files by *function*

A battle-tested file structure

/timeuse a brief, informative project name

Five essential subfolders:

/do	Stata .do files, Python .py files
/dta	Stata .dta files, other data files
/est	Stata .ster files, plotting points
/log	Stata .log files, other run logs
/out	tables, figures, & other output

You may want others:

/adm	grants, IRBs	/lit	related literature
/arc	archived files	/tex	drafts & slides
/doc	documentation	/trash	files to be deleted
/lib	software libraries	/xw	crosswalks

Within /do: two meta-files

`#settings.do` configures project-wide settings

- `global projdir ``/{`*`path`*`}/timeuse''`
- set other global macros
- set the `adopath`
- set the `scheme` & other graphics settings
- general-purpose `program` statements

`#master.do` executes files in correct order

- `do ``./#settings.do''`
- `do ``$projdir/do/{`*`subpath1`*`}/{`*`file1`*`}.do''`
- `do ``$projdir/do/{`*`subpath2`*`}/{`*`file2`*`}.do''`

Within /do: five subfolders

/ado	programs & settings
– /custom	user-written programs
– /import	imported programs (e.g., ssc)
– /fonts	custom fonts (e.g., CMU Serif)
– /scheme	custom graphics
/build	prepare data
– /fetch	acquire data (e.g., web-scraping)
– /prepare	map raw data into clean data
– /sample	construct estimation samples
/check	probe & understand data
/learn	early-stage analyses (internal consumption)
/share	later-stage analyses (external consumption)

Digression: nomenclature

Notice folder-naming conventions:

- Uniformly lowercase
- Similar in length: `/do`, `/dta`, `/est`, `/log`, `/out`
- Parallel structure: `/build`, `/check`, `/learn`, `/share`
- Special prefix `#` for “meta” files

What I’m going for here:

- Brief, memorable, easy to navigate
- ABC ordering $\iff$ logical progression
- Generic enough for all my projects

Largely a matter of taste—just be consistent

Within `/dta`: three types of data files

`/src` raw data files straight from source

- Subfolder for each dataset
- Leave completely unmodified
- Readme w/provenance (how/when acquired)
- If big: store as `.zip`, `unzip` in cleaning code

`/cln` clean data files created within this project*

- Distinguish cleaned files (all obs.) from samples (subsets)
- Sort by primary key, use efficient storage types
- Label every variable

`/trf` clean files transferred from elsewhere

- e.g., shovel-ready SIPP extract I don't want to re-clean
- Shortcut to get ball rolling—use sparingly/temporarily

*NB: `/src` & `/cln` aren't nicely alphabetized—I've opted for clarity over consistency here.

Within `/est`, `/log`, `/out`

These subfolders echo the structure of `/do`

`/log` :

- Subfolders: `/#master`, `/build`, `/check`, `/learn`, `/share`
- `‘‘do/#master.do’’` $\Longrightarrow$ `/log/#master/`{*logs*}
- `‘‘do/share/foo.do’’` $\Longrightarrow$ `/log/share/foo/`{*logs*}
- Time-stamped logs: `‘‘foo:20191031:01:30:00.log’’`

`/est` & `/out` :

- Subfolders: `/check`, `/learn`, `/share` (don't need `/build`)
- Give each analysis file `foo.do` its own `/est/`{*path*}`/foo` & `/out/`{*path*}`/foo` subfolders
- `‘‘do/share/foo.do’’` $\Longrightarrow$ `/out/share/foo/{figs}’’`

Virtues of this approach

1. Crystal-clear where to find any file
   - Always important, all the more so if collaborating
   - File structure embodies "meta-documentation"
2. Can name output files whatever I want
   - `.do` files share an output folder $\Longrightarrow$ need careful names
   - `.do` files get own output folders $\Longrightarrow$ the world is your oyster
3. Easy to remove deprecated materials
   - To delete ``‘‘foo’’``, just delete `foo.do` & `/foo` folders
   - Could easily write an `.ado` to archive deprecated analyses
4. Time-stamped logs $\Longrightarrow$ "poor man's version control"
   - Each run generates its own log file
   - Older log files retain archival copies of old code

maintenance

Maintenance: dull & indispensable

Analysis is fun, maintenance is $_{z}z^{z}$

But it's essential

- Clutter obscures logical flow
- Deprecated files invite confusion & error
- Keep the project folder tidy + timely

