Programming in Stata and Mata

Christopher F Baum

Boston College and DIW Berlin

University of Adelaide, June 2010

Introduction

In this talk, I will discuss some ways in which you can use Stata more effectively in your work, and present some examples of recent enhancements to Stata that facilitate that goal.

I first discuss the several contexts of what it means to be a Stata programmer. Then, given your role as a *user* of Stata rather than a developer, we consider your motivation for achieving proficiency in each of those contexts, and give examples of how such proficiency may be valuable.

I hope to convince you that "a little bit of Stata programming goes a long way" toward making your use of Stata more efficient and enjoyable.

First, some nomenclature related to programming:

- You should consider yourself a Stata programmer if you write *do-files*: sequences of Stata commands which you execute with the `do` command or by double-clicking on the file.
- You might also write what Stata formally defines as a *program*: a set of Stata commands that includes the `program` statement. A Stata program, stored in an *ado-file*, defines a new Stata command.
- As of version 9, you may use Stata's new programming language, *Mata*, to write routines in that language that are called by ado-files.

Any of these tasks involve *Stata programming*.

With that set of definitions in mind, we must deal with the *why:* why should you become a Stata programmer? After answering that essential question, we take up some of the aspects of *how:* how you can become a more efficient user of Stata by making use of programming techniques, be they simple or complex.

Using any computer program or language is all about *efficiency*: not computational efficiency as much as *human* efficiency. You want the computer to do the work that can be routinely automated, allowing you to make more efficient use of your time and reducing human errors. Computers are excellent at performing repetitive tasks; humans are not.

One of the strongest rationales for learning how to use programming techniques in Stata is the potential to shift more of the repetitive burden of data management, statistical analysis and the production of graphics to the computer.

Let's consider several specific advantages of using Stata programming techniques in the three contexts enumerated above.

Context 1: do-file programming

Using a *do-file* to automate a specific data management or statistical task leads to *reproducible research* and the ability to document the empirical research process. This reduces the effort needed to perform a similar task at a later point, or to document the specific steps you followed for your co-workers or supervisor.

Ideally, your entire research project should be defined by a set of do-files which execute every step from input of the raw data to production of the final tables and graphs. As a do-file can call another do-file (and so on), a hierarchy of do-files can be used to handle a quite complex project.

The beauty of this approach is *flexibility*: if you find an error in an earlier stage of the project, you need only modify the code and rerun that do-file and those following to bring the project up to date. For instance, an academic researcher may need to respond to a review of her paper—submitted months ago to an academic journal—by revising the specification of variables in a set of estimated models and estimating new statistical results. If all of the steps producing the final results are documented by a set of do-files, that task becomes straightforward.

I argue that *all* serious users of Stata should gain some facility with do-files and the Stata commands that support repetitive use of commands. A few hours' investment should save days of weeks of time over the course of a sizable research project.

That advice does not imply that Stata's interactive capabilities should be shunned. Stata is a powerful and effective tool for exploratory data analysis and *ad hoc* queries about your data. But data management tasks and the statistical analyses leading to tabulated results should not be performed with "point-and-click" tools which leave you without an audit trail of the steps you have taken.

Responsible research involves *reproducibility*, and "point-and-click" tools do not promote reproducibility. For that reason, I counsel researchers to move their data into Stata (from a spreadsheet environment, for example) as early as possible in the process, and perform all transformations, data cleaning, etc. with Stata's do-file language. This can save a great deal of time when mistakes are detected in the raw data, or when they are revised.

Context 2: ado-file programming

You may find that despite the breadth of Stata's official and user-writen commands, there are tasks that you must repeatedly perform that involve variations on the same do-file. You would like Stata to have a *command* to perform those tasks. At that point, you should consider Stata's *ado-file* programming capabilities.

Stata has great flexibility: a Stata command need be no more than a few lines of Stata code, and once defined that command becomes a “first-class citizen." You can easily write a Stata program, stored in an ado-file, that handles all the features of official Stata commands such as `if` *exp*, `in` *range* and command *options*. You can (and should) write a help file that documents its operation for your benefit and for those with whom you share the code.

Although ado-file programming requires that you learn how to use some additional commands used in that context, it may help you become more efficient in performing the data management, statistical or graphical tasks that you face.

My first response to would-be ado-file programmers: *don't!* In many cases, standard Stata commands will perform the tasks you need. A better understanding of the capabilities of those commands will often lead to a researcher realizing that a combination of Stata commands will do the job nicely, without the need for custom programming.

Those familiar with other statistical packages or computer languages often see the need to write a program to perform a task that can be handled with some of Stata's unique constructs: the *local macro* and the functions available for handling macros and lists. If you become familiar with those tools, as well as the full potential of commands such as `merge`, you may recognize that your needs can be readily met.

The second bit of advice along those lines: use Stata's search features such as `findit` and the Stata user community (via Statalist) to ensure that the program you envision writing has not already been written. In many cases an official Stata command will do almost what you want, and you can modify (*and rename*) a copy of that command to add the features you need.

In other cases, a user-written program from the *Stata Journal* or the SSC Archive (`help ssc`) may be close to what you need. You can either contact its author or modify (*and rename*) a copy of that command to meet your needs.

In either case, the bottom line is the same advice:
don't waste your time reinventing the wheel!

If your particular needs are not met by existing Stata commands nor by user-written software, and they involve a general task, you should consider writing your own ado-file. In contrast to many statistical programming languages and software environments, Stata makes it very easy to write new commands which implement all of Stata's features and error-checking tools. Some investment in the ado-file language is needed (perhaps by taking a *NetCourse* on Stata programming) but a good understanding of the features of that language—such as the `program` and `syntax` statements—is not hard to develop.

A huge benefit accrues to the ado-file author: few data management, statistical or graphical tasks are unique. Once you develop an ado-file to perform a particular task, you will probably run across another task that is quite similar. A clone of the ado-file, customized for the new task, will often suffice.

In this context, ado-file programming allows you to assemble a workbench of tools where most of the associated cost is learning how to build the first few tools.

Another rationale for many researchers to develop limited fluency in Stata's ado-file language:

- Stata's maximum likelihood (`ml`) capabilities involves the construction of ado-file programs defining the likelihood function.
- The `simulate`, `bootstrap` and `jackknife` commands may be used with standard Stata commands, but in many cases may require that a command be constructed to produce the needed results for each repetition.
- Although the nonlinear least squares commands (`nl`, `nlsur`) may be used in an interactive mode, it is likely that a Stata program will often be the easiest way to perform any complex NLLS task.

Context 3: Mata subroutines for ado-files

Your ado-files may perform some complicated tasks which involve many invocations of the same commands. Stata's ado-file language is easy to read and write, but it is *interpreted*: Stata must evaluate each statement and translate it into machine code. Stata's *Mata* programming language (`help mata`) creates *compiled* code which can run much faster than ado-file code.

Your ado-file can call a Mata routine to carry out a computationally intensive task and return the results in the form of Stata variables, scalars or matrices. Although you may think of Mata solely as a "matrix language", it is actually a general-purpose programming language, suitable for many non-matrix-oriented tasks such as text processing and list management.

The Mata programming environment is tightly integrated with Stata, allowing interchange of variables, local and global macros and Stata matrices to and from Mata without the necessity to make copies of those objects. A Mata program can easily generate an entire set of new variables (often in one matrix operation), and those variables will be available to Stata when the Mata routine terminates.

Mata's similarity to the C language makes it very easy to use for anyone with prior knowledge of C. Its handling of matrices is broadly similar to the syntax of other matrix programming languages such as MATLAB, Ox and GAUSS. Translation of existing code for those languages or from lower-level languages such as Fortran or C is usually quite straightforward. Unlike Stata's C plugins, code in Mata is platform-independent, and developing code in Mata is easier than in compiled C.

Tools for do-file authors

In this section of the talk, I will mention a number of tools and tricks useful for do-file authors. Like any language, the Stata do-file language can be used eloquently or incoherently. Users who bring other languages' techniques and try to reproduce them in Stata often find that their Stata programs resemble Google's automated translation of French to English: possibly comprehensible, but a long way from what a native speaker would say. We present suggestions on how the language may be used most effectively.

Although I focus on authoring do-files, these tips are equally useful for *ado*-file authors: and perhaps even more important in that context, as an ado-file program may be run many times.

Looping over observations is rarely appropriate

One of the important metaphors of Stata usage is that commands operate on the entire data set unless otherwise specified. There is rarely any reason to explicitly loop over observations. Constructs which would require looping in other programming languages are generally single commands in Stata: e.g., `recode`.

For example: do not use the "programmer's if" on Stata variables! For example,

```
if (race == 1) {
        (calculate something)
} else if (race == 2) {
...
```

will not do what you expect. It will examine the value of `race` in the *first observation* of the data set, not in each observation in turn! In this case the `if` qualifier should be used.

The by prefix can often replace a loop

A programming construct rather unique to Stata is the `by` prefix. It allows you to loop over the values of one or several categorical variables without having to explicitly spell out those values. Its limitation: it can only execute a single command as its argument. In many cases, though, that is quite sufficient. For example, in an individual-level data set,

```
bysort familyid : generate familysize = _N
bysort familyid : generate single = (_N == 1)
```

will generate a family size variable by using `_N`, the total number of observations in the `by`-group. Single households are those for which that number is one; the second statement creates an indicator (dummy) variable for that household status.

Repeated statements are usually not needed

When I see a do-file with a number of very similar statements, I know that the author's first language was not Stata. A construct such as

```
generate newcode = 1 if oldcode == 11
replace newcode = 2  if oldcode == 21
replace newcode = 3 if oldcode == 31
...
```

suggests to me that the author should read `help recode`. See below for a way to automate a `recode` statement.

A number of `generate` functions can also come in handy: `inlist( ), inrange( ), cond( ), recode( )`, which can all be used to map multiple values of one variable into a new variable.

Merge can solve concordance problems

A more general technique to solve *concordance* problems is offered by `merge`. If you want to map (or concord) values into a particular scheme—for instance, associate the average income in a postal code with all households whose address lies in that code—do not use commands to define that mapping. Construct a separate data set, containing the postal code and average income value (and any other available measurements) and `merge` it with the household-level data set:

```
merge n:1 postalcode using pcstats
```

where the `n:1` clause specifies that the postal-code file must have unique entries of that variable. If additional information is available at the postal code level, you may just add it to the using file and run the merge again. One `merge` command replaces many explicit `generate` and `replace` commands.

Some simple commands are often overlooked

Nick Cox's *Speaking Stata* column in the *Stata Journal* has pointed out several often-overlooked but very useful commands. For instance, the `count` command can be used to determine, in *ad hoc* interactive use or in a do-file, how many observations satisfy a logical condition. For do-file authors, the `assert` command may be used to ensure that a necessary condition is satisfied: e.g.

```
assert gender == 1 | gender == 2
```

will bring the do-file to a halt if that condition fails. This is a very useful tool to both validate raw data and ensure that any transformations have been conducted properly.

Duplicate entries in certain variables may be logically impossible. How can you determine whether they exist, and if so, deal with them? The `duplicates` suite of commands provides a comprehensive set of tools for dealing with duplicate entries.

egen functions can solve many programming problems

Every do-file author should be familiar with `[D] functions` (functions for `generate`) and `[D] egen`. The list of official `egen` functions includes many tools which you may find very helpful: for instance, a set of row-wise functions that allow you to specify a list of variables, which mimic similar functions in a spreadsheet environment. Matching functions such as `anycount, anymatch, anyvalue` allow you to find matching values in a `varlist`. Statistical `egen` functions allow you to compute various statistics as new variables: particularly useful in conjunction with the `by`-prefix, as we will discuss.

In addition, the list of `egen` functions is open-ended: many user-written functions are available in the SSC Archive (notably, Nick Cox's `egenmore`), and you can write your own.

Learn how to use return and ereturn

Almost all Stata commands return their results in the *return list* or the *ereturn list*. These returned items are categorized as *macros, scalars* or *matrices*. Your do-file may make use of any information left behind as long as you understand how to save it for future use and refer to it in your do-file. For instance, highlighting the use of `assert`:

```
summarize region, meanonly
assert r(min) > 0 & r(max) < 5
```

will validate the values of `region` in the data set to ensure that they are valid. `summarize` is an *r-class* command, and returns its results in `r( )` items. Estimation commands, such as `regress` or `probit`, return their results in the `ereturn list`. For instance, `e(r2)` is the regression R^2, and matrix `e(b)` is the row vector of estimated coefficients.

The values from the `return list` and `ereturn list` may be used in computations:

```
summarize famsize, detail
scalar iqr = r(p75) - r(p25)
scalar semean = r(sd) / sqrt(r(N))
display "IQR : " iqr
display "mean : " r(mean) " s.e. : " semean
```

will compute and display the inter-quartile range and the standard error of the mean of `famsize`. Here we have used Stata's scalars to compute and store numeric values.

In Stata, the `scalar` plays the role of a "variable" in a traditional programming language.

The local macro

The *local macro* is an invaluable tool for do-file authors. A local macro is created with the `local` statement, which serves to name the macro and provide its content. When you next refer to the macro, you extract its value by *dereferencing* it, using the backtick (‘) and apostrophe (’) on its left and right:

```
local george 2
local paul = ‘george’ + 2
```

In this case, I use an equals sign in the second local statement as I want to *evaluate* the right-hand side, as an arithmetic expression, and store it in the macro `paul`. If I did not use the equals sign in this context, the macro `paul` would contain the string `2 + 2`.

forvalues and foreach

In other cases, you want to *redefine* the macro, not evaluate it, and you should not use an equals sign. You merely want to take the contents of the macro (a character string) and alter that string. It is easiest to illustrate this concept by introducing the two key programming constructs for repetition: `forvalues` and `foreach`. These commands, defined in the *Programming* manual, are essential for do-file writers seeking to automate their workflow. Both commands make essential use of local macros as their "counter". For instance:

```
forvalues i=1/10 {
      summarize PRweek`i'
}
```

Note that the value of the local macro `i` is used within the body of the loop when that counter is to be referenced. Any Stata *numlist* may appear in the `forvalues` statement. Note also the curly braces, which must appear at the end of their lines.

In many cases, the `forvalues` command will allow you to substitute explicit statements with a single loop construct. By modifying the range and body of the loop, you can easily rewrite your do-file to handle a different case.

