The Political Methodologist

Newsletter of the Political Methodology Section
American Political Science Association
Volume 18, Number 2, Spring 2011

Editors:
Jake Bowers, University of Illinois at Urbana-Champaign
jwbowers@illinois.edu

Wendy K. Tam Cho, University of Illinois at Urbana-Champaign
wendycho@illinois.edu

Brian J. Gaines, University of Illinois at Urbana-Champaign
bjgaines@illinois.edu

Editorial Assistant:
Ashly Adam Townsen, University of Illinois at Urbana-Champaign
townsen5@illinois.edu

Contents

Notes from the Editors **1**

Articles **2**

Jake Bowers: Six steps to a Better Relationship with Your Future Self 2

Kieran Healy: Choosing Your Workflow Applications 9

Mark M. Fredrickson, Paul F. Testa and Nils B. Weidmann: Collaboration for Social Scientists, or Software is the Easy Part 19

Shawn Treier: Minimizing the Damage: Converting LaTeX to Word Using TeX2Word 23

Political Analysis **26**

Notes From the Editors

This issue of *The Political Methodologist* is devoted to work flow, namely how social scientists move from notions to plans, analyses, and finally reports, and how they can best ensure that all of their work is well organized, transparent, and reproducible. These are laudable, but lofty goals. Anyone who has ever assigned a replication homework knows that the modal political science article features tables and figures that cannot be easily reproduced, and often cannot be closely approximated even with hours of work and painstaking attention. Sometimes, even with the generous cooperation of the original author, important details in the original work remain shrouded in fog. The late David Freedman lamented that medical researchers were sticklers about protocols, but stingy about sharing data, while social scientists were generous with data sets, but hopeless at documenting analysis precisely. The articles in this issue outline the sketch of a remedy for our discipline's poor standards of record keeping and work documentation.

The Sixteenth Century poet and playwright John Heywood noted, "Many hands make light work." Alas, many hands writing together can also breed chaos. Fredrickson, Testa, and Weidmann discuss best practices and best software to minimize logistical hiccups associated with collaborative writing. Treier offers sage advice on one particular task familiar to (and hated by) many TPM readers, converting TeXfiles to Microsoft Word equivalents at the behest of a publisher. Healy and Bowers, in separate pieces, cast their nets more widely still, reviewing tools of various kinds to reduce errors and to minimize the difficulty of revisiting and revising one's work. Even lone wolves cannot avoid collaborating with their future selves, and everyone who has an imperfect memory and/or values time enough to prefer not having to retrace old footsteps can profit from taking seriously the ideas they broach.

Also included here is a brief update on recent changes at *Political Analysis* from the editors.

Our thanks to all of the contributors to this issue (only Wendy and Brian thank Jake, who is too reserved to thank his present-day self). As always, please feel free to contact the editors with any reactions, suggestions, or ideas for future issues of *TPM*.

If you are reading this issue in hardcopy but find yourself extra interested in, say, subversion or Emacs or Vim, we encourage you to peruse the pdf version on your screen. The authors have included hyperlinks throughout their articles and we reproduce them here.

Enjoy!

The Editors

Articles

Six steps to a better relationship with your future self.

Jake Bowers
University of Illinois
jwbowers@illinois.edu

> Do I contradict myself?
> Very well then I contradict myself,
> (I am large, I contain multitudes.)
> (Whitman, 1855)

An idea is born in a coffee shop, a seminar, a quiet walk. On this gray day in 2011, the idea dispels February's doldrums. The student rushes home, mind racing, the cold ignored.

This idea inspires a seminar paper in the spring. A conference paper arises from the seminar paper in collaboration with another student in 2012. A dissertation chapter descends from the conference paper in 2013. Other dissertation chapters take up 2014. A submission to a journal with the original co-author and a new collaborator happens in 2015. Revision and resubmission wait until 2017 while harried editors, reviewers and authors strive to balance research, teaching, service, and life. By now, the three lucky collaborators work as professors in three different universities. In 2018 a child is born and a paper is published. The United Nations takes an interest in the paper in 2019 and hosts a conference to discuss implications of the research. In 2020 a first year graduate student in a coffee shop has an idea that challenges the results in the now famous paper. She presents her paper at a conference in 2021. What would happen if the authors had controlled for X? Or included information now available but missing in 2012? Or chosen a different likelihood function? Will the United Nations (now eager to act based the paper) make a wrong move?

The first author convenes a three way video conference with the other collaborators during his homeward commute after putting his flying car in auto-drive mode.[1] The group must go back to the analyses. Which ones? The ones from 2011? Or 2012? Or 2018? Where are the files? The next day, one member of the group who has kept some hard-drives around out of nostalgia finds some of the files.[2] Now re-analysis should be easy. Right? The student, now professor, should remember the reason for those bits of code (or at least should remember which series of mouse clicks were used to produce the numbers for that crucial table as it was done in 2011 . . . or was it 2015?). Right? And, of course, the way Microsoft Word/Stata/SPSS/R/LISREL understands files and the way that machines in 2021 read and write them is the same — since Windows and Mac OS X have always existed and will always continue to exist more or less as they currently exist. Right? And the group knows exactly which bit of code produced which table and which figure, right? And they wrote their code following Nagler's Maxims (Nagler, 1995) and King's Replication Standard (King, 1995), right?

If the collaborators find themselves saying "Wrong" in answer to the questions posed here then reproducing, updating, or changing the original analyses will take a lot of time. If reproduction is hard to do, then the reputations of the scholars will suffer and, more importantly, world peace will have been delayed. This essay provides some suggestions for practices that will make such reproduction occur much more easily and quickly in the event that famous papers require special scrutiny. Specifically, this piece aims to amplify some of what we already ought to know King (1995) and Nagler (1995), and to add to some of those ideas given current practices, platforms, and possibilities.

Data analysis is computer programming.

All results (numbers, comparisons, tables, figures) should arise from code, not from a series of mouse clicks or copying and pasting. If I wanted to re-create the figure from 2011 but include a new variable or specification, I should be able to do so with just a few edits to the code rather than knowledge of how I (or you) used a pointing device in your graphical user interface some years ago.

Using R (R, 2011), for example, I might specify that

I owe many thanks to Mark Fredrickson, Brian Gaines, Kieran Healy, Kevin Quinn and Cara Wong for direct help on this document and to Mika LaVaque-Manty and Ben Hansen for many useful discussions on this topic. The source code for this document may be freely downloaded and modified from https://github.com/jwbowers/workflow

[1] One assumes that video chatting during manual driving of flying cars will have been outlawed in his state by 2020.

[2] This is the same guy who still owns cassette tapes and compact discs.

[3] The command `please-plot` and some other R functions used in this essay come from the `MayIPleaseDoStatistics` package which emphasizes politeness in data analysis. Functions like `please-plot` can be blocked and more polite versions such as `may-I-please-have-a-plot` can be required using `options(politeness=99)`

the file `fig1.pdf` was produced by the following commands in a file called `fig1.R`.[3]

```
#Read the data
thedata <- read.csv("Data/thedata-15-03-2011.csv")
pdf('fig1.pdf') ## begin writing to the pdf file
please-plot(y by x with thedata. red lines please.)
please-add-a-line(using model1)
#Note to self: a quadratic term does not add to the substance
#model2<-please-fit(y by x+x^2 with thedata)
#summary(abs(fitted(model1)-fitted(model2)))
dev.off() ## stop writing to the pdf file
```

Now, in the future if I wonder how "that plot on page 10" was created, I will know: (1) "that plot" is from a file called `fig1.pdf` and (2) `fig1.pdf` was created in `fig1.R`. In a future where R still exists, changing the figure will require quick edits of commands already written. In a future where R does not exist, I will at least be able to read the plain text R commands and use them to write code in my new favorite statistical computing language: R scripts are written in plain text, and plain text is a format that will be around as long as computer programmers write computer programs.[4]

Moreover, realize that file names send messages to your future self. Name your files with evocative and descriptive names. Your collaborators are less likely to call you at midnight asking for help if your files are named `inequality-and-protest-figures.R` than if your files are called `temp9` or `supercalifragilisticexpialidocious`. The extension `.R` tells us and the operating system that the file contains R commands. This part of the filename enables us to quickly search our antique hard drives for files containing R scripts.

Step 1 If we know the provenance of results, future or current collaborators can quickly and easily reproduce and thus change and improve upon the work.

No data analyst is an island for long.

Data analysis involves a long series of decisions. Each decision requires justification. Some decisions will be too small and technical for inclusion in the published article itself. These need to be documented in the code itself (Nagler, 1995). Paragraphs and citations in the publication will justify the most important decisions. So, one must code to communicate with yourself and others. There are two main ways to avoid forgetting the reasons you did something with data: comment your code and tightly link your code with your writing.[5]

Code to communicate: Comment your code.

Comments—unexecuted text inside of a script—are a message to collaborators (including your future self) and other consumers of your work. In the above code chunk, I used comments to explain the lines to readers unfamiliar with R and to remember that I had tried a different specification but decided not to use it because adding the squared term did not really change the substantive story arising from the model.[6] Messages left for your future self (or near-future others) help retrace and justify your decisions as the work moves from seminar paper to conference paper to poster back to paper to dissertation and onwards.

Notice one other benefit of coding for an audience: we learn by teaching. By assuming that others will look at your code, you will be more likely to write clearer code, or perhaps even to think more deeply about what you are doing as you do it.

Comment liberally. Comments are discarded when R runs analysis or LaTeX turns plain text into page images, so only those who dig into your work will see them.

Code to communicate: Literate programming.

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. (Knuth, 1984, p. 97)

Imagine you discover something new (or confirm something old). You produce a nice little report on your work for use in discussions of your working group or as a memo for a web or reviewer appendix. The report itself is a pdf file or some other format which displays page images to ease reading rather than to encourage reanalysis and rewriting. Eventually pieces of that report (tables, graphs, paragraphs) ought to show up in, or at least inform, the publishable paper. Re-creating those analyses by pointing, clicking, copying, or pasting would invite typing error and waste time. Re-creating your arguments justifying your analysis decisions would also waste time. More importantly, we and others want to know why we did what we did. Such explanations may not be very clear if we have some pages of printed code in one hand and a manuscript in the other. Keep in mind the distinction between the "source code" of a document (i.e. what computation was required to produce

[4] Since R is open source, I will also be able to download an old version of R, download an old-fashioned open-source operating system (like Ubuntu 10), and run the old-fashioned statistical computing environment in the old-fashioned operating system in a virtual machine on my new-fashioned actual machine.

[5] One can also try the R command `put-it-in-my-mind(reason,importance='high')` to firmly place a reason for a decision into the mind of the analyst. I myself have not had much luck with this function.

[6] R considers text marked with `#` as a comment.

it) and the visible, type-set page image. Page images are great for reading, but not great for reproducing or collaborating. The source code of any document exchanged by the group must be available and executable.

How might one avoid these problems? Literate programming is the practice of weaving code into a document — paragraphs, equations, and diagrams can explain the code, and the code can produce numbers, figures, and tables (and diagrams and even equations and paragraphs). Literate programming is not merely fancy commenting but is about enabling the practice of programming itself to facilitate easy reproduction and communication.

For example, I just suggested that we know where "that plot on page 10" comes from by making sure we had a `fig1.pdf` file produced from a clearly commented plain text file called something like `fig1.R`. An even easier solution would be to directly include a chunk of code to produce the figure inside of the paper itself. This paper, for example, was written in plain text using LaTeX markup with R code chunks to make things like Figure 1. This combination of LaTeX and R is called Sweave (Leisch, 2005).[7]

```
This paper, for example, was written in plain text using \LaTeX
markup with R code chunks to make things like
Figure~\ref{fig:giniprot}. This combination of \LaTeX and R is
called Sweave \citep{Leis:2005}.\footnote{Support for Sweave is
  included with R.}
\begin{figure}[h]
\centering
 <<fig1plot,fig=TRUE>>=
 ##  Make a scatterplot of Protest by Inequality
 with(good.df,plot(gini04,protac00,xlab='Gini Coefficient 2004',
 ylab='Mean Protest Activities\n(World Values Survey 1980-2000)'))
 ##  Label a few interesting points
 with(good.df[c("EGY","JOR","USA","SWE","CHL"),],
      text(gini04,protac00,labels=Nation))
 @
\caption{Protest activity by income inequality Norris (2009)}.}
\label{fig:giniprot}
\end{figure}
```

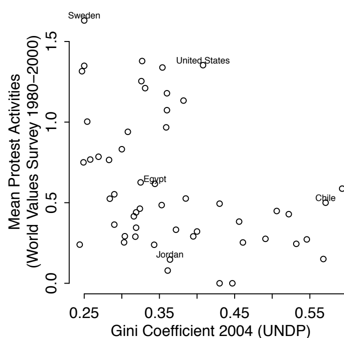


Figure 1: Protest activity by income inequality (Norris, 2009).

