Logbook for project

Jonathan I. Dingel   coauthors

Chicago Booth

Contents

Chapter A

Research infrastructure

This chapter is intended to introduce everyone to technology or tools that are integral to our workflow. It describes our research infrastructure in terms of three types of issues:

- **Code**: organize it, write it, run it, track it

- **Collaboration**: assigning tasks, sharing code, reviewing code, reporting results

- **Computing**: geeky details

We organize the project as a series of tasks, so our organization of code and data takes a task-based perspective. After writing code, we automate its execution via `make`. We track our code (and the rest of the project) using `Git`, a version control system. Collaboration occurs via Asana task assignments, GitHub/BitBucket code reviews, and logbook entries that share research designs and results.

Use a good text editor like SublimeText, Atom, or VSCode to write code, slides, and papers. Word processors aren't text editors. Your text editor should, at minimum, offer you syntax highlighting, tab autocomplete, and multiple selection.

Our approach assumes that you'll use Unix/Linux/MacOSX. Plain-text social science lives at the *nix command line. Gentzkow and Shapiro: "The command line is our means of implementing tools." Here are four intros to the Linux shell:

- https://ryanstutorials.net/linuxtutorial/

- http://swcarpentry.github.io/shell-novice/

- Grant McDermott's "Learning to love the shell" via his Data science for economists

- William E. Shotts, Jr's "Learning the Shell"

Getting started at the command line can be a little overwhelming, but it's well worth it. While you can use GUI apps to interact with most of our workflow (e.g., SourceTree for version control), automation of some key parts relies on shell scripts. See logbook entry A.9 for an assorted collection of tips.

Beyond *nix, the rest of the research workflow is language-agnostic: it applies to everything from Stata to Julia. In fact, the task-based approach naturally facilitates using different languages for different tasks.

I have five criteria in mind when evaluating a research workflow:

- Replicability: Can the research results be reproduced starting from the raw data?

- Portability: If I install a fresh copy of the project on a new computer, what are the startup costs before I can run the code?

- Modularity: Can a coauthor work on a task using the provided inputs without having to look upstream at the code that produced those inputs?

- Dependencies: In the event of a data update, how do you know which pieces of code need to be run (and in what order)?

- History: If results have changed, can I discern the relevant code changes and their authors?

After reading the rest of this chapter, you should be able to say how our workflow answers each of these questions.

A.1   Task-based organization of code and data

Eagleson's Law: Any code of your own that you haven't looked at for six or more months, might as well have been written by someone else.

A.1.1   A task-based approach

Many people create a project directory containing a distinct folders for each type of file in the project, like `raw`, `code`, and `data`. I no longer believe this is a helpful approach. Viewing a series of files in the `code` folder and a series of files in the `data` folder doesn't really make the relationships between them clear. One tries to list inputs and outputs at the top of each code file, but one inevitably fails to reliably update the metadata of each code file to accurately describe the inputs it relies upon and the outputs it produces. And in a team environment, one inevitably has to go back and partition the files into natural chunks of labor that can be divided among collaborators.

We take a task-based approach to organizing our research project. The key idea: the task is a quantum of workflow. Each task is a separate folder with three subfolders: `input`, `code`, and `output`. One task's input is another task's output. I was initially persuaded of the merits of this approach by Patrick Ball's principled data processing presentation. The key part of the talk starts at 15:50.

This approach has a number of advantages. One is that the project structure is self-documenting. You don't have to update the metadata at the top of the code because the input and output directories make the relationships obvious. In fact, these relationships between tasks can be automatically audited, as we show in the next section. Second, this structure makes the project modular. A team member can jump in and work on one task in isolation because she/he can take the inputs as given and focus on the task-specific code without having to navigate the rest of the project. Both of these advantages are also relevant when you return to a project after not looking at it for twelve months.

Some folks dislike the "quantum" nature of the task-based approach, because a research project involves many tasks and thus this approach generates many folders. While the optimality of a particular organizational approach is an empirical question, I have used both the "all code in one folder" approach and the task-based approach. I will only use the latter going forward.

In what follows, I describe how task flow works and provide a minimal example. For full-scale examples of this approach, see the replication packages posted at `https://github.com/jdingel`, such as DingelMiscioDavis.

A.1.2   Symbolic links

One task's input is another task's output. It would horrible if we had to move files from one task's `output` folder to another task's `input` folder. We don't. We automate the process using symbolic links.

Symbolic links are files that serve as aliases or pointers for other files. Processes like Stata, R, and Make follow the link to the target, so symbolic links allow your code to act as if one file is present in many different places. They're useful because a symbolic link in an `input` folder that refers to data in another task's `output` folder both makes the input source clear and means the input changes whenever the upstream output changes.

Symbolic links are created by the `ln -s` command. The syntax to create a link that points to a target is `ln -s targetpath linkpath`   The link points to the target. A few details:

- If `targetpath` is a file and `linkpath` is a folder, the symbolic link will inherit the target's filename and be placed within that folder.

- If `targetpath` is a folder, the link will point to (and serve as) a folder.

Here's an example of a sequential ("snake") task flow with three tasks.

```bash
#!/bin/bash

#Create example content
mkdir -p eg/task1/output/ eg/task2/input/ eg/task2/output/ eg/task3/input/
echo 'This is the first file' > eg/task1/output/result1.txt
echo 'This is the second file' > eg/task2/output/result2.txt

#Create symbolic links
ln -s ../../task1/output/result.txt eg/task2/input/result1.txt
ln -s ../../task2/output/result2.txt eg/task3/input/

#Report symbolic links
find eg -type l -ls
```

The result of that last command shows something like

```
lrwxr-xr-x 1 jdingel 30B Jun 2 10:14 eg/task3/input/result2.txt -> ../../task2/output/result2.txt
lrwxr-xr-x 1 jdingel 29B Jun 2 10:14 eg/task2/input/result1.txt -> ../../task1/output/result.txt
```

This directory listing lets you see how one task's input points to another task's output. If you want to see how a process like Stata or R will perceive this directory, use the `ls` command's `-L` option when listing the directory. For example, that lists the file size of the target, not the (trivial) size of the link.

The input-output links make the project's structure self-documenting. In fact, one can write a script to use the list of symbolic links to summarize the project's structure.[1] Commands like `sed` or `awk` can prune away the irrelevant `ls` output, leaving a list of symbolic links. That list is a directed graph that can be plotted using graphviz.[2]

Following from our previous example:

---

[1]I owe this strategy to Patrick Ball's principled data processing presentation. See 28:00-35:20 of his talk.

[2]The `dot` command requires graphviz. This is likely installed already on your Linux system; on OS X, type `brew install graphviz` if you have use the Homebrew package manager.

```bash
#!/bin/bash
#Graph the task input-output relationships based on symbolic links

mkdir -p eg/grapher/output/

echo -e 'digraph G {' > eg/grapher/output/graphviz.txt #Start graph
find eg -type l -ls | awk '{print $13 " -> " $11}' | #List symbolic links
sed 's/\.\.\///g' | sed 's/eg\///g' | #Drop relative paths
sed 's/\/\(input\)\/[a-zA-Z0-9_\.]*//g' | #Retain only task names; drop filenames
sed 's/\/\(output\)\/[a-zA-Z0-9_\.]*//g' >> eg/grapher/output/graphviz.txt #Ditto; write to file
echo '}' >> eg/grapher/output/graphviz.txt #Close graph

dot -Grankdir=LR -Tpng eg/grapher/output/graphviz.txt -o eg/grapher/output/task_flow.png
```

This produces a directed graph (and your project best be acyclic).

While in this case the flow chart is trivial, a real project will have a much more complicated structure that you can't remember. See our consumption segregation paper for an example.

A.1.3 All output is produced by tasks

In reality, every number reported in your paper is produced by code. Your workflow (and your replication package) therefore should produce every such number as a file in an output folder. A summary statistic that appears in a sentence rather than in a table is still a code-produced number. If you hardcode some number in your paper's LATEX file, I bet you'll forget to update it at least once (it's produced by code + human transcription and the latter is less reliable.) I've certainly produced drafts where the same number takes different values in a paragraph and in a table because code produce the latter but not the former.

All output is produced by tasks. Our task workflow produces the final paper. See Patrick Ball at 40:27.

A.2 Writing code

This section is for us to agree upon best practices in coding. It's light at the moment.

A.2.1 Language-free coding principles

Other people's principles

- See chapter 6, chapter 7, and appendix A of Gentzkow and Shapiro's Code and Data for the Social Sciences: A Practitioner's Guide.