The `foreach` command is even more useful. It defines an iteration over any one of a number of lists:

- the contents of a varlist (list of existing variables)
- the contents of a newlist (list of new variables)
- the contents of a numlist
- the separate words of a macro
- the elements of an arbitrary list

For example, we might want to `summarize` each of these variables:

```
foreach v of varlist price mpg rep78 {
      summarize `v', detail
}
```

Or, run a regression on variables for each country, and graph the data and fitted line:

```
local eucty DE ES FR IT UK
foreach c of local eucty {
    regress healthexp`c'   income`c'
    twoway (scatter healthexp`c'  income`c') || ///
           (lfit healthexp`c' income`c')
}
```

We can now illustrate how a local macro could be constructed by redefinition:

```
local eucty DE ES FR IT UK
local alleps
foreach c of local eucty {
    regress healthexp`c'   income`c'
    predict double eps`c', residual
    local alleps  "`alleps' eps`c'"
}
```

Within the loop we redefine the macro `alleps` (as a double-quoted string) to contain itself and the name of the residuals from that country's regression. We could then use the macro `alleps` to generate a graph of all five countries' residuals:

```
tsline  `alleps'
```

This technique can be used to automate a `recode` operation. Say that we had sequential codes for several countries (`cc`) coded as 1–4, and we wanted to apply IMF country codes to them:

```
local ctycode 111 112 136 134
local i 0
foreach c of local ctycode {
          local ++i
          local rc "`rc' (`i'=`c')"
}

display "`rc'"
 (1=111) (2=112) (3=136) (4=134)

recode cc `rc', gen(newcc)
(400 differences between cc and newcc)
```

extended macro functions, list functions, levelsof

Beyond their use in loop constructs, local macros can also be manipulated with an extensive set of *extended macro functions* and *list functions*. These functions (described in `[P] macro` and `[P] macro lists`) can be used to count the number of elements in a macro, extract each element in turn, extract the variable label or value label from a variable, or generate a list of files that match a particular pattern. A number of *string functions* are available in `[D] functions` to perform string manipulation tasks found in other string processing languages (including support for regular expressions, or *regexps*.)

A very handy command that produces a macro is `levelsof`, which returns a sorted list of the distinct values of *varname*, optionally as a macro. This list would be used in a `by`-prefix expression automatically, but what if you want to issue several commands rather than one? Then a `foreach`, using the local macro created by `levelsof`, is the solution.

Stata do-file programming: Selected recipes

I now present a number of “recipes” for solving common problems in Stata do-file programming. These are taken from my book, *An Introduction to Stata Programming*, Stata Press, 2009.

The concept behind this collection of recipes is that you may have a problem similar to one of those illustrated here. Differing somewhat, perhaps, but with enough similarity to provide guidance for the problem’s successful solution. Just as you might modify a recipe in the kitchen to deal with ingredients at hand or preferences of the dinner guests, you can modify a do-file “recipe” to meet your needs.

Tabulating a logical condition across a set of variables

The problem: considering a number of related variables, you want to determine whether, for each observation, *all* variables satisfy a logical condition. Alternatively, you might want to know whether *any* satisfy that condition (for instance, taking on inappropriate values), or you might want to *count* how many of the variables satisfy the logical condition.

This would seem to be a natural application of `egen` as that command already contains a number of row-wise functions to perform computations across variables. For instance, the `anycount()` function counts the number of variables in its *varlist* whose values for each observation match those of an integer *numlist*, while the `rowmiss()` and `rownonmiss()` functions tabulate the number of missing and non-missing values for each observation, respectively.

The three tasks above are all satisfied by `egen` functions from Nicholas Cox's `egenmore` package: `rall()`, `rany()` and `rcount()`, respectively. Why don't you just use those functions, then?

First, recall that `egen` functions are interpreted code. Unlike the built-in functions accessed by `generate`, the logic of an `egen` function must be interpreted each time it is called. For a large dataset, the time penalty can be significant. Second, to use an `egen` function, you must remember that there is such a function, and remember its name. In addition to Stata's official `egen` functions, documented in on-line help, there are many user-written `egen` functions available, but you must track them down.

For these reasons, current good programming practice suggests that you should avoid `egen` function calls in instances where the performance penalty might be an issue. This is particularly important within an ado-file program, but may apply to many do-files as well. In many cases, you can implement the logic of an `egen` function with a few lines of Stata commands.

Imagine that we have a dataset of household observations, where variables `child1...child12` contain the current age of each child (or missing values for nonexistent offspring). We would like to determine the number of school-age children. We could compute `nschool` with a `foreach` loop:

```
. generate nschool = 0
. foreach v of varlist child1-child12 {
.       replace nschool = nschool + inrange(`v', 6, 18)
. }
```

The built-in `inrange()` function will execute more efficiently than the interpreted logic within `rcount()`.

As a bonus, if we wanted to also compute an indicator variable signaling whether there are any school-age children in the household, we could do so within the same `foreach` loop:

```
. generate nschool = 0
. generate anyschool = 0
. foreach v of varlist child1-child12 {
.       replace nschool = nschool + inrange(`v', 6, 18)
.       replace anyschool = max( anyschool, inrange(`v', 6, 18))
. }
```

Note that in this case `anyschool` will remain at 0 for each observation unless one of the children's ages match the criteria specified in the `inrange` function. Once it is switched to 1, it will remain so.

An alternative (and more computationally efficient) way of writing this code takes advantage of the fact that `generate` is much faster than `replace`, as the latter command must keep track of the number of changes made in the variable. Thus, we could write

```
. foreach v of varlist child1-child12 {
.     local nschool "`nschool' + inrange(`v', 6, 18)"
. }
. generate byte nschool = `nschool'
. generate byte anyschool = nschool > 0
```

In this variation, the local macro is built up to include an `inrange()` clause for each of the possible 12 children. This version runs considerably faster on a large dataset.

Computing summary statistics over groups

The problem: your dataset has a hierarchical nature, where observations represent individuals who are also identified by their household id code, or records of individual patient visits which can be aggregated over the patient id or over the clinic id. In the latter case, you can define groups of observations belonging to a particular patient, or to a particular clinic.

With this kind of hierarchical data structure, you may often want to compute summary statistics for the groups. This can be readily performed in Stata by `tabstat` but that command will only display a table of summary measures. Alternatively, you could use `collapse` to generate a dataset of aggregated values for a variety of summary statistics, or `contract` to generate a collapsed dataset of frequencies. However, you may find that these options do not fit the bill.

What if you want to juxtapose the summary statistics for each aggregate unit with the individual observations in order to compute one or more variables for each record? For instance, you might have repeated-measures data for a physican's patients measuring their height, weight and blood pressure at the time of each office visit. You might want to flag observations where their weight is above their median weight, or when their blood pressure is above the 75^{th} percentile of their repeated measurements.

Computations such as these may be done with a judicious use of `by`-groups. For instance,

```
. by patientid: egen medwt = median(weight)
. by patientid: egen bp75 = pctile(bp), p(75)
```

We stress that you should avoid using variables to store constant values (which would occur if you omitted the `by patientid:` prefix). But in these cases, we are storing a separate constant for each `patientid`. You may now compute indicators for weight, blood pressure and at-risk status, using the `byte` datatype for these binary variables:

```
. generate byte highwt = weight > medwt & !missing(weight, medwt)
. generate byte highbp = bp > bp75 & !missing(bp, bp75)
. generate byte atrisk = highwt & highbp
```

If you need to calculate a sum for each group (`patientid` in this case), you can use the `total()` function for `egen`. Alternatively, to improve computational efficiency, you could use

```
. by patientid: generate atriskvisits = sum(atrisk)
. by patientid: generate n_atrisk = atriskvisits if _n == _N
. gsort -n_atrisk
. list patientid n_atrisk if inrange(n_atrisk, 1, .)
```

This sequence of commands uses the `sum()` function from `generate`, which is a *running sum*. Its value when `_n == _N` is the total for that `patientid`. We store that value as `n_atrisk` and sort it in descending order with `gsort`. The `list` command then prints one record per `patientid` for those patients with at least one instance of `atrisk` in their repeated measures.

Computing the extreme values of a sequence

The problem: you have hierarchical data such as observations of individual patient visits to a clinic. In the previous recipe, we described how summary statistics for each patient could be calculated. These include extrema: for instance, the highest weight ever recorded for each patient, or the lowest serum cholesterol reading. What you may need, however, is the *record* to date for those variables: the maximum (minimum) value observed *so far* in the sequence. This is a 'record' value in the context of setting a record: for instance, maximum points scored per game, or minimum time recorded for the 100-yard dash. How might you compute these values for hierarchical data?

First, let us consider a single sequence (that is, data for a single patient in our example above). You might be tempted to think that this is a case where looping over observations will be essential—and you would be wrong! We exploit the fact that Stata's `generate` and `replace` commands respect Stata's sort order. We need only record the first observation's value and then use `replace` to generate the 'record high':

```
. sort visitdate
. generate maxwt = weight in 1
. replace maxwt = max(maxwt[\_n - 1], weight) in 2/l
```

Unusually, you need not worry about missing values, as the `max()` function is smart enough to ignore them unless it is asked to compare missing with missing.

If we want to calculate a similar measure for each `patientid` in the dataset, we use the same mechanism:

```
. sort patientid visitdate
. by patientid: generate minchol = serumchol if _n == 1
. by patientid: replace minchol = ///
                      min(minchol[_n - 1], serumchol) if _n > 1
```

With repeated measures data, we cannot refer to observations 1,2, and so on as those are absolute references to the entire dataset. Recall that under the control of a `by`-group, the `_n` and `_N` values are redefined to refer to the observations in that `by`-group, allowing us to refer to `_n` in the `generate` command and the prior observation in that `by`-group with a `[_n - 1]` subscript.

Computing the length of spells

The problem: you have ordered data (for instance, a time series of measurements) and you would like to examine *spells* in the data. These might be periods during which a qualitative condition is unchanged, as signaled by an indicator variable. As examples, consider the sequence of periods during which a patient's cholesterol remains above the recommended level, or a worker remains unemployed, or a released offender stays clear of the law. Alternatively, they might signal repeated values of a measured variable, such as the number of years that a given team has been ranked first in its league. Our concern with spells may involve identifying their existence and measuring their duration.

One solution to this problem involves using a ready-made Stata command, `tsspell`, written by Nicholas J. Cox. This command can handle any aspect of our investigation. It does require that the underlying data be defined as a Stata time series (for instance, with `tsset`. This makes it less than ideal if your data are ordered but not evenly spaced, such as patient visits to their physician which may be irregularly timed. Another issue arises, though: that raised above with respect to `egen`. The `tsspell` program is fairly complicated interpreted code, which may impose a computational penalty when applied to a very large dataset. You may only need one simple feature of the program for your analysis. Thus, you may want to consider analyzing spells in do-file code, perhaps much simpler than the invocation of `tsspell`. You generally can avoid explicit looping over observations, and will want to do so whenever possible.

Assume that you have a variable denoting the ordering of the data (which might be a Stata date or date-and-time variable, but need not be) and that the data have been sorted on that variable. The variable of interest is `employer`, which takes on values `A`, `B`, `C`... or missing for periods of unemployment. You want to identify the beginning of each spell with an indicator variable. How do we know that a spell has begun? The condition

```
. generate byte beginspell = employer != employer[_n-1]
```

will suffice to define the start of each new spell (using the `byte` datatype to define this indicator variable). Of course, the data may be *left censored* in the sense that we do not start observing the employee's job history on her date of hire. But the fact that `employer[_n-1]` is missing for period 1 does not matter, as it will be captured as the start of the first spell. What about spells of unemployment? If they are coded as a missing value of `employer`, they will be considered spells as well.

First consider some fictitious data on an employee. She is first observed working for firm A in 1987, then is laid off in 1990. After a spell of unemployment, she is hired by firm B in 1992, and so on.

```
. list, sepby(employer) noobs
```

year	employer	wage	begins~l
1987	A	8.25	1
1988	A	8.50	0
1989	A	8.75	0
1990		.	1
1991		.	0
1992	B	7.82	1
1993	B	7.98	0
1994	B	8.12	0
1995	B	8.40	0
1996	B	8.52	0
1997	C	9.00	1
1998	A	9.25	1
1999		.	1
2000		.	0
2002	D	10.37	0

Notice that `beginspell` properly flags each change in employment status, including entry into unemployment. If we wanted to flag only spells of unemployment, we could do so with

```
. generate byte beginunemp = missing(employer) & (employer != employer[_n-1])
```

which would properly identify years in which unemployment spells commenced as 1990, 1999 and 2006.

With an indicator variable flagging the start of a spell, we can compute how many changes in employment status this employee has faced, as the count of that indicator variable provides that information. We can also use this notion to tag each spell as separate:

```
. list, sepby(employer) noobs
```

year	employer	wage	begins~l	beginu~p	spellnr
1987	A	8.25	1	0	1
1988	A	8.50	0	0	1
1989	A	8.75	0	0	1
1990		.	1	1	2
1991		.	0	0	2
1992	B	7.82	1	0	3
1993	B	7.98	0	0	3
1994	B	8.12	0	0	3
1995	B	8.40	0	0	3
1996	B	8.52	0	0	3
1997	C	9.00	1	0	4
1998	A	9.25	1	0	5
1999		.	1	1	6
2000		.	0	0	6
2002	D	10.37	0	0	7

What if we now want to calculate the average wage paid by each employer?

```
. sort spellnr
. by spellnr: egen meanwage = mean(wage)
```

Or the duration of employment with each employer (length of each employment spell)?

```
. by spellnr: gen length = _N if !missing(employer)
```

Here we are taking advantage of the fact that the time variable is an evenly spaced time series. If we had unequally spaced data, we would want to use Stata's date functions to compute the duration of each spell.

This example may seem not all that useful as it refers to a single employee's employment history. However, all of the techniques we have illustrated work equally well when applied in the context of panel or longitudinal data as long as they can be placed on a time-series calendar. If we add an `id` variable to these data and `xtset id year`, we may reproduce all of the results above by merely employing the `by id:` prefix. In the last three examples, we must sort by both `id` and spell: for example,

```
. sort id spellnr
. bysort id spellnr: egen meanwage = mean(wage)
```

is now required to compute the mean wage for each spell of each employee in a panel context.