By using `\label{fig:giniprot}`, I do not need to keep track of the figure number, nor do extra work when I reorganize the document in response to reviewer suggestions. Nor do I need a separate `fig1.R` file or `fig1.pdf` file. Tables and other numerical results are also possible to generate within the source code of a scholarly paper. Those who view the code for this essay will see how Table 1 was also generated directly from a regression object.[8]

	Coef	SE	95% CI	
Intercept	1.5	0.2	1.2	1.8
Income Inequality (lower=more equal)	−1.0	0.4	−1.9	−0.2
Mean Political Rights (lower=more rights)	−0.2	0.0	−0.2	−0.1
	n: 53, resid.sd: 0.28, R^2: 0.57			

Table 1: People living in countries with unequal income distributions report less protest activity to World Values Survey interviewers than people living in countries with relatively more equal income distributions, adjusting for average political rights as measured by Freedom House 1980–2000. Data from (Norris, 2009).

Literate data analysis is not the same as Sweave, even if Sweave is a nice implementation.[9] LyX offers a WYSIWYG environment for LaTeX that supports Sweave. And the `odfWeave` package in R allows the use of OpenOffice documents in exactly the same way.[10] If your workflow does not involve LaTeX and R, you can still implement some of the principles here. Imagine creating a style in Microsoft Word called "code" which hides your code when you print your document, but which allows you to at least run each code chunk piece by piece [or perhaps there are ways to extract all text of style "code" from a Microsoft Word document]. Or imagine just using some other kind of indication linking paragraphs to specific places in code files. There are many ways that creative people can program in a literate way.

Literate programming need not go against the principle of modular data analysis (Nagler, 1995). In my own work I routinely have several different Sweave files that fulfill different functions, some of them create LaTeX code that I can `\input` into my `main.tex` file, others setup the data, run simulations, or allow me to record my journeys down blind alleys. Of course, when we have flying cars running on autopilot, perhaps something other than Sweave will make our lives even easier. Then we'll change.

Step 2 We analyze data in order to explain something about the world to other scholars and policy makers. If we

[7] Support for Sweave is included with R.
[8] Beck (2010) inspired this particular presentation of a linear model.
[9] The R project has a task view devoted to reproducible research listing many of the different approaches to literate programming for R.
[10] A quick Google search of "Sweave for Stata" turned up lots of resources for literate programming with Stata.

focus on explaining how we got our computers to do data analysis to human beings, we will do a better job with the data analysis itself: we will learn as we focus on teaching, and we will avoid errors and save time as we ensure that others (including our future selves) can retrace our steps. A document than can be "run" to reproduce all of the analyses also instills confidence in readers and can more effectively spur discussion and learning and cumulation of research.

Meaningful code requires data.

All files containing commands operating on data must refer to a data file. A reference to a data file is a line of code the analysis program will use to operate on ("load"/ "open" / "get" / "use") the data file. One should not have to edit this line on different computers or platforms in order to execute this command. Using R, for example, all analysis files should have `load('thedata.rda')` or `read.csv('thedata.csv')` or some equivalent line in them, and `thedata.csv` should be stored in some place easy to find (like in the same directory as the file or perhaps in `'Data/thedata.rda'`). Of course, it never hurts to drop in a comment pointing to the data file.

Where should one store data files? An obvious solution is always to make sure that the data file used by a command file is in the same directory as the command file. More elegant solutions require all co-authors to have the same directory structure so that `load('Data/thedata.rda')` means the same thing on all computers used to work on the project. This kind of solution is one of the things that formal version control systems do well (as discussed a bit more in the next section).

The principle of modularity suggests that you separate data cleaning, processing, recoding, and merging from analysis in different files (Nagler, 1995). So, perhaps your analysis oriented files will `load('cleandata.rda')` and a comment in the code will alert the future you (among others) that `cleandata.rda` was created from `create-cleandata.R` which in turn begins with `read.csv(\url('http://data.gov/dirtydata.csv'))`. Such a data processing file will typically end with something like `save('cleandata.rda')` so that we are doubly certain about the provenance of the data.[11]

Now, if in the future we wonder where `cleandata.rda` came from, we might search for occurrences of 'cleandata' in the files on our system. However, if such searching among files is a burden, an even nicer solution is to maintain a file for each project called "MANIFEST.txt" or "INDEX.txt" or "README.txt" which lists the data and command files with brief descriptions of their functions and relations.

Step 3 We should know where the data came from and what operations were performed on which set of data.

In the good old days, when we executed our LISREL code in batch mode, we had no choice but to tell the machine clearly, in a few easy to understand and informative lines, what files (with filenames no longer than 8 characters) to use:

```
DA NI=19 NO=199 MA=CM
LA=basic.lab
CM FI=basic.cov
```

The fact that I need to articulate this idea at all arises because of graphical user interfaces: it is all too easy to use the mouse to load a data file into memory and then to write a script to analyze this file without ever noting the actual name or location of the data file.

Version control prevents clobbering and reconciles history.

Group work requires version control.[12] Many people are familiar with the "track changes" feature in modern WYSIWYG word processors or the fact that Dropbox allows one to recover previous versions of files. These are both kinds of version control. More generally, when we collaborate, we'd like to do a variety of actions with our shared files. Collaboration on data analytic projects is more productive and better when (1) it is easy to see what has changed between versions of files; (2) members of the team feel free to experiment and then to dump parts of the experiment in favor of previous work while merging the successful parts of the experiment into the main body of the paper; (3) the team can produce have "releases" of the same document (one to MPSA, one to APSR, one to your parents) without spawning many possibly conflicting copies of the same document; (4) people can work on the same files at the same time without conflicting with one another (and can reconcile their changes without too much confusion and clobbering). Clobbering is what happens when your future self or your current collaborator saves an old version of a file over a new version, erasing good work by accident.

Of course if you rely on Dropbox or "track changes" for version control, you must communicate with other folks in your group before you edit existing files. Only one of you can edit and save a given file at a time. This prevents your work (or your colleagues work) from getting lost when you both try to save the same file on top of each other.

[11] Of course, if you need math or paragraphs to explain what is happening in these files, you might prefer to make them into Sweave files, for which the conventional extension is .Rnw. So you'd have `create-cleandata.Rnw` which might explain and explore the different coding decisions you made, perhaps involving a factor analysis and diagnostic plots.

[12] Fredrickson et al. (2011) and Healy (2011) in this issue also explain what version control is and why we might want to use it.

If you find that you need to work on the same files at the same time, then you should work on establishing your own shared version control system. Free options include launchpad, github, sourceforge for open source projects (i.e. papers you are writing which you are happy to share with others as you write). Each of those services include paid versions too. One may also use Dropbox as a kind of server for version control: for example, one may copy files from the Dropbox directory into a local working directory so as to avoid clobbering and then work on merging changes by hand before copying back to the Dropbox directory and replacing existing files.

We use subversion with our own research group, and I use it for all of my own projects (except this one, for which I am experimenting with git). Subversion and bazaar and git are all great. They mainly differ in the extent to which you need to run a server. Subversion requires a server.[13]

Of course, one may take advantage of many of the benefits of formal version control systems with some disciplined systems for file and directory organization. An excellent, simple, and robust version control system is to rename your files with the date and time of saving them: thedoc.tex becomes thedoc25-12-2011-23:50.tex. Be sure to include year in the file names — remember, the life of an idea is measured in years. If you are wise enough to have saved your documents as plain text then you can easily compare documents using the many utilities available for comparing text files.[14] When you reach certain milestones you can rename the file accordingly: thedocAPSA2009.tex — for the one sent to discussants at APSA — or thedocAPSR2015.tex — for the version eventually sent to the APSR six years after you presented it at APSA. The formal version control systems I mentioned above all allow this kind of thing and are much more elegant and capable, but you can do it by hand too as long as you don't mind taking up a lot of disk space and having many "thedoc..." files around. If you do version control by hand, spend a little extra time to ensure that you do not clobber files when you make mistakes typing in the file-names. And, if you find yourself spending extra time reconciling changes made by different collaborators by hand, remember this is a task that modern version control systems take care of quickly and easily.

Step 4 Writing is rewriting. Thus, all writing involves versions. When we collaborate with ourselves and others we want to avoid clobbering and we want to enable graceful reconciliation of rewriting. One can do these things with formal systems of software (like subversion or git or bazaar) or with formal systems of file naming, file comparing, and communication or, even better, with both. In either case, plain text files will make such tasks easier, will take up less disk space, and be easier to read for the future you.

Minimize error by testing.

Now, back to that famous article of 2018. After reading the conference paper critique of 2021 (that came from the seminar paper of 2020), the statisticians at the UN worry about the bootstrap confidence intervals presented in the original paper.[15] So, now the authors would like to evaluate their bootstrap procedure. Although nice code exists for bootstrapping linear models, no nice code exists to bootstrap the bootstrap. Of course, the code required is not complex, but since they are writing custom code they worry about getting it right. As they've struggled to respond to the critiques of their paper, they've had lots of time to appreciate problems arising from bugs, errors, and typos in data analysis and code.

Now, if they had a moment to think in between teaching that new class, reading books for an awards committee, evaluating application files for the admissions committee, feeding popsicles to a sick child, and undertaking the odd bit of research, they might say to themselves, "Before I write new code, I should write a test of the code. I should write a little bit of code that lets me know that my double-bootstrap procedure actually does what it is supposed to do."

Of course, this idea, like most others, is not new. The desire to avoid error looms large when large groups of programmers write code for multi-million dollar programs. The idea of test driven development and the idea that one ought to create tests of small parts of one's code arose to address such concerns. For the social scientist collaborating with her future self and/or a small group of collaborators, here is an example of this idea in a very simple form: Say I want to write a function to multiply a number by 2. If my function works, when I give it the number 4, I should see it return the number 8 and when I give it -4, I should get -8.

```
## The test function:
test.times.2.fn <- function(){
  ## This function tests times.2.fn
  if (times.2.fn(thenumber=4) ==  8 &
      times.2.fn(thenumber=-4) == -8) {
```

[13]If you already pay to host a website, you may already have the right to run a subversion or git server there. Your university or institute may have a version control system running somewhere on campus. And Google will direct you to many helpful people who have installed such servers on their own diverse desktop machines. Github requires that you pay to host private repositories.

[14]Adobe Acrobat allows one to compare differences in pdf files. OpenOffice supports a "Compare Documents" option. And Google Docs will report on the version history of a document.

[15]Perhaps they should be worried about the deeper substantive critiques offered by the student, but they are statisticians and so focus on the stats. The policy makers of 2021 were cowed by the methodological virtuosity of the 2018 article, and so, even though they had the same substantive concerns as the student, they kept their mouths shut at the mini-conference to avoid looking dumb in front of their bosses.

```
    print("It works!")
  } else { print("It does not work!")
       }
}
## The function:
times.2.fn <- function(thenumber){
  ## This function multiplies a scalar number by 2
  ## thenumber is a scalar number
  thenumber+2
}
## Use the test function
test.times.2.fn()
```

```
[1] "It does not work!"
```

Ack! I mistyped "+" for "*". Good thing I wrote the test![16]

Step 5 No one can forsee all of the ways that a computer program can fail. One can, however, at least make sure that it succeeds in doing the task motivating the writing of the code in the first place.

Copy and improve on others' examples.

Lots of people are thinking about "reproducible research" and "literate programming" these days. Google those terms. Of course, the devil is in the details: Here I list a few of my own attempts at enabling reproducible research. You'll find many other inspiring examples on the web. Luckily, the open source ethos aligns nicely with academic incentives, so we are beginning to find more and more people offering their files online for copying and improvement. By the way, if you do copy and improve, it is polite to alert the person from whom you made the copy about your work.

I have experimented with three systems so far: (1) for one paper we simply included a Sweave document and data files into a compressed archive Bowers and Drake (2005); (2) for another more computing intensive paper we assembled a set of files that enabled reproduction of our results using the `make` system (Bowers et al., 2008); and (3) recently I have tried the "compendium" approach (Gentleman, 2005; Gentleman and Temple Lang, 2007) which embeds an academic paper within the R package system Bowers (2011). The benefit of this last approach is that one is not required to have access to a command line for `make`: the compendium is downloadable from within R using `install.packages()` and is viewable using the `vignette()` function in any operating system than runs R.[17] The idea that one ought to be able to install and run and use an academic paper just as one installs and uses statistical software packages is very attractive, and I anticipate that it will become ever easier to turn papers into R packages as creative and energetic folks turn their attention to the question of reproducible research.

Step 6 We all learn by doing. When we share reproduction materials we improve both cumulation of knowledge and our methods for doing social science (Freese, 2007; King, 1995). As we copy and improve upon each other's modes of doing work we enhance our collective ability to believe each other and for future scholars to believe us, too.

Remember that research ought to be credible communication.