- QuantEcon's "A Digression on Style and Naming". Read their comments on the following:

    - Code is read many more times than it is written

- Write code to be read in the future, not today
- Excess comments in code can make code harder to read
- Maintain the correspondence between the whiteboard math and the code

Cleaning data

I agree with Gentzkow and Shapiro that you should keep data in normalized files as long as possible.

> First, store your raw data in normalized files that preserve the information in the original data source and follows the rules above. Don't worry about how you plan to use the data. Rather, imagine that you are preparing the data for release to a broad group of users with differing needs. Do this because you, yourself, are likely to want to use the data in ways you do not currently anticipate.

Do not hardcode values

While I haven't fully embraced the functional programming paradigm, I do think that it is a mistake to use global variables or to hardcode values within functions. See Patrick Ball's argument and page 28 of Gentzkow and Shapiro's Code and Data.

For example, sample selection decisions are arguments. Constants do not belong inside functions. Name them once, impose the value/criterion once, and then refer to that repeatedly within your function. YAML is a good format for constants.

A.2.2   Sharing code across machines

We almost always want to run code on multiple machines. Absolute file paths are problematic for shared code, because `/Users/jdingel/` is not a valid path for almost every computer on earth. Here are two possible means of making shared code run on many machines:

1. Set absolute file paths with conditional statements.

2. Use only relative file paths.

I strongly prefer the latter.

Conditional absolute file paths

Conditional absolute file paths exploit environmental variables to identify the machine and then set absolute file paths based on that. For example, in Linux there is an environmental variable called `$USER` at the shell command line. Similarly, elements of Stata's c-class, such as '`c(pwd)`' and '`c(hostname)`', can be used to identify a machine and then run user-specific programs that set global variable to user-specific values.[3] Here is a brief example in Stata.

```
if "`c(os)'"=="Unix" & regexm("`c(hostname)'","midway")==1 {
    display "Running on RCC Midway server"
    global root "/project/xxx/xxxyyy/xyz_repo"
}
```

---

[3]Type `help creturn` in Stata for exhaustive documentation of the available arguments.

Relative file paths

Relative file paths work on any machine because they are relative. In Jonathan's opinion, this is generally preferable. It means that code can be cloned onto a new machine by any user and run immediately without having to write new user-specific code. See https://github.com/jdingel/DavisDingelMonrasMorales for an example of a repository that only uses relative paths.

GitHub agrees: "Relative links are easier for users who clone your repository. Absolute links may not work in clones of your repository - we recommend using relative links to refer to other files within your repository."

Package management

Package dependencies should be identified by `requirements.txt` (Python), `Manifest.toml` (Julia), and so forth. Stata lacks a built-in package-dependency management tool. I just write a do file that installs all the packages, but there is no official requirements/manifest file. My understanding is that R doesn't do package management properly natively. I think you want something like "packrat", but I have never used this myself.

These requirements should be committed to the repo so that anyone cloning the repo on a new machine has everything they need defined. This is also AEA journal policy: "packages/modules/etc. ... provide a setup program to install these (not manual instructions)."

I tend to write package-installation scripts in a distinct task that covers all packages used in the project. The best reason to concentrate everything in one task is that oftentimes package installation requires internet access: on RCC, login nodes can touch the external internet, while compute nodes cannot.

A.2.3 Limit lines to at most 100 characters

Files should have lines no longer than 100 characters in length. This is true both for code and TeX. The commands `diff` and `git diff` report lines that differ between files/commits. Really long lines of code therefore can be a nightmare, since one may have to scroll quite far to spot a change. This is really painful if a line is actually a full paragraph of text. Writing each new sentence on a separate line makes it much easier to view changesets.

See Directives for using LaTeX with version control systems

- Do not change line breaks without good reason.

- Turn off automatic line wrapping of your LaTeX editor.

- Start each new sentence in a new line.

- Split long sentences into several lines so that each line has at most about 80 characters.

- Verify that your code can be compiled flawlessly before committing your modifications to the repository.

- Use diff to critically review your modifications before committing them to the repository.

- Add a meaningful and descriptive comment when committing your modifications to the repository.

A.3   Running code with Make

Projects consist of tasks that depend on each other. If an upstream task changes, downstream tasks ought to be re-run using that new input. This is easy to do by hand when your project has a small number of tasks. This is hard to do when the tasks are computationally intensive and the input-output graph is complicated. Thankfully, computer programmers addressed these difficulties long ago with a Unix utility called Make. This is a popular form of the broader concept of build automation.

Makefiles are machine-readable documentation that make your workflow reproducible. They follow naturally from our task-based approach. Downstream tasks depend on upstream tasks. Using `make` requires you to specify those input-output relationship (dependencies). When generated files are missing, or when files they depend on have changed, needed files are re-made using a sequence of commands you specify. The commands are language-agnostic: if you can type it at the shell prompt, `make` can execute it.

A.3.1   Learn Make

The first fifteen minutes of Lecture 8 in MIT's "Missing Semester" CS class are a great introduction to Make. Start by watching that.

Here are four written introductions to `make`, listed in the order that I suggest reading them:

- Mike Bostock: Why Use Make

- Karl Broman: minimal make

- Kieran Healy: Pull It Together (The Plain Person's Guide to Plain Text Social Science)

- Zachary M. Jones: GNU Make for Reproducible Data Analysis

A.3.2   Some simple examples of Makefiles

The `Makefile` in the `logbook` folder is a simple example. It compiles this PDF if `logbook.tex` or one of the elements of `$(logbookentries)` is newer than `logbook.pdf`.

Here is an example of what a Makefile in an "initialdata" task folder might do. The following code is an excerpt from a Makefile that downloads Census data and produces a CSV by unzipping the downloaded file.

```
../output/mi_od_main_JT01_2010.csv: ../output/mi_od_main_JT01_2010.csv.gz
    gunzip -c $< > $@
../output/mi_od_main_JT01_2010.csv.gz: | ../output
    wget "https://lehd.ces.census.gov/data/lodes/LODES7/mi/od/$(@F)" -O ../output/$(@F)
../output:
    mkdir $@
```

The first rule concerns the target `../output/mi_od_main_JT01_2010.csv`. The target lists a compressed file as its one pre-requisite. If the target does not exist or if the target is older than (any of) the pre-requisite(s) listed after the colon, then the recipe for that target is executed. The recipe for this target is simple: use `gunzip` to decompress the pre-requisite and write it to the target.

Let's unpack that recipe in detail. gunzip takes a list of files on its command line and replaces each file whose name ends with .gz, -gz, .z, -z, _z or .Z with an uncompressed file without the original extension. `-c` is an option that writes output on standard output and keeps original files unchanged. The somewhat cryptic `$<`

and `$@` are automatic variables in Make. `$<` is the first pre-requisite (`../output/mi_od_main_JT01_2010.csv.gz`) and `$@` is the target (`../output/mi_od_main_JT01_2010.csv`). `>` redirects output to a file, overwriting the file. (By the way, `>>` appends redirected output to the end of a file (rather than overwriting).)

The second rule defines the compressed file from Census as the target. Its only pre-requisite is that the `../output` folder exist (an "order-only prerequisite" indicated by the pipe `|`). If `../output/` does not exist, the Makefile must provide a recipe to create it (see lines 5-6). The recipe for the compressed Census file is a `wget` command. `wget` is a free utility for non-interactive download of files from the web and `-O` is an option that concatenate all documents together and writes to the specified output file. The automatic variable `$(@F)` equals `mi_od_main_JT01_2010.csv.gz`, the file-within-directory part of the file name of the target. It is used in specify the web address of the Census file.

If you type `make ../output/mi_od_main_JT01_2010.csv` at the command line, `make` will try to make the CSV target and see the CSV.GZ file as a pre-requisite. If the pre-requisite has not been already downloaded, `make` will create the `../output` directory if needed, then run the recipe to produce that CSV.GZ file (the `wget` command), and then execute the recipe to produce the target.

