Code and Data for the Social Sciences: A Practitioner's Guide

Matthew Gentzkow Jesse M. Shapiro[1]

Chicago Booth and NBER

April 6, 2013

[1] E-mail: matthew.gentzkow@chicagobooth.edu, jesse.shapiro@chicagobooth.edu.

Contents

Chapter 1

Introduction

What does it mean to do empirical social science? Asking good questions. Digging up novel data. Designing statistical analysis. Writing up results.

For many of us, most of the time, what it means is writing and debugging code. We write code to clean data, to transform data, to scrape data, and to merge data. We write code to execute statistical analyses, to simulate models, to format results, to produce plots. We stare at, puzzle over, fight with, and curse at code that isn't working the way we expect it to. We dig through old code trying to figure out what we were thinking when we wrote it, or why we're getting a different result from the one we got the week before.

Even researchers lucky enough to have graduate students or research assistants who write code for them still spend a significant amount of time reviewing code, instructing on coding style, or fixing broken code.

Though we all write code for a living, few of the economists, political scientists, psychologists, sociologists, or other empirical researchers we know have any formal training in computer science. Most of them picked up the basics of programming without much effort, and have never given it much thought since. Saying they should spend more time thinking about the way they write code would be like telling a novelist that she should spend more time thinking about how best to use Microsoft Word. Sure, there are people who take whole courses in how to change fonts or do mail merge, but anyone moderately clever just opens the thing up and figures out how it works along the way.

This manual began with a growing sense that our own version of this self-taught seat-of-thepants approach to computing was hitting its limits. Again and again, we encountered situations like:

- In trying to replicate the estimates from an early draft of a paper, we discover that the code that produced the estimates no longer works because it calls files that have since been moved. When we finally track down the files and get the code running, the results are different from the earlier ones.
- In the middle of a project we realize that the number of observations in one of our regressions is surprisingly low. After much sleuthing, we find that many observations were dropped in a merge because they had missing values for the county identifier we were merging on. When we correct the mistake and include the dropped observations, the results change dramatically.
- A referee suggests changing our sample definition. The code that defines the sample has been copied and pasted throughout our project directory, and making the change requires updating dozens of files. In doing this, we realize that we were actually using different definitions in different places, so some of our results are based on inconsistent samples.
- We are keen to build on work a research assistant did over the summer. We open her directory and discover hundreds of code and data files. Despite the fact that the code is full of long, detailed comments, just figuring out which files to run in which order to reproduce the data and results takes days of work. Updating the code to extend the analysis proves all but impossible. In the end, we give up and rewrite all of the code from scratch.
- We and our two research assistants all write code that refers to a common set of data files stored on a shared drive. Our work is constantly interrupted because changes one of us makes to the data files causes the others' code to break.

At first, we thought of these kinds of problems as more or less inevitable. Any large scale endeavor has a messy underbelly, we figured, and good researchers just keep calm, fight through the frustrations, and make sure the final results are right. But as the projects grew bigger, the problems grew nastier, and our piecemeal efforts at improving matters—writing handbooks and protocols for our RAs, producing larger and larger quantities of comments, notes, and documentation—proved ever more ineffective, we had a growing sense that there must be a way to do better.

In the course of a project involving a really big dataset, we had the chance to work with a computer scientist who had, for many years, taught the course on databases at the University of Chicago. He showed us how we could organize our really big dataset so that it didn't become impossible to work with. Neat, we thought, and went home.

Around that time we were in the middle of assembling a small (but to us, very important) dataset of our own. We spent hours debating details of how to organize the files. A few weeks in we realized something. We were solving the same problem the computer scientist had shown us how to solve. Only we were solving it blind, without the advantage of decades of thought about database design.

Here is a good rule of thumb: If you are trying to solve a problem, and there are multi-billion dollar firms whose entire business model depends on solving the same problem, and there are whole courses at your university devoted to how to solve that problem, you might want to figure out what the experts do and see if you can't learn something from it.

This handbook is about translating insights from experts in code and data into practical terms for empirical social scientists. We are not ourselves software engineers, database managers, or computer scientists, and we don't presume to contribute anything to those disciplines. If this handbook accomplishes something, we hope it will be to help other social scientists realize that there are better ways to work.

Much of the time, when you are solving problems with code and data, you are solving problems that have been solved before, better, and on a larger scale. Recognizing that will let you spend less time wrestling with your RA's messy code, and more time on the research problems that got you interested in the first place.

Chapter 2

Automation

Rules

(A) Automate everything that can be automated.

(B) Write a single script that executes all code from beginning to end.

Let's start with a simple research project. We wish to test the hypothesis that the introduction of television to the US increased sales of potato chips. We receive an Excel file by e-mail with two worksheets: (i) "tv," which contains for each county in the US the year that television was first introduced; and (ii) "chips," which contains total sales of potato chips by county by year from 1940 to 1970. We wish to run a panel regression of log chip sales on a dummy variable for television being available with county and year fixed effects.

Here is one way we might proceed: Open the file in Excel and use "Save As" to save the worksheets as text files. Open up a statistical program like Stata, and issue the appropriate commands to load, reshape, and merge these text files. Define a new variable to hold logged chip sales, and issue the command to run the regression. Open a new MS Word file, copy the output from the results window of the statistical program into a table, write up an exciting discussion of the findings, and save. Submit to a journal.

Just about everybody learns early in graduate school, if not before, that this "interactive" mode of research is bad. They learn that the data building and statistical analysis should be stored in scripts—`.do` files in Stata, `.m` files in Matlab, `.r` files in R, and so forth.

It is worth pausing to remember why we don't like the interactive mode. There are many reasons, but two big ones.

