

Intermediate Stata

Christopher F Baum

Faculty Micro Resource Center

Academic Technology Services, Boston College

August 2004

`baum@bc.edu`

`http://fmwww.bc.edu/GStat/docs/StataInter.pdf`

Thanks to Petia Petrova and Vince Wiggins for their comments on this draft.

File handling

File extensions usually employed (but not required) include:

<code>.ado</code>	automatic do-file (defines a Stata command)
<code>.dct</code>	data dictionary, optionally used with <code>infile</code>
<code>.do</code>	do-file (user program)
<code>.dta</code>	Stata binary dataset
<code>.gph</code>	graphics output file (binary)
<code>.log</code>	text log file
<code>.smcl</code>	SMCL (markup) log file, for use with Viewer
<code>.raw</code>	ASCII data file

These extensions need not be given (except for `.ado`). If you use other extensions, they must be explicitly specified.

Loading external data: `insheet`

Comma-separated (CSV) files or tab-delimited data files may be read very easily with the `insheet` command—which despite its name does not read spreadsheet files. If your file has variable names in the first row that are valid for Stata, they will be automatically used (rather than default variable names). You usually need not specify whether the data are tab- or comma-delimited—but note that `insheet` cannot read space-delimited data (or character strings with embedded spaces, unless they are quoted).

If the file extension is `.raw`, you may just use

```
insheet using filename
```

to read it. If other file extensions are used, they must be given:

```
insheet using filename.csv
```

```
insheet using filename.txt
```

Loading external data: infile

A free-format ASCII text file with space-, tab-, or comma-delimited data may be read with the `infile` command. The missing-data indicator (`.`) may be used to specify that values are missing.

The command must specify the variable names. Assuming `auto.raw` contains numeric data,

```
infile price mpg displacement using auto
```

will read it. If a file contains a combination of string and numeric values in a variable, it should be read as string, and `encode` used to convert it to numeric with string value labels.

If some of the data are string variables without embedded spaces, they must be specified in the command:

```
infile str3 country price mpg displacement using auto2
```

would read a three-letter country of origin code, followed by the numeric variables.

The number of observations will be determined from the available data.

Loading external data: infile

The `infile` command may also be used with fixed-format data, including data containing undelimited string variables, by creating a dictionary file which describes the format of each variable and specifies where the data are to be found. The dictionary may also specify that more than one record in the input file corresponds to a single observation in the data set.

If data fields are not delimited—for instance, if the sequence ‘102’ should actually be considered as three integer variables—a dictionary must be used to define the variables’ locations.

The `byvariable()` option allows a variable-wise dataset to be read, where one specifies the number of observations available for each series.

Loading external data: infix

An alternative to infile with a dictionary is the `infix` command, which presents a syntax similar to that used by SAS for the definition of variables' data types and locations in a fixed-format ASCII data set: that is, a data file in which certain columns contain certain variables. The `_column()` directive allow contents of a fixed-format data file to be retrieved selectively.

`infix` may also be used for more complex record layouts where one individual's data are contained on several records in an ASCII file.

Loading external data

A logical condition may be used on the `infile` or `infix` commands to read only those records for which certain conditions are satisfied: i.e.

```
infix using employee if sex=="M"
```

```
infile price mpg using auto in 1/20
```

where the latter will read only the first 20 observations from the external file. This might be very useful when reading a large data set, where one can check to see that the formats are being properly specified on a subset of the file.

Loading external data: Stat/Transfer

If your data are already in the internal format of SAS, SPSS, Excel, GAUSS, MATLAB, or a number of other packages, the best way to get it into Stata is by using the third-party product Stat/Transfer. Stat/Transfer, the Swiss Army Knife of data converters, can be acquired for Windows machines from Research Services. It is available on the ecsa200.bc.edu and econ.bc.edu systems for Unix, Linux and Macintosh users.

Stat/Transfer will preserve variable labels, value labels, and other aspects of the data, and can be used to convert a Stata binary file into other packages' formats. It can also produce subsets of the data (selecting variables, cases or both) so as to generate an extract file that is more manageable. This is particularly important when the 2,047-variable limit on standard Stata data sets is encountered. Stat/Transfer also supports the Stata/SE database format.

Stat/Transfer is well documented, with on-line help available in both Windows, Mac and Unix versions, and an extensive manual.

Working with stored data: append and merge

In many empirical research projects, the raw data to be utilized are stored in a number of separate files: separate “waves” of panel data, timeseries data extracted from different databases, and the like. Stata only permits a single data set to be accessed at one time. How, then, do you work with multiple data sets? Several commands are available, including `append`, `merge`, and `joinby`. The `append` command combines two Stata-format data sets that possess variables in common, adding observations to the existing variables. The same variables need not be present in both files, as long as a subset of the variables are common to the “master” and “using” data sets. It is important to note that “PRICE” and “price” are different variables, and one will not be appended to the other.

