Managing Your Research*

Richard Stanton†

March 14, 2023

Abstract

In any research project, you need to keep track of, and be able to reproduce, complex analyses and their results (e.g., figures and tables) over a period of years or even decades. Using the right tools to manage your research makes a huge difference. This document describes some useful tools and methods for making your research more efficient and organized.

*Comments and suggestions on this document are welcome.

†Haas School of Business, U.C. Berkeley, 545 Student Services Building #1900, Berkeley, CA 94720-1900, phone: 510-642-7382, fax: 510-643-1412, e-mail: rhstanton@berkeley.edu.

Contents

1 Introduction

It doesn't much matter what tools you use to write a one-sentence email you'll never look at again, or a program whose only job is to display "Hello, world!" on the screen. However, research projects are *never* quick or simple, even if they start out seeming that way. As a result, it's important to invest some time up-front to avoid lots of repeated or unnecessary work later.

The most important consideration is that your work needs to be *reproducible*. People usually use this word referring to reproducibility by others, but much more important is that your work should be easily reproducible by *yourself*. For example, a referee might ask you to make some changes to Figure 2, which you created 6 months ago through multiple rounds of cleaning, modifying and processing several different data sets. Or, for a new project, you might find yourself needing to repeat some analysis from a 10-year-old paper.

Other important things to consider include working with coauthors who want to edit the paper at the same time as you do, without ever running the risk that anyone's edits will be lost.[1] You also want to be efficient, especially when your code takes a long time to run. This means not running long analyses more often than necessary, and avoiding writing the same code 15 times. And finally, one day you'll drop your laptop into a swimming pool. Or somewhere worse. Even if having to replace your laptop is painful, having to replace 10 years of work is an awful lot worse.

In any complex, long-lasting research project, the tools you use make an enormous difference. This document provides a quick survey of some useful tools and methods that I recommend for making your research more efficient and organized. Some other documents with similar goals are Healy (2020) and Broman (2016). While we don't agree on all of our recommendations, we do agree on the important issues that you need to think about; being aware of these issues is much more important than the exact choices you make.

2 Version control using Git and GitHub

Using Dropbox (or something similar) helps a great deal with keeping track of the most recent version of a paper (or tenure report or computer program) you're writing with one or more coauthors. Many conflicts can be avoided by having only one person edit a file at a time, but there are some significant drawbacks with this system:

- Especially with multiple files, it's easy to forget whose turn it is. What happens when

[1] This includes not only new edits, but also restoring text from an old version of the paper that everyone agreed 2 weeks ago should be deleted, but you've now changed your minds.

your coauthor (for the third time this week) accidentally edits the version of the file you sent her four days ago instead of the version containing all of your latest edits?
- How do you clean up the mess when you and your three coauthors all edit the same file at the same time in the rush to get something out for a conference deadline?
- What happens when you remember there was a section (or a Python function) in the version you presented at the WFA meetings six months ago, which is no longer in the active directory, but which you'd like to retrieve?[2]
- What happens when you realize that your code now produces results that differ from those from a year ago, but shouldn't? How do you track down what changes to the code during the last year caused the change in behavior?

A **Version Control System** (VCS) solves all of these problems. In particular:

- It allows you to keep track of every revision of every file that's ever existed, along with a log of what changed from one revision to the next. If you want to go back to a prior version, or see what's changed since two years ago, it's almost trivial.
- It makes it easy to merge simultaneous edits by different coauthors, even if they each started editing different revisions, so you don't have to worry any more about editing files sequentially. Edit whatever file you want whenever you want.

The first advantage makes using a VCS a good idea even when there's only one author, but it's particularly valuable when there is more than one.[3] There are numerous VCS to choose from, but I recommend using the most popular, third-generation distributed VCS, Git. An excellent introduction to using Git is "Version Control by Example" by Eric Sink, available at https://ericsink.com/vcbe/vcbe_a4_lo.pdf. A more advanced reference is "Pro Git" by Scott Chacon and Ben Straub, available at https://git-scm.com/book/en/v2. Some useful compact "cheat sheets" include

- https://education.github.com/git-cheat-sheet-education.pdf
- https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

[2]This problem is commonly "solved" by saving lots of versions with clever descriptive names like
- `program_WFA2011.m`
- `program_remember_this_as_I_might_want_to_go_back_to_this_version.m`

However, after a while there are so many files that you can't find the one you want in the crowd, and you'll eventually forget what those names, which seemed so clear when you first came up with them, mean.

[3]I've been using a VCS to keep track of every file I care about for over 20 years, starting with RCS (a "first generation" VCS), then CVS (a "second generation," centralized VCS, or CVCS), Mercurial (a "third generation," distributed VCS), and most recently Git.

2.1 The basic editing cycle

Similar to using Dropbox, when using a VCS each author edits and keeps track of his or her changes locally, and periodically "pushes" those changes to a shared central repository to keep everything synchronized. The basic editing cycle is the same whether you are using a VCS or Dropbox, though the specific commands differ:

	Activity	**Dropbox**	**Git**
1)	Make sure your local files are up to date	Copy from shared Dropbox folder to working directory	**Pull** remote repository with `git pull`
2)	Edit local files	Edit local files	• Edit local files. • **Commit** changes using `git commit`
3)	Make your changes public	Periodically copy edited files to shared Dropbox folder.	Periodically **push** changes to central repository using `git push`

If everyone always edited a fixed set of files sequentially, never forgetting to update their local files before editing, the commands above would be pretty much all you would ever need. An editing session would look something like this:

```
$ git pull

  [edit files]

$ git commit -am 'My changes'
$ git push
```

In practice, things can get more complicated, e.g., when both you and your coauthors have edited the same file at the same time. Without version control this is very hard to sort out, but Git makes it (relatively) straightforward. For details, see the references above.

2.2 GitHub

There are several public hosting services that make it extremely simple to host your central repository. The most widely used is GitHub, and I recommend you start by setting up an account there. The repositories you host there can be either public (open to everyone) or private (accessible only to those you explicitly designate).

2.3 Git interfaces

The sample editing session above showed the use of git at the command line. It is perfectly possible to use Git this way (and I often do so). However, there are lots of alternative interfaces available. In particular, a number of editors/IDEs (e.g., VS Code) have Git support built in, making it easy to commit and push your edits from the same environment you use to edit your files. Alternatively, there are Git-specific graphical interfaces available, such as GitKraken and GitHub Desktop. Experiment with a few alternatives and see which you like best. I tend to handle the commonest Git interactions from within my editor (GNU Emacs), but use the command line when I need to do something a little less common.

2.4 Using Git with Jupyter notebooks

Using version control with Jupyter notebooks is a little tricky, because every time you rerun the code in your notebook the file changes, even when you have made no changes to any of the code. nbstripout solves this problem by stripping out all the output before committing a change or checking whether a notebook has changed. It can be installed so it runs automatically every time you use Git to commit changes.

