Introduction to Stata Programming

Gabriel Rossman
rossman@soc.ucla.edu

October 15, 2010

Serious work in Stata is done entirely in do-files, but you may notice that your do-files get very repetitive. You may find yourself applying a series of very similar commands over and over again. For instance, you may have ugly, repetitive code like this:

```
recode var1 1=1 2=0 99=.
recode var2 1=1 2=0 99=.
recode var3 1=1 2=0 99=.
recode var4 1=1 2=0 99=.
recode var5 1=1 2=0 99=.
```

This is tedious but the real problem with it is that if you need to change it (for instance to make the missing data code be "-1" instead of "99"), you'll have to change it in each instance.

The goal of programming is to write a simple construct then see it repeated multiple times and any changes will propagate to each of the instances. Likewise you may want to have one command feed into another, which is impossible to do in a do-file without programming. Learning to program will help you write do-files that are concise, comprehensible, replicable, and can be easily modified. This is written with Stata in mind and using Stata terminology but the principles are similar in other scripting languages including both statistical languages like R and SAS and general purpose languages like Bash, Perl, and Python.[1]

Note that this lecture is loosely based on an earlier CCPR programming tutorial by Courtney Engel. I also suggest reading Long, J Scott. 2009. *The Workflow of Data Analysis Using Stata.* College Station, TX: Stata Press.

Directory Structure

The first step to a project is having a good directory structure. I like to keep mine in the Documents directory of my hard drive ("~/Documents" in Mac/Unix

[1]Although these general principles are relevant to other languages, much of the terminology is not. For instance, what Stata calls a "macro," most languages call a "variable;" what Stata calls a "variable," most languages call a "vector" (in memory) or "field" (on disk); and what Stata calls "commands," "programs," and "functions" are all just called "functions" in many object-oriented languages like Python and R.

or "My Documents" in Windows). Another way to do it is to keep it on a USB drive or server directory.

Within the directory are three main subdirectories: lit, stata, and writeup. Lit is just a place to put my pdfs of things I might like to cite. Stata is where I put my do-files, log-files, and subdirectories for data and output. Writeup is where I put my Word or LaTeX/Lyx files containing the write-up. Note that both "stata" and "writeup" contain "archive" subdirectories, which is where I hide old versions of my do-files and manuscripts (see "versioning" below). Also note that symbolic links (aka, "aliases" or "shortcuts") can be useful, for instance if two projects have overlapping lit reviews and/or datasets.[2] Likewise, you can use symbolic links so that your "rawdata" directory is really a shared directory on a server.

```
~/Documents/project
    /lit
    /stata
        /archive
        /cleandata
        /graphics
        /rawdata
        /tables
    /writeup
        /archive
```

The "stata" directory has several subdirectories. First, note that there are two data directories, one for raw and one for clean data. Your raw data should always be treated as *read-only*. You should modify the raw data only using do-files (rather than interactively) and you should *not* overwrite the raw file but instead put the modified data in the "cleandata" directory. The "graphics" directory is for saving graphics using the command "graph export".[3] The "tables" directory is for saving tables with commands like estout (which formats tables to meet journal style guidelines and saves them as plain text, RTF, or TEX). Note that if you're using LaTeX/Lyx you can have your write-up automatically include the most current version of the tables and graphics by targeting their locations.

[2]In Mac or Linux, try to use true symbolic links rather than aliases since they work at a lower-level. This makes it easier for Stata to use them and also makes it easier to migrate from a Mac desktop to a Linux server. To create a symbolic link in the Terminal (or Stata's "shell" command), type "ln -s target link". For instance, to create a link called rawdata.sym that targets ~/Documents/project/rawdata, you'd type "ln -s ~/Documents/project/rawdata rawdata.sym". You can also create symbolic links with the graphic interface using PathFinder or (in OS X 10.6) in the Finder by creating an Automator service.

[3]Note that only Stata can read the ".gph" format so you want "graph export" not "graph save". The format ".png" is good for display on the screen and casual use, but you want ".eps" for printing and typesetting. To get pdf on Mac/Unix get my ado file with "ssc install graphexportpdf". To do it on Windows first export as eps then use ghostscript or distiller.