The first is replicability. If the next day, or the next year, we want to reproduce our regression of chip sales on TV, we might dig up `tv.csv` and `chips.csv`, load them back into Stata, and set to work reshaping, merging, defining variables, and so forth. Perhaps we will get lucky, since this analysis is so simple, and get back the same coefficient when we run the regression. Or perhaps not. Even in this simple example, there are innumerable things that could go wrong: since writing the paper we have received an updated version of `tv.csv` and we inadvertently use the new one rather than the old one; we forget that we dropped several counties whose chip sales were implausibly large; we compute regular standard errors whereas before we computed robust standard errors; and so on.

On a deeper level, because there is no record of the precise steps that were taken, there is no authoritative definition of what the numbers in our paper actually are. If someone later asks why the number of observations reported in our table is different from the number of observations in the raw data, or how we computed our standard errors, or what we did with county-years with missing chip sales, and we ran the analysis interactively, we will have no way to say for sure.

The second reason is efficiency. If we decide to run a different regression, say using the level rather than the log of chip sales, we will have to go back and repeat all of the steps of building and cleaning the data. We can avoid this by saving the combined dataset before running any regressions, but if we later wish to change which observations we keep and which we drop, we will be back to square one.

In a real project, there might be a thousand steps from raw data to final results. For each of these, there could be several alternatives, detours, and experiments that were tried and discarded. Each step is typically run hundreds of times as the analysis is developed and refined. Trying to run and re-run all these steps interactively would be completely untenable.

For this reason, most researchers learn to script key steps, especially data manipulation and statistical analysis. Here is what the project directory for the paper above might look like after we switched to writing .do files, expanded our analysis a bit, and switched to $\LaTeX$ for word processing:

```
chips.csv       mergefiles.do          tv_potato_submission.pdf
cleandata.do    regressions_alt.do     tv_potato.tex
extract0B.xls   regressions_alt.log    tv.csv
fig1.eps        regressions.do         tvdata.dta
fig2.eps        regressions.log
figures.do      tables.txt
```

This is certainly a big improvement over our initial interactive approach. If we stare at these files for a while, we can probably work out more or less what they are. `Extract0B.xls` is the raw data file, `chips.csv` and `tv.csv` are the text files exported from Excel, and `tvdata.dta` is the combined data file in Stata format. `Mergefiles.do` and `cleandata.do` are the scripts that build the data, `figures.do`, `regressions.do`, and `regressions_alt.do` are the scripts that run the analysis, and the `.log` and `.eps` files are the output. `Tv_potato.tex` is the paper, `tables.txt` contains the tables, and `tv_potato_submission` is the PDF version we submitted to the journal.

But if we set about actually trying to reproduce `tv_potato_submission.pdf`, we'd immediately run into a bunch of questions. Should we export all observations from `extract0B.xls`, or just those with nonmissing data? Which should be run first, `cleandata.do` or `mergedata.do`? Does it matter in which order we run `regressions.do` and `figures.do`? Is the output from `regression_alt.do` actually used in the paper or is this file just left over from some experimentation? What is `tables.txt`? Is it produced manually or by code? Which numbers in the log files correspond to the numbers reported in the paper? Is `tv_potato_submission.pdf` just a PDF version of `tv_potato.tex` or did we do additional formatting, etc. before submitting to the journal?

We suspect that the experience of trying to reverse-engineer the build steps for a directory like this will feel familiar to many readers who have tried to make sense of directories their RAs or coauthors produced, or even directories that they produced themselves a few months in the past. In this toy example, the problems are probably surmountable and, assuming that we didn't do anything silly like modify and rerun `regressions.do` after the PDF was produced, we could probably reproduce the paper in a reasonable amount of time. But as most of us know from painful experience, the reverse-engineering process for a moderately complex project can easily become days or weeks of frustrating work, and the probability of those "silly" mistakes that render replication all but impossible is remarkably high.

To make the output of our directory replicable, we need to automate more steps. And we need a way to store the information about the order in which the steps are run.

First, let's add a Stat/Transfer script called `export_to_csv.stc` that handles the conversion from Excel. (Stata can also do this directly using the "import excel" command.) Next, let's switch from outputting `tables.txt` to outputting `tables.tex`, a LaTeX file produced by Stata's "outreg" command.

Finally, let's add another key script to the directory, called `rundirectory.bat`, which is a Windows shell script. Its contents look like this:

```
---- rundirectory.bat ----
stattransfer export_to_csv.stc
statase -b mergefiles.do
statase -b cleandata.do
statase -b regressions.do
statase -b figures.do
pdflatex tv_potato.tex
```

The `rundirectory.bat` script works like a roadmap, telling the operating system how to run the directory. Importantly, the rundirectory script also tells a human reader how the directory works. But unlike a readme file with notes on the steps of the analysis, `rundirectory.bat` cannot be incomplete, ambiguous, or out of date.

The proof in the pudding is that we can now delete all of the output files in the directory – the `.csv` files, the `.log` and `.eps` files, `tables.tex`, the `.pdf` – and reproduce them by running `rundirectory.bat`. This is the precise sense in which the output is now replicable.

Writing a shell script like `rundirectory.bat` is easy.[1] You may need a few tweaks, such as adding Stata to your system path, but many of these will be useful anyway. You could write all these steps into a Stata script (`rundirectory.do`), but a system shell provides a more natural interface for calling commands from multiple software packages, and for operating system commands like moving or renaming files.

[1] If you don't use Windows, Linux shell files work almost the same way. And if you're comfortable with Python, you can do even better, and write a `rundirectory.py` that will work on both Windows and Linux systems.

