Ado-file and Mata programming: Useful skills for many researchers

Christopher F Baum

Boston College

Canadian Stata Conference 2025

What level of Stata programming skill makes sense for you?

How advantageous might it be to acquire additional Stata programming skills? First, some nomenclature related to programming:

- You *are* 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.
- You can use Stata's programming language, *Mata*, to write routines in that language that are called by do-files or ado-files.

What level of Stata programming skill makes sense for you?

How advantageous might it be to acquire additional Stata programming skills? First, some nomenclature related to programming:

- You *are* 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.
- You can use Stata's programming language, *Mata*, to write routines in that language that are called by do-files or ado-files.

What level of Stata programming skill makes sense for you?

How advantageous might it be to acquire additional Stata programming skills? First, some nomenclature related to programming:

- You *are* 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.
- You can use Stata's programming language, *Mata*, to write routines in that language that are called by do-files or ado-files.

What level of Stata programming skill makes sense for you?

How advantageous might it be to acquire additional Stata programming skills? First, some nomenclature related to programming:

- You *are* 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.
- You can use Stata's programming language, *Mata*, to write routines in that language that are called by do-files or 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.

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.

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.

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.

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 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 or 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 community-contributed 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` and `frames`, you may recognize that your needs can be readily met.

The second bit of advice along those lines: use Stata's `search` command 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!

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, 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, tabulation 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.

A huge benefit accrues to the ado-file author: few data management, statistical, tabulation 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.

Context 3: Mata subroutines for do-files and ado-files

Your do-files or 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 do-file or 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.

Although I will not discuss it in this talk, this full integration is also available in recent versions of Stata for the Python language. `help python` for full details.

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.

Although I will not discuss it in this talk, this full integration is also available in recent versions of Stata for the Python language. `help python` for full details.

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.

Extensibility of official Stata

An advantage of the command-line driven environment involves *extensibility*: the continual expansion of Stata's capabilities. A *command*, to Stata, is a verb instructing the program to perform some action.

Commands can be "built in" commands—those elements so frequently used that they have been coded into the "Stata kernel." A relatively small fraction of the total number of official Stata commands are built in, but they are used very heavily.

Extensibility of official Stata

An advantage of the command-line driven environment involves *extensibility*: the continual expansion of Stata's capabilities. A *command*, to Stata, is a verb instructing the program to perform some action.

Commands can be "built in" commands—those elements so frequently used that they have been coded into the "Stata kernel." A relatively small fraction of the total number of official Stata commands are built in, but they are used very heavily.

The vast majority of Stata commands are written in Stata's own programming language–the "ado-file" language. If a command is not built in to the Stata kernel, Stata searches for it along the `adopath`. Like the `PATH` in Unix, Linux or DOS, the `adopath` indicates the several directories in which an ado-file might be located. This implies that the "official" Stata commands are not limited to those coded into the kernel.

The importance of this program design goes far beyond the limits of official Stata. Since the adopath includes both Stata directories and other directories on your hard disk (or on a server's filesystem), you can acquire new Stata commands from a number of web sites. The *Stata Journal (SJ)*, a quarterly peer-reviewed journal, is the primary method for distributing user contributions. Between 1991 and 2001, the *Stata Technical Bulletin* played this role.

The *SJ* is a subscription publication (articles more than three years old freely downloadable), but the `ado`- and `sthlp`-files may be freely downloaded from Stata's web site. The Stata `help` command accesses help on all installed commands; the Stata `search` command will locate commands that have been documented in the *STB* and the *SJ*, and with one click you may install them in your version of Stata. Help for these commands will then be available in your own copy.

User extensibility: the SSC archive

But this is only the beginning. Stata users worldwide participate in the *Statalist* forum, and when a user has written and documented a new general-purpose command to extend Stata functionality, they announce it on *Statalist*, to which you may freely subscribe: see Stata's web site.

Since September 1997, all items posted to `Statalist` (over 3,000) have been placed in the Boston College Statistical Software Components (SSC) Archive in *RePEc* (Research Papers in Economics), available from IDEAS (`http://ideas.repec.org`) and EconPapers (`http://econpapers.repec.org`).

User extensibility: the SSC archive

But this is only the beginning. Stata users worldwide participate in the *Statalist* forum, and when a user has written and documented a new general-purpose command to extend Stata functionality, they announce it on *Statalist*, to which you may freely subscribe: see Stata's web site.

Since September 1997, all items posted to `Statalist` (over 3,000) have been placed in the Boston College Statistical Software Components (SSC) Archive in *RePEc* (Research Papers in Economics), available from IDEAS (`http://ideas.repec.org`) and EconPapers (`http://econpapers.repec.org`).

Any component in the SSC archive can be readily inspected with a web browser, using IDEAS' or EconPapers' search functions, and if desired you may install it with one command from the archive from within Stata.

For instance, if you know there is a module in the archive named `mvsumm`, you could use `ssc describe mvsumm` to learn more about it, and `ssc install mvsumm` to install it if you wish. Anything in the archive can be accessed via Stata's `ssc` command: thus `ssc describe mvsumm` will locate this module, and make it possible to install it with one click.