A number of additional aspects of spells may be of interest. Returning to the single employee's data, we may want to flag only employment spells at least three years long. Using the `length` variable, we may generate such as indicator as:

```
. sort spellnr
. by spellnr: gen length = _N if !missing(employer)
(5 missing values generated)
. generate byte longspell = (length >= 3 & !missing(length))
. list year employer length longspell, sepby(employer) noobs
```

year	employer	length	longsp~l
1987	A	3	1
1988	A	3	1
1989	A	3	1
1990		.	0
1991		.	0
1992	B	5	1
1993	B	5	1
1994	B	5	1
1995	B	5	1
1996	B	5	1

Efficiently defining group characteristics and subsets

The problem: say that your cross-sectional dataset contains a record for each patient who has been treated at one of several clinics. You want to associate each patient's clinic with an location code (for urban clinics, the Standard Metropolitan Statistical Area (SMSA) in which the clinic is located). The SMSA identifier is not on the patient's record but it is available to you. How do you get this associated information on each patient's record without manual editing? One quite cumbersome technique (perhaps familiar to users of other statistical packages) involves writing a long sequence of statements with `if` *exp* clauses.

Let us presume that we have Stata dataset `patient` containing the individual's details as well as `clinicid`, the clinic ID. Assume that it can be dealt with as an integer. If it were a string code, that could easily be handled as well.

Create a text file, `clinics.raw`, containing two columns: the clinic ID (`clinicid`) and the SMSA FIPS code (`smsa`) For instance,

```
12367  1120
12467  1120
12892  1120
13211  1200
14012  4560
 ...    ...
23435  5400
29617  8000
32156  9240
```

where SMSA codes 1120, 1200, 4560, 5400, 8000 and 9240 refer to the Boston, Brockton, Lowell, New Bedford, Springfield-Chicopee-Holyoke and Worcester, MA SMSAs, respectively.

Read the file into Stata with `infile clinicid smsa using clinics`, and `save` the file as Stata dataset `clinic_char`. Now `use` the patient file and give the commands

```
. merge n:1 clinicid using clinic_char
. tab _merge
```

We use the `n:1` form of the `merge` command to ensure that the `clinic_char` dataset has a single record per clinic. After the merge is performed you should find that all patients now have an `smsa` variable defined. If there are missing values in `smsa`, list the `clinicid`s for which that variable is missing and verify that they correspond to non-urban locations. When you are satisfied that the merge has worked properly, type

```
. drop _merge
```

You have performed a *one-to-many merge*, attaching the same SMSA identifier to all patients who have been treated at clinics in that SMSA. You may now use the `smsa` variable to attach SMSA-specific information to each patient record with `merge`.

Unlike an approach depending on a long list of conditional statements such as

```
. replace smsa=1120 if inlist(clinicid,12367,12467,12892,...)
```

this approach leads you to create a Stata dataset containing your clinic ID numbers so that you may easily see whether you have a particular code in your list. This approach would be especially useful if you revise the list for a new set of clinics.

As Nicholas Cox has pointed out in a Stata FAQ, the above approach may also be fruitfully applied if you need to work with a subset of observations that satisfy a complicated criterion. This might be best defined in terms of an indicator variable that specifies the criterion (or its complement). The same approach may be used. Construct a file containing the identifiers that define the criterion (in the example above, the clinic IDs to be included in the analysis). Merge that file with your dataset and examine the `_merge` variable. That variable will take on values 1, 2 or 3, with a value of 3 indicating that the observation falls in the subset. You may then define the desired indicator:

```
. generate byte subset1 = _merge == 3
. drop _merge
. regress ... if subset1
```

Using this approach, any number of subsets may be easily constructed and maintained, avoiding the need for complicated conditional statements.

Handling parallel lists

The problem: For each of a set of variables, you want to perform some steps that involve another group of variables, perhaps creating a third set of variables. These are *parallel lists*, but the variable names of the other lists may not be deducible from those of the first list.How can these steps be automated?

First, let's consider that we have two arbitrary sets of variable names, and want to name the resulting variables based on the first set's variable names. For instance, you might have some time series of population data for several counties and cities:

```
. local county Suffolk Norfolk Middlesex Worcester Hampden
. local cseat Boston Dedham Cambridge Worcester Springfield
. local wc 0
. foreach c of local county {
.         local ++wc
.         local sn : word `wc´ of `cseat´
.         generate seatshare`county´ = `sn´ / `c´
. }
```

This `foreach` loop will operate on each pair of elements in the parallel lists, generating a set of new variables `seatshareSuffolk`, `seatshareNorfolk`...

Another form of this logic would use a set of numbered variables in one of the loops. In that case, you could use a `forvalues` loop over the values (assuming they were consecutive or otherwise patterned) and the extended macro function `word of...` to access the elements of the other loop. The `tokenize` command could also be used.

Alternatively, you could use a `forvalues` loop over both lists, employing the `word count` extended macro function:

```
. local n: word count `county´
. forvalues i = 1/`n´ {
.         local a: word `i´ of `county´
.         local b: word `i´ of `cseat´
.         generate seatshare`a´ = `b´/`a´
. }
```

yielding the same results as the previous approach.

You may also find this logic useful in handling a set of constant values that align with variables. Let's say that you have a cross-sectional dataset of hospital budgets over various years, in the wide structure: that is, separate variables for each year (e.g., `exp1994`, `exp1997`,.... You would like to apply a health care price deflator to each variable to place them in comparable terms. For instance:

```
. local yr 1994 1997 2001 2003 2005
. local defl 87.6 97.4 103.5 110.1 117.4
. local n: word count `yr´
. forvalues i = 1/`n´ {
.         local y: word `i´ of local yr
.         local pd: word `i´ of local defl
.         generate rexp`y´ = exp`y´ * 100 / `pd´
. }
```

This loop will generate a set of new variables measuring real expenditures, `rexp1994, rexp1997,...` by scaling each of the original (nominal valued) variables by (100/*defl*) for that year's value of the health care price deflator. This could also be achieved by using `reshape` to transform the data into the *long format*, but that is not really necessary in this context.

Requiring at least n observations per panel unit

The problem: if you have unbalanced panel data, how do you ensure that each unit has at least *n* observations available?

It is straightforward to calculate the number of available observations for each unit.

```
. xtset patient date
. by patient: generate nobs = _N
. generate want = (nobs >= n)
```

These commands will produce an indicator variable, `want`, which selects those units which satisfy the condition of having at least *n* available observations.

This works well if all you care about is the number of observations available, but you may have a more subtle concern: you want to count *consecutive* observations. You may want to compute statistics based on changes in various measurements using Stata's `L.` or `D.` time series operators Applying these operators to series with gaps will create missing values.

A solution to this problem is provided by Nicholas J. Cox and Vince Wiggins in a Stata FAQ, "How do I identify runs of consecutive observations in panel data?" The sequence of consecutive observations is often termed a *run* or a *spell*. They propose defining the runs in the timeseries for each panel unit:

```
. generate run = .
. by patient: replace run = cond(L.run == ., 1, L.run + 1)
. by patient: egen maxrun = max(run)
. generate wantn = (maxrun >= n)
```

The second command replaces the missing values of `run` with either 1 (denoting the start of a run) or the prior value + 1. For observations on consecutive dates, that will produce an integer series 1,. . . ,.*len* where *len* is the last observation in the run. When a break in the series occurs, the prior (lagged) value of `run` will be missing, and `run` will be reset to 1. The variable `maxrun` then contains, for each patient, the highest value of `run` in that unit's sequence.

Although this identifies (with indicator `wantn`) those patients who do (or do not) have a sequence or *spell* of *n* consecutive observations, it does not allow you to immediately identify this spell. You may want to retain only this longest spell, or run of observations, and discard other observations from this patient. To carry out this sort of screening, you should become familiar with Nicholas J. Cox's `tsspell` program (available from `ssc`), which provides comprehensive capabilities for spells in time series and panel data. A "canned" solution to the problem of retaining only the longest spell per patient is also available from my `onespell` routine, which makes use of `tsspell`.

Counting the number of distinct values per individual

The problem: If you have data on individuals that indicate their association with a particular entity, how do you count the number of entities associated with each individual? For instance, we may have a dataset of consumers who purchase items from various Internet vendors. Each observation identifies the consumer (`pid` and the vendor (`vid`), where 1=amazon.com, 2=llbean.com, 3=overstock.com, and so on. Several solutions were provided by Nicholas J. Cox in a Statalist posting.

```
. bysort pid vid: generate count = ( _n == 1)
. by pid : replace count = sum(count)
. by pid : replace count = count(_N)
```

Here, we consider each combination of consumer and vendor and set `count = 1` for their first observation. We then replace `count` with its `sum()` for each consumer, keeping in mind that this is a *running sum*, so that it takes on 1 for the first vendor, 2 for the second, and so on. Finally, `count` is replaced with its value in observation `_N` for each consumer: the maximum number of vendors with whom she deals.

A second, somewhat less intuitive but shorter solution:

```
. bysort pid (vid) : generate count = sum( vid != vid[_n-1] )
. by pid: replace count = count(_N)
```

This solution takes advantage of the fact that when `(vid)` is used on the `bysort` prefix, the data are sorted in order of `vid` within each `pid`, even though the `pid` is the only variable defining the `by`-group. When the `vid` changes, another value of 1 is generated and summed. When subsequent transactions pertain to the same vendor, `vid != vid[_n-1]` evaluates to 0, and those zero values are added to the sum.

This problem is common enough that an official `egen` function has been developed to *tag* observations:

```
. egen tag = tag( pid vid )
. egen count = total(tag), by(pid)
```

The `tag()` function returns 1 for the first observation of a particular combination of `pid vid`, and zero otherwise. Thus, its `total()` for each `pid` is the number of `vid`s with whom she deals.

As a last solution, Cox's `egenmore` package contains the `nvals()egen nvals()` function, which allows you to say

```
. egen count = nvals(vid), by(pid)
```

Computing firm-level correlations with multiple indices

The problem: a user on Statalist posed a question involving a very sizable dataset of firm-level stock returns and a set of index fund returns. He wanted to calculate, for each firm, the average returns and the set of correlations with the index funds, and determine with which fund they were most highly correlated.

We illustrate this problem with some actual daily stock returns data for 291 firms, 1992–2006, from CRSP (the Center for Research on Securities Prices): 311,737 firm-daily observations in total. We have constructed nine simulated index funds' returns. The hypothetical funds, managed by a group of Greek investment specialists, are labeled the Kappa, Lambda, Nu, Xi, Tau, Upsilon, Phi, Chi and Psi funds. These data are stored in `crspsubseta`.

To solve the problem, we define a loop over firms. For each firm of the `nf` firms, we want to calculate the correlations between firm returns and the set of `nind` index returns, and find the maximum value among those correlations. The variable `hiord` takes on values 1–9, while `permno` is an integer code assigned to each firm by CRSP. We set up a Stata matrix `retcorr` to hold the correlations, with `nf` rows and `nind` columns. The number of firms and number of indices are computed by the `word count` extended macro function applied to the local macro produced by `levelsof`.

```
. qui levelsof hiord, local(indices)
. local nind : word count `indices´
. qui levelsof permno, local(firms)
. local nf : word count `firms´
. matrix retcorr = J(`nf´, `nind´, .)
```

We calculate the average return for each firm with `summarize, meanonly`. In a loop over firms, we use `correlate` to compute the correlation matrix of each firm's returns, `ret`, with the set of index returns.

For firm *n*, we move the elements of the last row of the matrix corresponding to the correlations with the index returns into the n^{th} row of the `retcorr` matrix. We also place the mean for the n^{th} firm into that observation of variable `meanret`.

```
. local n 0
. qui gen meanret = .
. qui gen ndays = .
. local row = `nind´ + 1
. foreach f of local firms {
  2.          qui correlate index1-index`nind´ ret if permno == `f´
  3.          matrix sigma = r(C)
  4.          local ++n
  5.          forvalues i = 1/`nind´ {
  6.                  matrix retcorr[`n´, `i´] = sigma[`row´, `i´]
  7.          }
  8.          summarize ret if permno == `f´, meanonly
  9.          qui replace meanret = r(mean) in `n´
 10.          qui replace ndays = r(N) in `n´
 11. }
```

We now may use the `svmat` command to convert the `retcorr` matrix into a set of variables, `retcorr1-retcorr9`. The `egen` function `rowmax()` computes the maximum value for each firm. We then must determine which of the nine elements is matched by that maximum value. This number is stored in `highcorr`.

```
. svmat double retcorr
. qui egen double maxretcorr = rowmax(retcorr*)
. qui generate highcorr = .
. forvalues i = 1/`nind´ {
  2.          qui replace highcorr = `i´ if maxretcorr == retcorr`i´ ///
>         & !missing(maxretcorr)
  3. }
```

We now can sort the firm-level data in descending order of `meanret`, using `gsort` and list firms and their associated index fund numbers. These values show, for each firm, which index fund their returns most closely resemble. For brevity, we list only the fifty best-performing firms.

```
. gsort -meanret highcorr
. label values highcorr ind
. list permno meanret ndays highcorr in 1/50, noobs sep(0)
```

permno	meanret	ndays	highcorr
24969	.0080105	8	Nu
53575	.0037981	465	Tau
64186	.0033149	459	Upsilon
91804	.0028613	1001	Psi
86324	.0027118	1259	Chi
60090	.0026724	1259	Upsilon
88601	.0025065	1250	Chi
73940	.002376	531	Nu
84788	.0023348	945	Chi
22859	.0023073	1259	Lambda
85753	.0022981	489	Chi
39538	.0021567	1259	Nu
15667	.0019581	1259	Kappa
83674	.0019196	941	Chi
68347	.0019122	85	Kappa
81712	.0018903	1259	Chi
82686	.0017555	987	Chi
23887	.0017191	1259	Lambda
75625	.0017182	1259	Phi
24360	.0016474	1259	Lambda

Ado-file programming: a primer

A Stata *program* adds a command to Stata's language. The name of the program is the command name, and the program must be stored in a file of that same name with extension `.ado`, and placed on the *adopath*: the list of directories that Stata will search to locate programs.

A program begins with the `program define` *progname* statement, which usually includes the option `, rclass`, and a `version 10.1` statement. The *progname* should not be the same as any Stata command, nor for safety's sake the same as any accessible user-written command. If `findit progname` does not turn up anything, you can use that name. Programs (and Stata commands) are either *r-class* or *e-class*. The latter group of programs are for estimation; the former do everything else. Most programs you write are likely to be r-class.

The syntax statement

The `syntax` statement will almost always be used to define the command's format. For instance, a command that accesses one or more variables in the current data set will have a `syntax varlist` statement. With specifiers, you can specify the minimum and maximum number of variables to be accepted; whether they are numeric or string; and whether time-series operators are allowed. Each variable name in the `varlist` must refer to an existing variable.