3 LaTeX

While you *could* write your research papers using MS Word, don't! Use LaTeX, a document typesetting system based on Knuth's program TeX (Knuth, 1984a). LaTeX makes it relatively simple to produce papers with high-quality equations and error-free cross-references to equations, tables, and figures, and to handle creation and formatting of bibliographies and citations.[4]

3.1 Writing Finance Papers using LaTeX

A few years ago, I wrote a short introduction to LaTeX called "Writing Finance Papers using LaTeX" (Stanton, 2013). This is aimed at finance PhD students and highlights some tools and techniques I have found most useful. It can be found at `http://faculty.haas.berkeley.edu/stanton/texintro/index.html`. It includes instructions and sample files

[4] Also, unlike word-processor files, LaTeX files are plain text with no special control codes. This makes them easily portable from one machine (or architecture or OS) to another, easy to view using any plain-text file viewer, and also more likely to be rescuable if the file accidentally gets corrupted. This also makes it easy to keep track of your work using a Version Control System (see Section 2).

for formatting your papers to match the requirements of *Journal of Finance*, *Journal of Financial Economics*, and *Review of Financial Studies*.

3.2 Other references

If you're starting from scratch, one very popular reference, free to download, is the "Not So Short Introduction to LaTeX 2_ε" (Oetiker, Partl, Hyna, and Schlegl, 2021). Other standard references include Lamport (1994), written by the original author of LaTeX; Kopka and Daly (2003), a more recent, popular introductory text; and Mittelbach, Goossens, Braams, Carlisle, and Rowley (2004), an excellent reference on both LaTeX itself and many of the additional packages that have been created since LaTeX was created (the authors of this book are some of the main developers of LaTeX).

3.3 Overleaf

Overleaf (https://www.overleaf.com/) is a Web site that allows multiple people to edit a shared LaTeX document at the same time. Because everything is online, it is particularly useful for sharing documents with coauthors who prefer not to have to worry about using Git or installing and running LaTeX on their own machines. It even works pretty well with coauthors who don't know LaTeX at all — you take care of all the LaTeX stuff and they can just edit the text. For more information, see https://www.overleaf.com/for/authors.

If you want to share a document with others using Overleaf but you prefer working on your local machine with Git (I do, because I prefer editing in GNU Emacs rather than online using Overleaf), you can combine the two by using Overleaf as the remote repository for your project instead of GitHub. To do this, once you've created a repository for your paper on Overleaf,

- Go to Overleaf and open the paper.
- Click on Menu -> Git and copy the "git clone" command, e.g.,

  ```
  git clone https://git.overleaf.com/5e1e5dc63fe82300013bf0e7
  ```

- On your computer, go to the directory above where you want the paper to go (e.g., `c:\projects`), and paste the command above, followed by the name you want for the project directory, e.g.,

  ```
  $ git clone https://git.overleaf.com/5e1e5dc63fe82300013bf0e7 myproject
  ```

  This will create a directory `c:\projects\myproject` containing a git repository that is linked to the Overleaf document. When you run `git push` it will push your edits

to the Overleaf document (and if your coauthors are connected to Overleaf, they'll see your changes suddenly appear in front of them).

3.4 Using Overleaf and GitHub together

Overleaf is useful for sharing papers with coauthors, but its storage limits make it unsuitable as the main repository for a project's programs and data. My preferred solution is to keep the main project repository on GitHub, with the paper a `submodule` of the main repository, linked to a repository on Overleaf. This is similar to setting up Overleaf as your remote git repository, as described above. First, set up a local repository (in directory `project1`, say) linked to a remote repository for your project on Github. Now go to the `project1` directory and create a *submodule* containing just the write-up. To do this, as above, on Overleaf, click "Menu" → "Git" and copy the URL that appears after "git clone", e.g.,

```
https://git.overleaf.com/5e1e5dc63fe82300013bf0e7
```

Now type the command

```
$ git submodule add https://git.overleaf.com/5e1e5dc63fe82300013bf0e7 writeup
```

This will create a directory `project1/writeup` containing a git (sub)repository linked to the Overleaf document. When you run `git push` in directory `project1`, it will push your edits to the repository on GitHub. When you run `git push` in the subdirectory `writeup`, it will push the changes to just the write-up to the Overleaf document.

4 Use Python!

For many years, since first learning the language for a consulting project while a PhD student, I did almost all of my coding in C.[5] This gave me lots of speed and (eventually, after lots of hunting down code written by others or writing my own) lots of routines for doing common numerical/scientific/statistical operations. However, having to recompile the code every time you make a change makes exploring data a slow process, and C is *horrible* for reading, cleaning and manipulating external data. I kept thinking about switching to other language(s), such as Matlab, but the speed of C kept me from doing so until about 10–15 years ago, when I realized that Python gives me *almost* everything I want:

- A general purpose programming language with a large user base and lots of contributed routines to handle common numerical/scientific/statistical tasks.

[5] At the time I chose C, I had experience writing code in a number of other languages as well, including FORTRAN, Basic, Gauss, APL, PL/1, and assembly language for the Intel 8080 and Zilog Z80 microprocessors.

- Good data handling capabilities (in Python this means using the excellent `pandas` package, written by Wes McKinney).
- It's an interpreted language, which speeds up writing and debugging code.
- While code written with lots of loops in raw Python can be quite slow, certainly compared with C, there are a number of ways of massively speeding things up that take only a little extra work. For example:
  - Vectorizing your code (e.g., using the `numpy` library).
  - Using a just-in-time compiler (e.g., via the `numba` package).
  - Converting your code automatically to C and compiling it (via `Cython`).

Other languages may beat Python on one or two dimensions, but Python is close enough on all of them that I am happy using it for almost everything that I do.

4.1 Installing a working Python distribution

I use and recommend the Anaconda Python distribution, which provides a complete (and free) distribution of Python with almost every important library preinstalled. To install it, go to https://www.anaconda.com/products/individual and follow the instructions to download and install the default version (currently 3.9) for your operating system.

4.2 References

Here are some useful references on using and learning Python:

- Pilgrim, Mark, 2011, *Dive into Python 3* (Mark Pilgrim), https://diveintopython3.problemsolving.io/. Python for people who already know how to program.
- van Rossum, Guido, 2021, The Python tutorial, https://docs.python.org/3.9/tutorial/. The "official" Python tutorial.
- McKinney, Wes, 2018, *Python for Data Analysis*, second edition (O'Reilly Media, Inc., Sebastopol, CA) This book covers the `Pandas` package (written by McKinney), which is excellent for anything to do with data.
- McKinney, Wes, and the PyData Development Team, 2021, *Pandas: Powerful Python Data Analysis Toolkit*, Release 1.2.3, available at http://pandas.pydata.org/. The `Pandas` manual.

Additional useful references include

- The QuantEcon Web site has several useful sets of lectures on Python programming for economics and finance, written by Thomas J. Sargent and John Stachurski, and including large amounts of working code:

  - "Python Programming for Economics and Finance", https://python-programming.quantecon.org/intro.html.
  - "Quantitative Economics with Python", https://python.quantecon.org/.
  - "Advanced Quantitative Economics with Python", https://python-advanced.quantecon.org/.