Of course, `rundirectory.bat` does not automate *everything*. We could (and, admittedly, are tempted to) write a little Python script to submit the paper to a journal, but that seems like overkill even to us.

On the other hand, we have consistently found that pushing the boundaries of automation pays big dividends. The costs tend to be lower than they appear, and the benefits bigger. A rule of research is that you will end up running every step more times than you think. And the costs of repeated manual steps quickly accumulate beyond the costs of investing once in a reusable tool.

We used to routinely export files from Excel to CSV by hand. It worked ok until we had a project that required exporting 200 separate text files from an Excel spreadsheet. We followed our usual practice and did the export manually. Some time later, the provider sent us a new Excel file reflecting some updates to the data. We had learned our lesson.

Chapter 3

Version Control

Rules

(A) Store code and data under version control.

(B) Run the whole directory before checking it back in.

In the last chapter, we showed what the project directory for our seminal TV and potato chips project might look like. After we work on the directory for a while, the key files might look like this:

```
cleandata_022113.do     cleandata_022613.do      regressions.log
cleandata_022113a.do    cleandata_022613_jms.do  regressions_022413.do
chips.csv               tvdata.dta               regressions_022713_mg.do
regressions_022413.log
```

Dates are used to demarcate versions of files. Initials (JMS for Jesse, MG for Matt) are used to indicate authorship.

There are good reasons to store multiple versions of the same file. The most obvious is that it provides a quick way to roll back changes you want to discard. Another is that it facilitates comparison. Maybe Matt wants to show Jesse how he's thinking of changing their main specification. Creating `regressions_022713_mg.do` may be a good way to illustrate what he has in mind. If Jesse doesn't like it, he can always delete it.

The goal is admirable, but the method is wrong. There are two main reasons why. First, it is a pain. The researcher needs to decide when to "spawn" a new version and when to continue to edit the old one (hence `022113a`). The researcher needs to tag authorship and date every file. Failing to do that will result in confusion: Why is the date on the file name February 21 when the operating system says this was last edited in March?

And confusion is the second, and by far the more important, reason why this "date and initial" method is poor. Look at the file names above and answer the following questions: Which is the log file produced by `regressions_022713_mg.do`? Did the author (darn you, Matt!) fail to change the output file name in the code, overwriting `regressions_022413.log`? Did he simply not output a log?

Which version of `cleandata.do` produces the data file used by `regressions_022413.do`? Is it the one labeled `022113a`—the last one before February 24? Or was `regressions_022413` created on February 24 but edited later, raising the possibility that it needs output from `cleandata_022613.do` to run correctly? Unfortunately, we failed to tag `tvdata.dta` with a date and initial—probably because changing the file name in three different places with each new version is an enormous hassle.

Given a few minutes to look at the system dates and file contents, you could probably work out which inputs are needed by which scripts. And, having learned your lesson, next time you will harangue your coauthor and RAs to remember to date and initial every script, LOG file, and intermediate data file, so hopefully there's no more confusion.

This is too much work just to keep track of multiple versions of files. And it creates a serious risk that, later, you won't be able to sort out which file goes with which, and hence you won't be able to replicate your results. Fortunately, your computer can take care of this for you, automatically, using free software that you can set up in a few minutes.

Before we tell you how, we will start with a fact. (This is, after all, a handbook for empirical researchers.) *Not one piece of commercial software you have on your PC, your phone, your tablet, your car, or any other modern computing device was written with the "date and initial" method.*

Instead, software engineers use a tool called version control software to track successive versions of a given piece of code. Version control works like this. You set up a "repository" on your PC (or, even better, on a remote server). Every time you want to modify a directory, you "check it out" of the repository. After you are done changing it, you check it back in. That's it. You don't change file names, add dates, or anything. Instead, the software remembers every version that was ever checked in.

What happens if you change your mind about something? You ask the software for a history of changes to the directory and, if you want to go back to an old version of the directory or even of a single file, the operation just takes a click.

And what about your sneaky coauthor's decision to change the main regression model specification without telling you? The version control software automatically records who authored every change. And if you want to see what the changes were, most modern packages will show you a color-coded side-by-side comparison illustrating which lines of code changed and how.[1]

The main thing about this approach that is great, and the reason real software engineers *must* use a tool like this, is that it maintains a single, authoritative version of the directory at all times. In rare cases where two people try to make simultaneous and conflicting changes to the same file, the software will warn them and help them reconcile the conflicts.

A major ancillary benefit is therefore that you can edit without fear. If you make a mistake, or if you start in a new direction but later change your mind, you can always roll back all or part of your changes with ease. This requires no keeping track of dates and initials. All file names can remain just as nature intended. The software handles the versioning for you, so you can focus on writing the code and making it right. You didn't spend six years in grad school so you could type in today's date all over the place.

To visualize how much better your life would be with your code and data under version control, recall (if, gasp, you are old enough) what word processing was like before the invention of the "undo" command. A bad keystroke might spell doom. Version control is like an undo command for *everything.*

So our first rule is to keep everything—code and data—under version control. In fact, version control is fantastic for things like drafts of your papers, too. It allows you to overwrite changes without fear, to keep track of authorship, etc. (Some readers will have noticed the attractions of the "version history" feature of GoogleDocs, which is based on version control models from software

[1] For a slightly more advanced user, there are also well-defined methods for changing code in a way that is explicitly tentative, so you can "pencil in" some changes and let your coauthor have a look before you take them on board.

engineering. With version control software you get that functionality with LaTeX, LyX, or whatever is your favorite editing package.)