Any component in the SSC archive can be readily inspected with a web browser, using IDEAS' or EconPapers' search functions, and if desired you may install it with one command from the archive from within Stata.

For instance, if you know there is a module in the archive named `mvsumm`, you could use `ssc describe mvsumm` to learn more about it, and `ssc install mvsumm` to install it if you wish. Anything in the archive can be accessed via Stata's `ssc` command: thus `ssc describe mvsumm` will locate this module, and make it possible to install it with one click.

The command `ssc new` lists, in the Stata Viewer, all SSC packages that have been added or modified in the last month. You may click on their names for full details. The command `ssc hot` reports on the most popular packages on the SSC Archive.

The Stata command `ado update` checks to see whether all packages you have downloaded and installed from the SSC archive, the *Stata Journal*, or other user-maintained `net from...` sites are up to date.

`ado update` alone will provide a list of packages that have been updated. You can then use `ado update, update` to refresh your copies of those packages, or specify which packages are to be updated.

The command `ssc new` lists, in the Stata Viewer, all SSC packages that have been added or modified in the last month. You may click on their names for full details. The command `ssc hot` reports on the most popular packages on the SSC Archive.

The Stata command `ado update` checks to see whether all packages you have downloaded and installed from the SSC archive, the *Stata Journal*, or other user-maintained `net from...` sites are up to date.

`ado update` alone will provide a list of packages that have been updated. You can then use `ado update, update` to refresh your copies of those packages, or specify which packages are to be updated.

The command `ssc new` lists, in the Stata Viewer, all SSC packages that have been added or modified in the last month. You may click on their names for full details. The command `ssc hot` reports on the most popular packages on the SSC Archive.

The Stata command `ado update` checks to see whether all packages you have downloaded and installed from the SSC archive, the *Stata Journal*, or other user-maintained `net from...` sites are up to date.

`ado update` alone will provide a list of packages that have been updated. You can then use `ado update, update` to refresh your copies of those packages, or specify which packages are to be updated.

Ado-file programming: structure of an ado-file

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 Stata program begins with the `program define` *progname* statement, which usually includes the option `, rclass`, and a `version 19` statement. The *progname* must not be the same as any Stata command, nor for safety's sake the same as any accessible user-written command. If `search progname` does not turn up anything, you can use that name.

Programs (and Stata commands) are generally 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 distinction: *r-class* programs return results in `r()`, while *e-class* programs return results in `e()`. They also involve many requirements in order to play the role of an estimation program and permit post-estimation commands.

Strictly speaking, programs can also be declared *s-class*, returning results in `s()`, or not declared at all, in which case they are *n-class*. If not declared, programs cannot use `return`, `ereturn` or `sreturn`.

Programs (and Stata commands) are generally 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 distinction: *r-class* programs return results in `r()`, while *e-class* programs return results in `e()`. They also involve many requirements in order to play the role of an estimation program and permit post-estimation commands.

Strictly speaking, programs can also be declared *s-class*, returning results in `s()`, or not declared at all, in which case they are *n-class*. If not declared, programs cannot use `return`, `ereturn` or `sreturn`.

Programs (and Stata commands) are generally 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 distinction: *r-class* programs return results in `r()`, while *e-class* programs return results in `e()`. They also involve many requirements in order to play the role of an estimation program and permit post-estimation commands.

Strictly speaking, programs can also be declared *s-class*, returning results in `s()`, or not declared at all, in which case they are *n-class*. If not declared, programs cannot use `return`, `ereturn` or `sreturn`.

A program can receive positional arguments, which define a sequence of local macros depending on their order:

```
l . program stcty
  1. loc state = "`1´"
  2. loc county = "`2´"
  3. display _n  "`county´ county is located in the state of `state´"
  4. end
.         sjlog close, replace
. stcty Massachusetts Suffolk
Suffolk county is located in the state of Massachusetts
. stcty Michigan Emmet
Emmet county is located in the state of Michigan
. stcty Illinois Cook
Cook county is located in the state of Illinois
.         sjlog close, replace
```

The syntax statement

More commonly, the `syntax` statement is 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 or factor variables 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 only refer to *new* variables.

Including a subset of observations

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

Including a subset of observations

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

Using program 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.

Temporary variables and tempnames

Within your own command, you do not want to reuse the names of existing variables or matrices. You should 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).

Temporary variables and tempnames

Within your own command, you do not want to reuse the names of existing variables or matrices. You should 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).

The return statement

So after doing whatever computations or manipulations you need within your program, how do you return its results? You can 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 can 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.