- Sheppard, Kevin, 2021, Introduction to Python for econometrics, statistics and data analysis, 5th edition, Online book, Oxford University, https://www.kevinsheppard.com/teaching/python/notes/. This has lots of nice econometric code examples, e.g., GARCH or OLS with Newey and West (1987) standard errors. It also discusses speeding up mission-critical sections of your code using tools like `numba` and `Cython`, the existence of which is one of the most important reasons I use Python.
- For Matlab users, translating Matlab functions into Python equivalents:
  - http://www.scipy.org/NumPy_for_Matlab_Users/.
  - http://mathesaurus.sourceforge.net/matlab-python-xref.pdf

4.3 The Jupyter notebook interface and pandas

The notebook interface is a great way to work productively in Python, especially for playing around with data.[6] Probably the best way to learn how to use it productively (as well as learning the features of Pandas) is to work all the way through Wes McKinney's video tutorial (about three hours) at http://www.youtube.com/watch?v=w26x-z-BdWQ. The short version of getting started, once you've got everything installed, is to open a command prompt and execute the command

```
$ jupyter notebook
```

You can learn how to use Pandas in more detail from McKinney (2018).[7] This book and the video cover some of the same examples, but the video is more interactive, so you can see the process of solving data analysis problems using Python, Pandas, and the Jupyter notebook interface. Finally, a *very* short introduction to the main features of Pandas can be found at Pandas Development Team (2021).

[6] For many years, I never used a notebook-like interface for anything if I could possibly avoid it, preferring to write and debug code inside an Emacs buffer. However, while you can write Python code that way if you like, or using an Integrated Development Environment (IDE) such as Visual Studio Code (https://code.visualstudio.com/), I've found that the Jupyter interactive notebook interface (which you can use inside Emacs if you like, using the excellent EIN package) makes program development so easy that I almost never need to use a debugger (though there is one if you want).

[7] Data to accompany the book can be downloaded from https://github.com/pydata/pydata-book.

4.3.1 Using Jupyter notebooks with other languages

Jupyter notebooks can also be used to develop and run code in languages other than Python, including

- R (see https://github.com/IRkernel/IRkernel).
- Julia (see https://github.com/JuliaLang/IJulia.jl).
- Stata (see https://kylebarron.dev/stata_kernel/).

4.4 Using Jupyter notebooks outside the classical Web interface

You may prefer to interact with your Jupyter notebooks without using the standard browser-based interface. For example, editing in a real editor is a lot more pleasant than in a Web browser, and using an IDE makes it relatively painless to use a debug your code. Examples include

- PyCharm.
- VS Code.
- GNU Emacs (using the EIN package).

4.5 Statistics

An important statistical package is `statsmodels`.[8] An example of running a simple OLS regression (taken from the documentation) is:

```
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

# Load data
dat = sm.datasets.get_rdataset("Guerry", "HistData").data

# Fit regression model and print results
results = smf.ols('Lottery ~ Literacy + np.log(Pop1831)', data=dat).fit()
print(results.summary())
```

The output from this regression looks like this:

[8] For details see https://www.statsmodels.org/stable/index.html.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                Lottery   R-squared:                       0.348
Model:                            OLS   Adj. R-squared:                  0.333
Method:                 Least Squares   F-statistic:                     22.20
Date:                Mon, 05 Apr 2021   Prob (F-statistic):           1.90e-08
Time:                        16:53:49   Log-Likelihood:                -379.82
No. Observations:                  86   AIC:                             765.6
Df Residuals:                      83   BIC:                             773.0
Df Model:                           2
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept      246.4341     35.233      6.995      0.000     176.358     316.510
Literacy        -0.4889      0.128     -3.832      0.000      -0.743      -0.235
np.log(Pop1831) -31.3114      5.977     -5.239      0.000     -43.199     -19.424
==============================================================================
Omnibus:                        3.713   Durbin-Watson:                   2.019
Prob(Omnibus):                  0.156   Jarque-Bera (JB):                3.394
Skew:                          -0.487   Prob(JB):                        0.183
Kurtosis:                       3.003   Cond. No.                         702.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

4.6 Other languages

4.6.1 Stata

Many people in finance and economics use `Stata`. I use it use as little as possible, as I find it annoying to code in, but it is certainly very convenient for a lot of standard analyses. One particular package that is very useful — and does not currently have an equivalent in Python — is `reghdfe` (Correia, 2017), which allows the estimation of regressions with very large numbers of fixed effects without creating so many dummy variables that your computer runs out of memory.

4.6.2 Julia

`Julia` is the only other language I've seen that I might at some point consider switching to from Python. It has many features in common with Python, but one significant difference is that it *compiles* everything before running. This makes Julia very fast once you've run or

compiled everything once, but development can take a bit of getting used to, as it's quite a bit slower to run a program the first time. It has a very nice equivalent to Stata's `reghdfe` package, which runs a lot faster.

The QuantEcon quantitative economics notes mentioned above are also available in a Julia version at https://julia.quantecon.org/.

4.6.3 Other languages to consider

- R
  - Less used in finance, but popular in many other fields.
  - Lots of standard statistical estimations included.
  - Not very fast for numerical work.
- Matlab
  - Quite widely used in finance, but proprietary and expensive (though see GNU Octave).
  - Very easy for matrix calculations (hence the name).
  - Large number of statistical and other packages.
  - Not a general-purpose language — data manipulation not particularly convenient.
- SAS
  - Proprietary and not available on macOS.
  - Lots of standard analyses built-in.
  - Can handle datasets larger than computer memory.
  - Nancy uses it.
  - Incredibly clunky to code in!

5 Make your research reproducible!

The analysis for any project will often involve multiple steps and running multiple different programs. It can be easy even after just a few days to lose track of exactly what programs need to be run, in what order, on what input files, with what command-line options, to create your working data files, perform the analysis, and produce the output figures and tables. Even worse, imagine coming back to a project 6 months or a year later and trying to recreate a particular figure with slight modifications to satisfy a referee.

5.1 GNU Make

The Unix `make` utility is designed to solve almost exactly this problem. It was originally written to manage large software programs, keeping track of

1. What files need compiling and what libraries need linking to create the executable.
2. Making sure that you're always using the most recent version of every source file.
3. Recreating the executable efficiently; in particular, not recompiling things that haven't changed since you last compiled them.

This is very similar to the tasks you want to take care of in a big data-analysis project:

- Remembering what raw data files and source files are needed (in what order) to create your working data sets.
- Remembering what data and source files are needed to run each regression.
- Not rerunning regressions where you've already run the most recent version of the code on the most recent version of the data.