But if you want to get the most out of this approach, there is a second rule: you have to run the entire directory before you check in your changes. Return to our example, now dropping the annoying date and initial tags, and adding back `rundirectory.bat` (which, you'll recall, will run every script from top to bottom).

```
rundirectory.bat  tvdata.dta
cleandata.do      regressions.do
chips.csv         regressions.log
```

Suppose Jesse modifies `cleandata.do` and runs it to overwrite `tvdata.dta`. If Jesse checks in that change, Matt may find later that `regressions.do` breaks when he tries to run it, because of a change to `tvdata.dta` that `regressions.do` wasn't expecting.

The way to fix this problem and ensure it never happens again is just to execute `rundirectory.bat`, from start to finish, and check for errors before checking in the directory. If every version you check in has been run successfully via `rundirectory.bat`, then you know that, barring changes in the software itself, the next time you check it out, you will get back the output in `regressions.log` exactly.

Note that this is not a problem with the version control software. The "date and initial" method creates the same potential for this type of within-directory conflict, arguably more so, since a lot of effort is required to keep track of which input files are required for which scripts. Rather, version control, coupled with the rule of checking in complete runs of a directory, provides a comprehensive solution that guarantees both replicability and undo-ability with minimal effort.

OK, you're convinced. Now what? A step-by-step guide to setting up and using version control software is a bit outside our scope here. But for what it's worth, we use a SubVersion repository that we interact with using the very nice TortoiseSVN browser for Windows. Comparable software exists for Macs. More recent version control methods like Git or BitBucket may be worth checking out. It will probably take you a couple days to set up a repository and learn how you want to interact with it. You will break even on that time investment within a month or two.

Chapter 4

Directories

Rules

(A) Separate directories by function.

(B) Separate files into inputs and outputs.

(C) Make directories portable.

Let’s return to the main directory for our potato chip project:

```
                    ---C:/tv_and_potato/---
chips.csv      mergefiles.do        tv_potato_submission.pdf
cleandata.do   regressions_alt.do   tv_potato.tex
extract0B.xls  regressions_alt.log  tv.csv
fig1.eps       regressions.do       tvdata.dta
fig2.eps       regressions.log      rundirectory.bat
figures.do     tables.txt           export_to_csv.stc
```

The directory above contains all the steps for the entire project, governed by the single batch file `rundirectory.bat`.

Having a single directory that has and does everything has some appeal, but for most real-world research projects this organizational system is not ideal. Consider the following scenarios. (i) The researcher wants to change the way the data is built but not the way it is analyzed. This requires either rerunning the entire directory (which is slow) or modifying `rundirectory.bat` to execute some but not all scripts (which is a hassle). (ii) The researcher learns about a neat Stata command that makes the script `export_to_csv.stc` and its outputs `tv.csv` and `chips.csv` unnecessary. Before making this improvement, however, the researcher must search through `regressions.do` and `regressions_alt.do` to make sure these scripts do not depend on `tv.csv` and `chips.csv` in addition to `tvdata.dta`.

Consider the following alternative directory structure (leaving aside the TEX and PDF files for simplicity):

```
---C:/build---             ---C:/analysis---
/input                     /input
    extract0B.xls              tvdata.dta

/code                      /code
    rundirectory.bat           rundirectory.bat
    export_to_csv.stc          getinput.bat
    mergefiles.do              regressions.do
                               regressions_alt.do

/output                    /output
    tvdata.dta                 fig1.eps
                               fig2.eps
                               tables.txt

/temp                      /temp
    chips.csv                  regressions.log
    tv.csv                     regressions_alt.log
```

There are now two high-level directories. One contains the code to build a useable Stata file from the raw inputs. The other contains code to take the Stata file and turn it into figures and tables for the paper.

Within each high-level directory there is a consistent subdirectory structure that separates inputs, outputs, code, and temporary or intermediate files. Each directory is still controlled by a single script called `rundirectory.bat` that executes from start to finish. (In fact, we would advocate for having `rundirectory.bat` start by cleaning the contents of */temp* and */output*, so you can be sure all your output is produced by your current code.)

It is now easy to modify the data build without touching the analysis, and vice versa. And it is now clear from the directory structure that only `tvdata.dta` is required by the analysis code: `chips.csv` and `tv.csv` are explicitly designated as temp files.

A somewhat unusual feature of the structure we outline above is the use of `getinput.bat`, which copies `tvdata.dta` from *C:/build/output* to *C:/analysis/input*. Why not simply write `regressions.do` to load data directly from *C:/build/output*? There are three reasons. The first is that the structure above allows an unfamiliar user to immediately determine exactly what inputs are required by each directory, without trolling through each script to find all the files they call.

The second advantage is that keeping a separate "analysis" copy of `tvdata.dta` makes it possible to modify the build code without instantly changing the behavior of the analysis code. If Matt, for example, changes `tvdata.dta` to have a smaller number of observations, Jesse can run `regressions.do` without seeing any change in output until he refreshes the analysis copy of `tvdata.dta`.

The third, and perhaps the largest, advantage is that the code in `regressions.do` can now use relative file paths (*../input/tvdata.dta*) rather than absolute paths (*C:/build/output/tvdata.dta*). This makes the directory portable: the entire analysis directory, *sans* `getinput.bat`, will run successfully on someone else's PC, even if they do not possess the build directory. And if the location of `tvdata.dta` changes from *C:/build/output* to *C:/buildtv/output*, we can be confident that the only changes we need to make are to `getinput.bat`. No more trolling through `DO` files doing search and replace operations on path names.

We have only outlined a few of the advantages of using modular, functional directories to organize code. There are many others. For example, the output of *C:/build* is now easily accessible by any directory, which makes it easier to have multiple projects that use the same data file without creating multiple, redundant copies. And, separating scripts into functional groups makes debugging easier and faster when something goes wrong.