The return statement

So after doing whatever computations or manipulations you need within your program, how do you return its results? You can 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 can 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 18
syntax varname
return local varname `varlist´
tempvar new
quietly {
count if !mi(`varlist´)
return scalar N = r(N)
gen double `new´ = sum(`varlist´)
return scalar sum = `new´[_N]
return scalar mean = return(sum)/return(N)
}
end
```

A shortcut for a program allowing only one variable is to specify `syntax varname`. Despite that, the local macro returned is still `varlist`.

A sample program from `help return`:

```
program define mysum, rclass
version 18
syntax varname
return local varname `varlist´
tempvar new
quietly {
count if !mi(`varlist´)
return scalar N = r(N)
gen double `new´ = sum(`varlist´)
return scalar sum = `new´[_N]
return scalar mean = return(sum)/return(N)
}
end
```

A shortcut for a program allowing only one variable is to specify `syntax varname`. Despite that, the local macro returned is still `varlist`.

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:

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 18
syntax varname(numeric) [if] [in]
return local varname `varlist´
tempvar new
marksample touse
sort `touse´
quietly {
count if !mi(`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
```

Specifying `numeric` ensures that a numeric variable is provided. The `sort` command ensures that the observations selected by `if` and `in` qualifiers will appear at the end of the data in memory.

```
program define mysum2, rclass
version 18
syntax varname(numeric) [if] [in]
return local varname `varlist´
tempvar new
marksample touse
sort `touse´
quietly {
count if !mi(`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
```

Specifying `numeric` ensures that a numeric variable is provided. The `sort` command ensures that the observations selected by `if` and `in` qualifiers will appear at the end of the data in memory.

A second example of ado-file programming

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.

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 11
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
.............................................
```

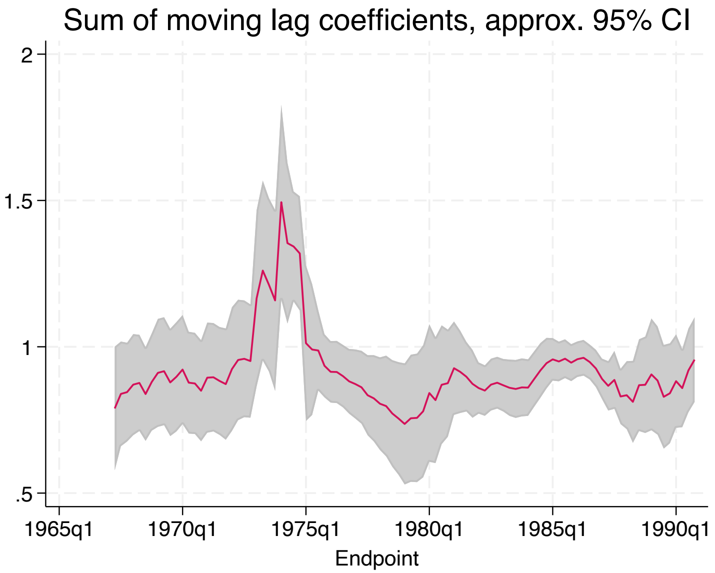
We can graph the resulting series and its approximate 95% standard error bands with `twoway rarea` and `tsline`:

```
 . 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) ylabel(,angle(0)) ///
> title("Sum of moving lag coefficients, approx. 95% CI") ///
> ||  tsline sum, legend(off)
```

```
 . 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 can 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) ylabel(,angle(0)) ///
> title("Sum of moving lag coefficients, approx. 95% CI") ///
> ||  tsline sum, legend(off)
```


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

A third example of ado-file programming

Let's say that you want to compute several statistics from the percentiles of a continuous variable. Researchers often use the interquartile range, (p75 $-$ p25), as an alternative to the standard deviation as a measure of a variable's spread. Those concerned with income distributions often use (p90 $-$ p10) as a measure of inequality. If we are concerned about outliers, we might compute (p95 $-$ p5) or (p99 $-$ p1) to compare the variable's range $(x_{max} - x_{min})$ with these percentile ranges.

Computing these percentile ranges in a do-file is easy enough. You merely need to use `summarize, detail` and access the appropriate percentiles in its stored results. But you might like to have a program that would calculate the ranges from these percentiles and make them available for later use. What must you do to write one? The first step is to choose a name. Here is our first crack at the code.

```
 . type pctrange.ado
*! pctrange v1.0.0  CFBaum 06feb2014
program pctrange
        version 13.1
        syntax varlist(max=1 numeric)
        quietly summarize `varlist´, detail
        scalar range = r(max) - r(min)
        scalar p7525 = r(p75) - r(p25)
        scalar p9010 = r(p90) - r(p10)
        scalar p9505 = r(p95) - r(p5)
        scalar p9901 = r(p99) - r(p1)
        display as result _n "Percentile ranges for `varlist´"
        display as txt "75-25: " p7525
        display as txt "90-10: " p9010
        display as txt "95-05: " p9505
        display as txt "99-01: " p9901
        display as txt "Range: " range
end
```

A statistical command should accept `if` *exp* and `in` *range* qualifiers if it is to be useful. Very little work is needed to add these features to our program. The definition of `if` *exp* and `in` *range* qualifiers and program options is all handled by the `syntax` statement. In the improved program, `[if]` and `[in]` denote that each of these qualifiers can be used.

The `marksample touse` command uses the information provided in a `if` *exp* or `in` *range* qualifier operative if one or both were given on the command line. The `marksample` command marks those observations which should enter the computations in an indicator variable, `` `touse' ``, equal to 1 for the desired observations, and 0 otherwise. The `` `touse' `` variable is a temporary variable, which will disappear when the ado-file ends, like a local macro.

After defining this indicator variable, we use `count if `touse'` to calculate the number of observations after applying the qualifiers and display an error if there are no observations. We must add `if `touse'` to each statement in the program which works with the input varlist. In this case, we need only modify the `summarize` statement to include `if `touse'`. The new version of the program:

A third example of ado-file programming