To use the `make` utility, you create a file called `Makefile` that summarizes all the rules for creating every figure, table, dataset, etc. Very importantly, `make` is not tied to a particular language, so it's perfectly happy if the rule to create a working data set involves running a Python script followed by a Stata DO file. Once you've created the `makefile`, to run all the regressions for a paper, recreating data sets if needed, you'll type a simple command like

```
$ make regressions
```

There are several different versions of `make`. I use `GNU Make` (see https://www.gnu.org/software/make/manual/make.html), which has some specific features I find useful, such as being able to write a rule to handle the common situation where running a single program file creates multiple output files.

Note that to use `GNU Make`, all your code needs to be run from the command line. This is not as much of a drawback as it might seem at first. It's much easier to make your work fully reproducible — by yourself or by others — if you can write down a set of instructions for doing so that can be run from the command line. "Click the left mouse button on the third menu item from the left" cannot be 100% relied on to do the same thing 10 years from now as it does today. In particular, this implies that the final version of your Python code should be an executable .py file, *not* a notebook file.[9]

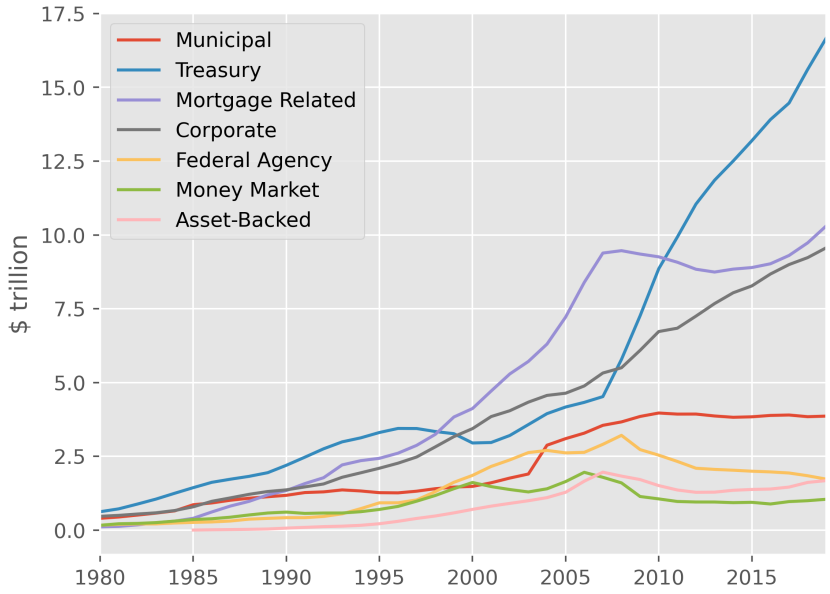
[9] Of course, a notebook is a great way to experiment.

5.1.1 References

For more information on using make to manage your research workflow, see Baker (2020), Jones (2013), or the GNU Make manual.

5.1.2 Example

Suppose we want to create the following plot of bond-market sizes over time from the spreadsheet `data.xls`, which has been downloaded from https://www.sifma.org/wp-content/uploads/2017/06/CM-US-Fixed-Income-SIFMA.xls.



Here is the Python code used to generate the graph and save it as file `plot.png`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use("ggplot")

lastYear = 2019

# Load data
df = pd.read_excel("data.xls", sheet_name="Outstanding", skiprows=3, index_col=0)
df.columns = ["Municipal", "Treasury", "Mortgage Related", "Corporate",
              "Federal Agency", "Money Market", "Asset-Backed", "Total"]
```

```
12
13 # Keep annual data up to 2019 and drop Total
14 df = df.iloc[0 : np.argmax(df.index == lastYear) + 1].drop(["Total"], axis=1)
15
16 # Plot
17 fig = plt.figure(figsize=(8, 5))
18 ax = (df / 1000).plot()
19 ax.set_xlim([1980, lastYear])
20 ax.set_ylabel("$ trillion")
21 plt.savefig("plot.png", dpi=400)
```

Creating the graph involves two tasks:

1. Download the remote file https://www.sifma.org/wp-content/uploads/2017/06/CM-US-Fixed-Income-SIFMA.xls and save it as local file `data.xls`.
2. Run `plot.py` to generate the plot and save it as `plot.png`.

Assuming both of these are time-consuming activities, we don't want to run them more than necessary. In other words, we only want to download the data if `data.xls` does not already exist on the disk, and we only want to run the Python code if `plot.png` either doesn't exist or is "out of date," which we define to mean `plot.png` is older than `plot.py` or `data.xls`.

Our `Makefile` will contain rules for making two *targets*: the data file `data.xls` and the graph file `plot.png`. Each target's rule specifies

- Its `prerequisites`: What other files it depends on.
- A recipe for creating the target, if necessary.

Here is a simple `Makefile` for our plot:

```
1  # Sample Makefile
2
3  # Definitions
4  PYTHON = python
5  WGET = wget -q
6
7  # Download data file if not present, and update time stamp
8  remotehost = https://www.sifma.org/wp-content/uploads/2017/06/
9  remotefile = CM-US-Fixed-Income-SIFMA.xls
10 data.xls:
11         $(WGET) $(remotehost)$(remotefile)  -O $@
12         touch $@
13
```

```
# Plot graph unless it is newer than data.xls and plot.py (and Makefile)
plot.png: plot.py data.xls Makefile
	$(PYTHON) $<
```

Note the use of some important automatically defined variables here:

- `$@` expands to the name of the target file
- `$<` expands to the name of the first prerequisite

Now we can run

- `make data.xls` to create the data file (if necessary).
- `make data.png` to create the graph (if necessary).
  - This also creates `data.xls` if needed, since this is a prerequisite of `data.png`.

Note that everything needs to be run from the command line. However, this is *not* really a drawback. After all, would you really want to rely 10 years from now on a rule that says something like

> "Click the left mouse button on the third menu item from the left"?

Also note that your final Python code should be an executable `.py` file, *not* a notebook file.[10]

5.1.3 Some suggestions for writing useful `Makefiles`

Use loops to shorten your code You can do a fair amount of programming inside a Makefile, which can make your code much shorter and less likely to contain errors. For example, I recently used this function to generate a long list of target names for a Makefile. Passed two lists of names, it generates a list of all pairs of names from list 1 followed by names from list 2, with specified text before, between, and after each pair of names.