Alternatively, you could specify a `newvarlist`, the elements of which must refer to new variables.

One of the most useful features of the `syntax` statement is that you can specify `[if]` and `[in]` arguments, which allow your command to make use of standard `if` *exp* and `in` *range* syntax to limit the observations to be used. Later in the program, you use `marksample touse` to create an indicator (dummy) temporary variable identifying those observations, and an `if 'touse'` qualifier on statements such as `generate` and `regress`.

The `syntax` statement may also include a `using` qualifier, allowing your command to read or write external files, and a specification of command options.

Option handling includes the ability to make options optional or required; to specify options that change a setting (such as `regress, noconstant`); that must be integer values; that must be real values; or that must be strings. Options can specify a *numlist* (such as a list of lags to be included), a *varlist* (to implement, for instance, a `by(`*varlist*`)` option); a *namelist* (such as the name of a matrix to be created, or the name of a new variable).

Essentially, any feature that you may find in an official Stata command, you may implement with the appropriate `syntax` statement. See `[P] syntax` for full details and examples.

Within your own command, you do not want to reuse the names of existing variables or matrices. You may use the `tempvar` and `tempname` commands to create "safe" names for variables or matrices, respectively, which you then refer to as local macros. That is, `tempvar eps1 eps2` will create temporary variable names which you could then use as `generate double 'eps1' = ...`

These variables and temporary named objects will disappear when your program terminates (just as any local macros defined within the program will become undefined upon exit).

So after doing whatever computations or manipulations you need within your program, how do you return its results? You may include `display` statements in your program to print out the results, but like official Stata commands, your program will be most useful if it also returns those results for further use. Given that your program has been declared `rclass`, you use the `return` statement for that purpose.

You may return scalars, local macros, or matrices:

```
return scalar teststat = `testval'
return local df = `N' - `k'
return local depvar "`varname'"
return matrix lambda = `lambda'
```

These objects may be accessed as `r(`*name*`)` in your do-file: e.g. `r(df)` will contain the number of degrees of freedom calculated in your program.

A sample program from `help return`:

```
program define mysum, rclass
    version 10.0
    syntax varname
    return local varname `varlist'
    tempvar new
    quietly {
        count if `varlist'!=.
        return scalar N = r(N)
        gen double `new' = sum(`varlist')
        return scalar sum = `new'[_N]
        return scalar mean = return(sum)/return(N)
    }
end
```

This program can be executed as `mysum` *varname*. It prints nothing, but places three scalars and a macro in the `return list`. The values `r(mean)`, `r(sum)`, `r(N)`, and `r(varname)` can now be referred to directly.

With minor modifications, this program can be enhanced to enable the `if` *exp* and `in` *range* qualifiers. We add those optional features to the `syntax` command, use the `marksample` command to delineate the wanted observations by `touse`, and apply `if 'touse'` qualifiers on two computational statements:

```
program define mysum2, rclass
    version 10.0
    syntax varname [if] [in]
    return local varname `varlist'
    tempvar new
    marksample touse
    quietly {
        count if `varlist'!=. & `touse'
        return scalar N = r(N)
        gen double `new' = sum(`varlist') if `touse'
        return scalar sum = `new'[_N]
        return scalar mean = return(sum)/return(N)
    }
end
```

Examples of ado-file programming

As another example of ado-file programming, we consider that the `rolling:` prefix (see `help rolling`) will allow you to save the estimated coefficients (`_b`) and standard errors (`_se`) from a moving-window regression. What if you want to compute a quantity that depends on the full variance-covariance matrix of the regression (VCE)? Those quantities cannot be saved by `rolling:`.

For instance, the regression

```
. regress y L(1/4).x
```

estimates the effects of the last four periods' values of `x` on `y`. We might naturally be interested in the sum of the lag coefficients, as it provides the *steady-state* effect of `x` on `y`. This computation is readily performed with `lincom`. If this regression is run over a moving window, how might we access the information needed to perform this computation?

A solution is available in the form of a *wrapper program* which may then be called by `rolling:`. We write our own `r`-class program, `myregress`, which returns the quantities of interest: the estimated sum of lag coefficients and its standard error.

The program takes as arguments the *varlist* of the regression and two required options: `lagvar()`, the name of the distributed lag variable, and `nlags()`, the highest-order lag to be included in the `lincom`. We build up the appropriate expression for the `lincom` command and return its results to the calling program.

```
 . type myregress.ado
*! myregress v1.0.0  CFBaum 11aug2008
program myregress, rclass
version 10.1
syntax varlist(ts) [if] [in], LAGVar(string) NLAGs(integer)
regress `varlist´ `if´ `in´
local nl1 = `nlags´ - 1
forvalues i = 1/`nl1´ {
        local lv "`lv´ L`i´.`lagvar´ + "
}
local lv "`lv´  L`nlags´.`lagvar´"
lincom `lv´
return scalar sum = `r(estimate)´
return scalar se = `r(se)´
end
```

As with any program to be used under the control of a prefix operator, it is a good idea to execute the program directly to test it to ensure that its results are those you could calculate directly with `lincom`.

```
 . use wpi1, clear
. qui myregress wpi L(1/4).wpi t, lagvar(wpi) nlags(4)

. return list
scalars:
                  r(se) =  .0082232176260432
                 r(sum) =  .9809968042273991
. lincom   L.wpi+L2.wpi+L3.wpi+L4.wpi
 ( 1)  L.wpi + L2.wpi + L3.wpi + L4.wpi = 0
```

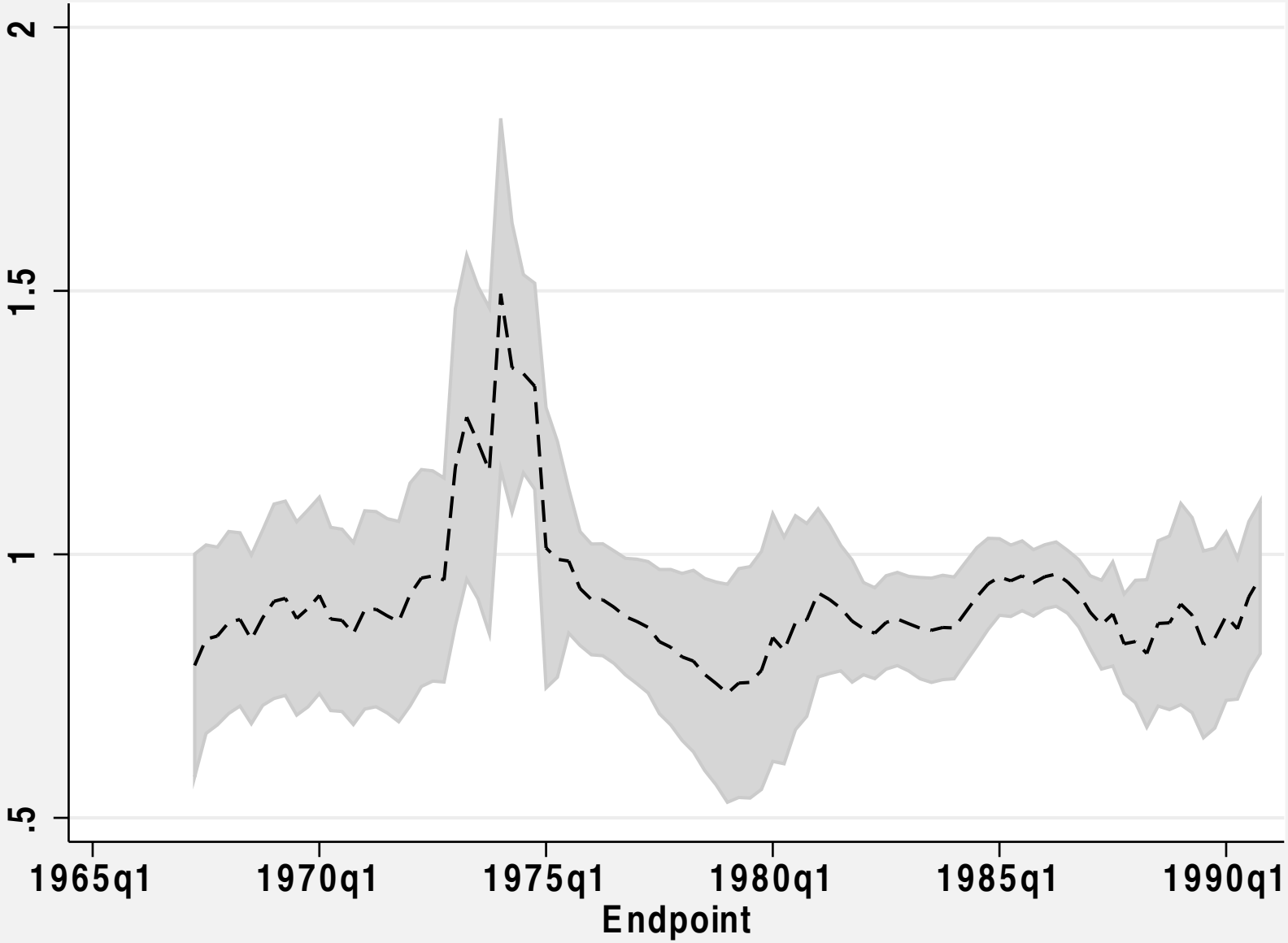
wpi	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
(1)	.9809968	.0082232	119.30	0.000	.9647067	.9972869

Having validated the wrapper program by comparing its results with those from `lincom`, we may now invoke it with `rolling`:

```
 . rolling sum=r(sum) se=r(se) ,window(30) : ///
> myregress wpi L(1/4).wpi t, lagvar(wpi) nlags(4)
(running myregress on estimation sample)
Rolling replications (95)
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
.....................................................    50
.....................................
```

We may graph the resulting series and its approximate 95% standard error bands with `twoway rarea` and `tsline`:

```
 . tsset end, quarterly
        time variable:  end, 1967q2 to 1990q4
                delta:  1 quarter
. label var end Endpoint
. g lo = sum - 1.96 * se
. g hi = sum + 1.96 * se
. twoway rarea lo hi end, color(gs12) title("Sum of moving lag coefficients, ap
> prox. 95% CI") ///
> ||  tsline sum, legend(off) scheme(s2mono)
```


Sum of moving lag coefficients, approx. 95% CI
2
1.5
1
.5
1965q1
1970q1
1975q1
1980q1
1985q1
1990q1
Endpoint

Examples of ado-file programming

As a third example of ado-file programming, consider the problem that in panel (longitudinal) data, many datasets contain *unbalanced panels*, with differing numbers of observations for different units in the panel. Some estimators commonly employed in a panel-data context can work with unbalanced panels, but expect to find a single *spell* for each unit: that is, a time-series without gaps.

Finding and retaining the single longest spell for each unit within the panel is quite straightforward in the case of a single variable. However, for our purposes, we want to apply this logic *listwise*, and delete shorter spells if *any* of the variables in a specified *varlist* are missing. The solution will entail creation of a new, smaller dataset in which only panel units with single spells are present.

We present a solution to this problem here in the context of an ado-file, `onespell.ado`. Dealing with this problem—finding and retaining the single longest spell for each unit within the panel—is quite straightforward in the case of a single variable. However, we want to apply this logic *listwise*, and delete shorter spells if *any* of the variables in a specified *varlist* are missing.

The program builds upon Nicholas J. Cox's excellent `tsspell` command, which examines a single variable, optionally given a logical condition that defines a spell and creates three new variables: `_spell`, indicating distinct spells (taking on successive integer values); `_seq`, giving the sequence of each observation in the spell (taking on successive integer values); and `_end`, indicating the end of spells. If applied to panel data rather than a single timeseries, the routine automatically performs these observations for each unit of a panel.

In this first part of the program, we define the syntax of the ado-file. The program accepts a *varlist* of any number of numeric variables, `if` *exp* and `in` *range* options, and requires that the user provide a filename in the `saving()` option in which the resulting edited dataset will be stored. Optionally, the user may specify a `replace` option (which, as is usual in Stata, must be spelled out). The `noisily` option is provided for debugging purposes. The `preserve` command allows us to modify the data and return to the original dataset.

The `tsset` command allows us to retrieve the names of the panel variable and time variable. If the data are not `tsset`, the program will abort. The `tsfill` command fills any gaps in the time variable with missing observations. We then use `marksample touse` to apply any qualifiers on the set of observations and define a number of `tempvar`s.

For ease of exposition, I do not list the entire ado-file here. Rather, the first piece of the code is displayed (as a text file), and the remainder (also as a text file) as a separate listing below a discussion of its workings.

```
. type onespell_part1.txt
*! onespell 1.1.1  CFBaum  13jan2005
* locate units with internal gaps in varlist and zap all but longest spell
program onespell, rclass
        version 10.1
        syntax varlist(numeric) [if] [in], Saving(string) [ REPLACE NOIsily]
        preserve
        quietly tsset
        local pv "`r(panelvar)´"
        local tv "`r(timevar)´"
        summarize `pv´, meanonly
        local n1 = r(N)
        tsfill
        marksample touse
        tempvar testgap spell seq end maxspell keepspell wantspell
        local sss = cond("`noisily´" != "", "noisily", "quietly")
```

The real work is performed in the second half of the program. The temporary variable `testgap` is generated with the `cond()` function to define each observation as either its value of the panel variable (`pv`) or missing. Cox's `tsspell` is then invoked on the `testgap` variable with the logical condition that the variable is non-missing. We explicitly name the three variables created by `tsspell` as temporary variables `spell, seq` and `end`.

In the first step of pruning the data, we note that any observation for which `spell = 0` may be discarded, along with any observations not defined in the `touse` restrictions. Now, for each panel unit, we consider how many spells exist. If `spell > 1`, there are gaps in the usable data. The longest spell for each panel unit is stored in temporary variable `maxspell`, produced by `egen max()` from the `seq` counter.

Now, for each panel unit, we generate a temporary variable `keepspell`, identified by the longest observed spell (`maxspell`) for that unit. We then can calculate temporary variable `wantspell` with `egen max()`, which places the `keepspell` value in each observation of the desired spell. What if there are two (or more) spells of identical length? By convention, the latest spell is chosen by this logic.