```
 *! pctrange v1.0.5  CFBaum 06feb2014
program pctrange, rclass
        version 13.1
        syntax varlist(max=1 numeric) [if] [in] [, noPRINT]
        marksample touse
        quietly count if `touse´
        if `r(N)´ == 0 {
                error 2000
        }
        local res range p7525 p9010 p9505 p9901
        tempname `res´
        quietly summarize `varlist´ if `touse´, detail
        scalar `range´ = r(max) - r(min)
        scalar `p7525´ = r(p75) - r(p25)
        scalar `p9010´ = r(p90) - r(p10)
        scalar `p9505´ = r(p95) - r(p5)
        scalar `p9901´ = r(p99) - r(p1)
        if "`print´" != "noprint" {
                display as result _n "Percentile ranges for `varlist´, N = `r(N)´"
                display as txt "75-25: " `p7525´
                display as txt "90-10: " `p9010´
                display as txt "95-05: " `p9505´
                display as txt "99-01: " `p9901´
                display as txt "Range: " `range´
        }
        foreach r of local res {
                return scalar `r´ = ``r´´
        }
        return scalar N = r(N)
```

Generalizing commands to handle multiple variables

It would be really handy to run `pctrange` for a number of variables with a single command. You could always loop over those variables with a `foreach` loop, but assembling the output afterward might be a bit of work. As the program produces five statistics for each variable, perhaps a nicely-formatted table would be useful—and that will require some rethinking about how the command's results are to be displayed and returned.

First, you must tell the `syntax` statement that more than one numeric variable is allowed. The program will perform as it does now for a single variable or produce a table if given several variables. Because we are constructing a table, a Stata matrix is a useful device to store the results we generate from `summarize`.

Rather than placing the elements in scalars, we declare a matrix with the `J()` function, calculating the number of rows needed with the macro extended function `word count` *string*. The `foreach` loop then cycles through the varlist, placing the percentile ranges for each variable into one row of the matrix. The local macro `rown` is used to build up the list of row names, applied with `matrix rownames`.

The one trick needed here appears in the `return` statement when only one variable is provided. We must doubly dereference `r`: once to specify the statistic being evaluated, and a second time to retrieve its content.

Rather than placing the elements in scalars, we declare a matrix with the `J()` function, calculating the number of rows needed with the macro extended function `word count` *string*. The `foreach` loop then cycles through the varlist, placing the percentile ranges for each variable into one row of the matrix. The local macro `rown` is used to build up the list of row names, applied with `matrix rownames`.

The one trick needed here appears in the `return` statement when only one variable is provided. We must doubly dereference `r`: once to specify the statistic being evaluated, and a second time to retrieve its content.

I added two additional options in the `syntax` statement for this version: a `format()` option, which allows you to specify the Stata format used to display the matrix elements, and the `mat` option.

```
 *! pctrange v1.0.6  CFBaum 11aug2008
program pctrange, rclass byable(recall)
        version 14
        syntax varlist(min=1 numeric ts) [if] [in] [, noPRINT FORmat(passthru)
> MATrix(string)]
        marksample touse
        quietly count if `touse´
        if `r(N)´ == 0 {
                error 2000
        }
        local nvar : word count `varlist´
        if `nvar´ == 1 {
                local res range p7525 p9010 p9505 p9901
                tempname `res´
                quietly summarize `varlist´ if `touse´, detail
                scalar `range´ = r(max) - r(min)
                scalar `p7525´ = r(p75) - r(p25)
                scalar `p9010´ = r(p90) - r(p10)
                scalar `p9505´ = r(p95) - r(p5)
                scalar `p9901´ = r(p99) - r(p1)
                if "`print´" != "noprint" {
                        display as result _n "Percentile ranges for `varlist´,  N = `r(N)´"
                        display as txt "75-25: " `p7525´
                        display as txt "90-10: " `p9010´
                        display as txt "95-05: " `p9505´
                        display as txt "99-01: " `p9901´
                        display as txt "Range: " `range´
                }
                foreach r of local res {
                        return scalar `r´ = ``r´´
                }
                return scalar N = r(N)
        }
        else {
                tempname rmat
                matrix `rmat´ = J(`nvar´,5,.)
                local i 0
                foreach v of varlist `varlist´ {
                        local ++i
                        quietly summarize `v´ if `touse´, detail
                        matrix `rmat´[`i´,1] = r(max) - r(min)
                        matrix `rmat´[`i´,2] = r(p75) - r(p25)
                        matrix `rmat´[`i´,3] = r(p90) - r(p10)
                        matrix `rmat´[`i´,4] = r(p95) - r(p5)
                        matrix `rmat´[`i´,5] = r(p99) - r(p1)
                        local rown "`rown´ `v´"
                }
                matrix colnames `rmat´ = Range P75-P25 P90-P10 P95-P05 P99-P01
                matrix rownames `rmat´ = `rown´
```