> [I]f the empirical basis for an article or book cannot be reproduced, of what use to the discipline are its conclusions? What purpose does an article like this serve? (King, 1995)

We all always collaborate. Many of us collaborate with groups of people at one moment in time as we race against a deadline. All of us collaborate with ourselves over time.[18] The time-frames over which collaboration are required — whether among a group of people working together or within a single scholar's productive life or probably both — are much longer than any given version of any given software will easily exist. Plain text is the exception. Thus, even as we extol version control systems, one must have a way to ensure future access to them in a form that will still be around when sentient cockroaches finally join political science departments (by then dominated by cetaceans after humans are mostly uploads).[19]

But what if the UN never hears of your work, or, by some cruel fate, your article does not spawn debate? Why then would you spend time to communicate with your future self and others? My own answer to this question is that I want my work to be credible and useful to myself and other scholars even if each article does not immediately change the world. What I report in my data analyses should have two main characteristics: (1) the findings of the work should not be a matter of opinion; and (2) other people should be able to reproduce the findings. That is, the work represents a shared experience — and an experience shared without respect to the identities of others (although requiring some common technical training and research resources).

---

[16] A more common example of this kind of testing occur everyday when we recode variables into new forms but look at a crosstab of the old vs. new variable before proceeding.

[17] Notice that my reproduction archives and/or instructions for using them are hosted on the Dataverse, which is another system designed to enhance academic collaboration across time and space.

[18] What is a reasonable time-span for which to plan for self-collaboration on a single idea? Ask your advisers how long it took them to move from idea to dissertation to publication.

[19] The arrival of the six-legged social scientists revives Emacs and finally makes Ctrl-a Ctrl-x Esc-x Ctrl-c a reasonable key combination.

Assume we want others to believe us when we say something. More narrowly, assume we want other people to believe us when we say something about data: "data" here can be words, numbers, musical notes, images, ideas, etc ... The point is that we are making some claims about patterns in some collection of stuff. Now, it might be easy to convince others that "this collection of stuff is different from that collection of stuff" if those people were looking over our shoulders the whole time that we made decisions about collecting the stuff and broke it up into understandable parts and reorganized and summarized it. Unfortunately, we can't assume that people are willing to shadow a researcher throughout her career. Rather, we do our work alone or in small groups and want to convince other distant and future people about our analyses.

Now, say your collections of stuff are large or complex and your chosen tools of analyses are computer programs. How can we convince people that what we did with some data with some program is credible, not a matter of whim or opinion, and reproducible by others who didn't shadow us as we wrote our papers? This essay has suggested a few concrete ways to enhance the believability of such scholarly work. In addition, these actions (as summarized in the section headings of this essay) make collaboration within research groups more effective. Believability comes in part from reproducibility and researchers often need to be able to reproduce in part or in whole what different people in the group have done or what they, themselves, did in the past.

In the end, following these practices and those recommended by Fredrickson et al. (2011) and Healy (2011) in this issue allows your computerized analyses of your collections of stuff to be credible. Finally, if the UN quibbles with your analyses, your future self can shoot them the archive required to reproduce your work.[20] You can say, "Here is everything you need to reproduce my work." To be extra helpful you can add "Read the README file for further instructions." And then you can get on with your life: maybe the next great idea will occur when your 4-year-old asks a wacky question after stripping and painting her overly cooperative 1-year-old brother purple, or teaching a class, or in a coffee shop, or on a quiet walk.

References

Beck, Nathaniel. 2010. "Making Regression and Related Output More Helpful to Users." *The Political Methodologist* 18 (1): 4–9.

Bowers, Jake. 2011. "Reproduction Compendium for: 'Making Effects Manifest in Randomized Experiments'." http://hdl.handle.net/1902.1 /15499.

Bowers, Jake and Katherine W. Drake. 2005. "Reproduction Archive for:'EDA for HLM: Visualization when Probabilistic Inference Fails'." http://hdl.handle.net/1902. 1/13376.

Bowers, Jake, Ben B. Hansen and Mark M. Fredrickson. 2008. "Reproduction Archive for: 'Attributing Effects to A Cluster Randomize Get-Out-The-Vote Campaign'." http://hdl.handle.net/1902.1/12174.

Freese, Jeremy. 2007. "Replication Standards for Quantitative Social Science: Why Not Sociology?" *Sociological Methods & Research* 36 (2):158–72.

Gentleman, Robert. 2005. "Reproducible Research: A Bioinformatics Case Study." *Statistical Applications in Genetics and Molecular Biology* 4 (1): 1034.

Gentleman, Robert and Duncan Temple Lang. 2007. "Statistical Analyses and Reproducible Research." *Journal of Computational and Graphical Statistics* 16 (1): 1–23.

King, Gary 1995. "Replication, Replication." *PS: Political Science and Politics* 28 (3): 444–52.

Knuth, Donald E. 1984. "Literate Programming." *The Computer Journal* 27 (2): 97–111

Leisch, Friedrich. 2005. *Sweave User Manual.*

Nagler, Jonathan. 1995. "Coding Style and Good Computing Practices." *PS: Political Science and Politics* 28 (3): 488–92.

Norris, Pippa. 2009. *Crossnational Data Release 3.0* http://www.hks.harvard.edu/fs/pnorris/Data/Data.htm.

R: A Language and Environment for Statistical Computing. 2011. R Development Core Team at the R Foundation for Statistical Computing http://www.R-project.org.

Whitman, Walt. 1855. "Leaves of Grass." *Song of Myself.* Project Gutenberg [2008], p. 51.

[20] Since you used plain text, the files will still be intelligible, analyzed using commented code so that folks can translate to whatever system succeeds R, or since you used R, you can include a copy of R and all of the R packages you used in your final analyses in 2018 in the archive itself. You can even throw in a copy of Ubuntu 10 and an open source virtual machine running the whole environment.

Choosing Your Workflow Applications

Kieran Healy
Duke University
kjhealy@soc.duke.edu

Introduction

You can do productive, maintainable and reproducible work with all kinds of different software set-ups. This is the main reason I don't go around encouraging everyone to convert to the applications I use. (My rule is that I don't try to persuade anyone to switch if I can't commit to offering them technical support during and after their move.) So this discussion is not geared toward convincing you there is One True Way to organize things. I do think, however, that if you're in the early phase of your career as a graduate student in, say, Sociology, or Economics, or Political Science, you should give some thought to how you're going to organize and manage your work.[1] This is so for two reasons. First, the transition to graduate school is a good time to make changes. Early on, there's less inertia and cost associated with switching things around than there will be later. Second, in the social sciences, text and data management skills are usually not taught to students explicitly. This means that you may end up adopting the practices of your advisor or mentor, continue to use what you are taught in your methods classes, or just copy whatever your peers are doing. Following these paths may lead you to an arrangement that you will be happy with. But maybe not. It's worth looking at the options.

Two remarks at the outset. First, because this discussion is aimed at beginning students, some readers may find much with which they are already familiar. Even so, some sections may still be of interest, as I have tried to keep the software references quite current. Second, although in what follows I advocate you take a look at several applications in particular, it's not really about the gadgets or utilities. The Zen of Organization is Not to be Found in Fancy Software. Nor shall the true path of Getting Things Done be revealed to you through the purchase of a nice Moleskine Notebook. Instead, it lies within—unfortunately.

Just Make Sure You Know What You Did

For any kind of formal data analysis that leads to a scholarly paper, however you do it, there are some basic principles to adhere to. Perhaps the most important thing is to do your work in a way that leaves a coherent record of your actions. Instead of doing a bit of statistical work and then just keeping the resulting table of results or graphic that you produced, for instance, write down what you did as a documented piece of code. Rather than figuring out but not recording a solution to a problem you might have again, write down the answer as an explicit procedure. Instead of copying out some archival material without much context, file the source properly, or at least a precise reference to it.

Why should you bother to do any of this? Because when you inevitably return to your table or figure or quotation nine months down the line, your future self will have been saved hours spent wondering what it was you thought you were doing and where you got that result from.

A second principle is that a document, file or folder should always be able to tell you what it is. Beyond making your work reproducible, you will also need some method for organizing and documenting your draft papers, code, field notes, datasets, output files or whatever it is you're working with. In a world of easily searchable files, this may mean little more than keeping your work in plain text and giving it a descriptive name. It should generally *not* mean investing time creating some elaborate classification scheme or catalog that becomes an end in itself to maintain.

A third principle is that repetitive and error-prone processes should be automated if possible. (Software developers call this "DRY", or Don't Repeat Yourself.) This makes it easier to check for and correct mistakes. Rather than copying and pasting code over and over to do basically the same thing to different parts of your data, write a general function that can be called whenever it's needed. Instead of retyping and reformatting the bibliography for each of your papers as you send it out to a journal, use software that can manage this for you automatically.

There are many ways of implementing these principles. You could use Microsoft Word, Endnote and SPSS. Or Textpad and Stata. Or a pile of legal pads, a calculator, a pair of scissors and a box of file folders. Still, software applications are not all created equal, and some make it easier than others to do the Right Thing. For instance, it is *possible* to produce well-structured, easily-maintainable documents using Microsoft Word. But you have to use its styling and outlining features strictly and responsibly, and most people don't bother. You can maintain reproducible analyses in SPSS, but the application isn't set up to do this automatically or efficiently, nor does its design encourage good habits. So, it is probably a good idea to invest some time learning about the alternatives. Many of them are free to use or try out, and you are at a point in your career where you can afford to play with different setups without too much trouble.

I thank Jake Bowers for helpful comments.

[1] This may also be true if you are about to move from being a graduate student to starting as a faculty member, though perhaps the rationale is less compelling given the costs.

What Sort of Computer Should You Use?

The earliest choice you will face is buying your computer and deciding what operating system to run on it. The leading candidates are Microsoft Windows, Apple's Mac OS X, and some distribution of Linux. Each of these platforms has gone some of the way—in some cases a long way—toward remedying the main defects stereotypically associated with it. I would characterize the present state of things this way:

- Windows dominates the market. Because of this, far more viruses and malware target Windows than any other OS. Long-standing design and usability problems have been somewhat ameliorated in recent years. The previous major version, Windows Vista, was not very popular, though its main problems were not primarily related to security. Its successor, Windows 7, is generally accepted to be a solid improvement.

- Mac OS X runs only on computers made by Apple (the existence of "hackintoshes" notwithstanding). Unlike in the past, Apple computers today have the same basic hardware as computers that run Windows. This has two consequences for those considering Mac OS X. First, one can now make direct price comparisons between Apple computers and PC alternatives (such as Dells, Lenovos, etc). In general, the more similarly kitted-out a PC is to an Apple machine, the more the price difference between the two goes away.[2] However, Apple does not compete at all price-points in the market, so it will always be possible to configure a cheaper PC (with fewer features) than one Apple sells. For the same reason, it is also easier to find a PC configuration precisely tailored to some particular set of needs or preferences (e.g., with a better display but without some other feature or other) than may be available from Apple.

  Second, because Apple now runs Intel-based hardware, installing and running Windows is easy, and even catered to by Mac OS's Boot Camp utility. Beyond installing OS X and Windows side-by-side, third-party virtualization software is available (for about $80 from VMWare or Parallels, or free from VirtualBox) that allows you to run Windows or Linux seamlessly within OS X. Thus, Apple hardware is the only setup where you can easily try out each of the main desktop operating systems.

- Linux is stable, secure, and free. User-oriented distributions such as Ubuntu are much better-integrated and well-organized than in the past. The user environment is friendlier now. Installing, upgrading and updating software—a key point of frustration and time-wasting in older Linux distributions—is also much better than it used to be, as Linux's package-management systems have matured. It remains true that Linux users are much more likely to be forced at some point to learn more than they might want to about the guts of their operating system.

These days, I use Mac OS X, and the discussion here reflects that choice to some extent. But the other two options are also perfectly viable alternatives, and most of the applications I will discuss are freely available for all of these operating systems.

The dissertation, book, or articles you write will generally consist of the main text, the results of data analysis (perhaps presented in tables or figures) and the scholarly apparatus of notes and references. Thus, as you put a paper or an entire dissertation together you will want to be able to easily **edit text**, **analyze data** and **minimize error**. In the next section I describe some applications and tools designed to let you do this easily. They fit together well (by design) and are all freely available for Windows, Linux and Mac OS X. They are not perfect, by any means — in fact, some of them can be awkward to learn. But graduate-level research and writing can also be awkward to learn. Specialized tasks need specialized tools and, unfortunately, although they are very good at what they do, these tools don't always go out of their way to be friendly.