We can now apply `keep` to retain only those observations, for each panel unit, associated with that unit's longest spell: those for which `wantspell` equals the `spell` number. The resulting data are then `save`d to the file specified in the `saving()` option, optionally employing `replace`, and the original data are `restore`d.

```
 . type onespell_part2.txt
        `sss´ {
* testgap is panelvar if obs is usable, 0 otherwise
                generate `testgap´ = cond(`touse´, `pv´, .)
                tsspell `testgap´ if !missing(`testgap´), spell(`spell´) seq(`s
> eq´) end(`end´)
                drop if `spell´ == 0 | `touse´ == 0
* if `spell´ > 1 for a unit, there are gaps in usable data
* calculate max length spell for each unit and identify
* that spell as the one to be retained
                egen `maxspell´ = max(`seq´), by(`pv´)
                generate `keepspell´ = cond(`seq´==`maxspell´, `spell´, 0)
                egen `wantspell´ = max(`keepspell´), by(`pv´)
* in case of ties, latest spell of max length is selected
                list `pv´ `tv´ `spell´ `seq´ `maxspell´ `keepspell´ `wantspell´
> , sepby(`pv´)
                summarize `spell´ `wantspell´
                keep if `wantspell´ == `spell´
                summarize `pv´, meanonly
                local n2 = r(N)
                drop \__*
        }
        display _n "Observations removed: " `n1´-`n2´
        save `saving´, `replace´
        restore
end
```

To illustrate, we modify the `grunfeld` dataset. The original dataset is a balanced panel of 20 years' observations on 10 firms. We remove observations from different variables in firms 2, 3 and 5, creating two spells in firms 2 and 3 and three spells in firm 5. We then apply `onespell`:

```
 . webuse grunfeld, clear
. quietly replace invest = . in 28
. quietly replace mvalue = . in 55
. quietly replace kstock = . in 87
. quietly replace kstock = . in 94
. onespell invest mvalue kstock, saving(grun1) replace
Observations removed: 28
file grun1.dta saved
```

A total of 28 observations are removed. The tabulation shows that firms 2, 3 and 5 now have longest spells of 12, 14 and 6 years, respectively.

```
. use grun1, clear
. tab company
```

company	Freq.	Percent	Cum.
1	20	11.63	11.63
2	12	6.98	18.60
3	14	8.14	26.74
4	20	11.63	38.37
5	6	3.49	41.86
6	20	11.63	53.49
7	20	11.63	65.12
8	20	11.63	76.74
9	20	11.63	88.37
10	20	11.63	100.00
Total	172	100.00	

Although this routine meets a specialized need, the logic that it employs may be useful in a number of circumstances for data management.

egen, nl and gmm programming

In this section of the talk, I will discuss writing `egen` functions and routines for use with the `nl` and `nlsur` (nonlinear least squares) and `gmm` (generalized method of moments) commands.

egen functions

The `egen` (Extended Generate) command is open-ended, in that any Stata user may define an additional `egen` function by writing a specialized ado-file program.The name of the program (and of the file in which it resides) must start with `_g`: that is, `_gcrunch.ado` will define the `crunch()` function for `egen`.

To illustrate `egen` functions, let us create a function to generate the 90–10 percentile range of a variable. The syntax for `egen` is:

`egen` [*type*] *newvar* = *fcn*(`arguments`) [*if*][*in*] [, *options*]

The `egen` command, like `generate`, may specify a data type. The `syntax` command indicates that a *newvarname* must be provided, followed by an equals sign and an *fcn*, or function, with *arguments*. `egen` functions may also handle `if` *exp* and `in` *range* qualifiers and options.

We calculate the percentile range using `summarize` with the `detail` option. On the last line of the function, we `generate` the new variable, of the appropriate type if specified, under the control of the `` `touse' `` temporary indicator variable, limiting the sample as specified.

```
. type _gpct9010.ado
*! _gpct9010 v1.0.0  CFBaum 11aug2008
        program _gpct9010
        version 10.1
        syntax newvarname =/exp [if] [in]
        tempvar touse
        mark `touse´ `if´ `in´
        quietly summarize `exp´ if `touse´, detail
        quietly generate `typlist´ `varlist´ = r(p90) - r(p10) if `touse´
end
```

This function works perfectly well, but it creates a new variable containing a single scalar value. As noted earlier, that is a very profligate use of Stata's memory (especially for large `_N`) and often can be avoided by retrieving the single scalar which is conveniently stored by our `pctrange` command. To be useful, we would like the `egen` function to be *byable*, so that it could compute the appropriate percentile range statistics for a number of groups defined in the data.

The changes to the code are relatively minor. We add an options clause to the `syntax` statement, as `egen` will pass the `by` prefix variables as a `by` *option* to our program. Rather than using `summarize`, we use `egen`'s own `pctile()pctile()` function, which is documented as allowing the `by` *prefix*, and pass the options to this function. The revised function reads:

```
. type _gpct9010.ado
*! _gpct9010 v1.0.1  CFBaum 11aug2008
        program _gpct9010
        version 10.1
        syntax newvarname =/exp [if] [in] [, *]
        tempvar touse p90 p10
        mark `touse´ `if´ `in´
        quietly {
                egen double `p90´ = pctile(`exp´) if `touse´, `options´ p(90)
                egen double `p10´ = pctile(`exp´) if `touse´, `options´ p(10)
                generate `typlist´ `varlist´ = `p90´ - `p10´ if `touse´
        }
end
```

These changes permit the function to produce a separate percentile range for each group of observations defined by the `by`-list.

To illustrate, we use `auto.dta`:

```
 . sysuse auto, clear
(1978 Automobile Data)
. bysort rep78 foreign: egen pctrange = pct9010(price)
```

Now, if we want to compute a summary statistic (such as the percentile range) for each observation classified in a particular subset of the sample, we may use the `pct9010()` function to do so.

nl and nlsur programs

You may perform nonlinear least squares estimation for either a single equation (`nl`) or a set of equations (`nlsur`). Although these commands may be used interactively or in terms of “programmed substitutable expressions," most serious use is likely to involve your writing a *function evaluator program*. That program will compute the dependent variable(s) as a function of the parameters and variables specified.

The techniques used for a maximum likelihood function evaluator, as described earlier, are quite similar to those used by `nl` and `nlsur` function evaluator programs. For instance, we might want to estimate a constant elasticity of substitution (CES) production function

$$\ln Q_i = \beta_0 - \frac{1}{\rho} \ln \left(\delta K_i^{-\rho} + (1 - \delta) L_i^{-\rho} \right) + \epsilon_i$$

which relates a firm's output Q_i to its use of capital, or machinery K_i and labor L_i. The parameters in this highly nonlinear relationship are β_0, ρ and δ.

We store the function evaluator program `nlces.ado`, as `nl` requires a program name that starts with the letters `nl`. As described in `nl`, the `syntax` statement must specify a *varlist*, allow for an `if` *exp*, and an option `at(`*name*`)`. The parameters to be estimated are passed to your program in the row vector `at`. In our CES example, the *varlist* must contain exactly three variables, which are extracted from the *varlist* by the `args` command. This command assigns its three arguments to the three variable names provided in the *varlist*.

For ease of reference, we assign `tempname`s to the three parameters to be estimated. The `generate` and `replace` statements make use of the `if` *exp* clause. The function evaluator program must replace the observations of the dependent variable: in this case, the first variable passed to the program, referenced within as `logoutput`.

```
. type nlces.ado
*! nlces v1.0.0  CFBaum 11aug2008
program nlces
   version 10.1
   syntax varlist(numeric min=3 max=3) if, at(name)
   args logoutput K L
   tempname b0 rho delta
   tempvar kterm lterm
   scalar `b0´ = `at´[1, 1]
   scalar `rho´ = `at´[1, 2]
   scalar `delta´ = `at´[1, 3]
   gen double `kterm´ = `delta´ * `K´^( -(`rho´ )) `if´
   gen double `lterm´ = (1 - `delta´) *`L´^( -(`rho´ )) `if´
   replace `logoutput´ = `b0´ - 1 / `rho´ * ln( `kterm´ + `lterm´ ) `if´
end
```

We invoke the estimation process with the `nl` command using Stata's `production` dataset. You specify the name of your likelihood function evaluator by including only the unique part of its name (that is, `ces`, not `nlces`), followed by `@`. The order in which the parameters appear in the `parameters()` and `initial()` options defines their order in the `at` vector. The `initial()` option is not required, but is recommended.

```
. use production, clear
. nl ces @ lnoutput capital labor, parameters(b0 rho delta) ///
>  initial(b0 0 rho 1 delta 0.5)
(obs = 100)
Iteration 0:  residual SS =  29.38631
Iteration 1:  residual SS =  29.36637
Iteration 2:  residual SS =  29.36583
Iteration 3:  residual SS =  29.36581
Iteration 4:  residual SS =  29.36581
Iteration 5:  residual SS =  29.36581
Iteration 6:  residual SS =  29.36581
Iteration 7:  residual SS =  29.36581
```

Source	SS	df	MS
Model	91.1449924	2	45.5724962
Residual	29.3658055	97	.302740263
Total	120.510798	99	1.21728079

Number of obs	100
R-squared	0.7563
Adj R-squared	0.7513
Root MSE	.5502184
Res. dev.	161.2538

lnoutput	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
/b0	3.792158	.099682	38.04	0.000	3.594316	3.989999
/rho	1.386993	.472584	2.93	0.004	.4490443	2.324941
/delta	.4823616	.0519791	9.28	0.000	.3791975	.5855258

```
  Parameter b0 taken as constant term in model & ANOVA table
```

After execution, you have access to all of Stata's postestimation commands. For instance, the elasticity of substitution $\sigma = 1/(1+\rho)$ of the CES function is not directly estimated, but is rather a nonlinear function of the estimated parameters. We may use Stata's `nlcom` command to generate point and interval estimates of σ using the *delta method:*

```
. nlcom (sigma: 1 / ( 1 + [rho]_b[_cons] ))
      sigma:  1 / ( 1 + [rho]_b[_cons] )
```

lnoutput	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
sigma	.4189372	.0829424	5.05	0.000	.2543194	.583555

This value, falling below unity in point and interval form, indicates that in the firms studied the two factors of production (capital and labor) are not very substitutable for one another.

The programming techniques illustrated here for `nl` carry over to the `nlsur` command (new in Stata version 10), which allows you to apply nonlinear least squares to a system of non-simultaneous (or *seemingly unrelated*) equations. Likewise, you could write a *wrapper* for `nlces`, as we illustrated before in the case of maximum likelihood, in order to create a new Stata command.

gmm programs

Like `nl`, Stata's new `gmm` command may be used with either substitutable expressions or a moment-evaluator program. We focus here on the development of moment-evaluator programs, which are similar to the function-evaluator programs you might develop for maximum likelihood estimation (with `ml` or nonlinear least squares (`nl, nlsur`).

A GMM moment-evaluator program receives a *varlist* and replaces its elements with the error part of each moment condition. An abbreviated form of the syntax for the `gmm` command is:

```
gmm moment_pgm [ if ] [ in ], equations( moment_names ) ///
parameters( param_names ) [instruments() options]
```

For instance, say that we wanted to compute linear instrumental variables regression estimates via GMM. This is unnecessary, as official `ivregress` and Baum–Schaffer–Stillman `ivreg2` provide this estimator, but let us consider it for pedagogical purposes. We have a dependent variable y, a set of regressors X and an instrument matrix Z which contains the exogenous variables in X as well as additional excluded exogenous variables, or instruments.

The moment conditions to be defined are the statements that each variable in Z is assumed to have zero covariance with the error term in the equation. We replace the population error term with its empirical counterpart, the residual $e = (y - Xb)$ where b is the vector of estimated parameters in the model. The moment-evaluator program computes this residual vector, while the `gmm` command specifies the variables to be included in Z.

Our moment evaluator program is:

```
. program gmm_ivreg
  1.      version 11
  2.      syntax varlist [if] , at(name) rhs(varlist) depvar(varlist)
  3.      tempvar m
  4.      quietly gen double `m´ = 0 `if´
  5.      local i 1
  6.      foreach var of varlist `rhs´ {
  7.          quietly replace `m´ = `m´ + `var´*`at´[1,`i´] `if´
  8.          local ++i
  9.      }
 10.      quietly replace `m´ = `m´ + `at´[1,`i´] `if´    // constant
 11.      quietly replace `varlist´ = `depvar´ - `m´ `if´
 12. end
```

The row vector `at` contains the current parameter values.

To invoke the program with the `auto` dataset, consider a model where `mpg` is the dependent variable, `gear_ratio` and `turn` are the explanatory variables, and we consider `turn` to be endogenous. The instrument matrix Z contains `gear_ratio, length, headroom` and a units vector. There is one equation (one vector of residuals) being computed, and three parameters to be estimated. As the `gmm` command does not have `depvar` or `rhs` options, the contents of those options are passed through to our moment-evaluator program.

```
. gmm gmm_ivreg, nequations(1) nparameters(3) ///
        instruments(gear_ratio length headroom) depvar(mpg) ///
        rhs(gear_ratio turn)
```

Executing this command gives us the GMM estimates:

```
. sysuse auto, clear
(1978 Automobile Data)
. gmm gmm_ivreg, nequations(1) nparameters(3) ///
>          instruments(gear_ratio length headroom) depvar(mpg) ///
>          rhs(gear_ratio turn) nolog
Final GMM criterion Q(b) =   .002489
GMM estimation
Number of parameters =   3
Number of moments    =   4
Initial weight matrix: Unadjusted                 Number of obs   =        74
GMM weight matrix:     Robust
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	.0034983	1.769734	0.00	0.998	-3.465116	3.472113
/b2	-1.218523	.1897103	-6.42	0.000	-1.590348	-.8466975
/b3	69.61528	12.14667	5.73	0.000	45.80824	93.42233

```
Instruments for equation 1: gear_ratio length headroom _cons
```

The `gmm` command may be used to estimate models that are not already programmed in Stata, including those containing nonlinear moment conditions, models with multiple equations and panel data models. This illustration merely lays the groundwork for more complex applications of GMM estimation procedures.

For instance, we might want to apply Poisson regression in a panel data context. A standard Poisson regression may be written as

$$y = \exp(x'\beta) + u$$

If the x variables are strictly exogenous, this gives rise to the moment condition

$$E[x\{y - \exp(x'\beta)\}] = 0$$

and we need only compute the residuals from this expression to implement GMM estimation.

In a panel context, with an individual heterogeneity term (fixed effect) η_i, we have

$$E(y_{it}|x_{it}, \eta_i) = \exp(x'_{it}\beta + \eta_i) = \mu_{it}\nu_i$$

where $\mu_{it} = \exp(x'_{it}\beta$ and $\nu_i = \exp(\eta_i)$.

With an additive error term ϵ, we have the regression model

$$y_{it} = \mu_{it}\nu_i + \epsilon_{it}$$

where η_i is allowed to be correlated with the regressors.

With strictly exogenous regressors, the sample moment conditions are

$$\sum_i \sum_t x_{it} \left(y_{it} - \mu_{it} \frac{\bar{y}_i}{\bar{\mu}_i} \right) = 0$$

where the bar values are the means of y and μ for panel i. As $\bar{\mu}_i$ depends on the parameters of the model, it must be recalculated within the residual equation.

Our moment evaluator program for this problem is then:

```
. program gmm_ppois
  1.    version 11
  2.    syntax varlist if, at(name) myrhs(varlist) ///
>    mylhs(varlist) myidvar(varlist)
  3.    quietly {
  4.       tempvar mu mubar ybar
  5.       gen double `mu´ = 0 `if´
  6.       local j = 1
  7.       foreach var of varlist `myrhs´ {
  8.       replace `mu´ = `mu´ + `var´*`at´[1,`j´] `if´
  9.       local ++j
 10.    }
 11.    replace `mu´ = exp(`mu´)
 12.    egen double `mubar´ = mean(`mu´) `if´, by(`myidvar´)
 13.    egen double `ybar´ = mean(`mylhs´) `if´, by(`myidvar´)
 14.    replace `varlist´ = `mylhs´ - `mu´*`ybar´/`mubar´ `if´
 15. }
 16. end
```

Using the `poisson1` dataset with three exogenous regressors, we estimate the model:

```
. webuse poisson1, clear
. gmm gmm_ppois, mylhs(y) myrhs(x1 x2 x3) myidvar(id) ///
>   nequations(1) parameters(b1 b2 b3) ///
>   instruments(x1 x2 x3, noconstant) vce(cluster id) ///
>   onestep nolog
Final GMM criterion Q(b) =  5.13e-27
GMM estimation
Number of parameters =   3
Number of moments    =   3
Initial weight matrix: Unadjusted                 Number of obs   =      409
                                  (Std. Err. adjusted for 45 clusters in id)
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	1.94866	.1000265	19.48	0.000	1.752612	2.144709
/b2	-2.966119	.0923592	-32.12	0.000	-3.14714	-2.785099
/b3	1.008634	.1156561	8.72	0.000	.781952	1.235315

```
Instruments for equation 1: x1 x2 x3
```

In this estimation, we use our program's `mylhs, myrhs` and `myidvar` options to specify the model. The `noconstant` is used in the instrument list, as a separate intercept cannot be identified in the model, and the covariance matrix is cluster-robust by the `id` (panel) variable. The one-step GMM estimator is used, as with strictly exogenous regressors, the model is exactly identified, leading to a Hansen *J* of approximately zero.

Introduction to Mata

Since the release of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with most of the capabilities of MATLAB, R, Ox or GAUSS. You can use Mata interactively, or you can develop Mata functions to be called from Stata. In this talk, we emphasize the latter use of Mata.

Mata functions may be particularly useful where the algorithm you wish to implement already exists in matrix-language form. It is quite straightforward to translate the logic of other matrix languages into Mata: much more so than converting it into Stata's matrix language.

A large library of mathematical and matrix functions is provided in Mata, including optimization routines, equation solvers, decompositions, eigensystem routines and probability density functions. Mata functions can access Stata's variables and can work with virtual matrices (*views*) of a subset of the data in memory. Mata also supports file input/output.

Circumventing the limits of Stata's matrix language

Mata circumvents the limitations of Stata's traditional matrix commands. Stata matrices must obey the maximum *matsize:* 800 rows or columns in Intercooled Stata. Thus, code relying on Stata matrices is fragile. Stata's matrix language does contain commands such as `matrix accum` which can build a cross-product matrix from variables of any length, but for many applications the limitation of *matsize* is binding.

Even in Stata/SE or Stata/MP, with the possibility of a much larger *matsize*, Stata's matrices have another drawback. Large matrices consume large amounts of memory, and an operation that converts Stata variables into a matrix or *vice versa* will require at least twice the memory needed for that set of variables.

The Mata programming language can sidestep these memory issues by creating matrices with contents that refer directly to Stata variables—no matter how many variables and observations may be referenced. These virtual matrices, or *views*, have minimal overhead in terms of memory consumption, regardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements (but not both). This implies that you can use Mata productively in a list processing environment as well as in a numeric context.

For example, a prominent list-handling command, Bill Gould's `adoupdate`, is written almost entirely in Mata. `viewsource adoupdate.ado` reveals that only 22 lines of code (out of 1,193 lines) are in the ado-file language. The rest is Mata.

Speed advantages

Last but by no means least, ado-file code written in the matrix language with explicit subscript references is *slow*. Even if such a routine avoids explicit subscripting, its performance may be unacceptable. For instance, David Roodman's `xtabond2` can run in version 7 or 8 without Mata, or in version 9 or 10 with Mata. The non-Mata version is an order of magnitude slower when applied to reasonably sized estimation problems.

In contrast, Mata code is automatically compiled into *bytecode*, like Java, and can be stored in object form or included in-line in a Stata do-file or ado-file. Mata code runs many times faster than the interpreted ado-file language, providing significant speed enhancements to many computationally burdensome tasks.

An efficient division of labor

Mata interfaced with Stata provides for an efficient division of labor. In a pure matrix programming language, you must handle all of the housekeeping details involved with data organization, transformation and selection. In contrast, if you write an ado-file that calls one or more Mata functions, the ado-file will handle those housekeeping details with the convenience features of the `syntax` and `marksample` statements of the regular ado-file language. When the housekeeping chores are completed, the resulting variables can be passed on to Mata for processing.

Mata can access Stata variables, local and global macros, scalars and matrices, and modify the contents of those objects as needed. If Mata's *view matrices* are used, alterations to the matrix within Mata modifies the Stata variables that comprise the view.

In the rest of this talk, I will discuss:

- Basic elements of Mata syntax
- Design of a Mata function
- Mata's interface functions
- Some examples of Stata–Mata routines

Operators

To understand Mata syntax, you must be familiar with its operators. The comma is the *column-join* operator, so

```
: r1 = ( 1, 2, 3 )
```

creates a three-element row vector. We could also construct this vector using the *row range operator* (..) as

```
: r1 = (1..3)
```

The backslash is the *row-join* operator, so

```
c1 = ( 4 \ 5 \ 6 )
```

creates a three-element column vector. We could also construct this vector using the *column range operator* (::) as

```
: c1 = (4::6)
```

We may combine the column-join and row-join operators:

```
m1 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
```

creates a 3×3 matrix.

The matrix could also be constructed with the row range operator:

```
m1 = ( 1..3 \ 4..6 \ 7..9 )
```

The prime (or apostrophe) is the transpose operator, so

```
r2 = ( 1 \ 2 \ 3 )'
```

is a row vector.

The comma and backslash operators can be used on vectors and matrices as well as scalars, so

```
r3 = r1, c1'
```

will produce a six-element row vector, and

```
c2 = r1' \ c1
```

creates a six-element column vector.

Matrix elements can be real or complex, so `2 - 3 i` refers to a complex number $2 - 3 \times \sqrt{-1}$.

The standard algebraic operators plus ($+$), minus ($-$) and multiply ($*$) work on scalars or matrices:

```
g = r1' + c1
h = r1 * c1
j = c1 * r1
```

In this example `h` will be the 1 $\times$ 1 dot product of vectors `r1, c1` while `j` is their 3 $\times$ 3 outer product.

Element-wise calculations and the colon operator

One of Mata's most powerful features is the *colon operator*. Mata's algebraic operators, including the forward slash (/) for division, also can be used in element-by-element computations when preceded by a colon:

```
k = r1' :* c1
```

will produce a three-element column vector, with elements as the product of the respective elements: $k_i = r1_i \; c1_i, \; i = 1, \ldots, 3$.

Mata's colon operator is very powerful, in that it will work on nonconformable objects. For example:

```
r4 = ( 1, 2, 3 )
m2 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
m3 = r4 :+ m2
m4 = m1 :/ r1
```

adds the row vector `r4` to each row of the 3 $\times$ 3 matrix `m2` to form `m3`, and divides the elements of each row of matrix `m1` by the corresponding elements of row vector `r1` to form `m4`.

Mata's scalar functions will also operate on elements of matrices:

```
d = sqrt(c)
```

will take the element-by-element square root, returning missing values where appropriate.

Logical operators

As in Stata, the equality logical operators are `a == b` and `a != b`. They will work whether or not `a` and `b` are conformable or even of the same type: `a` could be a vector and `b` a matrix. They return 0 or 1.

Unary not `!` returns 1 if a scalar equals zero, 0 otherwise, and may be applied in a vector or matrix context, returning a vector or matrix of 0, 1.

The remaining logical comparison operators (`>`, `>=`, `<`, `<=`) can only be used on objects that are conformable and of the same general type (numeric or string). They return 0 or 1.

As in Stata, the logical and (`&`) and or (`|`) operators may only be applied to real scalars.

Subscripting

Subscripts in Mata utilize square brackets, and may appear on either the left or right of an algebraic expression. There are two forms: *list* subscripts and *range* subscripts.

With list subscripts, you can reference a single element of an array as `x[i,j]`. But `i` or `j` can also be a vector: `x[i,jvec]`, where `jvec=(4,6,8)` references row `i` and those three columns of `x`. Missing values (dots) reference all rows or columns, so `x[i,.]` or `x[i,]` extracts row `i`, and `x[.,.]` or `x[,]` references the whole matrix.

You may also use *range operators* to avoid listing each consecutive element: `x[(1..4),.]` and `x[(1::4),.]` both reference the first four rows of `x`. The double-dot range creates a row vector, while the double-colon range creates a column vector. Either may be used in a subscript expression. Ranges may also decrement, so `x[(3::1),.]` returns those rows in reverse order.

Range subscripts use the notation `[| |]`. They can reference single elements of matrices, but are not useful for that. More useful is the ability to say `x[| i,j \ m,n |]`, which creates a submatrix starting at `x[i,j]` and ending at `x[m,n]`. The arguments may be specified as missing (dot), so `x[| 1,2 \4,. |]` specifies the submatrix ending in the last column and `x[| 2,2 \ .,.|]` discards the first row and column of `x`. They also may be used on the left hand side of an expression, or to extract a submatrix:
`v = invsym(xx)[| 2,2 \ .,.|]` discards the first row and column of the inverse of `xx`.

You need not use range subscripts, as even the specification of a submatrix can be handled with list subscripts and range *operators*, but they are more convenient for submatrix extraction and faster in terms of execution time.

Loop constructs

Several constructs support loops in Mata. As in any matrix language, explicit loops should not be used where matrix operations can be used. The most common loop construct resembles that of the C language:

for (*starting_value; ending_value; incr*) {
    statements
}

where the three elements define the starting value, ending value or bound and increment or decrement of the loop. For instance:

```
for (i=1; i<=10; i++) {
   printf("i=%g \n", i)
}
```

prints the integers 1 to 10 on separate lines.

If a single statement is to be executed, it may appear on the `for` statement.

You may also use `do`, which follows the syntax

do {
    statements
} while (*exp*)

which will execute the *statements* at least once.

Alternatively, you may use `while`:

while(*exp*) {
    statements
}

which could be used, for example, to loop until convergence.

Mata conditional statements

To execute certain statements conditionally, you use `if, else`:

if (*exp*) *statement*

if (*exp*) *statement1*
  else *statement2*

```
if (exp1) {
    statements1
}
  else if (exp2) {
    statements2
}
  else {
    statements3
}
```

You may also use the conditional `a ? b : c`, where `a` is a real scalar. If `a` evaluates to true (nonzero), the result is set to `b`, otherwise `c`. For instance,

```
if (k == 0)   dof = n-1
else          dof = n-k
```

can be written as

```
dof = ( k==0 ? n-1 : n-k )
```

The increment (++) and decrement (−−) operators can be used to manage counter variables. They may precede or follow the variable.

The operator `A # B` produces the Kronecker or direct product of `A` and `B`.

Element and organization types

To call Mata code within an ado-file, you must define a Mata function, which is the equivalent of a Stata ado-file program. Unlike a Stata program, a Mata function has an explicit *return type* and a set of *arguments*. A function may be of return type `void` if it does not need a return statement. Otherwise, a function is typed in terms of two characteristics: its *element type* and their *organization type*. For instance,

```
real scalar calcsum(real vector x)
```

declares that the Mata `calcsum` function will return a real scalar. It has one argument: an object `x`, which must be a `real vector`.

Element types may be `real, complex, numeric, string, pointer, transmorphic`. A `transmorphic` object may be filled with any of the other types. A `numeric` object may be either `real` or `complex`. Unlike Stata, Mata supports complex arithmetic.

There are five organization types: `matrix, vector, rowvector, colvector, scalar`. Strictly speaking the latter four are just special cases of `matrix`. In Stata's matrix language, all matrices have two subscripts, neither of which can be zero. In Mata, all but the `scalar` may have zero rows and/or columns. Three- (and higher-) dimension matrices can be implemented by the use of the `pointer` element type, not to be discussed further in this talk.

Arguments, variables and returns

A Mata function definition includes an *argument list*, which may be blank. The names of arguments are required and arguments are positional. The order of arguments in the calling sequence must match that in the Mata function. If the argument list includes a vertical bar (|), following arguments are optional.

Within a function, variables may be explicitly declared (and must be declared if `matastrict` mode is used). It is good programming practice to do so, as then variables cannot be inadvertently misused. Variables within a Mata function have *local scope*, and are not accessible outside the function unless declared as `external`.

A Mata function may only return one item (which could, however, be a multi-element *structure*. If the function is to return multiple objects, Mata's `st_...` functions should be used, as we will demonstrate.

Data access

If you're using Mata functions in conjunction with Stata's ado-file language, one of the most important set of tools are Mata's interface functions: the `st_` functions.

The first category of these functions provide access to data. Stata and Mata have separate workspaces, and these functions allow you to access and update Stata's workspace from inside Mata. For instance, `st_nobs(), st_nvar()` provide the same information as `describe` in Stata, which returns `r(N), r(k)` in its return list. Mata functions `st_data(), st_view()` allow you to access any rectangular subset of Stata's numeric variables, and `st_sdata(), st_sview()` do the same for string variables.

st_view( )

One of the most useful Mata concepts is the *view matrix*, which as its name implies is a view of some of Stata's variables for specified observations, created by a call to `st_view()`. Unlike most Mata functions, `st_view()` does not return a result. It requires three arguments: the name of the view matrix to be created, the observations (rows) that it is to contain, and the variables (columns). An optional fourth argument can specify `touse`: an indicator variable specifying whether each observation is to be included.

```
st_view(x, ., varname, touse)
```

States that the previously-declared Mata vector `x` should be created from all the observations (specified by the missing second argument) of `varname`, as modified by the contents of `touse`. In the Stata code, the `marksample` command imposes any `if` or `in` conditions by setting the indicator variable `touse`.