```
                if "`print´" != "noprint" {
                        local form ", noheader"
                        if "`format´" != "" {
                                local form "`form´ `format´"
                        }
                        matrix list `rmat´ `form´
                }
                if "`matrix´" != "" {
                        matrix `matrix´ = `rmat´
                }
                return matrix rmat = `rmat´
        }
        return local varname `varlist´
end
```

You can now invoke the program on a set of variables and, optionally, specify a format for the output of matrix elements:

```
. pctrange regday specneed bilingua occupday tot_day tchratio, form(%9.2f)
                Range     P75-P25     P90-P10     P95-P05     P99-P01
  regday      5854.00      918.50     2037.00     2871.00     4740.00
specneed     49737.01     2282.78     4336.76     5710.46    10265.45
bilingua    295140.00        0.00     6541.00     8817.00    27508.00
occupday     15088.00        0.00     5291.50     8096.00    11519.00
 tot_day      6403.00     1070.00     2337.50     3226.00     4755.00
tchratio        15.60        3.25        5.55        7.55       10.60
```

The `mat` option allows the matrix to be automatically saved as a Stata matrix with that name. This is useful if you are running `pctrange` several times (perhaps in a loop) and want to avoid having to rename the result matrix, `r(rmat)`, each time. You can use Baum and Azevedo's `outtable` routine (available from the SSC archive) to convert the matrix into a LaTeX table.

Table: MCAS percentile ranges

	Range	P75-P25	P90-P10	P95-P05	P99-P01
regday	5854.00	918.50	2037.00	2871.00	4740.00
specneed	49737.01	2282.78	4336.76	5710.46	10265.45
bilingua	295140.00	0.00	6541.00	8817.00	27508.00
occupday	15088.00	0.00	5291.50	8096.00	11519.00
tot day	6403.00	1070.00	2337.50	3226.00	4755.00
tchratio	15.60	3.25	5.55	7.55	10.60

Making commands byable

As a final touch, you might want the `pctrange` command to be *byable*: to permit its use with a `by` prefix. Because we are not creating any new variables with this version of the program, this can be done by simply adding `byable(recall)` to the `program` statement. The new `program` statement becomes:

```
 program pctrange, rclass byable(recall)
```

The other enhancement you might consider is allowing the *varlist* to contain variables with time-series operators such as `L.gdp` or `D.income`. We can easily incorporate that feature by changing the `syntax` statement to add the `ts` suboption:

```
 syntax varlist(min=1 numeric ts) [if] [in] ///
[, noPRINT FORmat(passthru) MATrix(string)]
```

Likewise, we could permit the use of factor variables in the varlist by adding the `fv` suboption. Because factor variables produce indicator variables taking on values 0 or 1, it would not be sensible to compute their percentiles.

With these modifications, we can apply `pctrange` using the `by` prefix or use time-series operators in the varlist. To illustrate the byable nature of the program, let's generate an indicator for teachers' average salaries above and below the mean and calculate the `pctrange` statistics for those categories.

Likewise, we could permit the use of factor variables in the varlist by adding the `fv` suboption. Because factor variables produce indicator variables taking on values 0 or 1, it would not be sensible to compute their percentiles.

With these modifications, we can apply `pctrange` using the `by` prefix or use time-series operators in the varlist. To illustrate the byable nature of the program, let's generate an indicator for teachers' average salaries above and below the mean and calculate the `pctrange` statistics for those categories.

```
 . discard
. summarize avgsalry, meanonly
. generate byte highsal = avgsalry > r(mean) & !missing(avgsalry)
. label define sal 0 low 1 high
. label val highsal sal
. tabstat avgsalry, by(highsal) stat(mean N)
Summary for variables: avgsalry
     by categories of: highsal
highsal |      mean         N
--------+--------------------
    low |   33.5616       101
   high |  38.60484        94
--------+--------------------
  Total |   35.9927       195
-----------------------------
```

We see that average salaries in low-salary school districts are over $5,000 less than those in high-salary school districts.