That's my way, but an alternative approach suggested in Scott Long's book is oriented around the concept of "posting" (similar to what a version control system calls "commits") where one distinguishes between drafts that are just for you and you're still working on and scripts that are ready to be permanently fixed and circulated with your collaborators. In Long's scheme if you want to change a posted script you write a new version with its own version number. Long's approach is especially appropriate to large-scale collaboration with interdependent components.

Also note that the Mac's Finder is designed for a low learning curve rather than a large feature set. I prefer to use PathFinder for file management.

Versioning

It's a good idea to have one current version of your scripts and your write-up but also to keep backups of old versions. This lets you replicate results based on older versions, revert any mistakes you make in new versions, and freely experiment knowing that you can always retrieve old code from the archival versions. I usually don't bother keeping old versions of my clean data, tables, and graphics because I can always replicate them from scratch with the appropriate do-file backup.

My quick and dirty way of handling this is to call my current version of a script something simple like ~/project/stata/recode.do. I put my old version in the folder archive and add the date to the name, e.g., ~/project/stata/archive/recode20101001.do. You can copy and rename the archival versions manually or you can use a Unix shell script that will do this for you. In Snow Leopard you can use Automator to save this script as a right-clickable Finder service.[4]

```
#bin/bash
TIMESTAMP=`date '+%Y%m%d'`
for f in "$@"
do
 EXTENSION=${f##*.}
 FILENAME=`basename $f | sed 's/\(.*\)\..*/\1/'`
 DIRPATH=`dirname $f`
 cp "$f" "$DIRPATH/archive/$FILENAME$TIMESTAMP.$EXTENSION"
done
```

This works pretty well for a solo project or a collaboration with a clear division of labor, but for a complex collaboration where you and your colleagues are working on the same files you might want to try version control software like Git, Subversion, or Mercurial. These systems provide similar functionality to the "track changes" feature in Word. These programs will work with any text-based format, including Stata do-files, and are designed to be integrated with cloud file-sharing/backup on either a client/server or distributed model.

[4]Like any shell script, you can make it a terminal command in Mac or Linux by saving it as a file, using the command "chmod +x filename" to make the file executable, and adding an alias targetting the file to ~/.bashrc.

Note that the Unix command "diff" is invaluable for comparing different versions of the same file. Many text editors and version control clients have analogous functions. I personally use the Diff bundle in TextMate.

Text Editors

Any kind of programming should be done in a text editor and never ever done in a word processor like Microsoft Word. The problem with word processors is they automatically do things that can crash your code like replace straight quotes with curly quotes. The main advantage of a text editor is syntax highlighting, where the editor automatically color-codes your code to make various aspects of the syntax more salient. This both makes it easier to read the code and helps you notice bugs, such as hanging quotes. A more advanced feature not found in older text editors is code-folding, where syntactic blocks of code (such as loops) are automatically flagged and can be hidden when you're not working on them. Editors can also be useful for cleaning text files, especially if you get good at regular expressions.

Beginning with Stata 11 (Windows) and 11.1 (Mac), the integrated do-file editor is actually pretty good, but I still prefer to use an external editor. My favorite editor for the Mac is TextMate and for Windows I like Notepad++. Editra and Open Komodo are also very good and will run on anything. Editra or the Stata do-file editor might be especially good options for beginners as they come with Stata support out of the box whereas with the others you have to add syntax parsing as a plug-in.

Header

A do-file (or any script) should begin with a header consisting of a series of comments saying who wrote the script, what it's for, what other script should have been run before it, etc. Note that Stata comments begin with "*" but many languages use "#". In addition to the header comments, you should sprinkle comments throughout the do-file to clarify the purpose of the code.

It's also a good idea to have your housekeeping commands (i.e., "set" and "log") right up-front.

I also like to have two sets of global macros in my header, first the directories and second a set of switches that describe different ideas (e.g., log all the independent variables).[5] The reason for doing both of these things is that it lets me easily change things without searching through the code to find all the places that they occur. For instance, by putting the directories in global macros in the header, I can easily migrate the project to a different computer. Likewise, I can use the switches to only run parts of the code that I've recently modified. This can save a lot of time but can also be problematic if the skipped code affects how the executed code would have run.