The `merge` command is very powerful. Like `append`, it works on a “master” data set—the current contents of memory—and a “using” data set. One or more merge variables are specified, and both master and using data sets must be sorted on those variables.

The distinction between “master” and “using” is important. When the same variable is present in each of the files, Stata’s default behavior is to hold the master data inviolate and discard the using dataset’s copy of that variable. This may be modified by the `update` option, which specifies that non-missing values in the using dataset should replace missing values in the master, and `update replace`, which specifies that non-missing values in the using dataset should take precedence.

Working with stored data: merge

A “**one-to-one**” merge specifies that each record in the using data set is to be combined with one record in the master data set. This would be appropriate if you acquired additional variables for the same observations. A new variable, `_merge`, takes on integer values indicating whether an observation appears in the master only, the using only, or appears in both. This may be used to determine whether the merge has been successful, or to remove those observations which remain unmatched (e.g. merging a set of households from different cities with a comprehensive list of ZIP codes; one would then discard all the unused ZIP code records). The `_merge` variable must be dropped before another merge is performed on this data set.

The `merge` command can also do a “**match merge**”, or “**one-to-N**” merge, in which each record in the using data set is matched with a number of records in the master data set. If a number of the households lived in the same ZIP code, then the match would place variables from the ZIP code file on the household records, repeating where necessary. This is a very useful technique to combine aggregate data with disaggregate data without dealing with the details.

Although “**one-to-N**” and “**N-to-one**” merges are commonplace and very useful, you never want to do a “**N-to-N**” merge, which will yield seemingly random results. To ensure that one data set has unique identifiers, specify the `uniquemaster` or `uniquusing` options, or use the `isid` command to ensure that a dataset has a unique identifier.

Writing external data: outfile, outsheet and file

If you want to transfer data to another package, Stat/Transfer is very useful. But if you just want to create an ASCII file from Stata, the `outfile` command may be used. It takes a *varlist*, and the `if` or `in` clauses may be used to control the observations to be exported. Applying `sort` prior to `outfile` will control the order of observations in the external file. You may specify that the data are to be written in comma-separated format.

The `outsheet` command can write a comma-delimited or tab-delimited ASCII file, optionally placing the variable names in the first row. Such a file can be easily read by a spreadsheet program such as Excel. Note that `outsheet` does not write spreadsheet files.

For customized output, the `file` command can write out information (including scalars, matrices and macros, text strings, etc.) in any ASCII or binary format of your choosing.

Creating external data: `postfile`

A very useful capability is provided by the `postfile` and `post` commands, which permit a Stata data set to be created in the course of a program. For instance, you may be simulating the distribution of a statistic, fitting a model over separate samples, or bootstrapping standard errors. Within the looping structure, you may `post` certain numeric values to the `postfile`. This will create a separate Stata binary data set, which may then be opened in a later Stata run and analysed. Note, however, that only numeric expressions may be written to the `postfile`, and the parens `()` given in the documentation, surrounding each `exp`, are required.

Reconfiguring data: collapse

Data are often provided in a different orientation than that required for statistical analysis. The most common example of this occurs with panel, or longitudinal, data, in which each observation conceptually has both cross-section (i) and time-series (t) subscripts. Often one will want to work with a “pure” cross-section or “pure” time-series. If the microdata themselves are the objects of analysis, this can be handled with sorting and a loop structure. If you have data for N firms for T periods per firm, and want to fit the same model to each firm, one could use the `statsby` command, or if more complex processing of each model’s results was required, a `foreach` block could be used. If analysis of a cross-section was desired, a `bysort` would do the job.

But what if you want to use average values for each time period, averaged over firms? The resulting dataset of T observations can be easily created by the `collapse` command, which permits you to generate a new data set comprised of summary statistics of specified variables. More than one summary statistic can be generated per input variable, so that both the number of firms per period and the average return on assets could be generated. `collapse` can produce counts, means, medians, percentiles, extrema, and standard deviations.

Reconfiguring data: reshape

Different models applied to longitudinal data require different orientations of those data. For instance, seemingly unrelated regressions (`sureg`) require the data to have T observations, with separate variables for each cross-sectional unit. Fixed-effects or random-effects regression models `xtreg`, on the other hand, require that the data be stacked or “vec” ’d. It is usually much easier to generate transformations of the data in stacked format, where a single variable is involved.