```
. bysort highsal: pctrange regday specneed bilingua occupday tot_day tchratio
________________________________________________________________________________

-> highsal = low
               Range     P75-P25     P90-P10     P95-P05     P99-P01
  regday        4858         703        1740        2526        3716
specneed   49737.008   2030.8198   3997.9497   5711.2104    11073.81
bilingua      295140           0        6235        8500       13376
occupday       11519           0        5490        7095       11286
 tot_day        5214         780        1770        2652        4597
tchratio        11.6   3.1999989   6.2999992   7.8000002   9.3999996
________________________________________________________________________________

-> highsal = high
               Range     P75-P25     P90-P10     P95-P05     P99-P01
  regday        5433        1052        2189        2807        5433
specneed   8570.4004   2486.3604   4263.9702     5620.54   8570.4004
bilingua       33968           0        8466       11899       33968
occupday       15088           0        5068        8100       15088
 tot_day        5928        1179        2572        3119        5928
tchratio        15.6   2.4000006   4.7000008   6.2999992        15.6
```

The salary differences carry over into the percentile ranges, where the ranges of `tot_day`, total spending per pupil, are much larger in the high-salary districts than in the low-salary districts.

Writing an egen function

The `egen` (Extended Generate) command is open-ended, in that any Stata user can 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`, can 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
       program _gpct9010
       version 14
       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()` 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
        program _gpct9010
        version 14
        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.

Writing an e-class program

The ado-file programs we have discussed in earlier sections are all `r`-class programs; that is, they provide results in the `return list`. Many statistical procedures involve fitting a model (rather than computing one or more statistics) and are thus termed *estimation commands*, or e-class commands.

One of Stata's great strengths derives from the common nature of its estimation commands, which follow a common syntax, leave behind the same objects, and generally support the same postestimation tools, such as `test`, `lincom`, and `margins` to compute marginal effects and `predict` to compute predicted values, residuals and similar quantities.

Although e-class commands are somewhat more complicated than r-class commands, it is reasonably simple for you to implement an estimation command as an ado-file. Many of the programming concepts discussed in earlier sections are equally useful when dealing with e-class commands. The additional features needed generally relate to postestimation capabilities.

One of Stata's great strengths derives from the common nature of its estimation commands, which follow a common syntax, leave behind the same objects, and generally support the same postestimation tools, such as `test`, `lincom`, and `margins` to compute marginal effects and `predict` to compute predicted values, residuals and similar quantities.

Although e-class commands are somewhat more complicated than r-class commands, it is reasonably simple for you to implement an estimation command as an ado-file. Many of the programming concepts discussed in earlier sections are equally useful when dealing with e-class commands. The additional features needed generally relate to postestimation capabilities.

As spelled out in [U] **18.9 Accessing results calculated by estimation commands**, there are a number of conventions that an `e`-class command must follow:

- The command must save its results in `e()`, accessed by `ereturn list`, rather than in `r()`.
- It should save its name in `e(cmd)`.
- It should save the contents of the command line in `e(cmdline)`.
- It should save the number of observations in `e(N)` and identify the estimation sample by setting the indicator variable (or "function") `e(sample)`.
- It must save the entire coefficient vector as Stata matrix `e(b)` and the variance–covariance matrix of the estimated parameters as Stata matrix `e(V)`.

As spelled out in [U] **18.9 Accessing results calculated by estimation commands**, there are a number of conventions that an `e`-class command must follow:

- The command must save its results in `e()`, accessed by `ereturn list`, rather than in `r()`.
- It should save its name in `e(cmd)`.
- It should save the contents of the command line in `e(cmdline)`.
- It should save the number of observations in `e(N)` and identify the estimation sample by setting the indicator variable (or "function") `e(sample)`.
- It must save the entire coefficient vector as Stata matrix `e(b)` and the variance–covariance matrix of the estimated parameters as Stata matrix `e(V)`.

As spelled out in [U] **18.9 Accessing results calculated by estimation commands**, there are a number of conventions that an `e`-class command must follow:

- The command must save its results in `e()`, accessed by `ereturn list`, rather than in `r()`.
- It should save its name in `e(cmd)`.
- It should save the contents of the command line in `e(cmdline)`.
- It should save the number of observations in `e(N)` and identify the estimation sample by setting the indicator variable (or “function”) `e(sample)`.
- It must save the entire coefficient vector as Stata matrix `e(b)` and the variance–covariance matrix of the estimated parameters as Stata matrix `e(V)`.

As spelled out in [U] **18.9 Accessing results calculated by estimation commands**, there are a number of conventions that an `e`-class command must follow:

- The command must save its results in `e()`, accessed by `ereturn list`, rather than in `r()`.
- It should save its name in `e(cmd)`.
- It should save the contents of the command line in `e(cmdline)`.
- It should save the number of observations in `e(N)` and identify the estimation sample by setting the indicator variable (or "function") `e(sample)`.
- It must save the entire coefficient vector as Stata matrix `e(b)` and the variance–covariance matrix of the estimated parameters as Stata matrix `e(V)`.

As spelled out in [U] **18.9 Accessing results calculated by estimation commands**, there are a number of conventions that an `e`-class command must follow:

- The command must save its results in `e()`, accessed by `ereturn list`, rather than in `r()`.
- It should save its name in `e(cmd)`.
- It should save the contents of the command line in `e(cmdline)`.
- It should save the number of observations in `e(N)` and identify the estimation sample by setting the indicator variable (or "function") `e(sample)`.
- It must save the entire coefficient vector as Stata matrix `e(b)` and the variance–covariance matrix of the estimated parameters as Stata matrix `e(V)`.

Correct capitalization of these result names is important. The coefficient vector is saved as a $1 \times k$ row vector for single-equation estimation commands, with additional rows added for multiple-equation estimators. The variance-covariance matrix is saved as a $k \times k$ symmetric matrix.

The presence of `e(b)` and `e(V)` in standardized locations enables Stata's postestimation commands (including those you write) to work properly. Estimation commands may set other `e()` scalars, macros or matrices.

Correct capitalization of these result names is important. The coefficient vector is saved as a $1 \times k$ row vector for single-equation estimation commands, with additional rows added for multiple-equation estimators. The variance-covariance matrix is saved as a $k \times k$ symmetric matrix.

The presence of `e(b)` and `e(V)` in standardized locations enables Stata's postestimation commands (including those you write) to work properly. Estimation commands may set other `e()` scalars, macros or matrices.

Whereas a `r`-class program, such as `pctrange`, uses the `return` command to return its results in `r()`, an `e`-class program uses the `ereturn` command. The command `ereturn` *name* = *exp* returns a scalar value, while `ereturn local` *name value* and `ereturn matrix` *name matname* return a macro and a Stata matrix, respectively. You do not use `ereturn` for the coefficient vector or estimated variance–covariance matrix.

The `ereturn post` command posts the estimates of `b` and `V` to their official locations. To return the coefficient vector and its variance–covariance matrix, you need to create the coefficient vector, say `` `beta' ``, and its variance–covariance matrix, say `` `vce' ``, and pass them back in the following fashion. We also can define the estimation sample flagged by the sample indicator temporary variable, `` `touse' ``:

```
    ereturn post `beta' `vce', esample(`touse')
```

You can save anything else in `e()`, using the `ereturn scalar`, `ereturn local`, or `ereturn matrix` commands, as described above. It is best to use the commonly used names for various quantities. For instance, `e(df_m)` and `e(df_r)` are commonly used to denote the numerator (model) and denominator (residual) degrees of freedom. `e(F)` commonly refers to the test against the null (constant-only) model for non-asymptotic results, while `e(chi2)` is used for an asymptotic estimator. `e(r2)` or `e(r2_p)` refer to the R^2 or pseudo-R^2, respectively.

Although you are free to choose other names for your `ereturn` values, it is most helpful if they match those used in common Stata commands. See [U] **18.10.2 Storing results in e()** for more details.

You can save anything else in `e()`, using the `ereturn scalar`, `ereturn local`, or `ereturn matrix` commands, as described above. It is best to use the commonly used names for various quantities. For instance, `e(df_m)` and `e(df_r)` are commonly used to denote the numerator (model) and denominator (residual) degrees of freedom. `e(F)` commonly refers to the test against the null (constant-only) model for non-asymptotic results, while `e(chi2)` is used for an asymptotic estimator. `e(r2)` or `e(r2_p)` refer to the R^2 or pseudo-R^2, respectively.

Although you are free to choose other names for your `ereturn` values, it is most helpful if they match those used in common Stata commands. See [U] **18.10.2 Storing results in e()** for more details.

Introduction to Mata

Since the release of version 9, Stata has contained 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 Stata/BE. 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.

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.

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 onwards 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. Results produced by Mata may then be accessed by Stata and formatted with commands like `estimates display`.

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.

Mata 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 )
```

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 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 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 1×1 dot product of vectors `r1, c1` while `j` is their 3×3 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 3×3 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`.

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.

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`.

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.

Calling Mata with a single command line

You can invoke Mata with a single Stata command, in either interactive mode or in a do-file or ado-file, with

```
. mata: one or more Mata commands, separated by semicolons
```

This context is most useful when you want to operate on one or more items in the Stata workspace, and return the results to the Stata workspace. In doing so, if you create items in Mata's workspace, they will remain there until you give the command `mata: mata clear`.