After obtaining the initial data, we will use these data in (multiple) downstream tasks. Here is an excerpt from the Makefile in a task called `LODES_datapreparation`.

```
../output/lodes_DetroitUA_2010.dta: DetroitUA_tract.do programs.do ../input/mi_od_main_JT01_2010.
    csv | ../output
  $(STATA) $<
../input/mi_od_main_JT01_2010.csv: ../../LODES_downloaddata/output/mi_od_main_JT01_2010.csv | ../
    input
  ln -s $< $@
../../%:
  $(MAKE) -C $(subst output/,code/,$(dir $@)) ../output/$(notdir $@)
```

This target `../output/lodes_DetroitUA_2010.dta` has four pre-requisites: `DetroitUA_tract.do`, `programs.do`, the CSV file that we described downloading above, and the `output` folder. The recipe for this target is to run the Stata script `DetroitUA_tract.do` (referenced by `$<`). `$(STATA)` is an alias for how we invoke Stata (modified to return a proper exit status and work with SLURM, ignore those details for now). In lines 3-4, `ln -s` creates a symbolic link called `../input/mi_od_main_JT01_2010.csv` that points to the upstream file `../../LODES_downloaddata/output/mi_od_main_JT01_2010.csv`. The final two lines define a generic recipe for producing upstream content. The `%` is a wildcard, so this recipe will be used for *any* target whose path starts with `../../`. `../../LODES_downloaddata/output/mi_od_main_JT01_2010.csv` is an example of such a target. If this file isn't already available, the recipe runs `make ../output/<target name>` in the upstream folder `../../LODES_downloaddata/code`. We achieve this by using Make's `$(dir)` and `$(notdir)` functions that split a filepath (e.g., `$@`) into the directory and filename components. Notice that this recipe invokes a different Makefile (the one assumed to be in the `LODES_downloaddata` task) in order to produce the target, and it assumes the Makefile in `LODES_downloaddata` task has a rule to build `../output/<target name>`.

A.3.3   Notes on writing Makefiles

An important reminder: Each line in the recipe must start with a tab. See Recipe Syntax.

A few notes on Makefile style.

- When you run `make` without specifying a target, its default goal is to build the first target listed in

the file. Convention is to define `all` as the first target of the Makefile and list all desired targets as pre-requisites of `all`.

- It is helpful to define variables containing long lists of files, such as `INPUTS= file1 file2 file3` so that you can summarize dependencies simply by writing `$(INPUTS)` and define targets simply by writing `all:    $(INPUTS) $(OUTPUTS)`. However, it is only sensible to write a target-dependency relationship as `$(OUTPUTS): $(INPUTS)` if there is one recipe that produces all those outputs jointly.

- Writing separate recipes for separate targets is advantageous because Make will return more informative errors by telling you which recipe failed.

- Automatic variables take values computed when the rule is executed based on the target and pre-requisites of the rule. Commonly used automatic variables include the file name of the target of the rule `$@`, the name of the first pre-requisite `$<`, and the file-within-directory part of the file name of the target `$(@F)`.

- In addition to normal prerequisites, it might be helpful to define order-only prerequisites. Order-only prerequisites must exist in order for the target to be produced, but the target does not need to be newer than the order-only prerequisites. An example of prerequisites that should be regarded as order-only is folders, since "the timestamps on directories change whenever a file is added, removed, or renamed, we certainly don't want to rebuild all the targets whenever the directory's timestamp changes" (GNU). Order-only prerequisites are defined by listing them after the pipe symbol: `output: normal prerequisites | order-only prerequisites`.

- Make provides several built-in functions for transforming text. Commonly used functions include `foreach` for performing text substitution by looping and `addprefix` for prepending prefix in front of individual names.

- In addition to executing recipes in the shell, you can define Make variables using shell commands by invoking Make's `shell` command. The simplest example would `$(shell echo {1..10})`, which would return the integers from 1 to 10 via Bash's brace expansion.

Grouped targets. Historically, Make (like most of neoclassical economics) has assumed "no joint production". If two targets are produced by the same recipe, Make assumes it needs to run the recipe twice. Consider this trivial shell script (`joint_producer.sh`)

```
echo 'done' > output1.txt
echo 'done' > output2.txt
```

accompanied by this trivial Makefile:

```
all: output1.txt output2.txt
output1.txt output2.txt:
	bash joint_producer.sh
```

Running the script once will produce both outputs. But what does Make think you need to do? Run it twice. In practice, after the first run, Make will see that the second target has been produced. But if you run Make with parallel threads, it will run the script twice simultaneously.

```
jdingel$ make -n
bash joint_producer.sh
bash joint_producer.sh
jdingel$ make
bash joint_producer.sh
jdingel$ rm output?.txt
jdingel$ make -j 2
bash joint_producer.sh
bash joint_producer.sh
```

Make 4.3 (January 2020 release) introduced "grouped targets", thereby allowing joint production. Make 4.3 would not make the mistake of submitting two copies of the same job when running parallel threads if you define the targets as grouped by `output1.txt output2.txt &:`, where the ampersand indicates grouped targets. I have typically refrained from using grouped targets because one cannot assume Make 4.3 will be available on most boxes (e.g., Midway2 is running GNU Make 3.82).

A.3.4   Reduce redundancies using pattern rules

Minimizing redundancies lowers future maintenance costs and makes it easier to scale up. Fortunately, Makefile has pattern rules that allow us to combine repeated or similar rules. The basic idea is to use the wildcard `%` to match the varying parts. Consider an example similar to the one in Section A.3.2, but now we are also downloading data for 2011:

```
all: ../output/mi_od_main_JT01_2010.csv ../output/mi_od_main_JT01_2011.csv
../output/mi_od_main_JT01_2010.csv: ../output/mi_od_main_JT01_2010.csv.gz
    gunzip -c $< > $@
../output/mi_od_main_JT01_2011.csv: ../output/mi_od_main_JT01_2011.csv.gz
    gunzip -c $< > $@
../output/mi_od_main_JT01_2010.csv.gz: | ../output
    wget "https://lehd.ces.census.gov/data/lodes/LODES7/mi/od/$(@F)" -O ../output/$(@F)
../output/mi_od_main_JT01_2011.csv.gz: | ../output
    wget "https://lehd.ces.census.gov/data/lodes/LODES7/mi/od/$(@F)" -O ../output/$(@F)
../output:
    mkdir $@
```

Notice that `gunzip -c $< > $@` and `wget "https://lehd.ces.census.gov/data/lodes/LODES7/mi/od/$(@F)" -O ../output/$(@F)` appeared twice in the example. Moreover, the targets and prerequisites for the 2010 and 2011 data only differ by the year in the filenames. Hence, we can consolidate these nearly redundant recipes by replacing the year in the targets and prerequisites with `%`. The consolidated, pattern-matching version of the example above is

```
all: ../output/mi_od_main_JT01_2010.csv ../output/mi_od_main_JT01_2011.csv
../output/mi_od_main_JT01_%.csv: ../output/mi_od_main_JT01_%.csv.gz
    gunzip -c $< > $@
../output/mi_od_main_JT01_%.csv.gz: | ../output
    wget "https://lehd.ces.census.gov/data/lodes/LODES7/mi/od/$(@F)" -O ../output/$(@F)
```

```
../output:
    mkdir $@
```

Note: the `%` in the prerequisite(s) will automatically be replaced with the the matched string from the `%` in the target. Pattern rules also allow for multiple targets, but all targets must be patterns (have `%` in the names). Furthermore, when two targets from the same rule have the same matched string, the rule will only be executed once; please see this stackoverflow post for more details. There are other ways of reducing redundancies in Makefiles, like implicit rules, double-colon rules, and secondary expansion.

Unfortunately, using pattern rules can produce opaque error messages. A target matches a pattern rule when the target pattern matches the target to be built and all the prerequisites exist or can be built. Therefore, if one of the prerequisites does not exist and there is no rule to build it, rather than seeing `No rule to make target '<prerequisite name>'`, you will see `No rule to make target '<target name>'`. The absence of the prerequisite (or a rule for the prerequisite) is reported as the absence of a rule for the target because it didn't match the pattern rule. Suppose we have the following Makefile with an explicit rule:

```
all: foo.csv
foo.csv: foo.dta
```

If `foo.dta` does not exist, when running `make`, we will see `No rule to make target 'foo.dta'`. By contrast, suppose we have the following Makefile using a pattern rule:

```
all: foo.csv
%.csv: %.dta
```

If `foo.dta` does not exist, when running `make`, we will see `No rule to make target 'foo.csv'`. This might not seem like a huge problem in our simple example, but, when you have a long makefile with multiple pattern rules chained together, it can be hard to find the missing link. You can guard against this by specifying the targets that can match the pattern:

```
all: foo.csv
foo.csv: %.csv: %.dta
```

This approach returns the error message `No rule to make target 'foo.dta', needed by 'foo.csv'`. Of course, if you want to build a file ending in `.csv` that is not `foo.csv`, this pattern rule will not match it. You would need to add that filename to the list preceding the colon before `%.csv`.

A.3.5   Running Make

When invoking Make, the following options are often relevant:

- `-n`: Prints the recipes that would be executed without executing them. Great to preview what jobs you're going to launch by running Make.