Modular directories like those we outline above are an especially powerful tool when combined with the version control methods we discuss in chapter 3. Version control makes it possible for getinput.bat to call versions of input files that remain completely stable even as you (or others) modify the code that builds them.

Chapter 5

Keys

Rules

(A) Store cleaned data in tables with unique, non-missing keys.

(B) Keep data normalized as far into your code pipeline as you can.

It is well known that television went to big cities first. So a good analysis of the effect of television on potato chip consumption requires good data on population as a control. We ask an RA to prepare a population dataset to facilitate our analysis. Here it is:

county	state	cnty_pop	state_pop	region
36037	NY	3817735	43320903	1
36038	NY	422999	43320903	1
36039	NY	324920	.	1
36040	.	143432	43320903	1
.	NY	.	43320903	1
37001	VA	3228290	7173000	3
37002	VA	449499	7173000	3
37003	VA	383888	7173000	4
37004	VA	483829	7173000	3

What a mess. How can the population of the state of New York be 43 million for one county but “missing” for another? If this is a dataset of counties, what does it mean when the “county” field is missing? If region is something like Census region, how can two counties in the same state be in different regions? And why is it that all the counties whose codes start with 36 are in New York except for one, where the state is unknown?

We can’t use these data, because we don’t understand what they mean. Without looking back at the underlying code, we could never say confidently what every variable is, or even every row. And we can forget trying to merge on attributes from another dataset. How would we know which state goes with county 36040? Or which region to use for 37003?

We know many researchers who spend time wrestling with datasets like this, and barking at RAs, students, or collaborators to fix them.

There must be a better way, because we know that large organizations like financial institutions, retailers, and insurers have to manage much more complex data in real time, with huge consequences of mistakes.

Long ago, smart people figured out a fundamental principle of database design: that the physical structure of a database should communicate its logical structure.

If you gave your county Census data to someone with training in databases, you’d probably get back something like this, called a *relational database*:

county	state	population
36037	NY	3817735
36038	NY	422999
36039	NY	324920
36040	NY	143432
37001	VA	3228290
37002	VA	449499
37003	VA	383888
37004	VA	483829

state	population	region
NY	43320903	1
VA	7173000	3

Now the ambiguity is gone. Every county has a population and a state. Every state has a population and a region. There are no missing states, no missing counties, and no conflicting definitions. The database is self-documenting. In fact, the database is now so clear that we can forget about names like county_pop and state_pop and just stick to “population.” Anyone would know which entity’s population you mean.

Note that when we say relational database here, we are referring to how the data are structured, not to the use of any fancy software. The data above could be stored as two tab-delimited text files, or two Stata `.dta` files, or two files in any standard statistical package that expects rectangular data.

Stepping back from this example, there are a few key principles at work here. To understand these it is helpful to have some vocabulary. Data is stored in rectangular arrays called *tables*. In the example above, there is a county table and a state table. We will refer to rows of tables as *elements* and columns of tables as *variables.*

Critically, each table has a *key* (rule A). A key is a variable or set of variables that uniquely identifies the elements of a table. The variables that form the key never take on missing values, and a key’s value is never duplicated across rows of the table. So, a state table has one and only one row for New York, and no rows where state is missing.

Each variable in a table is an attribute of the table’s elements. County population is a property of a county, so it lives in the county table. State population is a property of a state, so it cannot live in the county table. If we had panel data on counties, we would need separate tables for things that vary at the county level (like state) and things that vary at the county-year level (like population). This is the main reason that most data files researchers use for analysis do not satisfy the database rules: they typically combine variables defined at several different levels of aggregation.

The variables in a table can include one or more *foreign keys*. A foreign key is the key for another table in the database. In the county table, the variable “state” is a foreign key: it matches a county to an element of the state table. Foreign keys obey the same rules as all variables: they are included at the level of logical attribution. States are in regions, and counties are in states, so “state” shows up in the county table, and “region” in the state table.

Data stored in the form we have outlined is considered *normalized.* Storing normalized data means your data will be easier to understand and it will be harder to make costly mistakes.

Most statistical software won’t run a regression on a relational database. To perform our analysis we are going to need to merge (or join, in database-speak) the tables together to produce a single rectangular array. Plus, we might need to calculate some variables that aren’t in our source data, such as the log of population.

To get from the data you downloaded, entered, or bought from an original source to the matrix on which you will perform estimation, we recommend proceeding in three steps.

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.

Second, construct a second set of normalized files that includes the transformations of the original variables that you will need for your analysis. For example, you might add to the county table a variable indicating the county's population rank within its state. At this stage you can also bring in variables from other databases. For example, you might use a geography database to bring in county latitude and longitude.

Third, merge together the tables in the database to form the rectangular array on which you will estimate your model. At this stage, your database should still have unique, non-missing keys, but it will likely not be normalized. In our example, you will have a county-level file that includes variables like region that are not properties of counties. If you had panel data, your file would include both county-level and county-year-level variables. Do no data manipulation in this step. If your analysis requires the log of state population, calculate it while your database complies with the rules.

Following the steps above means you can keep your data in a normalized form until the last possible step in the code (rule B).

Chapter 6

Abstraction

Rules

(A) Abstract to eliminate redundancy.

(B) Abstract to improve clarity.

(C) Otherwise, don't abstract.