As an example of a single-line Mata command, say that we have cross-sectional data on nominal expenditures for 20 hospitals for selected years. For comparability across years, we want to create real (inflation-adjusted) measures. We have a price deflator for health care expenditures, benchmarked at 100 in 2000, for each of these years. Here are those data.

```
. list exp*, sep(0)

     ┌──────────────────────────────────────────────────────┐
     │  exp1994    exp1997    exp2001    exp2003    exp2005 │
     ├──────────────────────────────────────────────────────┤
  1. │ 8.181849   8.803116   9.842918   5.858101   9.190242 │
  2. │  9.23138   9.526636   7.415146    10.4168   8.252012 │
  3. │ 7.222272   6.583617   8.960152    6.53302   7.180707 │
  4. │ 9.448738   7.972495   8.766521    5.63679   6.681845 │
  5. │ 9.649901   6.769967   7.712472   9.241309   9.122219 │
  6. │ 6.856859   9.078987   7.766836   8.037149    8.25163 │
  7. │ 9.278242   7.264326   8.981012    8.23664   6.625589 │
  8. │ 6.841638   7.565176   6.927269   8.598692   10.75326 │
  9. │ 8.666395   7.525422   9.326843   8.239338    9.08489 │
 10. │ 7.644565   7.221323   8.915318   8.676471   9.277551 │
 11. │ 8.184382   9.080503   8.066475   8.371346   8.339171 │
 12. │ 8.145008   9.001379   7.540075   7.305631   8.677146 │
 13. │ 9.644306   9.578444   8.541085   7.431371   9.072832 │
 14. │ 7.397709   10.08259   7.648673   7.524302   7.143608 │
 15. │ 7.040617   8.258828   7.966858   10.83057   7.402671 │
 16. │ 7.861959   7.856915   8.849826   7.502389   8.290931 │
 17. │ 7.902071   9.412766   7.739596   7.646762   8.840546 │
 18. │ 9.576131   7.904189   9.243579   8.642523   9.428108 │
 19. │ 6.432031   8.056722    8.43427   8.755649   8.945447 │
 20. │ 8.228907   9.583271   8.321041   6.481467   10.23306 │
     └──────────────────────────────────────────────────────┘

. loc defl 87.6 97.4 103.5 110.1 117.4
```

In this example, we use a Mata view, labeled as *X*, to transform the Stata variables in place. The local macro containing each year's price deflator, `defl`, is transformed into a row vector *D* and scaled by 100. The last Mata command uses the colon operator (`:/`) to divide each hospital's nominal expenditures by the appropriate year's deflator.

```
. mata: expend = st_view(X=., ., "exp1994 exp1997 exp2001 exp2003 exp2005"); ///
> D = 0.01 :* strtoreal(tokens(st_local("defl"))); X[.,.] = X :/ D
```

We rename the Stata variables to reflect their new definitions:

```
. rename exp* rexp*
. list rexp*, sep(0)

       ┌──────────────────────────────────────────────────┐
       │ rexp1994   rexp1997   rexp2001   rexp2003   rexp2005 │
       ├──────────────────────────────────────────────────┤
    1. │  9.34001   9.038107   9.510066    5.32071   7.828145 │
    2. │  10.5381   9.780941   7.164392   9.461221   7.028971 │
    3. │ 8.244602    6.75936   8.657151   5.933715   6.116446 │
    4. │ 10.78623   8.185313   8.470069     5.1197    5.69152 │
    5. │ 11.01587   6.950685   7.451664   8.393559   7.770204 │
    6. │ 7.827465   9.321342   7.504189   7.299863   7.028646 │
    7. │  10.5916   7.458241   8.677306   7.481053   5.643602 │
    8. │ 7.810089   7.767121   6.693013   7.809893   9.159508 │
    9. │ 9.893146   7.726305   9.011443   7.483504   7.738408 │
   10. │ 8.726672   7.414089   8.613833   7.880537   7.902514 │
   11. │ 9.342902   9.322898   7.793695   7.603402   7.103212 │
   12. │ 9.297955   9.241662   7.285097   6.635451   7.391095 │
   13. │ 11.00948   9.834132   8.252256   6.749656   7.728137 │
   14. │ 8.444874   10.35174   7.390022   6.834062   6.084845 │
   15. │ 8.037233    8.47929   7.697447   9.837034   6.305512 │
   16. │ 8.974838   8.066648   8.550556   6.814159   7.062121 │
   17. │  9.02063    9.66403   7.477871   6.945288   7.530277 │
   18. │ 10.93166   8.115184   8.930994   7.849703   8.030757 │
   19. │ 7.342501   8.271789   8.149053   7.952451   7.619631 │
   20. │ 9.393729   9.839087   8.039653   5.886891   8.716402 │
```

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 construct an ado-file, `varextrema`, which takes a variable name and accepts optional `if` or `in` qualifiers. Rather than computing statistics in the ado-file, we call the `calcextrema` routine with two arguments: the variable name and the `` `touse' `` indicator variable.

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 18
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 18
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 18
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 18
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.

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, or the `teffects` suite) but we would like to approach it from first principles with a Mata routine.

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, or the `teffects` suite) 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.

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.

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 |
```

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.

```
. 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.

Additional Mata resources

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.

For more detail on Mata, see Chapters 13–14 of *An Introduction to Stata Programming, Second Edition* and Bill Gould's Stata Conference talk, "Mata: the Missing Manual" at `https://ideas.repec.org/p/boc/chic11/2.html`.

If you really want to become a serious Stata programmer, check out Bill Gould's book, *The Mata Book: A Book for Serious Programmers and Those Who Want to Be*, available from `http://stata-press.com`.

For more detail on Mata, see Chapters 13–14 of *An Introduction to Stata Programming, Second Edition* and Bill Gould's Stata Conference talk, "Mata: the Missing Manual" at `https://ideas.repec.org/p/boc/chic11/2.html`.

If you really want to become a serious Stata programmer, check out Bill Gould's book, *The Mata Book: A Book for Serious Programmers and Those Who Want to Be*, available from `http://stata-press.com`.