Regular pruning yields tangible benefits

- Gains in computational efficiency
- Detection of outstanding bugs
- Recovery of storage space
- Fresh eyes on data decisions

Make sure project is consistently "one-click-executable"

Avoid redundancy

Redundancy is a recipe for trouble

- Multiple versions $\implies$ which one is up-to-date?
- Bigger codebase $\implies$ harder to read/maintain

Law of Mutants: identical code blocks diverge over time

Don't replicate—automate

- If you do it more than 2–3×, automate it
- Don't hard-code file paths or parameters—use macros
- Easier to read, easier to extend, harder to mess up

For simple tasks: loops & macros

For complex tasks: `program define`, `.ado` files

Decluttering

Avoid creating clutter in the first place

- Save intermediate files sparingly—`tempfile` usually better
- Store concatenated pdfs, not one-pagers

Do "spring cleaning" on a regular basis

- If a file is woefully unnecessary, delete it outright
- If a file is obsolete but worth keeping, archive it
- When in doubt: archiving $\succ$ deletion

Before deleting code, consider sticking it in the `/trash`

- Slated for deletion, can fish out if needed
- Periodically "take out the `/trash`"
- Multiple users: let each control a `/trash/{user}` folder

Life cycle of the project folder

Projects go through a characteristic life cycle:

- Early: understand data, probe viability
- Later: fine-tune analyses, create pub-quality figures

Project folder should evolve in tandem

- Early: first passes at data cleaning, validation, analysis
- Later: revisit data decisions, streamline codebase, delete vestigial code, & archive peripheral analyses

Integrate any `/trf` files into main codebase

- Incorporate raw data straight from source
- Re-clean the data from first principles

Major overhauls

Sometimes need to do a major overhaul

- Fundamental changes to data organization
- Different approach to data construction
- Often big reductions in code base & runtime

Inherent risk of screwing something up

- Rebuild non-functional $\Longrightarrow$ want to retrace steps
- Estimates change, not sure why

Precaution: archive a snapshot of the project folder

- Space-permitting, include data files
- In any event, include everything else
- Another example of “poor man’s version control”

Create archival copies whenever you submit a paper computing

Working on a remote server

Pros:	Cons:
faster (parallel) processing	learning curve
ample storage space	requires internet connection
share access to same system	requires use of multiple apps
powerful shell tools	vulnerable to system outages
multiple persistent sessions	won't have admin privileges
regular backups	hard to use GUI (but possible)

My experience: servers $\succ$ alternatives

I use LaTeX locally, but I do all my data work on servers

A little Linux is a dangerous thing

Most servers run Unix/Linux. Learn the basics!

- Navigation: `pwd`, `ls`, `cd`, `cp`, `mv`, `rm`, `clear`, `mkdir`, `rmdir`
- Permissions: `chmod`, `chown`, `chgrp`, `umask`, `getent group`
- Searches/substitutions: `find`, `grep`, `locate`, `sed`
- Processes/storage: `du`, `jobs`, `kill`, `ps`, `unzip`/`gunzip`, `tar`
- Useful utilities: `cat`, `echo`, `head`/`tail`, `less`, `nano`, `touch`
- Existential questions: `whoami`, `whatis man`

Powerful combos using pipes, I/O redirection, & `exec`

Example: create a sorted list of lines containing "California":

```
cat counties.txt | grep 'California' | sort > ca.txt
```

Example: search `/hay` for `.do` files containing "needle":

```
find ./hay -type f -name '*.do' -exec grep 'needle' {} \;
```

Shell games

In Stata: lines starting with `shell` or `!` invoke the Linux shell

1. Run Python, Matlab, or `.bash` scripts directly from Stata*

   ```
   shell python niftywebscraping.py
   ```

   Recent example: invoking `lightgbm` from within Stata `.do` file to integrate machine-learning step into Stata workflow

2. Run Linux command-line utilities—for example, here's code to concatenate a list of .pdf files into a single .pdf:

   ```
   !gs -q -sPAPERSIZE=letter -dNOPAUSE -dBATCH
   -sDEVICE=pdfwrite -sOutputFile=out.pdf `pdflist'
   ```

   This is the key step in my short program `combine_pdfs`, which I store in `#settings.do` so I can use it project-wide

*Stata 16 introduces a `python` command for integrating Python statements directly into `.do` files. Thanks to Colin Cameron for bringing this to my attention.