```
define gen_names_2d
	$(foreach a,$(1),$(foreach b,$(2),$(3)$(a)$(4)$(b)$(5)))
endef
```

[10] Of course, a notebook is a great way to experiment.

5.1.4 Program files that generate multiple output files

GNU make is mainly designed to have a single "target" file for each rule, along with a recipe for creating that target. However, a single Stata DO file might create 10 or 20 different output files, and you don't want to run it 10 or 20 times. GNU Make allows you to handle this situation using code like this:

```
target1 target2 target3 &: doFile.do dataFile.dta
        $(STATA) $<
```

This rule says that the recipe should be invoked (just once) if it needs to recreate any of the files `target1`, `target1`, or `target3`.[11]

5.2 Embedding Python code in your document

An alternative (not necessarily mutually exclusive) approach is to embed your source code (Python, C, Matlab, Stata, etc.) directly *inside* your document. Keeping the code next to the figures and tables that it generates makes it much easier to keep track of what does what, and also makes it easy for other researchers, and you, to reproduce your results later (reproducible research). In addition, combining the text and the code makes both easier to understand (literate programming — see Knuth, 1984b). There are several ways of doing this; I shall describe two that I have found useful.

5.2.1 PythonTeX

`PythonTeX` (Poore, 2013, 2015) is a very convenient way to embed Python code in your LaTeX document. You can display the Python code if you like, and you can refer to the results of Python calculations directly from your LaTeX document. It is not great for writing and debugging your code, but it keeping your final code embedded in the document means you don't forget what you did. I find `PythonTeX` particularly useful for problem sets and exams, since doing all the calculations in Python and embedding them directly in the text means

1. You don't need to transcribe your results (with the attendant risk of error),
2. You can easily adjust the displayed precision,
3. It's trivial to change the numbers in the question without any extra work.

Figure 1 shows a sample LaTeX document including some Python code, with its output.

[11] This is a GNU Make-specific functionality, and actually requires a relatively recent version even of GNU Make.

```latex
\documentclass[12pt]{article}
\usepackage[makestderr]{pythontex}

\begin{document}
\centerline{\bf\Large PythonTeX Example}
\bigskip
\noindent Inline calculation using Python: $3^4 = \py{3**4}$. Next, a Python code block:
\begin{pyblock}[][numbers=left]
a = 3
b = 4
c = a + b
\end{pyblock}
Refer to Python variables: $c = a + b =\py{a} + \py{b} = \py{c}$. Then more calculations:
\begin{pyblock}[][numbers=left]
d = c + 1
\end{pyblock}
Result is $d = \py{d}$.

\end{document}
```

(a) LaTeX source

PythonTeX Example

Inline calculation using Python: $3^4 = 81$. Next, a Python code block:

```
a = 3
b = 4
c = a + b
```

Refer to Python variables: $c = a + b = 3 + 4 = 7$. Then more calculations:

```
d = c + 1
```

Result is $d = 8$.

(b) Output

Figure 1: Pythontex example

Although this document is written in LaTeX, processing it does require use of the external program `pythontex`, which means that this document will not work properly in `Overleaf`. If you really want to, you *can* run Python code inside your LaTeX document on Overleaf, albeit a bit less conveniently. Figure 2 shows a simple example. Note that Overleaf's Python installation is pretty basic, so you don't have access to many Python packages. If you need this, you will need to edit and compile your document on your local machine.

5.2.2 Emacs and org

Another method I find useful for keeping my documents and code together is to use the GNU Emacs package `org-mode`. GNU Emacs is a text editor, but it's a lot more than that. I have used Emacs as my main editor for over 35 years, and I have at various times used it to read and write email, edit and compile LaTeX, edit, debug and run C and Python code, take notes, and lots more.

I've recently started using the Emacs package `org-mode` in my teaching and research. This package enormously extends what you can do with Emacs. Out of the box, `org-mode` is great for taking notes, managing to do lists, and many other things. org files are plain text and use a markdown syntax similar to many others you'll have seen. For example, here's a simple org file:

```
#+include: "~/.org/setup"
#+title: A simple org file

* Section 1
Here's the first /section/. Let's put in a list:
- Item 1
- Item 2
** Subsection 1
Here's the first *subsection*.
```

One very nice feature of `org mode` is that an org file can be exported to many output formats, such as

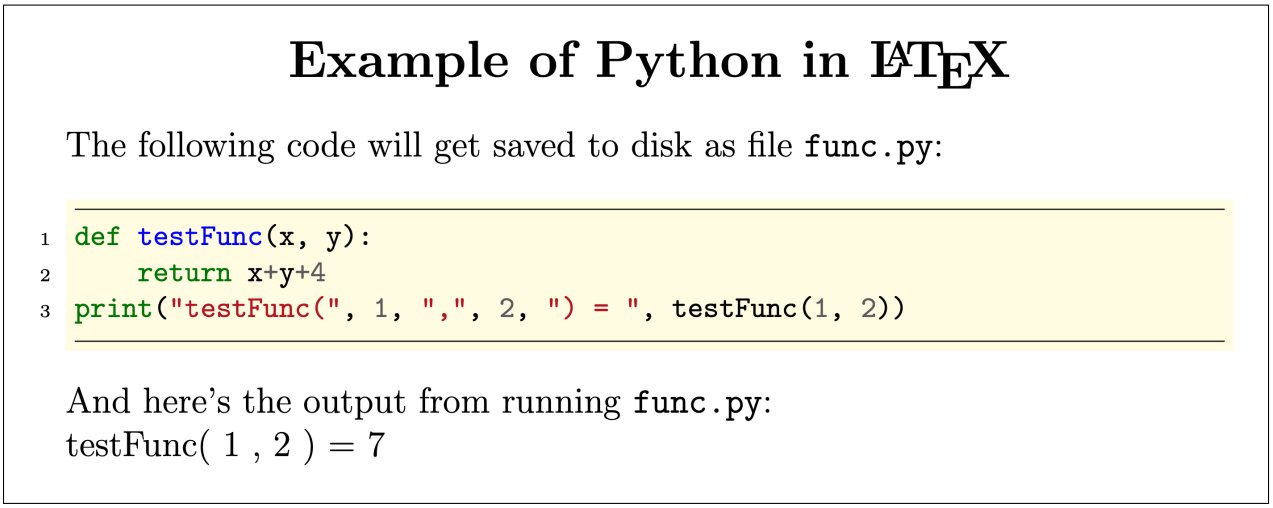
- LaTeX (and then PDF).
- Beamer (and then PDF).
- HTML.

Some examples are shown in figure 3.

One of the most useful features of `org mode` is that it allows you to include *and run* source code (Python, C, Matlab, Stata, etc.) inside a document, along with text, LaTeX equations,

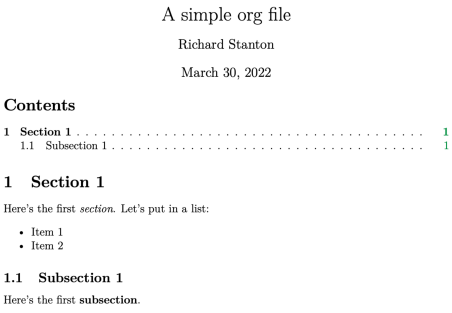
```latex
\documentclass[12pt]{article}
\usepackage{minted}
\newmintedfile[RSpython]{python}{}