The Mata statements

```
real matrix Z
st_view(Z=., ., .)
```

will create a view matrix of all observations and all variables in Stata's memory. The missing value (dot) specification indicates that all observations and all variables are included. The syntax `Z=.` specifies that the object is to be created as a void matrix, and then populated with contents. As `Z` is defined as a real matrix, columns associated with any string variables will contain all missing values. `st_sview()` creates a view matrix of string variables.

If we want to specify a subset of variables, we must define a string vector containing their names. For instance, if `varlist` is a `string scalar` argument containing Stata variable names,

```
void foo( string scalar varlist )
...
st_view(X=., ., tokens(varlist), touse)
```

creates matrix `X` containing those variables.

st_data( )

An alternative to view matrices is provided by `st_data()` and `st_sdata()`, which copy data from Stata variables into Mata matrices, vectors or scalars:

```
X = st_data(., .)
```

places a copy of all variables in Stata's memory into matrix `X`. However, this operation requires at least twice as much memory as consumed by the Stata variables, as Mata does not have Stata's full set of 1-, 2-, and 4-byte data types. Thus, although a view matrix can reference any variables currently in Stata's memory with minimal overhead, a matrix created by `st_data()` will consume considerable memory, just as a matrix in Stata's own matrix language does.

Similar to `st_view()`, an optional third argument to `st_data()` can mark out desired observations.

Using views to update Stata variables

A very important aspect of views: using a view matrix rather than copying data into Mata with `st_data()` implies that any changes made to the view matrix will be reflected in Stata's variables' contents. This is a very powerful feature that allows us to easily return information generated in Mata back to Stata's variables, or create new content in existing variables.

This may or may not be what you want to do. Keep in mind that any alterations to a view matrix will change Stata's variables, just as a `replace` command in Stata would. If you want to ensure that Mata computations cannot alter Stata's variables, avoid the use of views, or use them with caution. You may use `st_addvar()` to explicitly create new Stata variables, and `st_store()` to populate their contents.

A Mata function may take one (or several) existing variables and create a transformed variable (or set of variables). To do that with views, create the new variable(s), pass the name(s) as a *newvarlist* and set up a view matrix.

```
st_view(Z=., ., tokens(newvarlist), touse)
```

Then compute the new content as:

```
Z[., .] = result of computation
```

It is very important to use the `[., .]` construct as shown. `Z =` will cause a new matrix to be created and break the link to the view.

You may also create new variables and fill in their contents by combining these techniques:

```
st_view(Z, ., st_addvar(("int", "float"), ///
        ("idnr", "bp")))
Z[., .] = result of computation
```

In this example, we create two new Stata variables, of data type `int` and `float`, respectively, named `idnr` and `bp`.

You may also use *subviews* and, for panel data, *panelsubviews*. We will not discuss those here.

Access to locals, globals, scalars and matrices

You may also want to transfer other objects between the Stata and Mata environments. Although local and global macros, scalars and Stata matrices could be passed in the calling sequence to a Mata function, the function can only return one item. In order to return a number of objects to Stata—for instance, a list of macros, scalars and matrices as is commonly found in the `return list` from an *r*-class program—we use the appropriate *st_functions*.

For local macros,

```
contents = st_local("macname")
st_local("macname", newvalue )
```

The first command will return the contents of Stata local macro *macname*. The second command will create and populate that local macro if it does not exist, or replace the contents if it does, with *newvalue*.

Along the same lines, functions `st_global`, `st_numscalar` and `st_strscalar` may be used to retrieve the contents, create, or replace the contents of global macros, numeric scalars and string scalars, respectively. Function `st_matrix` performs these operations on Stata matrices.

All of these functions can be used to obtain the contents, create or replace the results in `r( )` or `e( )`: Stata's `return list` and `ereturn list`. Functions `st_rclear` and `st_eclear` can be used to delete all entries in those lists. Read-only access to the `c( )` objects is also available.

The `stata( )` function can execute a Stata command from within Mata.

A simple Mata function

We now give a simple illustration of how a Mata subroutine could be used to perform the computations in a do-file. We consider the same routine: an ado-file, `mysum3`, which takes a variable name and accepts optional `if` or `in` qualifiers. Rather than computing statistics in the ado-file, we call the `m_mysum` routine with two arguments: the variable name and the `` `touse' `` indicator variable.

```
program define mysum3, rclass
   version 11
   syntax varlist(max=1) [if] [in]
   return local varname `varlist´
   marksample touse
   mata: m_mysum("`varlist´", "`touse´")
   return scalar N = N
   return scalar sum = sum
   return scalar mean = mu
   return scalar sd = sigma
end
```

In the same ado-file, we include the Mata routine, prefaced by the `mata:` directive. This directive on its own line puts Stata into Mata mode until the `end` statement is encountered. The Mata routine creates a Mata *view* of the variable. A view of the variable is merely a reference to its contents, which need not be copied to Mata's workspace. Note that the contents have been filtered for missing values and those observations specified in the optional `if` or `in` qualifiers.

That view, labeled as `X` in the Mata code, is then a matrix (or, in this case, a column vector) which may be used in various Mata functions that compute the vector's descriptive statistics. The computed results are returned to the ado-file with the `st_numscalar( )` function calls.

```
version 11
mata:
void m_mysum(string scalar vname,
             string scalar touse)

    st_view(X, ., vname, touse)
    mu = mean(X)
    st_numscalar("N", rows(X))
    st_numscalar("mu", mu)
    st_numscalar("sum" ,rows(X) * mu)
    st_numscalar("sigma", sqrt(variance(X)))

end
```

For another example of a Mata function called from an ado-file, imagine that we did not have an easy way of computing the minimum and maximum of the elements of a Stata variable, and wanted to do so with Mata:

```
program varextrema, rclass
   version 11
   syntax varname(numeric) [if] [in]
   marksample touse
   mata: calcextrema( "`varlist´", "`touse´" )
   display as txt " min ( `varlist´ ) = " as res r(min)
   display as txt " max ( `varlist´ ) = " as res r(max)
   return scalar min = r(min)
   return scalar max = r(max)
end
```

Our ado-language code creates a Stata command, `varextrema`, which requires the name of a single numeric Stata variable. You may specify `if` *exp* or `in` *range* conditions. The Mata function `calcextrema` is called with two arguments: the name of the variable and the name of the `touse` temporary variable marking out valid observations. As we will see the Mata function returns its results in two numeric scalars: `r(min)`, `r(max)`. Those are returned in turn to the calling program in the `varextrema` return list.

We then add the Mata function definition to `varextrema.ado`:

```
version 11
mata:
mata set matastrict on
void calcextrema(string scalar varname, ///
                 string scalar touse)

  real colvector x, cmm
  st_view(x, ., varname, touse)
  cmm = colminmax(x)
  st_numscalar("r(min)", cmm[1])
  st_numscalar("r(max)", cmm[2])

end
```

The Mata code as shown is `strict`: all objects must be defined. The function is declared `void` as it does not return a result. A Mata function could return a single result to Mata, but we need to send two results back to Stata. The input arguments are declared as `string scalar` as they are variable names.

We create a *view matrix*, colvector `x`, as the subset of *varname* for which `touse==1`. Mata's `colminmax()` function computes the extrema of its arguments as a two-element vector, and `st_numscalar()` returns each of them to Stata as `r(min)`, `r(max)` respectively.

A multi-variable function

Now let's consider a slightly more ambitious task. Say that you would like to *center* a number of variables on their means, creating a new set of transformed variables. Surprisingly, official Stata does not have such a command, although Ben Jann's `center` command does so. Accordingly, we write Stata command `centervars`, employing a Mata function to do the work.

The Stata code:

```
program centervars, rclass
   version 11
   syntax varlist(numeric) [if] [in], ///
             GENerate(string) [DOUBLE]
   marksample touse
   quietly count if `touse´
   if `r(N)´ == 0  error 2000
   foreach v of local varlist
       confirm new var `generate´`v´

   foreach v of local varlist
       qui generate `double´ `generate´`v´ = .
       local newvars "`newvars´ `generate´`v´"

   mata: centerv( "`varlist´", "`newvars´", "`touse´" )
end
```

The file `centervars.ado` contains a Stata command, `centervars`, that takes a list of numeric variables and a mandatory `generate()` option. The contents of that option are used to create new variable names, which then are tested for validity with `confirm new var`, and if valid generated as missing. The list of those new variables is assembled in local macro `newvars`. The original `varlist` and the list of `newvars` is passed to the Mata function `centerv()` along with `touse`, the temporary variable that marks out the desired observations.

The Mata code:

```
version 11
mata:
void centerv( string scalar varlist, ///
              string scalar newvarlist,
              string scalar touse)

  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ ., . ] = X :- mean(X)

end
```

In the Mata function, `tokens( )` extracts the variable names from *varlist* and places them in a string rowvector, the form needed by `st_view`. The `st_view` function then creates a *view matrix*, `X`, containing those variables and the observations selected by `if` and `in` conditions.

The view matrix allows us to both access the variables' contents, as stored in Mata matrix `X`, but also to *modify* those contents. The colon operator (`:-`) subtracts the vector of column means of `X` from the data. Using the `Z[,]=` notation, the Stata variables themselves are modified. When the Mata function returns to Stata, the contents and descriptive statistics of the variables in *varlist* will be altered.

One of the advantages of Mata use is evident here: we need not loop over the variables in order to demean them, as the operation can be written in terms of matrices, and the computation done very efficiently even if there are many variables and observations. Also note that performing these calculations in Mata incurs minimal overhead, as the matrix `Z` is merely a view on the Stata variables in `newvars`. One caveat: Mata's `mean()` function performs *listwise deletion*, like Stata's `correlate` command.

Passing a function to Mata

Let's consider adding a feature to `centervars`: the ability to transform variables before centering with one of several mathematical functions (`abs(), exp(), log(), sqrt()`). The user will provide the name of the desired transformation, which defaults to the identity transformation, and Stata will pass the name of the function (actually, a pointer to the function) to Mata. We call this new command `centertrans`.

The Stata code:

```
program centertrans, rclass
   version 11
   syntax varlist(numeric) [if] [in], ///
             GENerate(string) [TRans(string)] [DOUBLE]
   marksample touse
   quietly count if `touse'
   if `r(N)' == 0  error 2000
   foreach v of local varlist {
       confirm new var `generate'`v'
   }
   local trops abs exp log sqrt
   if "`trans'" == "" {
     local trfn  "mf_iden"
    }
    else {
     local ntr : list posof "`trans'" in trops
     if !`ntr' {
     display as err "Error: trans must be chosen from `trops'"
     error 198
     }
    local trfn : "mf_`trans'"
    }
   foreach v of local varlist {
       qui generate `double' `generate'`trans'`v' = .
       local newvars "`newvars' `generate'`trans'`v'"
   }
   mata: centertrans( "`varlist'", "`newvars'", &`trfn'(), "`touse'" )
end
```

In Mata, we must define “wrapper functions" for the transformations, as we cannot pass a pointer to a built-in function. We define trivial functions such as

```
function mf_log(x) return(log(x))
```

which defines the `mf_log()` scalar function as taking the log of its argument.

The Mata function `centertrans()` receives the function argument as

```
pointer(real scalar function) scalar f
```

To apply the function, we use

```
Z[ ., . ] = (*f)(X)
```

which applies the function referenced by `f` to the elements of the matrix `X`.

The Mata code:

```
version 11
mata:
function mf_abs(x) return(abs(x))
function mf_exp(x) return(exp(x))
function mf_log(x) return(log(x))
function mf_sqrt(x) return(sqrt(x))
function mf_iden(x) return(x)

void centertrans( string scalar varlist, ///
                  string scalar newvarlist,
                  pointer(real scalar function) scalar f,
                  string scalar touse)

  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ , ] = (*f)(X)
  Z[ , ] = Z :- mean(Z)

end
```

A Mata-based estimation routine

Mata may prove particularly useful when you have an algorithm readily expressed in matrix form. Many estimation problems fall into that category. In this last example, I illustrate how "heteroskedastic OLS" (HOLS) can be easily implemented in Mata, with Stata code handling all of the housekeeping details. This section draws on joint work with Mark Schaffer.

HOLS is a form of Generalised Method of Moments (GMM) estimation in which you assert not only that the regressors X are uncorrelated with the error, but that you also have additional variables Z which are also uncorrelated with the error. Those additional "orthogonality conditions" serve to improve the efficiency of estimation when non-*i.i.d.* errors are encountered. This estimator is described in `help ivreg2` and in Baum, Schaffer & Stillman, *Stata Journal*, 2007.

A particularly important feature added to Mata in Stata version 10 is the suite of `optimize()` commands. These commands permit you to define your own optimization routine in Mata and direct its use. The routine need not be a maximum-likelihood nor nonlinear least squares routine, but rather any well-defined objective function that you wish to minimize or maximise.

Just as with `ml`, you may write a `d0`, `d1` or `d2` routine, requiring zero, first or first and second analytic derivatives in terms of the gradient vector and Hessian matrix. For ease of use in statistical applications, you may also construct a `v0`, `v1` or `v2` routine in terms of the score vector and Hessian matrix. For the first time, Stata provides a non-classical optimization method, Nelder–Mead simplex, in addition to the classical techniques available elsewhere in Stata.

The `hols` command takes a dependent variable and a set of regressors. The `exog()` option may be used to provide the names of additional variables uncorrelated with the error. By default, `hols` calculates estimates under the assumption of *i.i.d*. errors. If the `robust` option is used, the estimates' standard errors are robust to arbitrary heteroskedasticity.

Following estimation, the `estimates post` and `estimates display` commands are used to provide standard Stata estimation output. If the `exog` option is used, a Sargan–Hansen *J* test statistic is provided. A significant value of the *J* statistic implies rejection of the null hypothesis of orthogonality.