We are concerned about spatial correlation in potato chip consumption. We want to test whether per capita potato chip consumption in a county is correlated with the average per capita potato chip consumption among other counties in the same state. First we must define the "leave-out" mean of per capita consumption for each county:

```
egen total_pc_potato = total(pc_potato), by(state)
egen total_obs = count(pc_potato), by(state)
gen leaveout_state_pc_potato = (total_pc_potato - pc_potato) / (total_obs - 1)
```

We can now test whether `pc_potato` is correlated with `leaveout_state_pc_potato`. If so, we may need to adjust how we compute the standard errors in our model. We perform our analysis and are comforted to find little evidence of spatial correlation.

But what if we are using the wrong level of aggregation? Maybe spatial correlation will show up at the level of the metropolitan area. Let's copy and paste the code above and then adapt it to use metropolitan area instead of state as the level of aggregation:

```
egen total_pc_potato = total(pc_potato), by(metroarea)
egen total_obs = count(pc_potato), by(state)
gen leaveout_metro_pc_potato = (total_pc_potato - pc_potato) / (total_obs - 1)
```

And while we’re at it, let’s check if there is more spatial correlation in potato chip consumption when measured on a per-household rather than per-capita basis. For this we will need a third leave-out mean:

```
egen total_hh_potato = total(hh_potato), by(metroarea)
egen total_obs = count(hh_potato), by(state)
gen leaveout_metro_hh_potato = (total_hh_potato - pc_potato) / (total_obs - 1)
```

Note the errors. In the first “copy-and-paste” operation, we failed to replace an instance of `state` with `metroarea`. In the second, we propagated the first error, plus we failed to replace one use of the per-capita potato variable with the per-household analogue. The code will run, but everything after the first code block will be totally wrong.

Consider an alternative to the copy-and-paste approach, which is to write a general-purpose function that computes the leave-out mean of a variable:

```
program leaveout_mean

    syntax, invar(varname) outvar(name) byvar(varname)
    tempvar tot_invar count_invar
    egen ‘tot_invar’= total(‘invar’), by(‘byvar’)
    egen ‘count_invar’= count(‘invar’), by(‘byvar’)
    gen ‘outvar’ = (‘tot_invar’ - ‘invar’) / (‘count_invar’ - 1)

end
```

Having defined the function above, we can now replace our three code blocks with three lines:

```
leaveout_mean, invar(pc_potato) outvar(leaveout_state_pc_potato) byvar(state)
leaveout_mean, invar(pc_potato) outvar(leaveout_metro_pc_potato) byvar(metro)
leaveout_mean, invar(hh_potato) outvar(leaveout_metro_hh_potato) byvar(metro)
```

Now the amount of copying and pasting is minimized: each input is changed only once as we go from line to line. And because we wrote the `leaveout_mean` function to be totally general, we can use it for other projects as well as this one.[1] We will never again have to write code from scratch to compute a leave-out mean.

Key to achieving these goals is recognizing that all three code blocks were just specific instances of the same abstract idea: compute the mean of a variable across observations in a group, excepting the given observation. In programming, turning the specific instances of something into a general-purpose tool is known as *abstraction*.

Abstraction is essential to writing good code for at least two reasons. First, as we saw above, it eliminates redundancy, which reduces the scope for error and increases the value you can get from the code you write. Second, just as importantly, it makes code more readable.[2] A reader scanning one of the three code blocks above might easily miss their purpose. By contrast, a call to a function called `leaveout_mean` is hard to misunderstand.

Abstraction can be taken too far. If an operation only needs to be performed once, and the code that performs it is easy to read, we would not advise abstraction. Abstracting without a purpose can lead you to spend a lot of time dealing with cases that will never come up in your work.

When you do have a function you plan to use often, you should take the time to implement it carefully. One thing we have found helpful is the software engineering practice of "unit testing." This means writing a script that tests out the behavior of the function you've written to make sure it works as intended. For example, we might make some fake data and verify that the `leaveout_mean` calculates the leave-out mean correctly. An advantage of unit testing is that it allows you to safely change your function without fear that you will introduce errors that will break your code down the line. It also provides a convenient way to document how the function works: what inputs it requires, what inputs it will not accept, etc.

Abstraction is not just about code. It is relevant anywhere you find yourself repeating an operation. The principles in this chapter, for example, explain why word processing packages come with templates for standard document types like memos or reports. And these principles are

[1] In Stata, as with just about any program you are likely to use, it is easy to make a function portable and accessible anytime you use the program.

[2] In fact, we have found that the general version of a function is often easier to write as well as easier to read. (To see why, think about how much harder it would be to program a linear regression for a specific matrix of variables than for a general one.)

the reason why, rather than just repeatedly telling our RAs how we thought code should be written, we decided to write this handbook!

Chapter 7

Documentation

Rules

(A) Don't write documentation you will not maintain.

(B) Code should be self-documenting.

We have estimated the effect of television on potato chip consumption. To illustrate the pernicious consequences for society we wish to perform a welfare analysis, for which we will need to compute an elasticity. Fortunately, Jesse's dissertation studied the effect of a tax increase on demand for potatoes, from which we can back out the elasticity of demand.

Here is how a section of our Stata code might look:

```
* Elasticity = Percent Change in Quantity / Percent Change in Price
* Elasticity = 0.4 / 0.2 = 2
* See Shapiro (2005), The Economics of Potato Chips,
* Harvard University Mimeo, Table 2A.
compute_welfare_loss, elasticity(2)
```

Notice the helpful comments that provide a roadmap to the reader.

Many researchers we know spend a lot of energy haranguing themselves, their coauthors, and their research assistants to write more comments like the above, and in general, to carefully document the organization of their code and data outside of the scripts and data files themselves. You might expect us to say the same. After all, we love organizing things. But in this chapter we will try to convince you to document less, not more. To see why, we continue our story.