Edit Text

If you are going to be doing quantitative analysis of any kind then you should write using a good text editor. The same can be said, I'd argue, for working with any highly structured document subject to a lot of revision, such as a scholarly paper. Text editors are different from word processors. Unlike applications such as Microsoft Word, text editors generally don't make a big effort to make what you write look like as though it is being written on a printed page.[3] Instead, they focus on working with text efficiently and assisting you with visualizing the logical structure of what you're writing. If you are writing code to do some statistical analysis, for instance, then at a minimum a good editor will highlight keywords and operators in a way that makes the code more readable. Typically, it will also passively signal to you when you've done something wrong syntactically (such as forget a closing brace or semicolon or quotation mark), and automagically indent or tidy up your code as you write it. If you are writing a scholarly paper or a dissertation, a good text editor can make it easier to maintain control over the structure of your document, and

[2] Comparisons should still take account of remaining differences in hardware design quality, and of course the OS itself.

[3] For further argument about the advantages of text-editors over word processors see Allin Cottrell's polemic, "Word Processors: Stupid and Inefficient."

help ensure that cross-references and other paraphernalia are correct. Just as the actual numbers are crunched by your stats program—not your text editor—the typesetting of your paper is handled by a specialized application, too. Perhaps more importantly, a text editor *manipulates plain text* as opposed to binary file formats like `.doc` or `.pdf`, and plain text is the easiest format to manage, control, back up, and come back to later with some other application.

Emacs is a text editor, in the same way the blue whale is a mammal. Emacs is very powerful, and can become almost a complete working environment in itself, should you so wish. (I don't really recommend it.) Combining Emacs with some other applications and add-ons (described below) allows you to manage writing and data-analysis effectively. The Emacs Homepage has links to Windows and Linux versions. The two most easily available versions on the Mac are GNU Emacs itself and Aquamacs. The former is the "purest" version of Emacs and does not implement many Mac conventions out of the box. The latter makes an effort to integrate Emacs with the Mac OS. For Windows users who would like to use Emacs, the developers maintain an extensive FAQ including information on where to download a copy and how to install it.

While very powerful and flexible, Emacs is not particularly easy to learn. Indeed, to many first-time users (especially those used to standard applications on Windows or Mac OS) its conventions seem bizarre and byzantine. As applications go, Emacs is quite ancient: the first version was written by Richard Stallman in the 1970s. Because it evolved in a much earlier era of computing (before decent graphical displays, for instance, and possibly also fire), it doesn't share many of the conventions of modern applications.[4] Emacs offers many opportunities to waste your time learning its particular conventions, tweaking its settings, and generally customizing it. There are several good alternatives on each major platform, and I discuss some of them below.

At this point it's reasonable to ask why I am even mentioning it, let alone recommending you try it. The answer is that, despite its shortcomings, Emacs is nevertheless very, *very* good at managing the typesetting and statistics applications I'm about to discuss. It's so good, in fact, that Emacs has recently become quite popular amongst a set of software developers pretty much all of whom are much younger than Emacs itself. The upshot is that there has been a run of good, new resources available for learning it and optimizing it easily. Meet Emacs, a screencast available for purchase from PeepCode, walks you through the basics of the application. Emacs itself also has a built-in tutorial.

If text editors like Emacs are not concerned with formatting your documents nicely, then how do you produce properly typeset papers? You need a way to take the text you write and turn it into a presentable printed (or PDF) page. This is what **LaTeX** is for. LaTeX is a freely-available, professional-quality typesetting system. It takes text marked up in a way that describes the structure and formatting of the document (where the sections and subsections are, for example, or whether text should be **in bold face** or *emphasized*) and typesets it properly. If you have ever edited the HTML of a web page, you'll know the general idea of a markup language. If you haven't, the easiest way to understand what I mean is to look at a segment of LaTeX markup. An example is shown in Listing 1. You can get started with LaTeX for Mac OS X by downloading the MacTeX distribution. On Windows, ProTeXt and MiKTeX are both widely-used. Linux vendors have their own distributions, or you can install TeXLive yourself.[5]

```
\subsection{Edit Text}
This is what \textbf{LaTeX} is for. LaTeX is a
freely-available, professional-quality typesetting
system. It takes text marked up in a way that
describes the structure and formatting of the
document (where the sections and subsections are,
for example, or whether text should be \textbf{in
 bold face} or \emph{emphasized}) and typesets it
properly. If you have ever edited the HTML of a
web page, you'll know the general idea of a markup
language. If you haven't, the easiest way to
understand what I mean is to look at a segment of
LaTeX markup. An example is shown in Listing
\ref{lst:latex}.
```

Listing 1: Part of the LaTeX source for an earlier version of this document.

LaTeX works best with some tools that help you take full advantage of it with a minimum of fuss. You can manage bibliographical references in LaTeX documents using **BibTeX**. It does the same job as **Endnote**, the commercial plug-in for managing references in Microsoft Word. BibTeX comes with any standard LaTeX installation. Whichever text editor or word processor you use, you should strongly consider some kind of reference-manager software for your bibliographies. It saves a tremendous amount of time because

[4]One of the reasons that Emacs' keyboard shortcuts are so strange is that they have their roots in a model of computer that laid out its command and function keys differently from modern keyboards. It's that old.

[5]For more about these distributions of TeX, see the LaTeX project page. The proliferation of "-TeX" acronyms and names can be confusing to newcomers, as they may refer to a distribution of an entire TeX/LaTeX platform (as with MikTeX or MacTeX), or to a particular program or utility that comes with these distributions (such as BibTeX, for dealing with bibliographies), or to some bit of software that allows something else to work with or talk to the TeX system.

[6]If you plan to use BibTeX to manage your references, take a look at BibLaTeX, a package by Philipp Lehman, together with Biber, a replacement for BibTeX. BibLaTeX and Biber are not yet officially stable, but they are very well-documented, in wide use, and will soon jointly replace BibTeX as the standard way to process bibliographies in LaTeX. I recommend you use them instead of older configurations (such as BibTeX and

you can easily switch between bibliographical formats, and you don't have to worry whether every item referenced in your dissertation or paper is contained in the bibliography.[6]

AUCTeX and **RefTeX** are available for Emacs. These packages allow Emacs to understand the ins-and-outs of typesetting LaTeX documents, color-coding the marked-up text to make it easier to read, providing shortcuts to LaTeX's formatting commands, and helping you manage references to Figures, Tables and bibliographic citations in the text. These packages could also be listed under the "Minimize Error" section below, because they help ensure that your references and bibliography will be complete and consistently formatted.[7]

More information on Emacs and LaTeX is readily available via Google, and there are several excellent books available to help you get started. Kopka and Daly (2003) and Mittlebach et al. (2004) are good resources for learning LaTeX.

Analyze Data and Present Results

You will probably be doing some—perhaps a great deal—of quantitative data analysis. **R** is an environment for statistical computing. It's exceptionally well-supported, continually improving, and has a very active expert-user community who have produced many add-on packages. R has the ability to produce sophisticated and high-quality statistical graphics. The documentation that comes with the software is complete, if somewhat terse, but there are a large number of excellent reference and teaching texts that illustrate its use. These include Dalgaard (2002); Venables and Ripley (2002); Maindonald and Braun (2003); Fox (2002); Harrell (2001), and Gelman and Hill (2007). Although it is a command-line tool at its core, it has a good graphical interface as well. You can download it from The R Project Homepage.

R can be used directly within Emacs by way of a package called **ESS** (for "Emacs Speaks Statistics"). As shown in Figure 1, it allows you to work with your code in one Emacs frame and a live R session in another right beside it. Because everything is inside Emacs, it is easy to do things like send a chunk of your code over to R using a keystroke. This is a very efficient way of doing interactive data analysis while building up code you can use again in future.

Figure 1: An R session running inside Emacs using ESS. The R code file is on the left, and R itself is running on the right. You write in the left-hand pane and use a keyboard shortcut to send bits of code over to the right-hand pane, where they are executed by R.

the Natbib package) which you may come across in other introductory discussions.

[7] A note about fonts and LaTeX. It used to be that getting LaTeX to use anything but a relatively small set of fonts was a very tedious business. This is no longer the case. The XeTeX engine makes it trivially easy to use any Postscript, TrueType or OpenType font installed on your system. XeTeX was originally developed for use on the Mac, but is available now for Linux and Windows as well. If you want to use a variety of fonts with an absolute minimum of fuss, use the `xelatex` command to typset your documents instead of `pdflatex`.

You'll present your results in papers, but also in talks where you will likely use some kind of presentation software. Microsoft's PowerPoint is the most common application, but there are better ones. If you wish, you can use LaTeX, too, creating slides with the Beamer document class and displaying them as full-screen PDFs. Alternatively, on Mac OS X Apple's Keynote is very good. One benefit of using a Mac is that PDF is the operating system's native display format, so PDF graphics created in R can simply be dropped into Keynote without any compatibility problems. Microsoft's PowerPoint is less friendly toward the clean integration of PDF files in presentations.[8]

Minimize Error

We have already seen how the right set of tools can save you time by automatically highlighting the syntax of your code, ensuring everything you cite ends up in your bibliography, picking out mistakes in your markup, and providing templates for commonly-used methods or functions. Your time is saved twice over: you don't repeat yourself, and you make fewer errors you'd otherwise have to fix. When it comes to managing ongoing projects, minimizing error means addressing two related problems. The first is to find ways to further reduce the opportunity for errors to creep in without you noticing. This is especially important when it comes to coding and analyzing data. The second is to find a way to figure out, retrospectively, what it was you did to generate a particular result. These problems are obviously related, in that it's easy to make a retrospective assessment of well-documented and error-free work. As a practical matter, we want a convenient way to document work as we go, so that we can retrace our steps in order to reproduce our results. We'll also want to be able to smoothly recover from disaster when it befalls us.

Errors in data analysis often well up out of the gap that typically exists between the procedure used to produce a figure or table in a paper and the subsequent use of that output later. In the ordinary way of doing things, you have the code for your data analysis in one file, the output it produced in another, and the text of your paper in a third file. You do the analysis, collect the output and copy the relevant results into your paper, often manually reformatting them on the way. Each of these transitions introduces the opportunity for error. In particular, it is easy for a table of results to get detached from the sequence of steps that produced it. Almost everyone who has written a quantitative paper has been confronted with the problem of reading an old draft containing results or figures that need to be revisited or reproduced (as a result of peer-review, say) but which lack any information about the circumstances of their creation. Academic papers take a long time to get through the cycle of writing, review, revision, and publication, even when you're working hard the whole time. It is not uncommon to have to return to something you did two years previously in order to answer some question or other from a reviewer. You do not want to have to do everything over from scratch in order to get the right answer. I am not exaggerating when I say that, whatever the challenges of replicating the results of someone else's quantitative analysis, after a fairly short period of time authors themselves find it hard to replicate their *own* work. Computer Science people have a term of art for the inevitable process of decay that overtakes a project simply in virtue of its being left alone on the hard drive for six months or more: bit–rot.

Literate Programming with Sweave

A first step toward closing this gap is to use **Sweave** when doing quantitative analysis in R. Sweave is a *literate programming* framework designed to integrate the documentation of a data analysis and its execution. You write the text of your paper (or, more often, your report documenting a data analysis) as normal. Whenever you want to run a model, produce a table or display a figure, rather than paste in the results of your work from elsewhere, you write down the R code that will produce the output you want. These "chunks" of code are distinguished from the regular text by a special delimiter at their beginning and end. A small sample is shown in Listing 2. The code chunk begins with the line `<<load-data, echo=true>>=`. The character sequence `<<>>=` is the marker for the beginning of a chunk: `load-data` is just a label for the chunk and `echo=true` is an option. The end of each chunk is marked by the `@` symbol.

When you're ready, you "weave" the file: you feed it to R, which processes the code chunks, and spits out a finished version where the code chunks have been replaced by their output. This is now a well-formed LaTeX file that you can then turn into a PDF document in the normal way. Conversely, if you just want to extract the code you've written from the surrounding text, then you "tangle" the file, which results in an `.R` file. It's pretty straightforward in practice. Sweave files can be edited in Emacs, as ESS understands them.

The strength of this approach is that is makes it much easier to document your work properly. There is just one file for both the data analysis and the writeup. The output of the analysis is created on the fly, and the code to do it is embedded in the paper. If you need to do multiple but identical (or very similar) analyses of different bits of data,

[8] The actual business of *giving* talks based on your work is beyond the scope of this discussion. Suffice to say that there is plenty of good advice available via Google, and you should pay attention to it.

[9] For some real-world case-studies of reproductions of peer-reviewed studies using Sweave, and the errors uncovered as a result, see Hothorn and

Sweave can make generating consistent and reliable reports much easier.[9]

```
\subsection*{Some exploratory analysis}
In this section we do some exploratory analysis of
the data. We begin by reading in the data file:

<<load-data, echo=true>>=
 ## load the data.
 my.data <- read.csv("data/sampledata.csv",header=TRUE)
@ % The closing delimiter ends the code chunk.

We should \emph{plot the data} to take a look at
it:

<<plot-data, echo=true>>=
 ## make some plots.
 with(my.data, plot(x,y))
@

Maybe run some models, too.

<<ols-model echo=true>>=
 ## OLS model
 out <- lm(y ~x1 + x2,data=my.data)
 summary(out)
@

This concludes the exploratory analysis.
```

Listing 2: LaTeX and R code mixed together in an Sweave file.