\begin{document}
\centerline{\Large \bf Example of Python in \LaTeX}
\bigskip
\noindent The following code will get saved to disk as file \texttt{func.py}:
\begin{filecontents*}[overwrite]{func.py}
def testFunc(x, y):
    return x+y+4
print("testFunc(", 1, ",", 2, ") = ", testFunc(1, 2))
\end{filecontents*}
\RSpython{func.py}% Print code
\noindent And here's the output from running \texttt{func.py}:\\
\input{|python func.py}
\end{document}
```

(a) LaTeX source

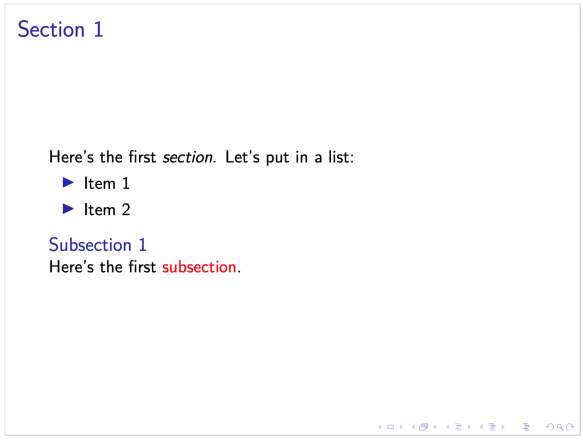


(b) Output

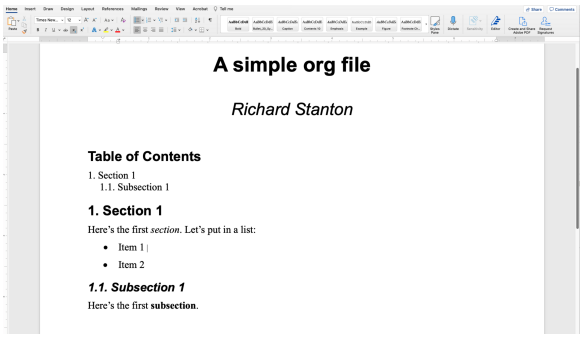
Figure 2: Manual inclusion of Python code in LaTeX document



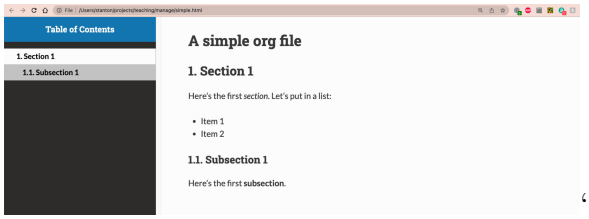
(a) PDF (via LaTeX)



(b) PDF (via Beamer)



(c) MS Word



(d) HTML

Figure 3: Examples of exporting simple org file

data and the results (e.g., figures and tables) of running the code. For a detailed description of how `org-mode` can help with both reproducible research and literate programming, see Schulte, Davison, Dye, and Dominik (2012); Schulte and Davison (2011); Stanisic, Legrand, and Danjean (2015). To see many different things you can do with `org-mode` in practice, with lots of examples, see this video by John Kitchin.

Thus, you can think of an org document as being like a Jupyter notebook that can also generate the paper (and Beamer slides and HTML)! Here is a somewhat more complex org file that demonstrates some of these features:

```
#+title: *org* document with live code, equations, and conditional text
#+OPTIONS: toc:nil num:2 h:1
#+include: setup
#+BIND: org-latex-image-default-width ".5\\linewidth"

* Conditional Compilation
- *Conditional compilation* allows article/Beamer text to differ.
  - This is *{{{ifslides(slide,article/HTML)}}}* output.

* Including LaTeX
- Easy to include mathematical text/equations, e.g., $E = mc^2$.
- Or an aligned equation:
  \begin{align}
  A &= 2 + 2 \\
    &= 4.
  \end{align}

* Including (live) source code
#+begin_src jupyter-python
  import matplotlib.pyplot as plt
  x = [1, 2, 3]
  y = [2, 7, 4]
  plt.figure(dpi=300)
  plt.plot(x, y)
  plt.show()
#+end_src
```

Note that the Python code block at the end can be run *within Emacs* by just pressing `C-c C-c`. Again, the output (including other the code or the results or both) can be exported to various formats. For example, Figure 4 shows the result of exporting the file to PDF (via LaTeX) and Figure 5 shows the result of exporting the file to PDF (via Beamer).

org document with live code, equations, and conditional text

Richard Stanton

March 31, 2022

1 Conditional Compilation

- **Conditional compilation** allows article/Beamer text to differ.
  - This is **article/HTML** output.

2 Including LaTeX

- Easy to include mathematical text/equations, e.g., $E = mc^2$.
- Or an aligned equation:

$$A = 2 + 2 \tag{1}$$
$$= 4. \tag{2}$$

3 Including (live) source code

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [2, 7, 4]
plt.figure(dpi=300)
plt.plot(x, y)
plt.show()
```