Game-changer: `tmux`

Problem: running long jobs

- Wi-fi/VPN connection briefly interrupted
- Have to pack up to change locations
- Back to square one ...

Solution: command-line utility `tmux`

- New session: `tmux new-session -s batman`*
- Work normally without fear of disruption, then log off
- Come back later, reload session: `tmux attach -t batman`
- Slow code done running, everything as I left it
- Can even have multiple concurrent sessions

Basic functionality is worth learning

*Use mnemonics, even arbitrary ones. I name my `tmux` sessions after Gotham residents.

Other essential software

Your favorite text editor

- My favorite: `Atom`
- Great user interface, high-quality Stata syntax highlighting, regular expressions, multiple cursors, lots of other functionality

A command-line SSH client

- Mac: built-in `Terminal` application; PC: `PuTTy`

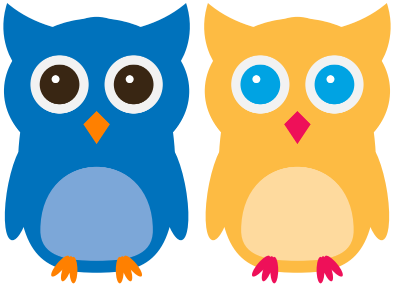
A drag-and-drop SFTP client

- `Cyberduck`: nice interface, crashes all the friggin' time
- Other options: `FileZilla`, `Transmit`, `WinSCP`

For papers & slides: a $\LaTeX$ distribution & editor

- I write my `.tex` files in `TeXShop` (can also use `Atom`)

collaboration



Getting along is hard to do

Collaborative work $\implies$ added challenges:

- Code works on my machine, breaks on yours
- I change a cleaning script, it affects your analysis
- I can't read your code, you can't read mine

Three viable approaches:

- Delegate model: a single user touches the code/data
- Partner model: 2+ users play equal roles
- Surgical team: "master surgeon" + "surgical assistants" (term comes from *The Mythical Man-Month*)

Partnership probably the hardest to pull off

- Advantages: fairness, fungibility, multiple pairs of eyes
- Requires open communication, version control

Version chaos vs. version control

Bad habit: `atus_v1.do`, `atus_v2.do`, `atus_FINAL_bpedits.do`

- We've all been there

Instead: use (real) version control

- Check files out, make changes, "commit" to shared repository
- System tracks changes from one commit to the next
- Easy to roll back changes (or just see what's changed)
- This is what actual programmers do

Many options on the market:

- Centralized repository: `RCS`, `CVS`, `subversion`
- Distributed repository: `git`, `Mercurial`

I'd go with `git` (and companion interfaces GitHub or GitLab)

Consider using a task manager

Hard to manage workflow in a collaborative project

- Zillions of email threads
- Some including everyone, others not
- Project documentation scattered all over

Instead: project-management software

- I like `Trello` (free version fine for small teams)
- Other options: `Asana`, `Evernote`

Style

Many ways to get the job done

- `gen byte emp = (empstat == 1 | empstat == 2)`
- `gen byte emp = (empstat==1 | empstat==2)`
- `gen byte emp = inlist(empstat, 1, 2)`

Best to adopt a shared style within each project

- Easier to read each other's code
- Easier to recycle code across scripts
- Learn coding tricks from each other

Even better: adopt common style across projects

- Easier to remember where stuff is, what stuff means
- Easier to recycle code across projects
- Easier for collaborators to understand your style

Next up

Part 2: Handling Data

- Coding practices
- Data cleaning
- Data validation
- Data analysis

Thurs. 11/14, same time/place

`fin`

Special thanks to LaTeX's `tikzlings` package
for making this little guy possible