A weakness of the Sweave model is that when you make changes, you have to reprocess all of the code to reproduce the final LaTeX file. If your analysis is computationally intensive this can take a long time. You can go a little ways toward working around this by designing projects so that they are relatively modular, which is good practice anyway. But for projects that are unavoidably large or computationally intensive, the add-on package `cacheSweave`, available from the R website, does a good job alleviating the problem.

Literate Programming with Org-mode

Org-mode is an Emacs mode originally designed to make it easier to take notes, make outlines and manage to-do lists. Very much in the spirit of Emacs itself, its users have extended it so that it is capable of all kinds of other things, too, such as calendar management, time-tracking, and so on. One very powerful extension to org-mode is Org-Babel, which is a generalized literate-programming framework for org-mode documents. It works like Sweave, except that instead of writing your papers, reports, or documentation in LaTeX and your code in R, you write text in Org-mode's lightweight markup syntax and your code in any one of a large number of supported languages. Org-mode has very powerful export capabilities, so it can convert `.org` files to LaTeX, HTML, and many other formats quite easily. Examples of it in use can be seen at the Org-babel website. This article was written as a plain-text `.org` file and the raw version is available for inspection as a repository on GitHub. You can treat Org-Babel just as you would Sweave, or you can take advantage of the fact that it's fully part of org-mode and get all of the latter's functionality for free.

For example, Figure 2 is generated on the fly from source-code blocks included in the `.org` source for this article. A piece of code can be executed, displayed, or both — as in the case of Listing 3. Then the figure can be created directly. I don't show the code for this here, but you can look in the source file for this article to see how it's done.

```
library(ggplot2)
tea <- rnorm(100)
biscuits <- tea + rnorm(100,0,1.3)
```

Listing 3: "Live" code contained in this document.

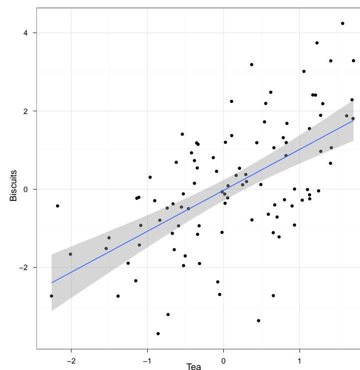


Figure 2: A figure produced from code embedded in the source (`.org`) file for this article.

[9] Leisch (2011).

Use Revision Control

The task of documenting your work at the level of particular pieces of code or edits to paragraphs in individual files can become more involved over time, as projects grow and change. This can pose a challenge to the Literate Programming model. Moreover, what if you are not doing statistical analysis at all, but still want to keep track of your work as it develops? The best thing to do is to institute some kind of **version control system** to keep a complete record of changes to a file, a folder, or a project. This can be used in conjunction with or independently of a documentation method like Sweave. A good version control system allows you to easily "rewind the tape" to earlier incarnations of your notes, drafts, papers and code, and lets you keep track of what's current without having to keep directories full of files with confusingly similar names like `Paper-1.txt`, `Paper-2.txt`, `Paper-conferenceversion.txt`, and so on.

In the social sciences and humanities, you are most likely to have come across the idea of version control by way of the "Track Changes" feature in Microsoft Word, which lets you see the edits you and your collaborators have made to a document. Think of true version control as a way to keep track of whole projects (not just individual documents) in a much better-organized, comprehensive, and transparent fashion. Modern version control systems such as Subversion, Mercurial and Git can, if needed, manage very large projects with many branches spread across multiple users. As such, you have to get used to some new concepts related to tracking your files, and then learn how your version control system implements these concepts. Because of their power, these tools might seem like overkill for individual users. (Again, though, many people find Word's "Track Changes" feature indispensable once they begin using it.) But version control systems can be used quite straightforwardly in a basic fashion, and they increasingly come with front ends that can be easily integrated with your text editor.[10] Moreover, you can meet these systems half way. The excellent DropBox, for example, allows you to share files between different computers you own, or with collaborators or general public. But it also automatically version-controls the contents of these folders.

Revision control has significant benefits. A tool like Git or Mercurial combines the virtues of version control with backups, because every repository is a complete, self-contained, cryptographically signed copy of the project. It puts you in the habit of recording (or "committing") changes to a file or project as you work on it, and (briefly) documenting those changes as you go. It allows you to easily test out alternative lines of development by branching a project. It allows collaborators to work on a project at the same time without sending endless versions of the "master" copy back and forth via email, and it provides powerful tools that allow you to automatically merge or (when necessary) manually compare changes that you or others have made. Perhaps most importantly, it lets you revisit any stage of a project's development at will and reconstruct what it was you were doing. This can be tremendously useful whether you are writing code for a quantitative analysis, managing field notes, or writing a paper.[11] While you will probably not need to control everything in this way (though some people do), I *strongly* suggest you consider managing at least the core set of text files that make up your project (e.g., the code that does the analysis and generates your tables and figures; the dataset itself; your notes and working papers, the chapters of your dissertation, etc). As time goes by you will generate a complete, annotated record of your actions that is also a backup of your project at every stage of its development. Services such as GitHub allow you to store public or (for a fee) private project repositories and so can be a way to back up work offsite as well as a platform for collaboration and documentation of your work.

You don't need backups until you really, really need them

Regardless of whether you choose to use a formal revision control system, you should nevertheless have *some* kind of systematic method for keeping track of versions of your files. The task of backing up and synchronizing your files is related to the question of version control. Apple's Time Machine software, for example, backs up and versions your files, allowing you to step back to particular instances of the file you want. Other GUI-based file synchronization tools, such as DropBox and SugarSync, are available across various platforms.

Even if you have no need for a synchronization application, you will still need to back up your work regularly. Because you are lazy and prone to magical thinking, you will not do this responsibly by yourself. This is why the most useful backup systems are the ones that require a minimum amount of work to set up and, once organized, back up everything automatically to an external (or remote) hard disk without you having to remember to do anything. On Macs, Apple's **Time Machine** software is built in to the operating system and makes backups very easy. On Linux, you can use rsync for backups. It is also worth looking into a secure, peer-to-peer, or offsite backup service like Crash-

[10] Emacs comes with support for a variety of VCS systems built in. There's also a very good add-on package, Magit, devoted specifically to Git.

[11] Mercurial and Git are *distributed* revision control systems (DVCSs) which can handle projects with many contributors and very complex, decentralized structures. Bryan O'Sullivan's *Distributed Version Control with Mercurial* is a free, comprehensive guide to one of the main DVCS tools, but also provides a clear account of how modern version-control systems have developed, together with the main concepts behind them. For Git, I recommend starting at this site and following the links to the documentation.

plan, Spider Oak, or Backblaze. Offsite backup means that in the event (unlikely, but not unheard of) that your computer *and* your local backups are stolen or destroyed, you will still have copies of your files.[12] As Jamie Zawinski has remarked, when it comes to losing your data "The universe tends toward maximum irony. Don't push it."

Pulling it Together: An Emacs Starter Kit for the Social Sciences

A step-by-step guide to downloading and installing every piece of software I've mentioned so far is beyond the scope of this discussion. But let's say you take the plunge and download Emacs, a TEX distribution, R, and maybe even Git. Now what? If you're going to work in Emacs, there are a variety of tweaks and add-ons that are very helpful but not set by default. To make things a little easier, I maintain an Emacs Starter Kit for the Social Sciences. It's designed to smooth out Emacs' rough edges by giving you a drop-in collection of default settings, as well as automatically installing some important add-on packages. It will, I hope, help you skirt the abyss of Setting Things Up Forever. The original version of the kit was written by Phil Hagelberg and was made to go with the "Meet Emacs" screencast mentioned above. It was aimed at software developers in general. Eric Schulte, one of the authors of Org-babel, modified and further extended the kit. My version adds support for AucTeX, ESS, and other bits and pieces mentioned here. As you can see if you follow the links, the kit is stored on GitHub and you are free to fork it and modify it to your own liking.

Do I Have to Use this Stuff?

Pros and Cons

Using Emacs, LATEX and R together has four main advantages. First, these applications are all free. You can try them out without much in the way of monetary expense. (Your time may be a different matter, but although you don't believe me, you have more of that now than you will later.) Second, they are all open-source projects and are all available for Mac OS X, Linux and Windows. Portability is important, as is the long-term viability of the platform you choose to work with. If you change your computing system, your work can move with you easily. Third, they deliberately implement "best practices" in their default configurations. Writing documents in LATEX encourages you to produce papers with a clear structure, and the output itself is of very high quality aesthetically. Similarly, by default R implements modern statistical methods in a way that discourages you from thinking about statistics in terms of canned solutions to standard problems. It also produces figures that accord with accepted standards of efficient and effective information design. And fourth, the applications are closely integrated. Everything (including version control systems) can work inside Emacs, and all of them talk to or can take advantage of the others. R can output LATEX tables, for instance, even if you don't use Sweave.

None of these applications is perfect. They are powerful, but they can be hard to learn. However, you don't have to start out using all of them at once, or learn everything about them right away — the only thing you *really* need to start doing immediately is keeping good backups. There are a number of ways to try them out in whole or in part. You could try LATEX first, using any editor. Or you could try Emacs and LATEX together. You could begin using R and its GUI.[13] Sweave or Org-babel can be left till last, though I have found these increasingly useful since I've started using them, and wish that all of my old project directories had some documentation in one or other of these formats. Revision control is more beneficial when implemented at the beginning of projects, and best of all when committing changes to a project becomes a habit of work. But it can be added at any time.

You are not condemned to use these applications forever, either. LATEX and (especially) Org-mode documents can be converted into many other formats. Your text files are editable in any other text editor. Statistical code is by nature much less portable, but the openness of R means that it is not likely to become obsolete or inaccessible any time soon.

A disadvantage of these particular applications is that I'm in a minority with respect to other people in my field. This is less and less true in the case of R, but remains so for LATEX. (It also varies across social science disciplines.) Most people use Microsoft Word to write papers, and if you're collaborating with people (people you can't boss around, I mean) this can be an issue. Similarly, journals and presses in my field often do not accept material marked up in LATEX, though again there are important ex-

[12] I know of someone whose office building was hit by a tornado. She returned to find her files and computer sitting in a foot of water. You never know.

[13] If you already know Emacs, you should certainly try R using ESS instead of the R GUI, though.

[14] Getting from LATEX to Word is easiest via HTML. But if you really want to maximize the portability of your papers or especially your reading notes or memos, consider writing them in a modern lightweight markup format. Org-mode's native format is effectively one of these already, and it provides built-in support for export to many others. An org-mode file can also be exported easily to rich-text or HTML, and from there Word or Google Docs will open it. Other options for lightweight markup include Markdown or its close relation, MultiMarkdown. Documents written in these formats are easy to read in their plain-text form but can be simply and directly converted into HTML, Rich Text, LATEX, Word, or other formats. TextMate has good support for Markdown and MultiMarkdown, allowing you to do these conversions more or less automatically. John MacFarlane's Pandoc is a powerful tool that can read markdown and (subsets of) reStructuredText, HTML, Org, and LATEX; and it can write

ceptions. Converting files to a format Word understands can be tedious (although it is quite doable).[14] I find these difficulties are outweighed by the day-to-day benefits of using these applications, on the one hand, and their longer-term advantages of portability and simplicity, on the other. Your mileage, as they say, may vary.

Some Alternatives

There are many other applications you might put at the center of your workflow, depending on need, personal preference, willingness to pay some money, or desire to work on a specific platform. For text editing, especially, there is a plethora of choices. On the Mac, quality editors include BBEdit (beloved of many web developers), Smultron, and TextMate (shown in Figure 3). TextMate has strong support for LaTeX and good (meaning, ESS-like) support for R. Because it is a modern application written specifically for the Mac it can take advantage of features of OS X that Emacs cannot, and is much better integrated with the rest of the operating system. It also has good support for many of the ancillary applications discussed above, such as version control systems.[15] On Linux, an alternative to Emacs is vi or Vim, but there are many others. For Windows there is Textpad, WinEdt, UltraEdit, and NotePad++ amongst many others. Most of these applications have strong support for LaTeX and some also have good support for statistics programming.

For statistical analysis in the social sciences, the main alternative to R is Stata. Stata is not free, but like R it is versatile, powerful, extensible and available for all the main computing platforms. It has a large body of user-contributed software. In recent versions its graphics capabilities have improved a great deal. ESS can run Stata inside Emacs in the same way as it can do for R. Other editors can also be made to work with Stata: Jeremy Freese provides an UltraEdit syntax highlighting file for Stata. There is a Stata mode for WinEdt. Friedrich Huebler has a guide for integrating Stata with programming editors. Gabriel Rossman's blog Code and Culture has many examples of using Stata in the day-to-day business of analyzing sociological data.

