Why should you become a Stata programmer?

Christopher F Baum

Boston College and DIW Berlin

January 2009

Introduction

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

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

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

Introduction

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

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

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

Introduction

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

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

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

First, some nomenclature related to programming:

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

Any of these tasks involve *Stata programming*.

First, some nomenclature related to programming:

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

Any of these tasks involve *Stata programming*.

First, some nomenclature related to programming:

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

Any of these tasks involve *Stata programming*.

First, some nomenclature related to programming:

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

Any of these tasks involve *Stata programming*.

First, some nomenclature related to programming:

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

Any of these tasks involve *Stata programming*.

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

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

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

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

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 or supervisor.

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

Context 1: do-file programming

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

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

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

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

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

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

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

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

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

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

Context 2: ado-file programming

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Context 3: Mata subroutines for ado-files

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

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

Context 3: Mata subroutines for ado-files

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

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

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

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

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

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

Tools for do-file authors

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

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

Tools for do-file authors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
sort postalcode
merge postalcode using pcstats, uniqusing
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
tsline  `alleps'
```

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

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

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

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

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

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

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

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

Example: `estimates` and `estout`

Anyone who performs empirical research is familiar with the tedious task of turning estimation output into tables, with appropriate handling of standard errors or t-statistics, p-values, significance stars, the alignment of explanatory variables and presentation of summary statistics. Stata's own `estimates` commands make that a bit simpler by allowing you to `estimates store` and produce a crude but readable table from several sets of output, with some control over the format and contents of the table, with `estimates table`. But that is a long way from publishable-quality results.

Thankfully, Ben Jann's `estout` suite of programs provides complete, easy-to-use routines to turn sets of estimates into publication-quality tables in LaTeX, MSWord or HTML formats. The routines have been described in two *Stata Journal* articles, 5:3 (2005) and 7:2 (2007), and `estout` has its own website:

`http://repec.org/bocode/e/estout`

which has explanations of all of the available options and numerous worked examples of its use.

To use the facilities of `estout`, you merely preface the estimation commands with `eststo`:

```
eststo: regress y x1 x2 x3
eststo: probit z a1 a2 a3 a4
eststo: ivreg2 y3 (y1 y2 = z1-z4) z5 z6, gmm2s
```

Then, to produce a table, just give command

```
esttab using myests.tex
```

which will create the LaTeX table in that file. A file destined for Excel would use the `.csv` extension; for Word, use `.rtf`. You may also use extension `.html` or `.smcl` (Stata's own markup language).

The `esttab` command is a easy-to-use wrapper for `estout`, which has many options to control the exact format and content of the table. Any of the `estout` options may be used in the `esttab` command. For instance, you may want to suppress the coefficient listings of year dummies in a panel regression.

You may also use `estadd` to include user-generated statistics in the `ereturn list` (such as elasticities produced by `mfx`) so that they can be accessed by `esttab`.

The `esttab` command is a easy-to-use wrapper for `estout`, which has many options to control the exact format and content of the table. Any of the `estout` options may be used in the `esttab` command. For instance, you may want to suppress the coefficient listings of year dummies in a panel regression.

You may also use `estadd` to include user-generated statistics in the `ereturn list` (such as elasticities produced by `mfx`) so that they can be accessed by `esttab`.

One very useful feature is the `margin` option of `esttab`, which allows you to display only the marginal effects (and not the estimated coefficients) of a limited dependent variable model such as `logit` or `probit`. You may also display the results of multiple-equation models using `estout`.

It may be necessary to change the format of your estimation tables when submitting a paper to a different journal: for instance, one which wants t-statistics rather than standard errors reported. This may be easily achieved by just rerunning the estimation job with different `estout` options.

One very useful feature is the `margin` option of `esttab`, which allows you to display only the marginal effects (and not the estimated coefficients) of a limited dependent variable model such as `logit` or `probit`. You may also display the results of multiple-equation models using `estout`.

It may be necessary to change the format of your estimation tables when submitting a paper to a different journal: for instance, one which wants t-statistics rather than standard errors reported. This may be easily achieved by just rerunning the estimation job with different `estout` options.

Ado-file programming: a primer

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

A program begins with the `program define` *progname* statement, which usually includes the option `, rclass`, and a `version 10.0` statement. The *progname* should not be the same as any Stata command, nor for safety's sake the same as any accessible user-written command. If `findit progname` does not turn up anything, you can use that name. Programs (and Stata commands) are either *r-class* or *e-class*. The latter group of programs are for estimation; the former do everything else. Most programs you write are likely to be r-class.

Ado-file programming: a primer

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

A program begins with the `program define` *progname* statement, which usually includes the option `, rclass`, and a `version 10.0` statement. The *progname* should not be the same as any Stata command, nor for safety's sake the same as any accessible user-written command. If `findit progname` does not turn up anything, you can use that name. Programs (and Stata commands) are either *r-class* or *e-class*. The latter group of programs are for estimation; the former do everything else. Most programs you write are likely to be r-class.

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

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

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

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

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

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

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

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

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

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

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

You may return scalars, local macros, or matrices:

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

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

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

You may return scalars, local macros, or matrices:

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

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

A sample program from `help return`:

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

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

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

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

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

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

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

```
program define mysum3, rclass
    version 10.0
    syntax varlist(max=1) [if] [in]
    return local varname `varlist'
    marksample touse
    mata: m_mysum("`varlist'", "`touse'")
    return scalar N = N
    return scalar sum = sum
    return scalar mean = mu
    return scalar sd = sigma
end
```

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

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

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

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

```
version 10.0
mata:
void m_mysum(string scalar vname,
             string scalar touse)
{
    st_view(X, ., vname, touse)
    mu = mean(X)
    st_numscalar("N", rows(X))
    st_numscalar("mu", mu)
    st_numscalar("sum" ,rows(X) * mu)
    st_numscalar("sigma", sqrt(variance(X)))
}
end
```

Although it might appear that the Mata subroutine is an unnecessary complication for this simple set of computations, the example illustrates how Mata may be used in conjunction with the ado-file language.

Mata can easily access or modify any object in Stata's workspace. The ado-file language can be used to parse the syntax of the command, and hand off the appropriate variables, observations and parameters to Mata for numerical processing. Mata can return its results, including new variables, scalars, macros, and matrices to Stata.

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

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

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

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