The `reshape` command allows you to transfer the data from the former format (known as `wide`) to the latter (known as `long`) or vice versa. It is a complicated command, because of the many variations on this process one might encounter, but it is very powerful.

Reconfiguring data: reshape

As an example, a dataset from the World Bank, provided as a spreadsheet, has rows labelled by both country (`ccode`) and variable (`vcode`), and columns labelled by years. Two applications of `reshape` were needed to transfer the data to the desired `long` format, where the observations have both country and year subscripts, and the columns are variables:

```
reshape long d, i(ccode vcode) j(year)
```

```
reshape wide d, i(ccode year) j(vcode) string
```

The resulting data set is in the appropriate format for `xtreg` modelling. If it were to be used in `sureg`-type models, a further `reshape wide` could be applied to transform it into that format.

Repeating commands: `foreach` and `forvalues`

One of Stata's great strengths is the ability to perform repetitive tasks without spelling out the details (e.g. the `by` prefix). However, the `by` prefix can only execute a single command; so that while you may run a regression for each country in your sample, you cannot also save the residuals or predicted values for those country-specific regressions.

Stata provides two commands that allow construction of a true block structure or loop: `foreach` and `forvalues`. These commands permit a delimited block of commands to be repeated over elements of a *varlist* or *numlist*. Indeed, the target of `foreach` may be any list of names, and can be a list of new variables to be created. The `forvalues` *numlist* may include an increment, so that it could for instance count from 10 to 100 in steps of 10, or count down from 10 to 1.

This code fragment loops over a varlist, calculates (but does not display) the descriptives of each variable, and then summarizes the observations of that variable that exceed its mean. Note the use of *'var'*, in particular the backtick (') on the left of the word. This syntax is mandatory when referring to the placeholder.

```
foreach var of varlist pri-rep t* {  
    quietly summarize 'var'  
    summarize 'var' if 'var' > r(mean)  
}
```

Generally a `forvalues` or `foreach` loop is the best way to solve any programming problem that involves repetition. It is usually much faster, in the long run, to figure out how to place a problem in this context. Nested loops may also be defined with these commands.

Repeating commands: while and if

A loop structure may also be explicitly defined by the `while` command, which is akin to the “do while” construct in other programming languages. A `while` structure often will make use of an `if` command—not to be confused with the `if` clause on other commands—which will create conditional logic. The `if` command may also use an `else` clause to express conditional logic.

For many purposes, it is more efficient (in terms of your time) to employ `foreach` or `forvalues`, since those commands handle the logic of repetition without explicit detail. Programs written with these commands are easier to maintain and modify. We demonstrate below how a loop could be written in both of these formats.

Local macros, scalars and results

In programming terms, **local macros** and **scalars** are the “variables” of Stata programs (not to be confused with the variables of the data set). The distinction: a local macro can contain a string, while a scalar can contain a single number (at maximum precision). You should use these constructs whenever possible to avoid creating variables with constant values merely for the storage of those constants. This is particularly important when working with large data sets.

When you want to work with a scalar object—such as a counter in a `foreach` or `forvalues` command—it will involve defining and accessing a local macro. In addition, all Stata commands that compute results or estimates generate one or more objects to hold those items, which are scalars (numbers), local macros (strings) or matrices.

This behavior of Stata's computational commands allows you to write a do-file that makes use of these quantities. We saw one example of this above. The command `summarize` generates a number of scalars, such as `r(N)`, the number of observations; `r(mean)`, the mean; `r(Var)`, the variance; etc. The available items are shown by `return list` for a "r-class" command. The contents of these scalars may be used in expressions. In the example above, the mean of a variable was used to govern a following `summarize` command:

```
quietly summarize price  
summarize price if price > r(mean)
```

In this example, the scalar `r(mean)` may be used directly. But what if you wanted to issue another command that generated results, which would wipe out all of the `r()` returns? Then you use the `tt` local statement to preserve the item in a macro of your choosing:

```
local mu `r(mean)`
```

Later in the program, you could use

```
regress mpg weight length if price > `mu`
```

Note the use of the backtick (‘) on the left of the local macro. This syntax is mandatory, as it makes it clear that you are referring to the *value* of the local macro `mu` rather than the contents of the variable `mu`.

Stata commands are either *r-class* commands like `summarize`, that return results, or *e-class* commands, that return estimates. You may examine the set of results from a *r-class* command with the command `return list`. For an *e-class* command, use `ereturn list`. An *e-class* command will return `e()` scalars, macros and matrices: for instance, after `regress`, the local macro `e(N)` will contain the number of observations, `e(r2)` the R^2 value, `e(depvar)` will contain the name of the dependent variable, and so on.