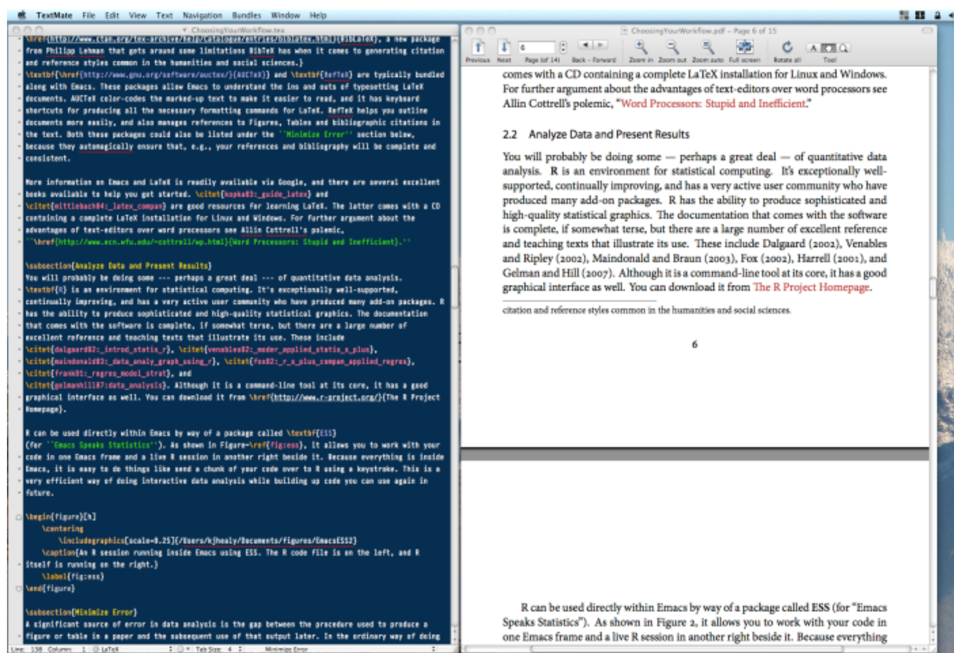


Figure 3: An earlier version of this document being edited in TextMate.

to MarkDown, reStructuredText, HTML, LaTeX, ConTeXt, RTF, DocBook XML, groff man, and S5 HTML slide shows. Pandoc is terrifically useful and I recommend checking it out. Lightweight markup languages like Markdown and Textile have a harder time dealing with some of the requirements of scholarly writing, especially the machinery of bibliographies and citations. If they could handle this task elegantly they would be almost perfect, but in practice this would probably just turn them back into something much less lightweight. Even here, though, good progress is being made as Pandoc will soon include support for citations.

[15] Its next major version, TextMate 2, has been in development for a very long time and is awaited with a mixture of near-religious hope, chronic anxiety and deep frustration by users of the original.

Amongst social scientists, revision control is perhaps the least widely-used of the tools I have discussed. But I am convinced that it is the most important one over the long term. While tools like Git and Mercurial take a little getting used to both conceptually and in practice, the services they provide are extremely useful. It is already quite easy to use version control in conjunction with some of the text editors discussed above: Emacs and TextMate both have support for various DVCSs. On the Mac, CornerStone and Versions are full-featured applications designed to make it easy to use Subversion. Taking a longer view, version control is likely to become more widely available through intermediary services or even as part of the basic functionality of operating systems.

A Broader Perspective

It would be nice if all you needed to do your work was a box software of software tricks and shortcuts. But of course it's a bit more complicated than that. In order to get to the point where you can write a paper, you need to be organized enough to have read the right literature, maybe collected some data, and most importantly asked an interesting question in the first place. No amount of software is going to solve those problems for you. Too much concern with the details of your setup hinders your work. Indeed — and I speak from experience here — this concern is itself a kind self-imposed distraction that placates work-related anxiety in the short term while storing up more of it for later.[16] On the hardware side, there's the absurd productivity counterpart to the hedonic treadmill, where for some reason it's hard to get through the to-do list even though the caf you're at contains more computing power than was available to the Pentagon in 1965. On the software side, the besetting vice of productivity-enhancing software is the tendency to waste a lot of your time installing, updating, and generally obsessing about your productivity-enhancing software.[17] Even more generally, efficient workflow habits are themselves just a means to the end of completing the projects you are really interested in, of making the things you want to make, of finding the answers to the questions that brought you to graduate school. The process of idea generation and project management can be run well, too, and perhaps even the business of choosing what the projects should be in the first place. But this is not the place — and I am not the person — to be giving advice about that.

All of which is just to reiterate that it's the principles of workflow management that are important. The software is just a means to an end. One of the smartest, most productive people I've ever known spent half of his career writing on a typewriter and the other half on an IBM Displaywriter. His backup solution for having hopelessly outdated hardware was to keep a spare Displaywriter in a nearby closet, in case the first one broke. It never did.

References

Dalgaard, Peter. 2002. *Introductory Statistics with R.* New York, NY: Springer.

Fox, John. 2002. *An R and S-Plus Companion to Applied Regression.* Thousand Oaks, CA: Sage.

Harrell, Frank E. 2001. *Regression Modeling Strategies.* New York, NY: Springer.

Gelman, Andrew and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models.* New York, NY: Cambridge University Press.

Hothon, Torsten and Friedrich Leisch. 2011. "Case Studies in Reproducibility." *Briefings in Bioinformatics* XX (January):1–13.

Kopka, Helmut and Patrick W. Daly. 2003. *Guide to LaTeX.* New York, NY: Addison Wesley.

Maindonald, John and John Braun. 2003. *Data Analysis and Graphics Using R: An Example-Based approach.* New York, NY: Cambridge University Press.

Mittelbach, Frank and Michel Goossens with Johannes Braams, David Carlisle, and Chris Rowley. 2004. *The LaTeX Companion.* New York, NY: Addison Wesley.

Venables, W.N. and B.D. Ripley. 2002. *Modern Applied Statistics with S.* New York, NY: Springer.

[16] See Merlin Mann, amongst others, for more on this point.

[17] Mike Hall's brilliant "Org-Mode in your Pocket is a GNU-Shaped Devil" makes this point very well.

Collaboration for Social Scientists, or Software is the Easy Part

Mark M. Fredrickson, Paul F. Testa & Nils B. Weidmann
University of Illinois & Yale University
mark.m.fredrickson@gmail.com

Collaboration Basics

In this article, we consider how to improve two different modes of collaboration: synchronous and asynchronous. When working synchronously, contributors focus on *the same portions of the research at the same time*. Of course, virtually any research project will require collaborators to spend time working on either different portions of the project or working on the same sections but at different times. We label this form of collaboration asynchronous. Asynchronous collaboration requires more careful attention to dividing labor, and we spend more time providing software solutions in this domain. These suggestions are based on what has worked *for us*. These suggestions are grounded in experience, and we think they are useful techniques for any team to adopt. We have also found that software is the easy part of any collaboration while the personal and intellectual parts of collaboration are both more difficult and more fulfilling than playing with software tools. Hopefully, adopting some of these techniques may help your team get past technical details faster and down to the real business of producing research.

Synchronous Collaboration

While it might appear that only collaborators at the same institution or who can frequently meet face to face will benefit from synchronous collaboration techniques, many of these techniques rely on networked computers or can be applied over video chat or speaker phone.

We begin by importing some techniques from software engineering. In recent years, so-called Agile programming and project management approaches to software engineering have become popular, especially at start ups and younger development shops. Many techniques fall under the umbrella of "Agile" methods, including suggestions for organizing teams, minimizing unnecessary meetings, and handling client requests. While social scientists could benefit from these suggestions, we tend to be our own clients and work in smaller teams than programmers. One technique we do think would be of benefit to social scientists would be the concept of pair programming. Pair programming places two programmers at the same computer: one screen, one keyboard, two heads. One programmer takes the lead to write software, while the second provides suggestions, acts as a sounding board, catches errors, and questions assumptions made by the first programmer.

While it may sound wasteful to place two collaborators in front of a single computer and have them both focus on the same task, the technique can lead to *more* code being written and *higher quality* software as well. The key insight is that typing is rarely the bottleneck for producing code. Having a second person on hand to help with the concepts, design, and implementation cuts down on time spent chasing dead-ends or time wasted on simple bugs. If you have spent several hours on a problem only to realize your mistake while explaining the problem to someone, you will see the immediate benefits of pair programming.

In practice, pair programming need not have both subjects staring at the same screen at the same time. One programmer may be writing code, while the other looks into API documentation, writes unit tests, or provides documentation, but is immediately available to support the first programmer. Social scientists might additionally adapt this design to having one collaborator writing code, while the other simultaneously writes text in support of the analysis.

Collaborators need not be in the same physical space. There are several tools for real-time co-editing of documents. Wikipedia provides a fairly detailed list of collaborative real-time editors. All of these editors allow multiple authors to simultaneously edit documents, which may even be a useful feature to pair programmers in the same physical space. Since this issue of TPM is strongly encouraging learning and using a text editor, you may wish to favor editors that allow for simultaneous editing. At a minimum, GNU Screen provides an immediate solution for Emacs and VIM users who wish to pair program. More advanced uses may require a editor plugin or separate editor.

In addition to managing file editing, some real time editors also facilitate verbal communication. Of course, if your editor does not immediately provide this service, a call via Skype or Google Chat can fulfill communication needs. Of course, these tools can also be of use to collaborators, even they forgo the pair programming model.

Asynchronous Collaboration

Even the closest of collaborators need to spend some time apart. The primary challenge for effective asynchronous collaboration is ensuring that this time is spent contributing to the final product and not wondering where one's changes went and why one's partner is working off an obsolete draft. We suggest three tools for improving asynchronous collaboration: sharing files to provide common, immediate access to the latest documents; version control systems to safely manage simultaneous updates from collaborators; and build scripts to encapsulate the techniques necessary to create artifacts, such as PDF documents.

Shared Files

Early in collaboration projects, we find it is common to exchange ideas and drafts via email. While simple to use, managing files and staying up-to-date with collaborators quickly becomes a burden to the project. Is the current version the document in the email from yesterday? Or the version you have on your hard drive? If you've asked these questions, you understand the problem of sharing files via email.

Services such as Dropbox provide a relatively simple and elegant solution to the challenge of keeping collaborators on same page. Users download a desktop application that creates a DropBox folder on their computer. Files stored in this folder are available online through the user's account, as well as on any other computer the user has installed DropBox on or shared his or her folder with. Changes made to a file are automatically synchronized with a user's online account and other computers, ensuring that collaborators are always working off the most recent copy.

Think of DropBox as the sort of iPhone of file sharing services (in fact there are several DropBox-supporting applications for the iPhone and other mobile devices). It's relatively easy to use, synchronizes changes automatically and offers a degree of version control (up to thirty days for free users and unlimited for paying customers). DropBox is well suited for the less technically savvy social scientists. More advanced users and frequent collaborators may chafe at the limitations of its "freemium" services and find that some of its more user friendly features promote bad collaboration habits, specifically simultaneous editing of documents.

Version Control

While shared files solve the problem of all collaborators having access to common resources, simple file servers provide no guarantee that collaborators will not unintentionally overwrite each other's changes. Consider for example the following scenario, both you and your collaborator are working on the same LaTeX file. You are editing the abstract, while your partner changes a few lines in the conclusion. You save your work to the shared area, while unknown to you, your partner saved her work only a few minutes before. Even though you were working in an entirely different part of the file, your changes overwrite those of your partner, silently dropping her work and reverting back to the old conclusion. Your partner's work has been lost.

This is exactly what so-called version control (VC) systems are designed to avoid. Developed for software engineering, these systems enable multiple authors to work on the same documents and safely merge their changes. Version control systems come in two flavors: centralized and distributed. The basic setup of a centralized version control system is simple. Collaborators work off a central repository, where the version-controlled documents reside. Upon joining a project, each collaborator obtains a working copy of each of these documents. Versions are tracked by means of revision numbers, assigned and maintained by the VC system. When you obtain a document ("check out"), your local copy is assigned the revision number of the repository at that time.