- `-j`: this option allows parallelization of jobs, i.e., contemporaneous execution of several recipes. Normally, make executes one recipe at a time, but `-j` enables simultaneous execution. `-j` is followed by an integer specifying the number of parallel processes. The default number of jobs run is 1 (serial processing).

- **--debug**: this option is useful to know how make analyzes your dependency graph. This option provides the most detailed information available other than running a debugger. We make use of 2 suboptions of **--debug**: "basic" and "verbose". When the "basic" suboption is used, make will print each target that is found out-of-date and the status of the update action. "verbose" replicates the "basic" suboption and includes additional information about files that were parsed, prerequisites that did not need to be rebuilt, etc.

A note on running Makefiles in a cluster computing environment that has a job scheduler:

- A problem with batch jobs is that the shell sees the submission command as completed upon job submission any output files are produced. Failing to see the output, Make will repeatedly submit the job.

- You need the job submission command to not exit back to the shell until the job completes, so that the target will be produced before Make evaluates that recipe's success or failure.

- With PBSPro's `qsub` at Census RDC, use `qsub -W block=true`.

- With SLURM's `sbatch` at Chicago Midway, use `sbatch -W` or `srun`[4].

- Read http://wresch.github.io/2012/11/01/qsub-submit-jobs-from-makefile.html for more discussion of `qsub` and the `sbatch` manual for discussion of `-W` or `--wait`.

- I am not sure about the best practice for the Torque scheduler.

A.4 Tracking code with Git

This section introduces the idea of version control and applies it to a single-machine user. (This is sufficient for applying version control to confidential-data settings where you only work on one server.) A later section in the **Collaboration** block addresses using version control when sharing across machines.

A.4.1 Introduction to version control

Version control is the management of changes to code and documents. It should be applied to all textual content in a research project. Version control tools allow us to

- Keep a complete history of our project without any additional effort or files

- Hit "undo" (at arbitrarily large scale) at will

- Collaboration: Edit code simultaneously without creating disastrous conflicts

- Collaboration: Easily track coauthors' contributions and changes

All real programmers use version control. The dominant version control software is Git.

To understand version control, please read the following short introductions:

---

[4]Please refer to this stackoverflow post for details on the differences between `sbatch` and `srun`. According to the SLURM documentation, "`srun` is used to submit a job for execution in real time", while "`sbatch` is used to submit a job script for later execution." In essence, `srun` is interactive and blocking (output are in your terminal ,and you can't do anything in the terminal until `srun` is finished), while `sbatch` is batch processing and non-blocking (outputs are redirected to other files and, without the `-W` flag, you can run other command while `sbatch` is running).

- Gentzkow & Shapiro - "Chapter 3: Version Control"

- Michael Stepner - "git vs. Dropbox from a researcher's perspective"

A.4.2   Which files should be kept in version control?

Version control is appropriate for any plaintext file. Binaries (like PDFs) are less suitable, since you can't examine their contents.

Obviously all code should be in version control. What about writing?

- LaTeX is perfect; it's plaintext.

- LyX is suitable for version control, because `.lyx` files are plaintext.

- Jonathan prefers to err on the side of documenting and committing too much. There's little downside.

A file called `.gitignore` identifies paths that should not be tracked by Git.

- We don't commit data. I therefore typically put `tasks/*/output` in `.gitignore` so that the repository doesn't track output files.

- Git sees symbolic links as their targets. I therefore put `tasks/*/input` in `.gitignore` so that the repository doesn't track data files.

- To produce logbook content (and the paper), you may want to commit tables and figures from these output folders. To do so, you can always type, for example, `git add table1.tex -f`, where the `-f` force option causes the file to be tracked despite the gitignore rule.

A.4.3   Version control in a single-machine environment

To productively use git in a single-machine environment, you only need to know a half-dozen commands.

- `git init` initializes a repository in the current working directory

- `git status` tells you what files have been modified since you last committed

- `git diff` shows you those modifications (and can also show you the modifications between any two commits)

- `git add` stages a file

- `git commit` commits a set of changes

- `git log` lists the history of commits

See "Git Basics" on these.

A couple more resources:

- For command-line Git, see Atlassian's Git cheatsheet.

- You may also want to glance at Jess Johnson - How To Use Source Control Effectively.

- If your University subscribes to Lynda, they offer a number of Git courses (with which I have no experience).

Some notes on command-line Git by Ningyin Xu, a former RA, follow.

Setting up a repository

If you want to create a brand-new repo, you'll use the `git init` command.

```
cd /path/to/your/existing/project
git init
```

Saving changes to the repository

Now that we have a repository, we can edit its contents and then commit changes to it. The basic commands are:

```
cd /path/to/repo
# make some changes to a file named "test.txt"
git add test.txt
git commit -m "edited test.txt"
```

This example introduced two additional git commands: `add` and `commit`. The `git add` command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, `git add` doesn't really affect the repository in any significant way — changes are not actually recorded until you run `git commit`. The `git commit` command commits the staged snapshot to the project history. Committed snapshots can be thought of as "saved" versions of a project, Git will never change them unless you explicity ask it to. In conjunction with these commands, you'll also need `git status` to view the state of the working directory and the staging area. In general cases, you could do "git status" before "git add", and Git will tell you what files you have made changes and how you could deal with them.

Another common use case for `git add` is the `--all` option. Executing `git add --all` will take any changed and untracked files in the repo and add them to the repo and update the repo's working tree. This may include hidden files. You should list files that you do not want to be part of the repository in the file called `.gitignore` ([documentation](#)).

Undoing changes

The `git checkout` command is mostly used for its three functions: checking out files, checking out commits, and checking out branches. Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched. Checking out branches will be discussed in the **Collaboration** section.

```
git checkout <commit> <file>
# Check out a previous version of a file. This turns the <file> that resides
# in the working directory into an exact copy of the one from <commit> and
# adds it to the staging area.

git checkout <commit>
# Update all files in the working directory to match the specified commit.
```

```
# But this is a read-only operation, the ``current'' state of your project
# remains untouched in the master branch.

git checkout master
# This lets you to get back to the "current" state (under the condition you
# don't have more than one branches).
```

In some case, you might want to undo your last commit. To do that, use `git reset --soft HEAD~1`. This command will remove the last commit from the current branch and keep previously commited files in the staged area. For a more detailed description, see Atlassian guide.

A.5   Task assignments on Asana

Asana is a project management tool used to assign tasks, impose deadlines, and track progress. Email is not a task management tool. To get started, sign up at https://asana.com.

Other potential project management tools would be the issue trackers on BitBucket and GitHub. These support assigning tasks and accompanying discussion, but they don't make it quite as easy to assign deadlines to issues (but try GitHub's milestones), don't allow hierarchical issues (i.e., subtasks, but try checklists within GitHub issues), and limit the types of supported file attachments. You might try to use Slack as a project management tool, but I have not enjoyed it. It is better for free-flowing conversation than keeping a to-do list.