[5] You could also do this with local macros but I prefer global macros because they persist after the do-file completes or hits an error, which is convenient for debugging, interactive work, etc.

Here's the beginning of the do-file that makes all the graphs in my book project

```
global bookdir "~/Documents/book"
global images "~/Documents/book/images"
global payola "~/Documents/Sjt/payola/"
global dixiedata
"~/Documents/mediacongl/dixiechicks/daily_mediabase"
global sh_parentpath  "~/Documents/Sjt/songhistoryfiles_mjk082708"
global survey "~/Documents/Sjt/radio/survey"
global humpsdata "$bookdir/stata/myhumpsweekly"

capture log close
log using $bookdir/stata/graph.log, replace

*switches -- chapters
global ch2 0
global ch3 0
global ch4 0
global ch5 1

*switches
*shared frailty analysis
global earlywin     60     /*how far ahead of p05event may adds come*/
global iterations   25     /*how many iterations can each MLE run?*/
global frailtysims  100    /*how many random shuffles of each song*/
```

Macros

In Stata, macros are small data objects (often just a single word) held in memory.[6] The concept is similar to a "variable" in many other languages. Stata assigns some macros automatically and others can be assigned by the programmer. However the programmer can still read the ones Stata assigns automatically.

There are two basic types of macros, "locals" and "globals." A *local* is shown by left and right apostrophes (on the keyboard they are below the ~ and the " respectively). A *global* is shown by $, just like a Unix variable. The difference is that globals are always accessible whereas locals are only accessible at or below the level in the programming hierarchy (see "loops" and "programs") where the macro is first mentioned (or "declared"). It's a good idea to use locals within loop constructs and to use globals for issues that apply to the entire script, such as the directory map. Using macros in Stata can be simpler than using variables in other languages because Stata doesn't require extra steps for declaration and concatenation.

[6]Don't think of this as being at all analogous to what Microsoft Office calls a "macro." Microsoft macros are more analogous to what Stata calls a "program" and most languages call "subroutines" or user-defined "functions."

For instance, in the code excerpt above I called the main project directory "$bookdir", a global. If I wanted it to be a local I would have assigned and called it like so:

```
local bookdir "~/Documents/book"
cd ‘bookdir’
```

I also like to use macros to define lists of variables that I tend to use together. This is extremely useful when creating nested regression models (e.g., model 1 is controls, model two is controls plus human capital, model three is controls plus human capital and networks) because changes propogate. For instance if you want to change one of the control variables from a quadratic to a spline you only need to do it in one place rather than once for each model (which is both time-consuming and allows the possibility of inconsistency).

```
local controls "FPY00 g_drama g_comedy g_biography major castsize date female"
local hc "pacting pactsq pAnom"
local team "OTH_pAnom0 pDnom0 pWnom0"
eststo clear
eststo: logit actor_nom ‘controls’
eststo: logit actor_nom ‘controls’ ‘hc’
eststo: logit actor_nom ‘controls’ ‘hc’ centrality
eststo: logit actor_nom ‘controls’ ‘hc’ centrality ‘team’
esttab , se b(3) se(3) scalars(ll rho) nodepvars nomtitles label
eststo clear
```

Another application for macros is switches. I like to rely on the "if" syntax to only execute a piece of code if a particular global is specified. In the example above this corresponds to graphs by chapter. Here's a simplified example:

```
if $ch2==1 {
    use $bookdir/stata/cleandata/music.dta, clear
    histogram radiostations
    cd $images
    graphexportpdf radiostations
}
```

This code will only run if the header defines $ch2 as 1. This is useful if I want to debug a certain section of code without waiting for the well-behaved code to execute.

Furthermore, as described below the "program," "foreach," "forvalues," and "while" syntaxes rely heavily on local macros.