A few months after writing the first version of our script we return to our code to revise the analysis. We find the following:

```
* Elasticity = Percent Change in Quantity / Percent Change in Price
* Elasticity = 0.4 / 0.2 = 2
* See Shapiro (2005), The Economics of Potato Chips,
* Harvard University Mimeo, Table 2A.
compute_welfare_loss, elasticity(3)
```

Notice the conflict in red. Maybe someone noticed a typo in the original calculation, or decided to use the estimates from Table 2B instead of Table 2A of Jesse's dissertation. Whatever its origin, the problem is clear: the comments contradict the code, and it is now unclear which (if either) is correct. Someone will have to go back to the source to figure out what number we should be using.

Readers will notice that this is an instance of a more general problem: anytime you have more than one representation of the same information (in this case, an elasticity), you run the risk that the two will someday come in conflict. In the best case scenario, you will need to do some work to untangle the mess. In the worst case scenario, your results will be wrong or internally inconsistent.

The problem of internal inconsistency is especially severe when it comes to documentation—comments, notes, readmes, etc.—because you don't *have* to keep them up to date for your code to work or for your results to be quantitatively right. It is therefore tempting to make improvements to the code without making parallel improvements to the comments, only to find later that your comments are confusing or misleading. In the case above, the practice of letting comments go stale resulted in code that is probably less clear than it would have been if we had not had so much documentation in the first place.

To avoid such confusion, you will need to keep your comments up to date, meaning just as up to date as your code. If it's not worth maintaining a piece of documentation up to that standard, it probably isn't worth writing it in the first place (rule A).

That raises the important question of how to make the code clear without extensive comments. Imagine the selection above with no comments at all. How would a reader know why the elasticity is 2 and not 3?

To solve that problem we turn to the code itself. Much of the content of the comments above can be readily incorporated into the code:

```
* See Shapiro (2005), The Economics of Potato Chips,
* Harvard University Mimeo, Table 2A.
local percent_change_in_quantity = -0.4
local percent_change_in_price = 0.2
local elasticity = ‘percent_change_in_quantity’/‘percent_change_in_price’
compute_welfare_loss, elasticity(‘elasticity’)
```

This code block contains just as much documentation as the one we started with. It makes clear both the formula for the price elasticity and the quantitative components we are using. But it is far better than the original code, because it has far less scope for internal inconsistency. You can't change the percent change in quantity without also changing the elasticity, and you can't get a different elasticity number with these percent changes.

When possible, then, you should write your code to be self-documenting (rule B). Use the naming of variables and the structure of the code to help guide a reader through your operations. That's a good idea anyway, because even the best comments can't untangle a coding mess. To boot, writing such code will mean you don't have to write comments and other notes only to find that they have later lost their grip on what the code is really doing.

These principles apply far beyond code, and indeed they underlie many of the other chapters in this handbook. Organizing your data files so that their structure makes their meaning clear lets you avoid pairing every dataset you make with extensive documentation (chapter 5). Naming files, directories, and other objects intelligently means their names declare their function (chapter 4). A cleverly drawn figure or table will often say so much that notes are present only to confirm the obvious or clarify minor details. And so on.

Documentation does have its place. In the example above, if we don't include the citation to Jesse's stellar thesis, how will a reader know where 0.4 comes from? There is no (practical) way to script the link back to the original paper, so a comment is appropriate.

Documentation can be used to make clear that something is right when it at first may seem wrong. Suppose, for example, we have a variable y distributed lognormal with location μ and scale σ. If we wish to compute the log of the variable's expectation, it might be wise to write

```
* Log of the expectation is not the expectation of the log
* See http://en.wikipedia.org/wiki/Log-normal_distribution
log_expected_y = ‘mu’ + 0.5*(‘sigma’^2)
```

so that a reader isn't surprised that the expression is not simply $\log(E(y)) = \mu$. Of course, what to document is in the eye of the beholder: if you and your collaborators are not likely to forget the expression for the expectation of a lognormal, then the comment above is probably superfluous.

Documentation can also be used to prevent unintended behavior. Suppose you write a command to estimate a regression model via maximum likelihood. If two or more variables are collinear, your solver will iterate forever. So, you may wish to put a warning in the code: "Don't try to estimate an unidentified model." But be careful. As we note above, nothing documents code quite like code. Writing a function to test whether your $(X'X)$ matrix has full rank will provide just as much documentation, will not require the user to be conscientious enough to read the comments, and will likely lead to a faster resolution of the problem.

Which brings us to a related point. In Jesse's house there is a furnace room with two switches. One controls a light. The other turns off the hot water for the whole house. When he first moved in, people (let's not name names) conducting innocent business would occasionally shut off the hot water while fumbling for the light switch. He tried having a sign: "Do not touch this switch." But in the dark, in a hurry, a sign is worthless. So he put a piece of tape over the switch. If there are some inputs you really, really want to prevent, comments that say "don't ever do X" are not the way to go. Write your code so it will not let those inputs in the door in the first place.

Chapter 8

Management

Rules

(A) Manage tasks with a task management system.

(B) E-mail is not a task management system.

> From: Jesse Shapiro
>
> To: Matthew Gentzkow
>
> Re: potato chips
>
> Hey Matt,
>
> Do you have that robustness check where we control for the amount of ranch dip sold in each county? I am writing the section on dipping sauces and wanted to mention it.
>
> -Jesse

> From Matthew Gentzkow
>
> To: Jesse Shapiro
>
> Sorry, I thought you were doing that because it's similar to that other thing you were doing with controlling for salsa sales.
>
> Let me know if you want to do it or if you want me to take over.
>
> -MG

From: Jesse Shapiro

To: Matthew Gentzkow, Michael Sinkinson

I thought Matt was doing ranch dip and Mike was doing salsa?