As the project proceeds, collaborators change different parts of their working copies of the same document, which brings us to the situation described above: how do your modified abstract and the conclusion written by your collaborator end up in the same document? Let's assume that you are finished writing the abstract, while your partner is still working on the conclusion. Your and your collaborator's working copies are both currently in revision 22. You upload your changes to the central repository, a step that in the VC world is called a "commit". When you do so, a new revision (rev. 23) is created in the central repository. Once your partner attempts to upload the new document with the conclusion, the VC system prevents her from doing so, because the working copy she is using is not up to date and still in revision 22. Prior to committing her changes, she has to update her working copy to the current revision of the repository (rev. 23). This is done by performing an "update" operation in the VC system, during with the updated parts in the main document (the abstract) are merged into the local working copy of your partner, while, and this is important, preserving her newly written conclusion.[1]

Various implementations of this centralized version control model exist. One of the most popular systems is Subversion (SVN), which is available free of charge (both the server required to set up a central repository, and a command line client to check out and maintain a local working copy). However, various alternatives exist that make life easier for less tech-savvy people. Projectlocker offers Subversion hosting, and a free account can store up to 300 MB. If the documents you put under version control mainly include plain-text files, such as those containing Latex and R code, this amount of space is more than enough. TortoiseSVN is a graphical Subversion client for Windows users that integrates nicely with the Windows Explorer. A similar project exists for OS X (SCPlugin), but due to its early development stage, some people may prefer commercial prod-

[1] Things become more tricky if you and your partner modify the same parts of a document (for example, you both provide an abstract). In this case, human intervention is required: the VC system highlights the conflicting changes, but lets you decide what should be the final version. Most importantly, the VC system will not delete work with out your explicit consent. VC systems (generally) assess changes on a line-by-line basis. You can minimize your conflicts by editing the fewest lines possible for any change and by making frequent check-ins. You may find it helpful to use hard wrapping in your editor of choice when editing `.tex` files, at say 80 characters. This will automatically break long sentences into smaller chunks from the perspective of the VC system and provide for fewer headaches down the road.

ucts (Cornerstone or Versions).

As the name implies, distributed version control systems spread out the work of managing repositories while still emphasizing collaboration. The main difference is that instead of a single, server-side repository, each collaborator has a local repository and a local working copy. The advantage of this mode of version control is that each collaborator can work offline, make small, frequent, and fast commits, and still communicate with other repositories (usually by pushing to a server as with a centralized system). By their decentralized nature, distributed VCs make "branching" a natural technique. "Branching" refers to making parallel copies of the repository to try out ideas, make complex changes, or just play around in a sandbox. For example, say you wish to add an additional data source to a paper to see if it adds strength to the argument, but you and/or your collaborators also will be editing the document simultaneously. By adding a branch, you have a safe place to add data and change wide sections of the analysis, but can still edit text on the main document. If the additional data source proves useful, you can merge the branch back to the main code base, maintaining any edits. If the additional data is not useful, you have not harmed your primary document. In a distributed version control system, every collaborator is working his or her own branch, so these merges are natural and well supported.[2]

The downside of distributed systems is that they require slightly more overhead than a centralized system. To commit changes in a centralized system, a user pushes his changes to a centralized server and they are immediately available to other users. In a decentralized system, a user commits locally, then pushes to a remote server. This remote server can either be commonly shared or unique to the individual collaborator. In the latter case, the other collaborators will need to pull in changes from their associates' online repositories. To some degree this additional complexity can be hidden by tools. There are number of online services providing tooling around distributed version controls systems (GitHub for Git, Launchpad for Bazaar, and BitBucket for Mercurial — three popular DVCs). Friendly graphical client side tools for working with these version control systems also exist.

So which to use? In our experience, centralized version control (specifically SVN) is the easiest for collaborators to use, but has required the most work to set up the server. If you have technical support through your institution or plan to use an online service, the maturity and simplicity of SVN could be a big benefit to your team. On the other hand, distributed version control systems can be quick to get up and running. For example, for a paper involving people with a mix of technical ability, we used a hybrid approach: Dropbox held the files and a shared distributed VC repository for the group. The less technical contributors edited the files directly on Dropbox. Others pushed and pulled to their own local repositories.

Single users can also make quick use of distributed systems to have an undo stack of previous documents and a place to create sandbox branches. In both the local user and the Dropbox case, it would be easy to move from such simple used of VC systems to the use of a shared remote server for more serious collaboration.[3]

Scripts

While version control systems will keep track of changes and make sure the source is in a consistent state, source code alone (and here we include things such as .tex files and data), does not completely describe *how* to create the research. Consider, for example, creating a figure for a paper. Being a good collaborator, you take the time to write the figure generating code in `figure.R` and insert it into the main LaTeXfile using `\includegraphics`. The figure relies on data in `data.csv` and some code in `models.R`. While it is straightforward for the original author to create this graphic, will it be obvious to others in the project that if either the data or models change, the figure should also change? For lengthy or large projects, even the original author may forget which files depend on others.

Again borrowing for software engineering, we suggest the use of build files to solve this problem. Build files explicitly state dependencies between files and explain how to generate artifacts, such as PDF files. As an added benefit, build files automate the creation of artifacts and ensure that files are built in the proper order. Most importantly, build files ensure that artifacts are updated when source documents, including data, change. Returning to the figure example above, we could notate the necessary conditions for updating the figure and the main PDF with the following GNU `Makefile`[4]:

```
paper.pdf: figure.pdf paper.tex
  latexmk -pdf paper.tex

figure.pdf: figure.R data.csv
  R --silent --file = figure.R
```

[2]It is fair to note that systems like Subversion support branching; however, SVN places more burden on the user to merge branches than the distributed systems. This situation is improving for SVN and will likely be a non-issue in the future.

[3]To collaboratively produce this article we used the DVCS Git and the free-for-public-repositories GitHub service. We designated Fredrickson's repository as the "canonical" repository, but Testa and Weidmann had their own repositories. When Testa or Weidmann wished to incorporate a change into the canonical repository, Fredrickson pulled in the changes. You can see the repository and the entire history of the project online .

[4]We use the classic and widely available GNU `make` system, but other build systems exist. Some of these systems, for example `Rake` for Ruby, allow more programming and customization within the build scripts. Your team may benefit from this extended functionality

The unindented lines indicate *targets*, with a list of dependencies after the colon and a build command on the following indented line. The `make` command checks each target and compares the time stamp on the target with the time stamp on each dependency. If any dependency is newer than the target, the dependency is rebuilt using its command (perhaps recursively building further dependencies) and then builds the target using its command. For example, if `data.csv` is updated, `make` will automatically rebuild `figure.pdf` before rebuilding `paper.pdf`.

Using a `Makefile` simplifies the amount of knowledge any individual on the team has to have regarding creating artifacts. Instead of having to remember and manually implement the build process, all a collaborator has to do is type `make` at the command line and he will be certain to have a properly built version of an artifact, say a PDF document.

To some degree, literate programming tools, such as `Sweave`, minimize the need establish a clear dependency tree in a build script. `Sweave` chunks take the place of having separate files for loading and transforming data, building models, and generating figures. Since files are evaluated top-down, there is an implicit dependency structure, with later chunks depending on earlier chunks. In order to weave the file, all chunks are rebuilt, guaranteeing any changes in early code chunks flow downstream.

While we are heavy users of `Sweave`, we still think explicit build scripts have a role to play. First, certain computations can be time consuming, but do not need to be frequently updated. Simulations and other iterative computations within a `Sweave` document increase the time from making an edit (perhaps to the text) and final output as a PDF. Writing these computations in separate `.R` files eliminates the need to rerun the computations when no code or data has changed.[5] Second, even when using a single `Sweave` file to merge text and code, projects of a reasonable size will include additional files, especially data. These files may have their own build steps or simply be dependencies for the `Sweave` file. Finally, even when using `Sweave`, your team may wish to split up the work into different files for logical or practical reasons. While version control systems are powerful tools, sometimes the best way to collaborate is to divide the work into separate files. Build scripts help with merging the separate files into a unified whole.

While build files indicate which files should be updated when data change, we encourage social scientists to write scripts to update data as well. Again borrowing from software engineering, we have found "database migrations" to be a useful technique in capturing exactly how and why data should change.[6] While a version control system could capture how a `.csv` file changes from one commit to the next, a migration provides the exact steps by which the data are manipulated. For example, consider downloading data from the ANES and the United States Census and joining it into a single table for analysis in `R`. The most familiar approach might be to load both datasets into an interactive `R` session, use the `merge` function to combine them into a single table, and then save the merged data into a `.rda` file. We suggest two alternatives, both of which could be considered "migrations," that provide more information about the steps undertaken to combine the disparate data sources.

The first technique we call "one file to rule them all." In this scenario, you add your ANES and Census data to your version control repository, along with a file `data.R`. This file contains the code to load, clean, and merge the data, possibly saving a `data.rda` file in the process. You can enter `data.rda` as a dependency in your `Makefile`:

```
analysis.tex: analysis.Rnw data.rda
  R CMD Sweave analysis.Rnw

data.rda: data.R anes.csv census.csv
  R --silent --file = data.R
```

If you or your collaborators need to edit the data at a later time, you can update `data.R`. Since it is included in the `Makefile`, downstream files will be appropriately updated as well.

For more complex data needs, you may consider employing a series of migrations, each building off the previous. As a convention, label your migrations in order: `001_load_data.R`, `002_fix_coding.R`, etc. Rather than having a single file manage all data manipulations, each migration is a separate file that loads, manipulates, and saves the updated data. To run the migrations, collaborators run the scripts in sequence starting with `001_...`. This migration strategy has been most successful, for us, when using mixed languages to update the data. Here, for example, is a listing of migrations on a project that pulls in hate crime data, survey information, and Census information from the web and builds a relational database:[7]

```
001_initialize.sql
002_populate.clj
003_remove_redundant_data.sql
004_connect_census_tables.sql
005_hate_groups.clj
006_splc_hatewatch_events.clj
007_coding_events # is a directory of relevant files for 007
```

Where, as an example, `003_connect_census_tables.sql` presupposes a `data.sql` file to create a new table `census` from various sub-tables of Census data:

[5] Recently, caching systems for `Sweave` have appeared. However, a comparison of such caching systems and the `make` system is beyond the scope of this article.

[6] As the name implies, database migrations are most often applied to databases, as compared to flat files, such as `.csv` or `.dta` files.

[7] `.sql` files are SQL files, a relational database language, and `.clj` are Clojure files, a LISP language for the JVM.

```
-- see the .schema for what is in these tables
-- fips gets duplicated as fips:1, fips:2, ...

CREATE VIEW census AS
SELECT * FROM
  counties c LEFT JOIN census_area_pop ap ON c.fips = ap.fips
  LEFT JOIN census_employment e ON c.fips = e.fips
  LEFT JOIN census_foreign_moved fm ON c.fips = fm.fips
  LEFT JOIN census_income i ON c.fips = i.fips
  LEFT JOIN census_language_education le ON c.fips = le.fips
  LEFT JOIN census_occupation o ON c.fips = o.fips
  LEFT JOIN census_race r ON c.fips = r.fips;
```

Splitting migrations into separate files is more overhead, but provides a finer grained record of changes, even in a version control scenario. We consider either single files (like the `data.R` example above) or multiple files good practice. Your team should select the method that best suits your work style and the amount of data cleaning and manipulation required for your project.

Conclusion

While we have stressed software throughout this article, the technology is the easiest part of collaboration. Habits, conventions, and best practices are much harder to achieve. At the same time, software suggests (or makes easier) certain methods of collaboration. Using a version control system requires collaborators to think about which files to add to the shared repository and which files are transitory or local. Similarly, build files help us communicate the steps necessary to build documents to our collaborators. Alternatively, we could just write detailed `README` files, but tools such as subversion, git, and make, add value above and beyond pure description, though they serve a similar purpose.

Nevertheless, agreeing to use `SVN` or `make` is a relatively simple decision, adhering to best practices is much harder. Version control frees us from calling our collaborators on the phone and saying, "Don't touch this file. I'm working on it." But just because one can check in a file without merge conflicts does not mean the document is in a good state. We can still add mutually non-conflicting changes that lead to disastrous results. Working through such disasters is still a matter of communication between collaborators. But even such conflicts are usually quickly resolved. More difficult is maintaining common style and usage throughout a project.[8]

This article has focused on collaboration, but these suggestions also have benefits for *replication*. In a sense, someone replicating your research is simply a future collaborator. Encapsulating changes within a version control system, providing build scripts, and adhering to a consistent style guide all make it easier for a future researcher to replicate our work.[9] These tools also make it easier for other researchers to evaluate our work by adjusting assumptions, using different data sets, or applying new methods to our data. Writing code with a partner at your elbow ensures that at least one other person understands what the code is doing, raising the probability that someone replicating your work will understand it too. Good practices for collaboration make it easier for future partners to understand what we have done and how to do it even as it saves us from headaches and lost time in the here and now.

Minimizing the Damage: Converting LaTeXto Word Using TeX2Word

Shawn Treier
University of Minnesota
satreier@umn.edu

The arduous process of submission, review, and revision, ought to end with a joyful final submission, a cold drink, and a dumb, relaxing movie. Yet, for many LaTeX users, the long ignored or forgotten author guidelines rob the moment of all joy with the line "only Microsoft Word format accepted". For a LaTeX user, converting a document from LaTeX to Word can be painful both because it is not simple to execute but also because it can amount to uglifying a manuscript whose beauty is in part the result of time and effort by the author. In this essay, I hope to return at least some joy to (or at least remove some pain from) those final moments in the publication cycle by discussing what I think is the least painful way to convert LaTeX documents to Word.