Macros are a very important way to talk to Stata and to program Stata so the output of one command feeds the arguments of another command. The three commands that show you what macros Stata has generated automatically are:

```
creturn list
return list
ereturn list
```

The first of these, creturn, shows you system settings. Most of the creturn values are preferences that can be changed with the “set” command. For instance, this code checks the memory allocation and if it is less that 100 mb, sets it to 100 mb.

```
if ‘c(memory)’<1.049e+08 {

    set mem 100m

}
```

Others of the creturn values can’t be changed but you can still use them to feed into the program. Here’s an example where a program will create a pdf on a Mac but eps on anything else:[7]

```
if "‘c(os)’"=="MacOSX" {

    graph export mygraph.pdf, replace

}
else {

    graph export mygraph.eps, replace

}
```

Return and ereturn macros are produced by commands and only last until you issue another similar command (which will overwrite them). One of my favorite applications of this is to use “summarize” then use some of the return macros to feed into the next command. For instance, this code uses “summarize” to learn the range of a variable then uses return macros to adjust the graph that follows so it has a nice number of tick marks and labeled points.

```
sum date
local mindate=‘r(min)’
local maxdate=‘r(max)’
local interval=(‘maxdate’-‘mindate’)/10
local interval=round(‘interval’,7)
twoway (line x date) , /*
 */xmtick(‘mindate’(7)‘maxdate’) xlabel(‘mindate’(‘interval’)‘maxdate’)
```

[7]Graph exporting is one of very view things in Stata where the operating system matters. You should also use the if "‘c(os)’" construct if you are making use of the “shell” command and expect it to run on different systems.

A more complicated type of macro is the matrix, which is a little table.[8] Cells in the matrix are identified as "matrixname[row,column]". You can use matrices to record things too complicated to fit in a local, but one of the most obvious uses is return matrices. Most Stata commands that give output as some kind of table will allow you to return the table. The option "matcell(name)" lets you save the results of a tabulate command and you can use the saved matrix to do things like calculate odds-ratios.

```
tab candidat inc [fweight= pop], matcell(elec)
disp "A wealthy person was about " /*
 */ round((elec[2,5]*elec[1,1])/(elec[1,5]*elec[2,1])) /*
 */ " times more likely to choose Bush over Clinton than a very poor person"
```

Likewise regression commands return an ephemeral matrix called "e(b)". You can copy e(b), and having copied it, manipulate it.

```
sysuse auto, clear
reg mpg foreign
matrix betas = e(b)
local foreignadvantage = round(betas[1,1])
disp "in 1978, foreign cars got about ‘foreignadvantage’ more miles to the gallon than
```

Functions

Stata has a variety of functions that will process an argument enclosed in parentheses. For instance the function "log()" returns the natural logarithm of whatever is in the parentheses.

```
gen income_ln=log(income)
```

Although functions are often used for transforming variables they are much more versatile and can also be used for things like processing macros, complex expressions, and even other functions. That is, you can have nested functions like "log(real(income))" which will take a variable called income (but coded as a string), turn it into a numeric, and then take the log.

The simplest Stata functions are random number functions, most of which start with the letter "r." These can be very useful for sampling, simulations, permutation analysis, etc.

Many of the functions are most useful for hard-core programmers, but the math functions, string functions, and date functions are very useful even for fairly simple do-files. The aforementioned "log()" is one of the most useful math functions and Stata has functions for most of the other things you learned in elementary and high school math, especially anything having to do with rounding, trig, or logarithms/exponentiation. If you understand the math, the code is straightforward.

[8]Note that Stata is somewhat unusual in distinguishing between "matrices" and "the dataset."

String functions are for cleaning text. They are a little harder to use than the math functions but they are invaluable for cleaning dirty data like IMDB. Although sometimes it's best to give up and use Perl or Python for cleaning text, many things can be done very well using Stata's extensive library of string functions. There are a lot of specialized but fairly straightforward functions like "trim()" and "subinstr()" but Stata also has the "regexm()" and "regexs()" functions for full-blown regular expressions, which are very flexible but have a learning curve. By using regular expressions you can do things like taking the "city, state" line of an address and splitting it into one "city" component and another "state" component.