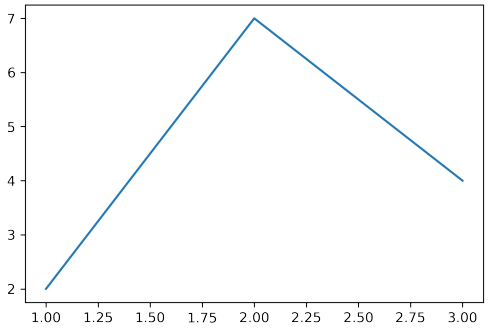


Figure 4: Exporting to LaTeX

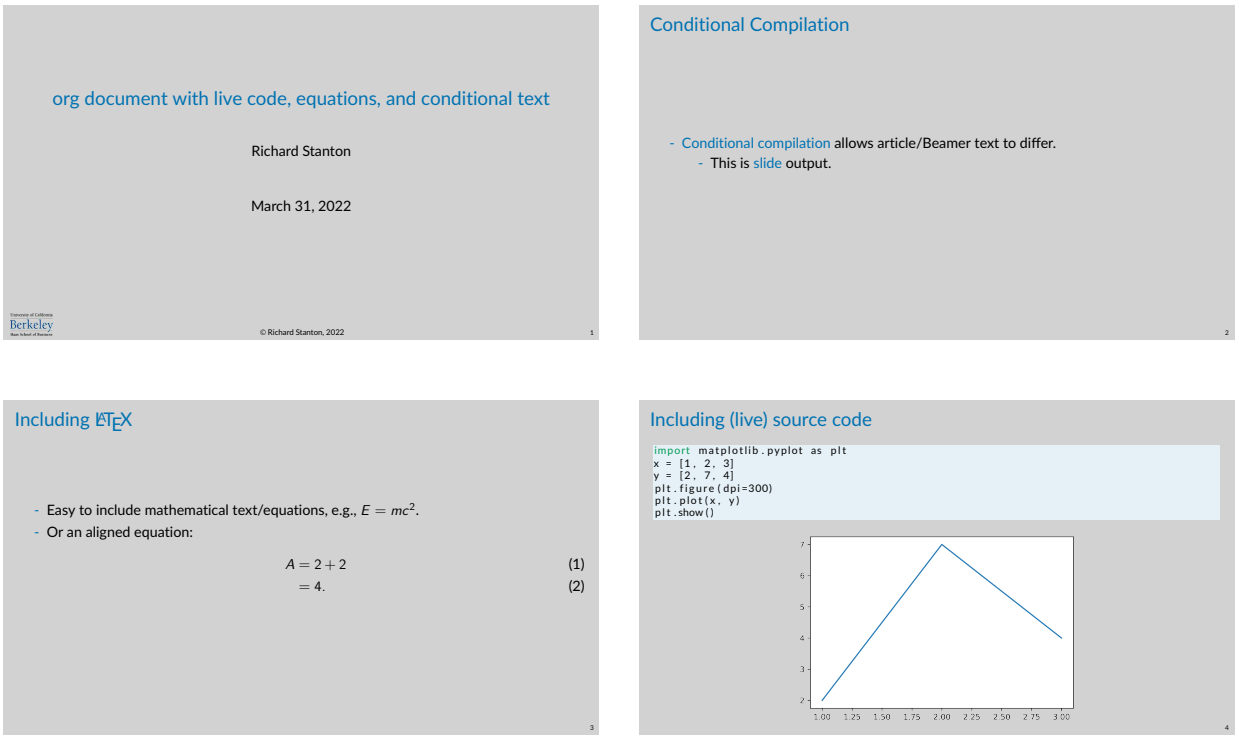


Figure 5: Exporting to Beamer

6 Backing up your work

Computers and disks are pretty reliable, and you can get by for a long time without backing up any of your files. But one day a disk *will* fail, or you'll drop your laptop onto a concrete floor, or you'll accidentally delete lots of files you don't have stored anywhere else, and once you've stopped swearing, crying, and working out how many thousands of hours it'll take to recreate everything you had before, you'll never again wonder whether it's important to back up your important files. There are multiple way to back-up your files, and the more different methods you use, the more confident you can be that you can recover your files regardless of what happens.

6.1 Local backups

I use Carbon Copy Cloner to keep an external copy of my hard drive that I can boot from if my main drive ever becomes unusable. This is very convenient in the event of a total disk failure, but because it only keeps the most recent version of every file, it leaves me vulnerable to files becoming corrupted without my noticing. I therefore also use Time Machine to back up on a different disk, which allows me to keep track of historical versions of my files.

6.2 Cloud backups

Local backups are not very useful if, for example, the house they're in burns down. As a result, you should also keep a remote backup. I use SpiderOak One to backup all my important files (including historical and deleted versions) to the cloud. Note that your project repositories on GitHub serve as another form of remote back-up.

6.3 Synchronizing your work across multiple computers

I work primarily on three different computers: a Mac Pro in my office, another Mac Pro at home, and a MacBook Pro laptop. It is important to keep the files on the three machines synchronized, so that whatever machine I'm sitting at, I know I'm working on the most recent version of each file. To handle this, I use a program called Unison, available for MacOS, Windows and Unix.[12] Synchronizing multiple computers also serves a backup function, as the most recent versions of your files are stored on multiple machines.

7 Project directory structure

One thing that helps find your way around a project is to keep the files in a sensible directory structure. One that I think is very reasonable is `https://drivendata.github.io/cookiecutter-data-science/`, though you may well not need all of these directories and I prefer calling the directory that contains the paper `paper` rather than `reports`.

Another important thing is that you want to be able to use exactly the same code on multiple machines, and for your coauthors to be able to run your code, without needing to edit anything. In particular, this means *never* hard-coding full directory names — in either your `Makefile` or your program files — like `/bulk/refm-lab/misc/projects/test.do`, since the exact location is likely to be different on different machines. Instead, use relative paths like `../data/data.csv`, relative to the current directory (typically the directory in which the program files are located).

Of course, you may not be able to keep everything in exactly the same relative position on different machines. For example, large data files might be stored in locations outside your control. Symbolic links are one way around this — see `https://en.wikipedia.org/wiki/Symbolic_link`, but sometimes you have to use different directory names on different machines. Similarly, how you invoke, say, Python may differ across hosts. In this case, your code/Makefile should start with code that determines automatically what host the code is

[12] I used to use SpiderOak One for synchronization as well as for back-up. However, I often found files getting overwritten by older versions, so I switched to Unison and now use SpiderOak One only for back-up.

running on, and sets things up accordingly. For example, here's the start of a Makefile for a recent project of mine that checks what host the code is running on and sets some environment variables accordingly (specifically containing an indicator for what host the code is running and what command to run to execute Python and Stata on that host).

```
# Are we on the refm main node? If so, submit all requests to compute nodes
ifneq (,$(findstring refm,$(HOSTNAME)))
        machine = refm
        STATA = stata-mp -b
        PYTHON = refm python
# If on compute nodes cn13, etc., call programs directly.
else ifneq ($(wildcard /home/refm/.),)
        machine = refm
        STATA = /home/apps/STATA15/stata-mp -b
        PYTHON = python
# Laptop [$(HOSTNAME) not set. Don't recreate working data sets.]
else ifneq (,$(findstring MacBook,$(shell hostname)))
        machine = laptop
        STATA = stata-se -b
        PYTHON = python
# Desktop Mac
else
        machine = mac
        STATA = stata-se -b
        PYTHON = python
endif
```

8 Getting started

This section is a quick summary of how to set up your system to get ready for a research project. There are a lot of steps, but most of them only need to be done once ever or once at the start of each project.

Get installed

1. If you do not already have one, set up an account on GitHub.[13]
2. If you do not already have one, set up an account on Overleaf.[14]

[13] Students are entitled to a free GitHub Pro account plus some other benefits (see https://education.github.com/pack).

[14] U.C. Berkeley students are entitled to a free Overleaf Professional account (see https://www.overleaf.com/edu/berkeley).

3. Install `Git` on your personal computer.[15]
4. Install a full LaTeX distribution on your personal computer (see Stanton, 2013).[16]
5. Install the full Anaconda Python distribution (version 3.8) on your personal computer.[17] This includes Jupyter.
6. (Optional) If you want to use Julia, install it on your personal computer.[18]
7. Install GNU Make on your personal computer.[19]
8. Install an Integrated Development Environment (IDE) on your personal computer. I recommend VS Code because it is easy to use, popular, actively developed, and has good support for both Python and Julia. For Python alone, I also like PyCharm and Spyder (which is automatically installed as part of the Anaconda Python distribution). However, neither of these has very good (or, in the case of Spyder, any) support for Julia.

Learn the Basics of your Tools

9. Read this document cover to cover. If you are not familiar with LaTeX, also read Stanton (2013). Then skim the references listed in the bibliography to learn the basics of the other tools you've just installed (and so you know where to look when you need help later). Mainly, you'll learn the tools by using them.

Set Things Up

10. Create a new GitHub repository for your project (see `https://github.com/new`) called, say, `project1`, Make the repository private (you probably don't want the rest of the world to be able to see it), click `Create Repository`, then *immediately*...
11. Follow either the first or second set of instructions that pop up after you've clicked `Create Repository` to create a corresponding repository on your personal computer, linked to the repository you just created on GitHub. For example, I would create a directory called, say, `/project1`, then I would go to directory `project1` and type:

[15] This can be a command-line version (I use the MacPorts version on my Mac; you can download Git for Windows at `https://git-scm.com/download/win`), or you may prefer a GUI-based client such as GitKraken or GitHub Desktop.