I will focus on the use of a commercial program TeX2Word, a product of Chikrii Softlab. It is an extremely cheap program ($45 for academic purposes). One disadvantage is that it works only with Microsoft Word on Windows, and is unavailable for Mac OS X. In my installation, I have

[8]You may wish to adopt a style guide for both coding and text. Google publishes a style guide for R. Style guides for the English language are numerous.

[9]For an example of a paper using these techniques, see the replication archive for Attributing Effects to A Cluster Randomized Get-Out-the-Vote Campaign by Ben B. Hansen and Jake Bowers. Interested researchers can build the project from the ground up using a set of build scripts, including an additional appendix to the paper including details on modifying and extending the replication process.

Word 2003 installed on Windows XP within Parallels running on Mac OS X. Second, it does require another software application: MathType ($57 academic), produced by Design Science, the same company that produces Equation Editor for Word. This requirement is more helpful than hurtful. MathType is an extended version of Equation Editor, and TEX2Word requires the extensions in the program to faithfully convert LATEX equations. The resulting equations look much better than standard Equation Editor, and much easier to manipulate in Word (Among many other advantages, MathType allows the user to enter LATEX commands directly).[1]

Getting Started The conversion process itself is straightforward. TEX2Word installs as an add-in to Word, and adds "*.tex" files as an option to "File → Open". Installed with TEX2Word is a pseudo-compiler, with a standard installation of a LATEX distribution. Opening a TEX file processes the file through the pseudo-compiler, with an output file that is Word format. The biggest disadvantage to this process is that non-standard packages cannot be used. Consequently, before conversion, one must translate anything using a non-standard package into standard LATEX. For instance, use of `dcolumn`, `endfloat`, `setspace`, `rotating` or `multirow` need to be removed. Nevertheless, this is not typically a critical problem, as long as the unsupported package was primarily for the purposes of appearance, and not essential to display the substance of the manuscript (e.g., you are out of luck if you really need to display hierglyphs — incidentally, `hyperref` is included). Most of the tasks handled by these packages will be handled by the typsetter anyway. Furthermore, redefinition of commands through `\newcommand` does not convert, so all instances must be replaced by the full LATEX commands (same is true with user defined math operators, which I replace with `\text{operatorname}`).

Mathematical Notation Because of the reliance on MathType, the conversion of the math works extremely well; even fairly complicated documents look surprisingly good after the conversion. Naturally, there will be some some minor adjustments that will need to be made after the conversion. For instance, in converting an equation such as $y_i \sim N(\mu, \sigma^2)$, the tilde (produced by `\sim`) appears in the document, but in a version of Word without MathType, did not appear. This was corrected simply by editing the MathType object and retyping `\sim`. TEX2Word had difficulties with commands such as `\bigg`, but rarely with any consequence. In the rare cases where a problem arose, replacement with `\left` and `\right` or typing directly into MathType (such as `\big|` for $\big|$) was sufficient. The command `\intertext`, while producing the text correctly, appears at the end of the previous line, and will need to be moved to the next line within the MathType object. Simple equation alignments work perfectly fine (although equations longer than a page should be broken into separate environments), but alignments on two or more positions can be problematic; not surprisingly, since these often require fine-tuning the spacing in LATEX, and this does not translate well. Adjustment simply requires editing the MathType object and typing a number of `\quad`'s and `\qquad`'s to obtain the desired spacing.

Probably the only aspect of math conversion that was disconcerting was the treatment of `\boldsymbol`, which is recognized by MathType, but does not produce a boldface font (at least as of version 6.0); furthermore, `\pmb` is unrecognized (`\mathbf` works fine though for **x** and **y**), and there is no support for the package `bm`. Within MathType, one can insert these characters from menu and select a boldface font, but that is potentially quite cumbersome with numerous boldfaced parameters. Nevertheless, all of these issues, and any other small adjustment that may be required, are extremely minor relative to the amazing conversion for most math notation.[2]

Note, displaying program code will be problematic, since TEX2Word does not convert `verbatim` environments or `\verb` commands. Any "as-is" display will need to be replaced with copy and paste (one can search for the `\begin{vertatim}'s` that will appear in the converted document).[3]

Tabular Floats Conversion of tables works well, and table objects are created, but these tables will require some revisions within Word. First, `\hline` or `\cline` do not work, so you will need to add the lines yourself, a straightforward task in Word. Second, you will need to adjust the column widths. Again, these do not have to be perfect, just cleaned up sufficiently for the typesetter to see the structure of the table. Third, alignments in the table will not carry over from LATEX so those will need to be set in Word. Fourth, `multicolumn` specifications will not always carry over. Sometimes everything is fine, but more often you will need to delete the row, insert a new row, and recreate the multicolumn setup by merging cells in the table object. In my own experience, I once had an extremely large table for which the multicolumn setup changed the format-

[1] Note, you should alert the editor that the equations are MathType objects and are not editable without the add-on.

[2] An occasional issue with text involves comments. The conversion works perfectly fine, but with the TEXshop editor (on Mac OS X) returns do not automatically appear in the document. The consequence is that text commented out may occasionally appear in the converted document.

[3] Surprisingly for a conversion program, the commands to display LATEX and TEX do not produce satisfactory output (I have my own arrangement of the letters saved that I can copy and paste). It should be highly unlikely, though, for any article that needs to be converted to Word to discuss LATEX specifically (such as this one)!

ting so much (making every column *extremely* narrow) that commenting out the row *before* conversion was necessary. Multirow entries must be created by merging cells; a tricky operation in Word (typically requiring several attempts to recreate the appearance from the original document).

Figures Figures do not cause any particular problems during conversion, but do not necessarily appear; all that appears is the caption (this is true whether the format of figures is PDF, PNG, JPEG or EPS). In many cases, this is fine because the journal requires only captions and figure locations, but no actual figures. The figures though, actually are buried in the document as picture objects. If one copies and pastes the caption (and area surrounding the caption), say to the end of the document, the figure as a picture environment will be revealed. These are lower quality images, and the size of the figure will reflect the original dimensions of the figure in the file, not as adjusted by the options to `\includegraphics`. You will still need to send the figures in original format (likely required anyway); I have also sent pages from the original LaTeX document and indicated those as the "camera ready" version, indicating that the versions in the document are useful only in terms of layout. Another option is to delete the hidden figures and insert PNG versions of the figures near the captions.

Bibliography An important issue is how to handle the bibliography. BibTeX works in the pseudo-compiler within WordTeX, but it only provides citations in `plain` style (i.e., numerical references instead of author-year). First, in order for TeX2Word to create the citations and bibliography, *include* at the end of the document `filename.tex` the BibTeX generated document (using `\include{filename.bbl}`). Since this is generated under the desired format in the original LaTeX document, the format of the bibliography will match that format (except for the numerical designations). The difficult aspect is replacing numerical citations with author-year. One approach, if one is inclined, is pattern match the citations themselves in `filename.tex`, replacing all of the citation commands with the actual citations (leaving the only conversion for the list of references). One could also conduct the pattern matching within Word, which I imagine (even with libraries to do so within Perl or Python) is not worth the trouble. My personal choice is to simply "find and replace" on single citations, copying the designations from the open `filename.bbl`, and editing directly the few citations that contain multiple parenthetical entries. This takes about an hour. I compile a list of the more complex entries during a final proofreading of the entire document (a necessary step anyway given the minor conversion problems that will appear).

General Formatting Finally, a common aspect of the final edits is formatting: double spacing, endnotes instead of footnotes, tables and figures at the end of the document. Because `setspace` and `endfloat` do not work in the conversion, these need to be set within Word. Double spacing, margins, and justification are easy to set within Word. The one issue that commonly occurs is the conversion of footnotes to endnotes, which will need to be conducted within Word. The actual conversion to endnotes is straightforward (select to add a footnote, then instead convert all to endnotes), but the placement of those endnotes can be problematic. By default, endnotes will be placed at the end of the document, but they usually must occur after the text but before the references. The solution to this problem is to (1) insert a section break between the text and the references, (2) suppress endnotes for the entire document, then (3) unsuppress the endnotes, but only for the section before the references. Then (4) insert a page break between the text and the references to ensure the endnotes are contained on a separate page.

Overall, my assessment of TeX2Word matches a column in the TUG journal PracTeX: "a good bit of cleanup was required after my TeX2Word conversion; but... I can't imagine a program that could correctly guess what the perfect conversion would be. All in all, using TeX2Word saves a good bit of time over much more manual methods" (Walden, 2005). I have described the conversion of text, math environments, tables and figures, references, and document formatting. In all of these aspects, the author will need to make revisions within the Word document. And as annoying and time-consuming these revisions may be, they are straightforward and relatively minor compared to alternatives requiring more extensive reconstruction of the document.

References

Walden, David. 2005. "Travels in TeX Land: Word2TeX Redux, TeX2Word, Plain TeX with Eplian, and Playing with 'Thought Breaks'." *The Prac TeX Journal* 4.

Political Analysis

Political Analysis Update

R. Michael Alvarez and Jonathan N. Katz, co-editors

In January 2010, we began our term as co-editors of *Political Analysis*. Our immediate task was to try to provide a seamless transition from the previous editorial team to ours; we wish to thank Wendy Tam Cho, Robert Franzese, Andrew Martin and Christopher Zorn for their assistance with this transition. During this same period of time, we have also implemented a number of changes: First, have transitioned all of the manuscript submission, review, and revision processes to ScholarOne (http://mc.manuscriptcentral.com:80/pa). All submissions and revisions are now processed using this new system, and we believe it has greatly improved our ability to quickly and efficiently process manuscripts. Second, we have begun to implement a new replication policy, the details of which are available with our Information for Authors (http://www.oxfordjournals.org/ our journals/polana/forauthors/index.html) page. The *Political Analysis* Dataverse page, which is the primary location for the replication materials associated with the journal, is located at http://dvn.iq.harvard.edu/dvn/dv/pan. Third, we have established the Editors' Choice Article award, representing papers that the editors see as providing an especially significant contribution to political methodology. Editors' Choice articles are available for free online access at http://www.oxfordjournals.org/our journals/polana/editor schoice.html. The first Editors' Choice Article is "Estimation of Heterogeneous Treatment Effects from Randomized Experiments, with Application to the Optimal Planning of the Get-Out-The-Vote Campaign," by Kosuke Imai and Aaron Strauss. Fourth, in addition to the journal's website (http://pan.oxfordjournals.org), *Political Analysis* is now on Facebook (http://www.facebook.com/pages/Political-Analysis/104544669596569) and Twitter (http://twitter.com/polanalysis). You can use these social media tools to stay updated on topics relating to the journal and political methodology. We have other innovations in the works, so keep an eye on the journal website, our Facebook page, or the Twitter feed.

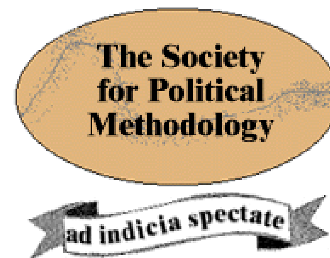
University of Illinois at Urbana-Champaign
Department of Political Science
240 Computing Applications Building
605 E Springfield Ave
Champaign, IL 61820

The Political Methodologist is the newsletter of the Political Methodology Section of the American Political Science Association. Copyright 2011, American Political Science Association. All rights reserved. The support of the Department of Political Science at the University of Illinois in helping to defray the editorial and production costs of the newsletter is gratefully acknowledged.

Subscriptions to *TPM* are free for members of the APSA's Methodology Section. Please contact APSA (202-483-2512) if you are interested in joining the section. Dues are $25.00 per year and include a free subscription to *Political Analysis*, the quarterly journal of the section.

Submissions to *TPM* are always welcome. Articles may be sent to any of the editors, by e-mail if possible. Alternatively, submissions can be made on diskette as plain ascii files sent to Wendy K. Tam Cho, 240 Computing Applications Building, 605 E. Springfield Ave., Champaign, IL 61820 . LaTeX format files are especially encouraged.

TPM was produced using LaTeX.



President: Jeff Gill
Washington University in St. Louis
jgill@wustl.edu

Vice President: Robert Franzese
University of Michigan
franzese@umich.edu

Treasurer: Suzanna Linn
Pennsylvania State University
slinn@la.psu.edu

Member-at-Large: Brad Jones
University of California, Davis
bsjjones@ucdavis.edu

Political Analysis Editors:
Michael Alvarez and Jonathan Katz
California Institute of Technology
rma@hss.caltech.edu and *jkatz@caltech.edu*