The date functions have two purposes. First, they can take dates coded as strings (e.g., "November 5, 1985") and convert them to a number of time increments since January 1, 1960. Stata can count time out in milliseconds (%tc), days (%td), weeks (%tw), months (%tw), quarters (%tq), half-years (%th), or any arbitrary increment (%tg).[9] Of these, %td is the most popular. Second, the date functions can convert one date format to another or extract a component (e.g., day of the week) from a date. For instance, my radio data comes with a variable called "firstplayed" that is a string formatted as "MM/DD/YYYY". To get this into Stata and stored as a date ("fpdate") and a date rounded to the nearest Sunday ("fp_w"), I use these commands:[10]

```
gen fp_w=date(firstplayed,"MDY")-dow(date(firstplayed,"MDY"))
format fp_w %td
```

Loops

Loops execute some commands several times based on a set of values in a macro. The two basic commands are foreach, which runs the loop over a series of words (separated by spaces) and forvalues, which runs over a number series. The "while" loop is a more general loop command that runs until a specified condition is met. You can think of forvalues as being special cases of while, and indeed some programming languages require programmers to jerry-rig a forvalues algorithm using while. These loops are useful for all sorts of repetitive tasks. Because you only write the command once then loop it you both save time and avoid inconsistency.

For instance, the Stata standard for dummies is that 0 means no and 1 means yes, but the Survey of Public Participation in the Arts codes "no" as "2." Here's a loop that corrects several of the dummies (and renames them to avoid confusion with the original versions):

```
lab def yesno 0 "N" 1 "Y"
foreach var in PEX4A PEX4B PEX5 PEQ1A PEQ2A PEQ3A PEQ4A {
```

[9]Unix time is measured in seconds and starts 1/1/1970 so Unix Time is approximately (%tc*1000)+(86400*3652).

[10]I keep fp_w as %td instead of %tw because 365 days doesn't divide evenly by 7 days a week and I don't like how %tw handles the odd days.

```
        recode ‘var’ 2=0 1=1 .=.
        lab val ‘var’ yesno
        ren ‘var’ ‘var’r
    }
```

First note the syntax of the “foreach” command itself (the second line). The syntax goes “foreach local in list”. So “var” is a local that draws values from “list,” one at a time. Next note that foreach ends with an open curly bracket and is followed by several indented commands. This indentation is called *whitespace*. Like most languages, Stata doesn’t need whitespace but it’s considered good programming practice as (much like syntax highlighting) it helps *you* understand the script and immediately see the logical structure. Finally the loop ends with a closed curly bracket. When executed the loop will run once treating the local “var” as meaning “PEX4A” then again treating it as “PEX4B”, etc.

We can also use a local as the list. For instance imagine that we wanted to import all the comma-separated-values files in “stata/rawdata” and save them as Stata files in “stata/cleandata.” By using Nick Cox’s ado-file “fs,” we can get a return macro listing all the csv files and then use that local to run the loop. Note that you only need to install “fs” once, after that it’s just another command.

```
    ssc install fs, replace /*this line on first use only*/
    cd ~/Documents/project/stata/rawdata
    fs *.csv
    foreach file in ‘r(files)’ {
        insheet using ‘file’, clear
        save ../cleandata/‘file’.dta, replace
    }
```

Forvalues has a similar syntax except that list is replaced by a series defined as local=min/max or local=min(interval)max. In the former, the interval is assumed to be one. The local in this loop is often called “i” and programmers nickname it a “tick,” as in a ticking clock. Note that you don’t need to call on the tick within the loop, for instance if you just want to repeat something a few times. Non-trivial applications of loops that do not call on the tick might involve things like calculating standard error through resampling or permutation.

```
    forvalues count=1/10 {
        disp ‘count’
    }
    forvalues countbytwos=2(2)10 {
        disp ‘countbytwos’
    }
    forvalues i=1/10 {
          disp “exact same thing each time”
     }
```

While takes a conditional statement and repeats so long as the statement holds. For instance, here’s a while loop that replicate the first forvalues example. (This is only an illustration in Stata, but is a very common algorithm in languages like Perl that lack a forvalues loop).

```
     local count=1
     while ‘count’ <= 10 {
          disp ‘count’
          local count=‘count’+1
     }
```

Also note that you can nest loops within each other and even build a manual tick counter for added flexibility. For instance, here’s a little script that creates every combination of several numeric and string values.

```
     clear
     set obs 12
     gen piecesof=.
     gen fruit=’”’
     local i=1
     foreach fruit in apple pear tomato pepper {
          forvalues num=1/3 {
               replace piecesof=‘num’ in ‘i’
               replace fruit=”‘fruit’” in ‘i’
               local i=‘i’+1
          }
     }
```