[16] You can get by using only Overleaf, but you will be much more productive if you edit your files locally. First, this allows you to use the editor of your choice instead of the (rather limited) editor provided by Overleaf. Second, Overleaf doesn't have a concept of directories, so any special files you need for every project (including your master bibliography file) need to be copied from project to project, while on your own machine they can just be installed once.

[17] See `https://www.anaconda.com/products/individual`.

[18] See `https://julialang.org/downloads/`.

[19] I use the MacPorts version on my Mac. For Windows, the easiest way to install GNU Make (and lots of other Unix tools) is via Cygwin.

```
echo "# PROJECT 1" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:rhstanton/project1.git
git push -u origin main
```

12. On your personal computer, create subdirectories `notebooks` and `code` underneath `project1` to put your notebooks and final Python/Julia code in, respectively.
13. If you will be using Jupyter notebooks (which I strongly recommend for initial exploration, even if at some later point you decide to switch to code you can run from the command line), also install nbstripout to make Git play nicely with your notebooks.
14. Create an Overleaf project that will contain the write-up for your problem set.[20]
15. Now go to the `project1` directory on your personal computer, and create a submodule that will contain just the write-up. To do this,
    - On Overleaf, click "Menu" → "Git" and copy the URL that appears after "git clone", e.g.,
      ```
      https://git.overleaf.com/5e1e5dc63fe82300013bf0e7
      ```
    - On your personal computer, go to the directory `project1` and run the command
      ```
      git submodule add https://git.overleaf.com/5e1e5dc63fe82300013bf0e7 writeup
      ```
    - This will create a directory `project1/writeup` containing a git (sub)repository that is linked to the Overleaf document.
    - Now when you run `git push` in directory `project1` or any subdirectory other than `writeup`, it will push your edits to the repository on GitHub. If you run `git push` in the `writeup` subdirectory, it will push the changes to just the write-up to the Overleaf document (and if you or your coauthors are connected to Overleaf, you'll see your changes suddenly appear in front of them).

Now you're ready to get going! Do your exploratory coding in Jupyter notebooks in `notebooks/` and write up your results in, say, `writeup/paper.tex`.[21] Eventually, you should also create final Python or Julia programs to run the code for each question, and a Makefile to go along with them, all stored in `project1/code`. Keep track of everything using Git by

- running `git add` to add each new file you create e.g., `table1.py`, `Makefile`.

[20] In Overleaf, click "New Project" → "Blank Project", then select a descriptive name for the project, e.g., "My first project."

[21] I suggest you practice editing `main.tex` both directly on Overleaf and on your local machine.

- running `git commit` on a regular basis to commit your changes (locally).
- running `git push` both in the main project and in `project1/writeup` to push your edits to the remote repositories.

References

Baker, Peter, 2020, Using GNU Make to manage the workflow of data analysis projects, *Journal of Statistical Software* 94.

Broman, Karl, 2016, Tools for reproducible research, https://kbroman.org/Tools4RR/.

Chacon, Scott, and Ben Straub, 2021, *Pro Git*, second edition (Apress, Berkeley, CA), https://git-scm.com/book/en/v2.

Correia, Sergio, 2017, Linear models with high-dimensional fixed effects: An efficient and feasible estimator, Working paper, Duke University.

Healy, Kieran, 2020, The plain person's guide to plain text social science, Working paper, Duke University, http://kieranhealy.org/resources/.

Jones, Zach, 2013, GNU Make for reproducible data analysis, http://zmjones.com/make/.

Knuth, Donald E., 1984a, *The TEXbook* (Addison-Wesley, Reading, MA).

Knuth, Donald E., 1984b, Literate programming, *The Computer Journal* 27, 97–111.

Kopka, Helmut, and Patrick W. Daly, 2003, *Guide to LaTeX*, fourth edition (Addison-Wesley, Reading, MA).

Lamport, Leslie, 1994, *LaTeX: A Document Preparation System*, second edition (Addison-Wesley, Reading, MA).

McKinney, Wes, 2018, *Python for Data Analysis*, second edition (O'Reilly Media, Inc., Sebastopol, CA).

McKinney, Wes, and the PyData Development Team, 2021, *Pandas: Powerful Python Data Analysis Toolkit*, Release 1.2.3.

Mittelbach, Frank, Michel Goossens, Johannes Braams, David Carlisle, and Chris Rowley, 2004, *The LaTeX Companion*, second edition (Addison-Wesley, Reading, MA).

Newey, Whitney K., and Kenneth D. West, 1987, A simple, positive definite heteroskedasticity and autocorrelation consistent covariance matrix, *Econometrica* 55, 703–708.

Oetiker, Tobias, Hubert Partl, Irene Hyna, and Elisabeth Schlegl, 2021, The not so short introduction to LaTeX 2_ε: Or LaTeX 2_ε in 157 minutes, v. 6.4 (http://www.ctan.org/tex-archive/info/lshort/english).

Pandas Development Team, 2021, 10 minutes to pandas, https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html.

Pilgrim, Mark, 2011, *Dive into Python 3* (Mark Pilgrim), https://diveintopython3.problemsolving.io/.

Poore, Geoffrey M., 2013, Reproducible documents with PythonTeX, in *Proceedings of the 12th Python in Science Conference (SCIPY 2013)*, 74–80.

Poore, Geoffrey M., 2015, PythonTeX: Reproducible documents with LaTeX, Python, and more, *Computational Science & Discovery* 8, 014010.

Schulte, Eric, and Dan Davison, 2011, Active documents with org-mode, *Computing in Science and Engineering* 13, 66–73, http://www.cs.unm.edu/~eschulte/data/CISE-13-3-SciProg.pdf.

Schulte, Eric, Dan Davison, Thomas Dye, and Carsten Dominik, 2012, A multi-language computing environment for literate programming and reproducible research, *Journal of Statistical Software* 46, 1–24.

Sheppard, Kevin, 2021, Introduction to Python for econometrics, statistics and data analysis, 5th edition, Online book, Oxford University, https://www.kevinsheppard.com/teaching/python/notes/.

Sink, Eric, 2011, *Version Control by Example* (Pyrenean Gold Press, Champaign, IL), https://ericsink.com/vcbe/index.html.

Stanisic, Luka, Arnaud Legrand, and Vincent Danjean, 2015, An effective Git and org-mode based workflow for reproducible research, *ACM SIGOPS Operating Systems Review* 49, 61–70.

Stanton, Richard, 2013, Writing finance papers using LaTeX, http://faculty.haas.berkeley.edu/stanton/texintro/.

van Rossum, Guido, 2021, The Python tutorial, https://docs.python.org/3.9/tutorial/.