-Jesse

From: Michael Sinkinson

To: Matthew Gentzkow, Jesse Shapiro

I did the salsa robustness check two weeks ago. See my e-mail from 8/14, 9:36am.

-Mike

From: Jesse Shapiro

To: Michael Sinkinson, Matthew Gentzkow

Right, but in that e-mail you were controlling for the log of salsa consumption. I thought we agreed we wanted the level of consumption?

-Jesse

From: Michael Sinkinson

To: Jesse Shapiro, Matthew Gentzkow

On it!

-Mike

What's wrong with this picture? Mainly, it's ambiguity. Mike thought his task was done, when Jesse thought it was not. Matt thought Jesse was working on the ranch dressing task, but Jesse thought Matt was doing it. In fact, a careful reader will notice that even after all that e-mail, it's *still* not clear who is going to do the ranch dressing robustness check!

It's worse than that. If we come back to the salsa task in two weeks, where will we look to find out its status? This thread? The one Mike mentions from 8/14? And how will we reference our discussion? By date? By forwarding this whole thread, including all the extraneous exchanges about ranch dressing?

If you work alone, these problems are small. You probably have a legal pad or a Word document or a spot on your whiteboard where you keep track of what you need to do. Every now and again you might forget what you were planning to do or where you jotted something down, but if you are organized you probably get by ok.

The minute two people need to work together, however, the problems exemplified in the thread above are big. And although we haven't proved this formally, we think they grow more than arithmetically with the number of people (coauthors, RAs, etc.) involved in a project.

Software firms handle project and task management systematically. Microsoft does not just say, "Hey Matt, when you get a chance, can you add in-line spell-checking to Word?"

Rather, enterprises engaged in collaborative work use project and task management systems that enforce organized communication and reporting about tasks. In the old days, those often involved handing physical reports up the chain of command. Now, they increasingly involve the use of browser-based task-management portals.

In one of these portals, Mike's salsa task would have looked like this:

Task: Salsa Robustness Check

Assigned To: Michael Sinkinson

Assigned By: Jesse Shapiro

Subscribed to Comments: Matthew Gentzkow

Status: Completed.

Description:

Run main specifications adding a control for per capita salsa consumption.

Add a line to our robustness table reflecting the results.

Comment by: Michael Sinkinson

On it!

Comment by: Michael Sinkinson

See the new version of the paper posted in /drafts/Potato Chips and the supporting code in /analysis/Potato Chips. Is this what you had in mind?

Comment by: Jesse Shapiro

Almost. Our econometric model implies that salsa consumption should enter in levels not logs. Can you revise?

Comment by: Michael Sinkinson

Ok, how about now?

Comment by: Jesse Shapiro

Yup, looks good.

Completed By: Michael Sinkinson

Notice that now there is no ambiguity about whose responsibility the task is or what the goals are. Anyone looking at the task header will know that Mike is expected to do it, and no explicit communication is needed to figure out who is doing what.

There is also a natural place to store communication about the task. Everyone expects that questions and answers will be posted to the appropriate task. And, weeks, months, or years later, there will still be a task-specific record of who did what and why.

There are lots of good online systems for task management available at the moment that look something like the example above. Many are free or at least have a free no-frills option. Most have apps for mobile devices and offer some kind of e-mail integration so, for example, Mike's comments above would be e-mailed to Jesse so he knows there's something he needs to look at.

These systems are changing all the time and which one you want is a matter of taste, style, budget, and the like, so we won't review them all here. Good free options as of this writing include Asana (www.asana.com), Wrike (www.wrike.com) and Flow (www.getflow.com). We use a program called JIRA, which is not free and requires a little more work to install.

While we're on the subject of useful tools, you should probably get yourself set up with some kind of collaborative note-taking environment. That way, you're not bound by the limitations of your task management system in what you can share or record. It's helpful to have a place to jot down thoughts or display results that are less structured than the code that produces your final paper, but more permanent than an e-mail or conversation.

The best system is one that lets you easily organize notes by project and share them with other users. It's great if you can add rich attachments so you can show your collaborators a graph, a code snippet, a table, etc.

There are a bunch of options, and again, many are free. Evernote (www.evernote.com) has a free basic option and is available across lots of platforms and interfaces. Another option for Windows users is OneNote, which is included with Microsoft Office.

For Further Reading

The ideas in this handbook are not new. The chapters are an attempt to communicate well-trod ideas from software engineering and computer science to a social science audience.

Here we list additional resources that have influenced our thinking. You may find these helpful if you wish to see some of the topics we have covered in greater depth.

Note that the single best resource we know of is not a book, but a website called software carpentry (http://software-carpentry.org/) devoted to teaching computing to scientists.

Bowman, Judith S., Sandra L. Emerson and Marcy Darnovsky. 2001. *The Practical SQL Handbook: Using SQL Variants.* New York: Addison-Wesley. [Chapter 2 contains a nice overview of database design.]

Brooks, Frederick P. 1975. *The Mythical Man-month.* Reading: Addison-Wesley.

Immon, William H. 2005. *Building the Data Warehouse*. New York: Wiley.

Martin, Robert C. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship.* New York: Prentice Hall.

Lutz, Mark and David Ascher. 1999. *Learning Python.* New York: O'Reilly Media.

Acknowledgments

We benefited greatly from the input of coauthors and colleagues on the methods described in this handbook. Ben Skrainka's Institute for Computational Economics lecture slides showed us that we were not alone.

Most importantly, we acknowledge the tireless work of the research assistants who suffered through our obsessions and wrong turns.