Program and Arguments

The “program” syntax in Stata lets you write your own commands. You can put them at the beginning of a do-file for later use or you can save them as “ado” files to always have access to them. Of course, you can also share ado files. Even though the core of Stata is proprietary, the ease of writing and sharing ado files gives it a quasi-open source feel and many of the most popular commands (e.g., estout and gllamm) are ado files. These user-written ado-files don’t come on the Stata cd, but you can look for and install useful ones with the commands “ssc” and “findit.”

When you write a program, it takes as an argument whatever follows the name of the program. By default, this argument is just a series of local macros called ‘1’, ‘2’, ‘3’, etc., but you can use the “syntax” and “token” commands to help you parse the argument in complex ways. For simple commands (especially ones that are just for your own use) it is often good enough to just use the default behavior. For instance, here's a trivial program that takes arguments by order, assigns them to mnemonic locals, and applies them in a "disp" command.

```
capture program drop switcheroo
program define switcheroo

    local omega `1'
    local alpha `2'
    disp "`alpha' `omega'"

end
switcheroo first last
```

As a non-trivial example, I wrote a program for use in writing my book called "songgraph_all" that shows the cumulative distribution function of radio stations playing the pop song given in the argument. I also have a more complicated command called "graphlines_group" that will break this out out by format or owner within format. Here's an example of the syntax for the two commands which relies simply on word order:

```
songgraph_all "RIHANNA" "Umbrella fJayZ"
songgraph_all "RIHANNA" "Umbrella fJayZ" f "format" "Top_40"
graphlines_group  "RIHANNA" "Umbrella fJayZ" f format
graphlines_group "RIHANNA" "Umbrella fJayZ" fo owner f format Top_40 "% of Stations Pla
```

Here's the beginning of the code for the "songgraph_all" command. Note that one program can invoke another program and also that programs should use whitespace to show structure:

```
capture program drop songgraph_all
program define songgraph_all

    local artist      `1'
    local song        `2'
    local suffix      `3' /*file suffix to use */
    local subsample2  `4' /*variable to restrict sample to, eg format*/
    local subsample2x `5' /*subsample2 value (eg, Top 40)*/
```

Piping

One of my favorite commands is "shell" which gives you direct access to the command-line of your operating system.[11] This lets you access many other

[11]Note that an important limitation of "shell" is that unlike a true terminal or shell script, the Stata "shell" command has no concept of a session and doesn't load preference files like ~/.bashrc. If you need a session, have Stata launch an executable shell script rather than directly sending the shell one command at a time. If you want Stata to write this script use the "file write" command.

programs from directly within Stata, which is useful since versatile as Stata is, some things are difficult or impossible to do in Stata. This has some applications in Windows, but really becomes powerful in Unix (i.e., Mac, Linux, etc.) as your average Unix installation comes standard with or can easily install many powerful tools that can be run directly from the command-line.[12] By using the "shell" command you can let Stata do what it does best, then "pipe" problems to tools that are better suited to handle them. Alternately, instead of starting with a Stata do-file and reaching other programs with "shell" you can have a "bash" or "make" script that invokes many programs, including Stata.

The big advantage to piping is that it lets you script things that are difficult or impossible in Stata. As always, the advantage of scripting is that it makes it easier to repeat the operation (usually with slight variations). The two big downsides are a) your do-files won't be portable to computers that use a different operating system or have a different panopoly of tools and b) the different languages each have slightly different syntax so there's a learning curve.