Asana is organized into tasks, projects, and teams. Tasks are the basic unit of work in Asana, projects are lists of tasks, and teams are groups of people that work together on tasks and projects.
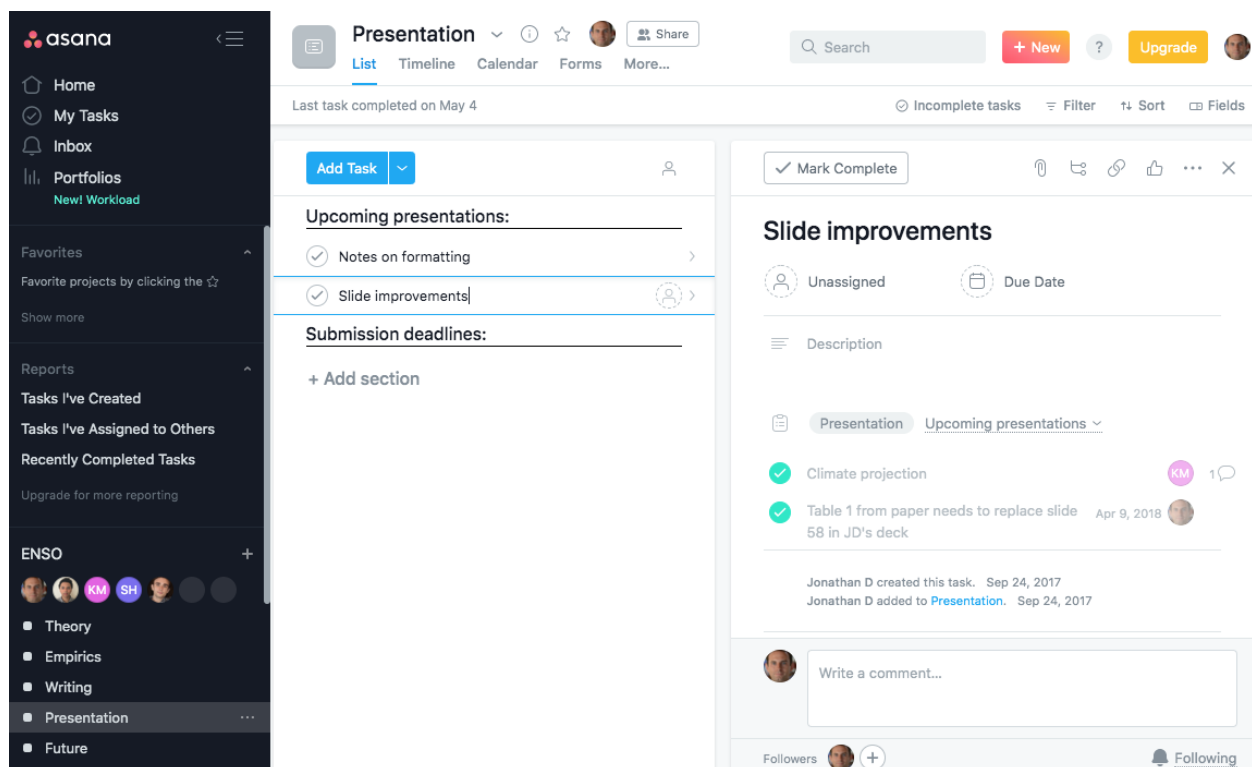
In this screenshot, you see a list of projects in the left column of the ENSO workspace. We're on the "Presentation" project right now, looking at a list of tasks. One task is "Slide improvements", which is not assigned to anyone and lacks a due date. It contains two subtasks: one assigned to Kyle Meng without a due date and one assigned to Jonathan due April 9. Both subtasks have been completed, so they are checked off and faded out.

To better organize projects, you can divide tasks into "sections". To create a section, just create a new task within a project and end the task name with a colon.
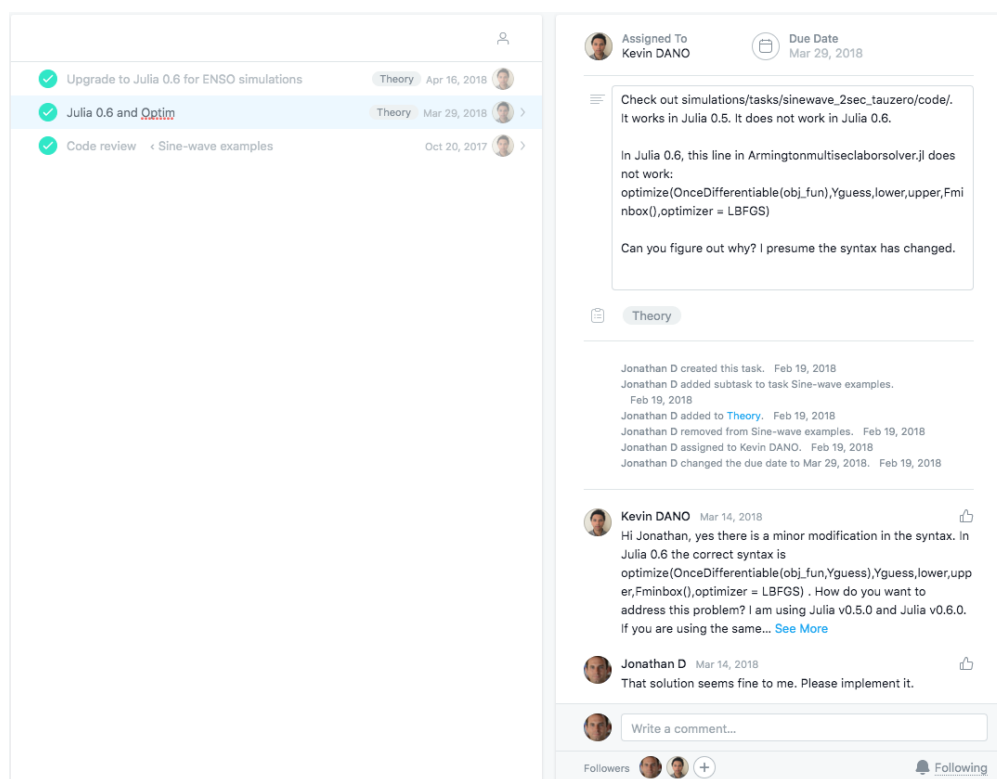
There is a comment thread associated with each task. Asana conveniently records discussions that are categorized by task and searchable. Kyle likely never saw the interaction in this screenshot because he was not a "follower" of this task. That's good, because he was not interested in this task/conversation. To make sure someone receives "notifications" related to a task in their inbox, add them as a follower. The task assignee is automatically a follower (hence Kevin is following this task). Commenters are automatically added as followers (hence Jonathan is also a follower). You can grab someone's attention by using "@" and their name in a comment.

Jonathan loves the "Inbox" tab. Following those notifications lets him track others' progress on tasks and alerts him to comments. The "My Tasks" tab alerts him each day to tasks that are coming due soon. Jonathan recommends (1) turning off Asana email notifications (Jonathan doesn't like email for workflow, though some of his collaborators do) and (2) installing the Asana app on your smartphone (great way to see research progress while trapped in seminars!).

To learn more about using Asana, see these introductory lessons.

A.6 Sharing code via Git

Git's true powers lie in the fact it is a *distributed* version-control system for tracking changes. Every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server.

In projects involving multiple machines, collaborating on a repository involves distinguishing between the "local" and "remote" repos. It involves (at least) three new git commands: `git clone`, `git pull`, and `git push`.

Take a look at Figures 3 and 4 in Blischak JD, Davenport ER, Wilson G (2016) "A Quick Introduction to Version Control with Git and GitHub." *PLoS Comput Biol* 12(1): e1004668.

Try Roger Dudler - git - the simple guide. After reading the material below, you might also circle back to read the "Git tutorial" from UChicago's Open Source Macroeconomics Laboratory.

A.6.1 Online hosting services and a GUI client

Don't confuse Git and GitHub. Git is free, open-source software. GitHub, BitBucket, and GitLab are online repository hosting services. They also provide a number of user-friendly GUI tools for code review and commenting that complement Git.

While you can run Git from the command line, you might find it a little easier to have a GUI as you learn Git. I recommend using SourceTree, a free Git GUI for OS X, although it's the only Git GUI I've used.[5] SourceTree is by Atlassian, the folks also behind BitBucket. You can use the app to pull, commit, push, diff, etc. If you're on Windows, perhaps try GitKraken.

A.6.2 Clone the remote repo

If you want to create a local development clone of an existing repo, give the `git clone` command a repository URL to create a copy of the remote repo. Then the content is on your machine as a local repo. For example, to clone the (public) remote repo "lights_to_cities"

```
cd path/you/want/to/put/the/repo
git clone https://github.com/jdingel/lights_to_cities.git
```

Here's another example, this time cloning a private repository hosted by BitBucket. Log in to Bitbucket.org and go to the `beacensus` repository page: https://bitbucket.org/jdingel/beacensus. On the repository page, click the ⬆ button and you'll be prompted to clone the repository to your own machine using the SourceTree application.

---

[5]Some of my coauthors have had poor experiences using SourceTree on their Windows machines.
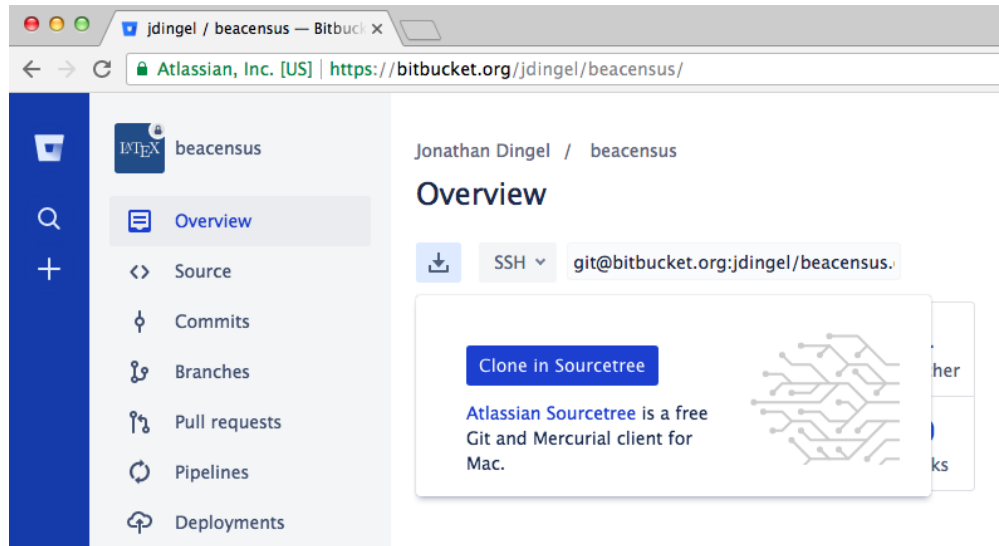
Name the folder on your machine, and a copy of all the files in the remote repository will be cloned to your local repository.