The Stata code:

```
program hols, eclass
version 11
syntax varlist [if] [in] [, exog(varlist) robust ]
local depvar: word 1 of `varlist'
local regs: list varlist - depvar
marksample touse
markout `touse'  `exog'
tempname b V
mata: m_hols("`depvar'",  "`regs'", "`exog'", "`touse'", "`robust'")
mat `b' = r(beta)
mat `V' = r(V)
local vnames  `regs' _cons
matname `V' `vnames'
matname `b' `vnames', c(.)
local N = r(N)
ereturn post `b' `V', depname(`depvar') obs(`N') esample(`touse')
ereturn local depvar = "`depvar'"
ereturn scalar N = r(N)
ereturn scalar j = r(j)
ereturn scalar L = r(L)
ereturn scalar K = r(K)
if "`robust'" != "" {
    ereturn local vcetype "Robust"
}
local res = cond("`exog'" != "", "Heteroskedastic", "")
display _newline "`res' OLS results" _col(60) "Number of obs = " e(N)
ereturn display
display "Sargan-Hansen J statistic: " %7.3f e(j)
if ( e(L)-e(K) > 0 ) {
display "Chi-sq(" %3.0f e(L)-e(K) "  )       P-val = " ///
    %5.4f chiprob(e(L)-e(K), e(j)) _newline
}
end
```

The Mata code makes use of a function to compute the covariance matrix: either the classical, non-robust *VCE* or the heteroskedasticity-robust *VCE*. For ease of reuse, this logic is broken out into a standalone function.

```
version 11
mata:
real matrix m_myomega(real rowvector beta,
                      real colvector Y,
                      real matrix X,
                      real matrix Z,
                      string scalar robust)
{
        real matrix QZZ, omega
        real vector e, e2
        real scalar N, sigma2
// Calculate residuals from the coefficient estimates
N = rows(Z)
e = Y - X * beta'
if (robust=="") {
// Compute classical, non-robust covariance matrix
    QZZ = 1/N * quadcross(Z, Z)
    sigma2 = 1/N * quadcross(e, e)
    omega = sigma2 * QZZ
}
else {
// Compute heteroskedasticity-consistent covariance matrix
    e2 = e:^2
    omega = 1/N * quadcross(Z, e2, Z)
}
_makesymmetric(omega)
return (omega)
}
```

The main Mata code takes as arguments the dependent variable name, the list of regressors, the optional list of additional exogenous variables, the marksample indicator (`touse`) and the `robust` flag. The logic for linear GMM can be expressed purely in terms of matrix algebra.

The Mata code:

```
void m_hols(string scalar yname,
                string scalar inexognames,
                string scalar exexognames,
                string scalar touse,
                string scalar robust)
{
real matrix Y, X2, Z1, X, Z, QZZ, QZX, W, omega, V
real vector cons, beta_iv, beta_gmm, e, gbar
real scalar K, L, N, j
st_view(Y, ., tokens(yname), touse)
st_view(X2, ., tokens(inexognames), touse)
st_view(Z1, ., tokens(exexognames), touse)
// Constant is added by default.
cons = J(rows(X2), 1, 1)
X = X2, cons
Z = Z1, X
K = cols(X)
L = cols(Z)
N = rows(Y)
QZZ = 1/N * quadcross(Z, Z)
QZX = 1/N * quadcross(Z, X)
// First step of 2-step feasible efficient GMM.  Weighting matrix = inv(Z'Z)
W = invsym(QZZ)
beta_iv = (invsym(X'Z * W * Z'X) * X'Z * W * Z'Y)'
```

Mata code, continued...

```
// Use first-step residuals to calculate optimal weighting matrix for 2-step FEGMM
omega = m_myomega(beta_iv, Y, X, Z, robust)
// Second step of 2-step feasible efficient GMM
W = invsym(omega)
beta_gmm = (invsym(X'Z * W * Z'X) * X'Z * W * Z'Y)'
// Sargan-Hansen J statistic: first we calculate the second-step residuals
e = Y - X * beta_gmm'
// Calculate gbar = 1/N * Z'*e
gbar = 1/N * quadcross(Z, e)
j = N * gbar' * W * gbar
// Sandwich var-cov matrix (no finite-sample correction)
// Reduces to classical var-cov matrix if Omega is not robust form.
// But the GMM estimator is "root-N consistent", and technically we do
// inference on sqrt(N)*beta.  By convention we work with beta, so we adjust
// the var-cov matrix instead:
V = 1/N * invsym(QZX' * W * QZX)
// Return results to Stata as r-class macros.
st_matrix("r(beta)", beta_gmm)
st_matrix("r(V)", V)
st_numscalar("r(j)", j)
st_numscalar("r(N)", N)
st_numscalar("r(L)", L)
st_numscalar("r(K)", K)
}
end
```

Comparison of HOLS estimation results

```
. hols price mpg headroom, robust
OLS results                                     Number of obs = 74
```

price	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
mpg	-259.1057	66.15838	-3.92	0.000	-388.7737	-129.4376
headroom	-334.0215	314.9933	-1.06	0.289	-951.3971	283.3541
_cons	12683.31	2163.351	5.86	0.000	8443.224	16923.4

```
Sargan-Hansen J statistic:   0.000
. hols price mpg headroom, exog(trunk displacement weight) robust
Heteroskedastic OLS results                     Number of obs = 74
```

price	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
mpg	-287.4003	60.375	-4.76	0.000	-405.7331	-169.0675
headroom	-367.0973	313.6646	-1.17	0.242	-981.8687	247.6741
_cons	13136.54	2067.6	6.35	0.000	9084.117	17188.96

```
Sargan-Hansen J statistic:   4.795
Chi-sq(  3  )        P-val = 0.1874
```

As shown, this user-written estimation command can take advantage of all of the features of official estimation commands. There is even greater potential for using Mata with nonlinear estimation problems, as its new `optimize()` suite of commands allows easy access to an expanded set of optimization routines. For more information, see Austin Nichols' talk on *GMM estimation in Mata* from Summer NASUG 2008 at `http://ideas.repec.org`.

moremata

If you're serious about using Mata, you should familiarize yourself with Ben Jann's `moremata` package, available from SSC. The package contains a function library, `lmoremata`, as well as full documentation of all included routines (in the same style as Mata's on-line function descriptions).

Routines in `moremata` currently include kernel functions; statistical functions for quantiles, ranks, frequencies, means, variances and correlations; functions for sampling; density and distribution functions; root finders; matrix utility and manipulation functions; string functions; and input-output functions. Many of these functions provide functionality as yet missing from official Mata, and ease the task of various programming chores.

In summary, then, it should be apparent that gaining some familiarity with Mata will expand your horizons as a Stata programmer. Mata may be used effectively in either its interactive or function mode as an efficient adjunct to Stata's traditional command-line interface. We have not illustrated its usefulness for text processing problems (such as developing a concordance of words in a manuscript) but it could be fruitfully applied to such tasks as well.

Example: Finding nearest neighbors

As an example of Mata programming, consider the question of how to find "nearest neighbors" in geographical terms: that is, which observations are spatially proximate to each observation in the dataset? This can be generalized to a broader problem: which observations are *closest* in terms of similarity of a number of variables? This might be recognized as a problem of calculating a *propensity score* (see Leuven and Sianesi's `psmatch2` on the SSC Archive) but we would like to approach it from first principles with a Mata routine.

We allow a match to be defined in terms of a set of variables on which a close match will be defined. The quality of the match can then be evaluated by calculating the correlation between the original variable's observations and its values of the identified "nearest neighbor." That is, if we consider two units (patients, cities, firms, households) with similar values of $x_1, \ldots, x_m$, how highly correlated are their values of y?

Although the original example is geographical, the underlying task is found in many disciplines where a control group of observations is to be identified, each of which is the closest match to one of the observations of interest. For instance, in finance, you may have a sample of firms that underwent a takeover. For each firm, you would like to find a “similar” firm (based on several characteristics) that did not undergo a takeover. Those pairs of firms are nearest neighbors. In our application, we will compute the Euclidian distance between the standardized values of pairs of observations.

To implement the solution, we first construct a Stata ado-file defining program `nneighbor` which takes a *varlist* of one or more measures that are to be used in the match. In our application, we may use any number of variables as the basis for defining the nearest neighbor. The user must specify `y`, a response variable; `matchobs`, a variable to hold the observation numbers of the nearest neighbor; and `matchval`, a variable to hold the values of `y` belonging to the nearest neighbor.

After validating any `if` *exp* or `in` *range* conditions with `marksample`, the program confirms that the two new variable names are valid, then generates those variables with missing values. The latter step is necessary as we construct view matrices in the Mata function related to those variables, which must already exist. We then call the Mata function, `mf_nneighbor()`, and compute one statistic from its results: the correlation between the `y()` variable and the `matchvals()` variable, measuring the similarity of these `y()` values between the observations and their nearest neighbors.

```
. type nneighbor.ado
*! nneighbor 1.0.1  CFBaum 11aug2008
program nneighbor
        version 11
        syntax varlist(numeric) [if] [in], ///
        Y(varname numeric) MATCHOBS(string) MATCHVAL(string)
        marksample touse
        qui count if `touse´
        if r(N) == 0 {
                error 2000
        }
// validate new variable names
        confirm new variable `matchobs´
        confirm new variable `matchval´
        qui     generate long `matchobs´ = .
        qui generate `matchval´ = .
        mata: mf_nneighbor("`varlist´", "`matchobs´", "`y´", ///
                "`matchval´", "`touse´")
        summarize `y´ if `touse´, meanonly
        display _n "Nearest neighbors for `r(N)´ observations of `y´"
        display    "Based on L2-norm of standardized vars: `varlist´"
        display    "Matched observation numbers: `matchobs´"
        display    "Matched values: `matchval´"
        qui correlate `y´ `matchval´ if `touse´
        display    "Correlation[ `y´, `matchval´ ] = " %5.4f `r(rho)´
end
```

We now construct the Mata function. The function uses a view on the `varlist`, constructing view matrix `X`. As the scale of those variables affects the Euclidian distance (L2-norm) calculation, the variables are standardized in matrix `Z` using Ben Jann's `mm_meancolvar()` function from the `moremata` package on the SSC Archive. Views are then established for the `matchobs` variable (`C`), the response variable (`y`) and the `matchvals` variable (`ystar`).

For each observation and variable in the normalized *varlist*, the L2-norm of distances between that observation and the entire vector is computed as `d`. The heart of the function is the call to `minindex()`. This function is a fast, efficient calculator of the minimum values of a variable. Its fourth argument can deal with ties; for simplicity we do not handle ties here. We request the closest two values, in terms of the distance `d`, to each observation, recognizing that each observation is its own nearest neighbor. The observation numbers of the two nearest neighbors are stored in vector `ind`. Therefore, the observation number desired is the second element of the vector, and `y[ind[2]]` is the value of the nearest neighbor's response variable. Those elements are stored in `C[i]` and `ystar[i]`, respectively.

```
 . type mf_nneighbor.mata
mata: mata clear
mata: mata set matastrict on
version 11
mata:
// mf_nneighbor 1.0.0  CFBaum 11aug2008
void function mf_nneighbor(string scalar matchvars,
                           string scalar closest,
                           string scalar response,
                           string scalar match,
                           string scalar touse)
{
        real matrix X, Z, mc, C, y, ystar
        real colvector ind
        real colvector w
        real colvector d
        real scalar n, k, i, j
        string rowvector vars, v
        st_view(X, ., tokens(matchvars), touse)
// standardize matchvars with mm_meancolvar from moremata
        mc = mm_meancolvar(X)
        Z = ( X :- mc[1, .]) :/ sqrt( mc[2, .])
        n = rows(X)
        k = cols(X)
        st_view(C, ., closest, touse)
        st_view(y, ., response, touse)
        st_view(ystar, ., match, touse)
```

```
 (continued)
// loop over observations
        for(i = 1; i <= n; i++) {
// loop over matchvars
            d = J(n, 1, 0)
            for(j = 1; j <= k; j++) {
                d = d + ( Z[., j] :- Z[i, j] ) :^2
            }
        minindex(d, 2, ind, w)
        C[i] = ind[2]
        ystar[i] = y[ind[2]]
        }
}
end
```

We now can try out the routine. We employ the `usairquality` dataset used in earlier examples. It contains statistics for 41 U.S. cities' air quality (`so2`, or sulphur dioxide concentration) as well as several demographic factors. To test our routine, we first apply it to a single variable: population (`pop`). Examining the result, we can see that it is properly selecting the city with the closest population value as the nearest neighbor:

```
. use usairquality, clear
. sort pop
. nneighbor pop, y(so2) matchobs(mo1) matchval(mv1)
Nearest neighbors for 41 observations of so2
Based on L2-norm of standardized vars: pop
Matched observation numbers: mo1
Matched values: mv1
Correlation[ so2, mv1 ] = 0.0700
. list pop mo1 so2 mv1, sep(0)
```

	pop	mo1	so2	mv1
1.	71	2	31	36
2.	80	1	36	31
3.	116	4	46	13
4.	132	3	13	46
5.	158	6	56	28
6.	176	7	28	94
7.	179	6	94	28
8.	201	7	17	94
9.	244	10	11	8
10.	277	11	8	26
11.	299	12	26	31
12.	308	11	31	26
13.	335	14	10	14
14.	347	13	14	10
15.	361	14	9	14
16.	448	17	18	23
17.	453	16	23	18
19.	497	20	24	14

We must note, however, that the response variable's values are very weakly correlated with those of the `matchvar`. Matching cities on the basis of one attribute does not seem to imply that they will have similar values of air pollution. We thus exercise the routine on two broader sets of attributes: one adding `temp` and `wind`, and the second adding `precip` and `days`, where `days` measures the mean number of days with poor air quality.

```
. nneighbor pop temp wind, y(so2) matchobs(mo3) matchval(mv3)
Nearest neighbors for 41 observations of so2
Based on L2-norm of standardized vars: pop temp wind
Matched observation numbers: mo3
Matched values: mv3
Correlation[ so2, mv3 ] = 0.1769
. nneighbor pop temp wind precip days, y(so2) matchobs(mo5) matchval(mv5)
Nearest neighbors for 41 observations of so2
Based on L2-norm of standardized vars: pop temp wind precip days
Matched observation numbers: mo5
Matched values: mv5
Correlation[ so2, mv5 ] = 0.5286
```

We see that with the broader set of five attributes on which matching is based, there is a much higher correlation between the `so2` values for each city and those for its nearest neighbor.

Workshop: delegates' programming needs and solutions

In this last session, I encourage each of you to specify a statistical or data management problem for which you need programming assistance. We will develop possible solutions to these problems, emphasizing the use of programming principles discussed in the course.