The most basic thing to do is to just use the basic command line tools. For instance, suppose I have a bunch of files organized by song, and I'm interested in finding all the song files that mention a particlar radio station, say KIIS-FM. I can run the following command that finds all the song files in my song directory (or its subdirectories) and puts the names of these files in a text file called "kiis.txt"

```
shell grep -l -r 'KIIS' ~/Documents/book/stata/rawsongs/ > kiis.txt
```

The command "grep" will work with any Unix computer, but it can be slow when applied to large directories (or the entire file system). If this is a problem, you can make it much faster by accessing the index of your desktop search.[13] (It's analogous to trying to search for a passage in a book by reading the index rather than reading the whole text). On a Mac, "mdfind" is the command-line interface for the Spotlight index and you can invoke it from within Stata like this.

```
shell mdfind -onlyin ~/Documents/book/stata/rawsongs/ "kiis" > kiis.txt
```

Whether I use grep or mdfind (or for that matter, Windows Search or the Unix command "locate"), I now have a query saved as a text file. By extension, I could then write a Stata program around this shell command that will let me

[12]Windows also has a shell but traditionally it's not as powerful as Unix. Beginning with Vista (optional) and Windows 7 (standard), Windows now includes the "PowerShell" tool which is similar to Unix but not a true POSIX standard bash shell like Mac/Linux. The upshot is that Stata "shell" commands and entire bash scripts written for Mac/Linux will probably require some debugging (but not a 100% rewrite) before they run on Windows PowerShell.

[13]Note that desktop search only lists the files containing a keyword and not the location in the file. If you want to both flag files and extract the relevant lines, it's possible to combine the speed of desktop search with the flexibility of grep, as described here.

query station data from my song files (or vice versa). You could do something similar to see what saved news or blog stories contain a certain keyword.

More broadly, versatile as Stata is at data management, statistical analysis, and graphing, it can't do everything. For instance Stata is very picky about how text files are formatted and it's almost impossible to get Stata to do things like read field-tagged data (e.g., BibTex, IMDB). Likewise, you might be interested in treating header information from a bunch of scraped websites as a dataset. Rather than spend days doing it manually or drive yourself crazy getting Stata to do it, you're probably best off using perl to do the bulk of the work then insheeting the clean output into Stata.

Perl and Stata are both inspired by C, so once you get comfortable with Stata (especially the programming constructs) you should find learning perl reasonably easy. I highly recommend the online course notes and exercises "Unix and Perl Primer for Biologists." They assume no prior familiarity with programming or Unix and although the examples involve genetics, it's well-suited for social scientists as, like us, biologists are not computer scientists but are reasonably technically competent and they often deal with large text based data sets. The course should be useful to any social scientist who deals with large dirty datasets, in other words, basically anyone who is a quant but doesn't just download clean ICPSR or Census data. This is especially relevant for anyone who wants to scrape data off the web, use IMDB, do large-scale content analysis, etc.

The most glaring ommission in Stata is that as of yet it has no social network analysis features. However you can integrate social networks by piping data to one of the several scriptable programs that can do this (Pajek, igraph or statnet in R, NWB, Mathematica), run the analysis in that package, and then import the results back to Stata.[14] Likewise, through piping you can have Stata ask for R's helpin reading SPSS (or many other file formats) into Stata.

Although Stata graphing has gotten very good beginning with Stata 10, it still has limitations and so for some purposes (e.g., network graphs, heat density plots) you might like to have another program (R, gnuplot, Pajek) do the graphing for Stata. By using the "shell" command you can integrate these graphs directly into your do-file. In the short run it would be faster to use the graphic interface to do this (Excel can do almost any non-network graph) but if you have to do it several times with slight changes each time, it's better to script it. Likewise, Stata has good eps graphs for all platforms but only mediocre pdf graphs for mac and no pdf graphs at all for other platforms so I wrote a command "graphexportpdf" that uses "shell" to pipe to the Ghostscript command "ps2pdf" to translate the eps to a high-quality pdf.

[14]I have a Stata ado command for exporting Stata data to Pajek format (which most packages can read) and a perl script for importing Pajek/NWB/Mathematica output back into Stata. For the former type "ssc install stata2pajek" and for the latter get http://gabrielr.bol.ucla.edu/pajek_labelvector.pl

Have a Nice Day

I like to end all of my scripts with the comment

```
*have a nice day
```

The reason is that Stata will only execute a command that has an end-of-line character. This means that the very last line of the do-file is not executed. One way to solve this is just to put a blank line at the bottom but it's hard to tell at a glance whether you remembered to do this. Hence, I put a silly phrase at the end to ensure that the last *real* command gets executed.