It's also possible to clone remote repositories to which you have access simply by opening the SourceTree app and following these instructions.

Pull and push

Two commands, **git pull** and **git push**, allow us to sync the local repo and remote repo. You "pull" commits from the remote repo down to your local repo; you "push" commits from your local repo to the cloud.

You must pull before you push. You can pull after you commit, but if you commit and there are already new commits in the remote repo, you have to pull those commits and merge them with yours before you're allowed to push your commit to the remote repo.[6] It's best to pull before you commit, just to reduce the number of "merge" commits that appear in the project history. But if you're working on the same file as a collaborator, merges are unavoidable. Git isn't Dropbox: if you ignore a project for a couple weeks, nothing will sync automatically. But there's no reason to pull regularly if you're not looking at a project. Pull when you sit down to resume work.

Relative to the single-machine process of doing a chunk of work and committing it, the collaborative process involves only one extra step: push after you commit.

```
cd /path/to/repo
git pull
# make some changes to a file named "test.txt"
git add test.txt
git commit -m "edited test.txt"
git push
```

---

[6]Git Basics 2.5 on `git push`: "This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push."

Branching and merging

A branch is series of commits that can be distinguished from another collection of commits. Branches allow you to make commits without immediately having that code be imposed on the master project. The merit of branching is most obvious when you're doing web development in which your service is always "live". If the "master" branch is your live website, you don't want version 0.1 of a new feature to immediately go live. In that case, the master branch must be immaculate, and branches are where development happens. Once new features (or rewrites of old features) are tested and proven, they're merged into the master branch.

Research doesn't quite follow this model. It's unlikely that there's a "live" version. On the other hand, we often rewrite code when something is slow or inefficient. For large chunks of work (or for rewrites that should deliver identical output), it makes sense to work on your own branch so that others' downstream work isn't altered until you're ready to merge into the master branch. We create a new branch whenever we create a new task or substantially rewrite an old task. See Section A.7.1 for more about the role of branching within our code writing and reviewing processes.

The `git branch` command lets you create, list, rename, and delete branches. It is tightly integrated with the `git checkout` and `git merge` commands.

```
git branch
# List all of the branches in your repository.


git branch <branch>
# Create a new branch called <branch>. This does not check out the new branch.


git branch -d <branch>
# Delete the specified branch. This is a ``safe'' operation in that Git prevents
# you from deleting the branch if it has unmerged changes.


git branch -D <branch>
# Force delete the specified branch, even if it has unmerged changes. This is
# the command to use if you want to permanently throw away all of the commits
# associated with a particular line of development.


git branch -m <branch>
# Rename the current branch to <branch>.


git checkout <existing-branch>
# Check out the specified branch, which should have already been created with
# git branch. This makes <existing-branch> the current branch, and updates the
# working directory to match.


git checkout -b <new-branch>
# Create and check out <new-branch>. The -b option is a convenience flag that
# tells Git to run git branch <new-branch> before running git checkout
# <new-branch>. git checkout -b <new-branch> <existing-branch>


git merge <branch>
# Merge the specified branch into the current branch. Git will determine the
```

```
# merge algorithm automatically (discussed below).


git merge --no-ff <branch>
# Merge the specified branch into the current branch, but always generate a
# merge commit (even if it was a fast-forward merge).
```

In some cases, you might forget to pull the latest commit before making a new one. As a result, you might be unable to push your commit due to a conflict with the latest commit on a repository. To find a solution, read Chapter 28 Pull, but you have local work.

A.7 Reviewing code on BitBucket

BitBucket and GitHub offer a bunch of browser-based tools supporting code review and comments. Jonathan now prefers this method of providing feedback rather than writing comments in files. One merit is that the history of feedback persists without cluttering up active code. Another is that threaded conversations are much cleaner than writing back-and-forth within code.

Note: BitBucket offers some experimental features from time to time, which might change the interface and make the following instructions not applicable. Please opt out experimental features by going to "Settings - Labs", and disable all the "BETA" features.

A.7.1 Default code review process

Given a task, the default process for writing code and submitting it for review is the following:

1. Create a new branch when you create a new task (G&S: "The assignee should create a separate git branch for each task that involves code or output stored in the repository.")

2. Commit your work on this new branch. A complete task includes a `Makefile`.

3. In addition to creating a `code` folder, write a `readme.md` in the task folder that explains what it does.

4. Submit a pull request when your code is ready to be reviewed.

5. Team uses the pull request to review code and have a conversation about the task

6. Additional commits are made on the branch in response to discussion and feedback.

7. Finalized task is added to master branch using a merge that squashes the branch's commits.

A.7.2 Commenting on commits

Comment on a file

Jonathan wrote two comments about data visualization on commit `abf9191`. Scrolling through the commit webpage, Jonathan clicks on "comment" for `output/figures/summarystats/device_byzip.eps` and says the following:

output/figures/summarystats/device_byzip.eps MODIFIED

View file Comment ...

Jonathan Dingel REPO OWNER
- Please impose graphregion(color(white)) on every twoway plot created in Stata.
- Please plot points; there are too many observations for plotting the 5-digit ZIP as the label.
- Instead of plotting the two levels against each other, please compute the difference and plot a kernel density of it. Compute the average as (device_share − population_share) / (device_share/2 + population_share/2). This isn't the perfect measure, but it should be mean zero and let us see the dispersion in the differences.
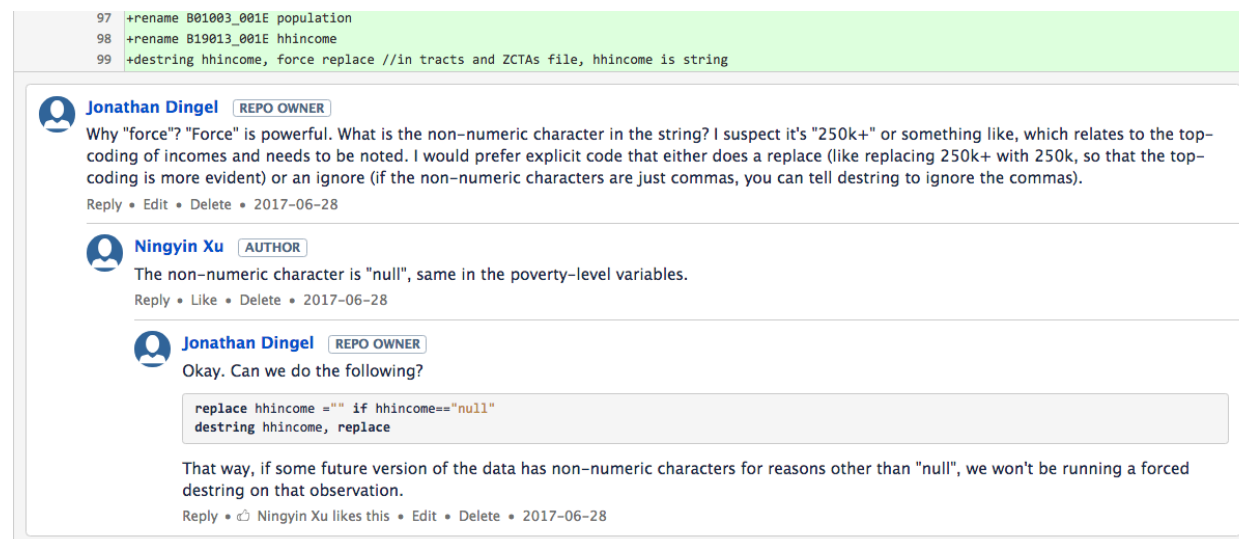- Also, to try another visualization, please make a scatterplot in which the vertical axis is device_share − population_share and the horizontal axis is population_share.

Reply • Edit • Delete • 3 days ago

File contents unchanged.

Ningyin can reply to comment to ask for clarification, and Jonathan can link to this commit comment when assigning tasks.

Comment on a line

You can also write comments on individual lines of a file. Jonathan spotted a "force" option on line 99 of code/gen_census_data.do and wrote a comment seeking clarification. He embeds his code suggestion as part of the comment, complete with BitBucket's language-specific syntax highlighting.

```
97  +rename B01003_001E population
98  +rename B19013_001E hhincome
99  +destring hhincome, force replace //in tracts and ZCTAs file, hhincome is string
```

Jonathan Dingel REPO OWNER
Why "force"? "Force" is powerful. What is the non-numeric character in the string? I suspect it's "250k+" or something like, which relates to the top-coding of incomes and needs to be noted. I would prefer explicit code that either does a replace (like replacing 250k+ with 250k, so that the top-coding is more evident) or an ignore (if the non-numeric characters are just commas, you can tell destring to ignore the commas).
Reply • Edit • Delete • 2017-06-28

Ningyin Xu AUTHOR
The non-numeric character is "null", same in the poverty-level variables.
Reply • Like • Delete • 2017-06-28

Jonathan Dingel REPO OWNER
Okay. Can we do the following?

```
replace hhincome ="" if hhincome=="null"
destring hhincome, replace
```

That way, if some future version of the data has non-numeric characters for reasons other than "null", we won't be running a forced destring on that observation.
Reply • 👍 Ningyin Xu likes this • Edit • Delete • 2017-06-28

A.8 Sharing results via the logbook

This logbook is where we document the research project. It can present preliminary results, sketch empirical specifications, etc. There are separate chapters for research infrastructure, call notes, and research results. These categories are easy to revise – individual logbook entries should be modular pieces suitable for re-ordering. Within chapters, entries are usually ordered chronologically. Entries are identified by date and author. At their best, logbook entries presenting results can be directly pasted into the first draft of the paper.

A.8.1 Committing logbook entries

To add an entry this logbook:

1. Write the logbook entry as a TeX file. The TeX file will be embedded within the logbook TeX file, so do not include any LaTeX preamble.

2. Insert the entry into the logbook by editing `logbook.tex`. The logbook entry (section) title should include an informative title, the date, and author. Follow it by `\input{}` with the filename. All other content is in the entries' individual TeX files.

```
\chapter{Call notes}
\section{20180327 JD} \input{./entries/20180327Dingel.tex}
```

What not to commit

- Please do not commit the logbook PDF to the repository. Doing so often creates merge conflicts when users preview their own logbook entries by compiling the logbook PDF locally. Similarly, do not commit the paper PDF to the repository.

- Do not commit the `logbook.aux`, `logbook.log`, nor `logbook.out` files generated by your LaTeXcompiler. Either add these to your `.gitignore` file or have your Makefile delete these after succesful compilation of the PDF.

A.9   Unix/Linux shell commands

This section documents basic Unix/Linux shell commands and then some clever combinations for tasks we often encounter during research. At the command line, type `man <<command>>` to get the manual page for `<<comand>>`.

A.9.1   Mac OS X is Unix and POSIX-compliant, not GNU/Linux

Mac OS X is based on the Darwin operating system, which is based on BSD. It counts as UNIX and is POSIX-compliant. But it is not GNU/Linux. You will therefore run into annoying differences when trying to use some utilities, mostly when you see a solution on StackExchange that works on GNU/Linux but doesn't work on Mac OS X. Here's a short list:

- `tree` not available in OS X by default, please run `brew install tree` to install it via Homebrew.

- `cut` lacks the complement option in OS X

- `wc` lacks the `-L, --max-line-length` option available in Linux.

- Unfortunately, `sed` differs considerably between GNU and OS X.

  - The `-i` option syntax differs between GNU and OS X

  - GNU sed supports `\|`, `\+`, and `\?` in regular expressions but OS X (and POSIX) don't.

  - Other warnings (1,2), like the `-e` flag will give you extended regular expressions in OS X but isn't really compatible with Linux.

A.9.2 Navigating the file system

- `pwd`: identify the "present working directory"

- `cd`: "change directory" to the named destination (e.g., `cd <<destination>>`)

- `ls -lht`: lists the current directory's contents. The `-lht` options list the files in detail, with human-readable file sizes, ordered by time last modified.

- tab completion: You do not have to type a complete filename. Starting typing the file name and hit the `tab` key. Commands with long or difficult to spell filenames can be entered by typing the first few characters and pressing a completion key, which completes the command or filename.

- To recall a command from your history, type `ctrl-R` to search and type a fragment of the command

- hashtag comments: comments in the shell are set off by `#`. add a comment to your command to tag it for easier retrieval via search in the future

- copy files using `cp`, move files using `mv`

- copy files across different servers using scp

A.9.3 Navigating text

- `ctrl-A` jumps to beginning of line

- `ctrl-E` jumps to end of line

- `ctrl-K` kills content (cuts) from cursor to end of line

- `ctrl-U` kills content (cuts) from cursor to beginning of line

A.9.4 Piping and writing to file

A pipeline is a sequence of processes chained together by their standard streams, so that the output of each process feeds directly as input to the next one. Pipe using `|`.

To write output to a file, use `>`. This overwrites the file if it already exists. Use `>>` to append to an existing file.

- parse a directory listing using `grep`: the command

  ```
  ls -l | grep 'key'
  ```

  will output the directory listing and select only the lines containing the phrase "key"

- write hello world to a file:

  ```
  echo 'hello world' > file.txt
  ```

- look for missing files in a numbered sequence:

```
ls ../output/isoindices_{1..500}.dta > /dev/null
```

This returns only the files that are not found in that sequence. (the null device is a device file that discards all data written to it but reports that the write operation succeeded)

A.9.5   Text processing

- `cat`: Reads files sequentially, writing them to standard output. The name is derived from its function to concatenate files. At the command line, think of this as "print the file".

- `head -n <filename>` outputs the first `n` lines of the file. the default is ten lines

- `tail -n <filename>` outputs the last `n` lines of the file. the default is ten lines

- `grep`: returns all lines of a file matching a specified expression (use `-v` option to return all lines not containing the expression)

    - how to detect invalid utf8 unicode/binary in a text file: `grep -axv '.*' file.txt`

- `sed`: stream editor with many functions; I mostly use it to substitute one expression for another

- `tr`: change or delete characters. It is useful for changing filenames (e.g. deleting whitespace).

- `awk`: find and replace text, print columns, a number of other text editing functions

    - An intro to the great language with the strange name (Daniel Robbins, 1 Dec 2000)
    - Why you should learn just a little Awk (Greg Grothaus, 29 Sep 2010)

- `paste`: horizontally concatenate files with equal number of lines

Combine these well and you get something like "Command-line Tools can be 235x Faster than your Hadoop Cluster".

A.9.6   Emacs

Most often, you'll be an environment where you get to choose your own text editor. However, in some (i.e., confidential) computing environments, you will not be free to install arbitrary software. Emacs will typically be installed everywhere, so it it worth knowing some basic info.

Emacs is a family of text editors, dating to the 1970s, that is "the extensible, customizable, self-documenting, real-time display editor." But there's a learning curve. Even the introductions can be overwhelming.

With regard to Emacs key notation, `C` means the "control" key and `M` means the alt/option key.

- Quitting/exiting: `C-x C-c`

- Saving: `C-x C-s`

- Copy and paste: The selected region is where your cursor is relative to where you set a mark. Set a mark with C-space. Then move your cursor to end of region and hit C-w to cut (kill) or M-w to copy. Paste using C-y (yank).

A.9.7 Multiple "windows"

[Screen](#) is "a full-screen window manager that multiplexes a physical terminal between several processes, typically interactive shells." Consider using this if you want, e.g., to run Stata interactively in the bottom half of your screen while working on your do file in a text editor in the top half.

A.9.8 Data compression

In macOS, using the Finder to compress files produces hidden files that are useless to non-Mac users.[7] In addition, the Finder does not use the ZIP64 extension to compress files what may make large ZIP files unreadable by other machines. To avoid these issues, compress data by using the Terminal command line instead of the Finder. The following Unix utilities are available to compress or decompress data:

- `tar` for tar archive format

- `zip` for zip archive format.

- `gzip` for gz archive format.

When compresing files, it might be also useful to store the names of saved file without storing the directories. In zip utility, it can be done using option `-j` to junk the paths.

A.9.9 SLURM (Simple Linux Utility for Resource Management)

Read the [RCC introduction](#) and then head over to the [official documentation](#). To understand how the RCC compute ecosystem and get futher information on RCC usage, read [slides from UChicago Slurm workshop.](#) This [two-page PDF](#) lists almost all the relevant commands you might need.

- `sbatch`: this command submits jobs (`.sbatch` scripts) to the job scheduler on the cluster

    - Instead of writing a separate sbatch file for each script you might want to run using SLURM, you can pass the particular script/filename as an argument using the `--export` option.

    - It is possible to set job dependencies in Slurm: Slurm will not start a job until the specified dependecies are satified. To set job dependencies, specify the dependency type and job ID in `--dependency` option.

- `sinteractive`: start an interactive session on the server

- `squeue --user=jdingel`: list running and queued jobs for the relevant users

- My preferred `squeue` command is the following:

```
squeue --user=jdingel --format="%.17i %.13P %.20j %.8u %.8T %.9M %.9l %.6D %R" #jdjobs
```

This provides a good bit more information about each job.

- `rcchelp sinfo`: produce a summary of the partitions on Midway

- How to only see really costly jobs:

---

[7]For details, see https://perishablepress.com/remove-macosx-ds-store-zip-files-mac/.

```
rcchelp usage --byjob | grep '[0-9][0-9][0-9]\.[0-9][0-9][[:blank:]]|'
```

Without options, `rcchelp usage --byjob` provides a complete history of job submissions. Piping it to grep to select only lines containing a number in the form ###.## returns only jobs that used at least 100 service units.

A.9.10 Jumping between MacOS GUI and Terminal

A few tips if you're using Terminal on your Mac but not a 100% command-line ninja:

- You can drag the path of a Mac folder into Terminal (or Stata) by dragging the folder icon at the top of its Finder window into the Terminal prompt (via Luke Stein)

- Typing '`open .` ' in any directory in the Terminal will open that folder in the Finder (via Florian Oswald)

A.9.11 Other resources

- https://unix.stackexchange.com/questions/6/what-are-your-favorite-command-line-features-or-tricks

- MIT's Hacker Tools course

- UChicago RCC Workshops and Training

A.10 Aesthetics and visualization

These are Jonathan's standards and he's willing to argue for them.

- Generally, follow Edward Tufte's *Visual Display of Quantitative Information*. Mostly important, maximize the data-ink ratio.

- At minimum, let's adhere to Schwabish "An Economist's Guide to Visualizing Data" (*JEP* 2014)

- Never make a graphic with fewer than a dozen data points. A dozen data points belong in a table.

- Please impose `graphregion(color(white))` on every `twoway` plot created in Stata.

A.11 Manuscript preparation

Before distributing a draft, use automated tools to validate the text. Within `paper/reviewing`, there is a Makefile that

- reports repeated words to avoid "the the" errors

- changes references to be "tight": `Table~\ref` rather than `Table \ref`

- prints hard-coded numbers for verification or replacement

- runs ChkTeX to check LaTeX semantics

- runs TeXtidote to check grammar and spelling

- If you haven't installed ChkTeX, Macintosh installation instrutions are available at `http://www2.hawaii.edu/~ramonf/ChkTeXonMacOSX/index.html`.

- TeXtidote is available from `https://github.com/sylvainhalle/textidote`. To install on Mac OS X, install the JDK.

A.12   Notes on Julia

Jonathan prefers Julia to Matlab.

Julia resources

Jonathan learned Julia by starting with the QuantEcon introduction to Julia.
Here are some helpful resources:

1. julialang.org: for installation and general info about julia: blogs, publications, conference

2. docs.julialang.org/en/stable/manual/introduction: excellent manual for julia

3. lectures.quantecon.org/jl: an excellent manual for Julia with a macro vibe

4. github.com/bkamins/Julia-DataFrames-Tutorial: excellent tutorial on how to read/use DataFrames in Julia

5. juliabloggers.com: to keep up with advancements in Julia

6. johnmyleswhite.com: for interesting discussions about performance in Julia

List of very useful packages:

- DataFrames.jl: DataFrames in Julia

- ReadStat.jl: read `dta` files in Julia

- JLD.jl: fantastic way of storing output in Julia

- Optim.jl: General optimization in julia

- Calculus.jl: for automatic diffentiation in Julia (very practical to check your analytical expressions of gradients and hessians)

Notes on using Julia on UChicago's Midway computing cluster

- You cannot connect to the external internet from Midway's compute nodes. Julia's packages are pulled from Github, so you must install packages in Julia (and in Stata) when running on a login node.

- You can `Pkg.activate(".")` on a compute node, but you cannot `Pkg.instantiate()` on a compute node. You need to install packages, define the `Project.toml`, `Pkg.instantiate()` and precompile (`using` commands once) on a login node before moving to the compute nodes.

Notes on managing package dependencies in Julia code

Running Julia code will involve packages. Here's how we manage them in our repositories.

- Project.toml and Manifest.toml

    - These two files are central to `Pkg`, Julia's builtin package manager. They make it possible to instantiate the exact same package environment on different machines.

    - `Project.toml` describes the project on a high level. The package dependencies and compatibility constraints are listed in the project file.

    - `Manifest.toml` is an absolute record of the state of the packages in the environment. This is not pleasant to read.

- In each repository, we create a task called `setup_environment` that defines all package dependencies using the `Project.toml` and `Manifest.toml` files.

- In a Julia script, we set the active environment using the following commands:

```
import Pkg
Pkg.activate("../input/Project.toml")
using package_name
```

    While we only explicitly name `Project.toml` as an input in this script, this presumes that the corresponding `Manifest.toml` file is available in the same directory.

- In each Makefile that executes a Julia script, we provide these `toml` files as inputs by creating symbolic links to `setup_environment`:

```
../input/Project.toml: ../../setup_environment/output/Project.toml | ../input/Manifest.toml
    ../input
  ln -s $< $@
../input/Manifest.toml: ../../setup_environment/output/Manifest.toml | ../input
  ln -s $< $@
```

    Note that we define `Manifest.toml` as a pre-requisite for `Project.toml`. If a list of pre-requisities is generated by parsing the Julia script for all files that start with `../input/`, only `Project.toml` will be flagged. Making `Manifest.toml` a pre-requisite of `Project.toml` ensures that both files appear in the `input` folder.

A.13 Notes on Stata

Stata usage among older economists may just reflect path dependence. It has a number of shortcomings, such as no proper package mangagement. Beware of dumb Stata gotchas.

Command-line execution

For old versions (before Stata 16), a few Stata commands only work in "interactive" mode and not when Stata is invokved at the command line.

- `graph export`: "ps and eps are available for all versions of Stata; png and tif are available for all versions of Stata except Stata(console) for Unix; pdf is available only for Stata for Windows and Stata for Mac." See https://www.stata.com/manuals13/g-2graphexport.pdf#g-2graphexport. We usually choose eps format.

Temporary files

Stata loads at most one dataset at a time, thus opening a dataset will cause Stata to discard the dataset that is currently in memory. This constraint may push you to consider saving lots of intermediate output or temporary files to the `output` folder of your task. Don't. Use Stata's `tempfile` command instead.

Here is a trivial example

```
clear
use "exampw1.dta", clear
tempfile temp1  // create a temporary file
save "`temp1'"    // save memory into the temporary file
use "exampw2.dta", clear
merge 1:1 v001 v002 v003 using "`temp1'" // use the temporary file
```

Evaluating inequalities

In Stata, missing numeric observations take the value of positive infinity. Thus, when evaluating inequalities, $. > X$ is true for any X. I prefer using the command `inrange( )` to avoid this issue. Consider the following two ways of generating a dummy indicating that x1 takes a positive value:

```
gen byte positive1 = 1 if x1>=0
gen byte positive2 = (inrange(x1,0,.)==1)
```

The first command will cause positive1 to be true if x1 is non-negative or if x1 is missing. By contrast, the second command will set positive2 to true only if x1 is non-negative and non-missing.

A.14   Running Windows

If scientific computing lives at the Linux command line, then being a Windows users can be tricky. The two parts of our workflow that are most difficult to implement on Windows are executing Makefiles and creating symbolic links.

For now, we recommend simply installing Ubuntu on Windows Subsystem for Linux (WSL). Here are instructions: Install Ubuntu on WSL2 on Windows 10. You will need to install the Linux versions of all applications, such as R, LaTeX, and Stata. This is genuine Ubuntu; it will not invoke your Windows executables.

Chapter B

Call notes

Chapter C

Research design

Chapter D

Data description

Chapter E

Research results

Chapter F

Feedback