Commands may also return matrices. For instance, `regress` will return the matrix `e(b)`, a row vector of point estimates, and the matrix `e(V)`, the estimated variance–covariance matrix of the estimated parameters.

Use `display` to examine the contents of a scalar or local macro. For the latter, you must use the backtick and apostrophe to indicate that you want to access the contents of the macro: `contrast display r(mean)` with `display "The mean is 'mu' "`. The contents of matrices may be displayed with the `matrix list` command.

Since items are accessible in local macros, it is very easy to write a program that makes use of results in directing program flow. Local macros can be created by the `local` statement, and used as counters (e.g. in `while`).

Let us form a “rolling forecast” of volatility from a moving-window regression, and we had not learned that Baum’s `rollreg` command would do this job for us. Assume that we have 120 time-series observations which have been `tsset`:

```
gen volfc=.
local win 12
forv i=13/120 {
    local first = 'i'-'win'+4
    quietly regress y L(1/4).y in 'first'/'i'
    quietly replace volfc = e(rmse) in 'i'/'i'
}
```

This program will generate the series `volfc` as the RMS error of an AR(4) model fit to a window of 12 observations for the `y` series.

Let us redo the example above using the `while` looping construct:

```
gen volfc=.
local win 12
local i 13
local iend 120
while 'i' <= 'iend' {
    local first = 'i'-'win'+4
    quietly regress y L(1/4).y in 'first'/'i'
    quietly replace volfc = e(rmse) in 'i'/'i'
    local i = 'i' + 1
}
```

The most common error made with `while` is to forget the loop increment—which creates an infinite loop! Hit `break` (`ctrl-C`) to abort a runaway program.

The use of local macros and the appropriate loop constructs make it possible to write a Stata program that is fairly general, and requires little modification to be reused on different series, or with different parameters. This makes your work with Stata very productive, since much of the code is reusable and adaptable to similar tasks. Let us consider how this approach might be pursued in the context of the volatility forecast example.

We show here a complete Stata program, `volfc`, which is stored in the file `volfc.ado` on the `adopath`. Since this is a personally-authored program, it should be placed in the personal subdirectory of the `ado` directory at the top level of the hard disk (not the Stata directory's `ado` subdirectory!)

This program makes use of Stata's syntax parsing capabilities to allow this user-written command to emulate all Stata commands' syntax. It does not make use of many of the features that might be useful in such a command: handling `if` and `in` clauses, providing more specific error messages for inappropriate option values, and so on.

The program generalizes the do-file shown above by allowing the moving-window volatility estimate to be generated from a specified variable, and placed in a new variable specified in the `vol()` option. The window width (option `win()`) and AR length (option `AR()`) take on default values 12 and 4, but may be overridden by the user. The program automatically calculates the first and last observations to be used in the loop from the data and specified options. It could readily be generalized to use a different volatility measure from the rolling regression (e.g. mean absolute error).

To be complete, we should provide a help file for `volfc`, in the file `volfc.hlp`. The help file would specify the syntax of the command, explain its purpose, define each of the options, and provide any references to other Stata commands that might be useful.

```

program define volfc, rclass
version 7.0
syntax varname(numeric) ,Vol(string) [Win(integer 12) AR(integer 4)]
quietly tsset
if `win' < `ar' {
    dis "You must have a longer window than AR length!"
    error 198
}
quietly gen `vol'=.
local start = `win'+`ar'
quietly summ `varlist', meanonly
local last = r(N)
dis _n "'vol': volatility forecast for `varlist' with
        window=`win', AR(`ar')"
```

(continues...)

```
forv i='start'/'last' {  
    local first = 'i'-'win'+1  
    quietly regress 'varlist' L(1/'ar').'varlist' in 'first'/'i'  
    quietly replace 'vol' = e(rmse) in 'i'/'i'  
}  
  
exit  
end
```

This program defines the `volfc` command, which will appear like any other Stata command on your machine. It may be executed as

```
use http://fmwww.bc.edu/ec-p/data/macro/bdh, clear
volfc pcrude, vol(vv)
volfc pcrude, vol(vv24) win(24)
volfc pcrude, vol(vv126) ar(6)
volfc pcrude, vol(vv248) win(24) ar(8)
```

The volatility series might then be graphed (presuming a time variable `date` which is the variable that has been `tsset`) with

```
tsline vv vv24 vv126 vv248
```

This illustrates the relative simplicity of developing a quite general tool in Stata's programming language. Although you may use Stata without ever authoring an "ado-file", much of the productivity enhancement that a Stata user may enjoy is likely to be tied to this sort of development. Many research tasks are quite repetitive in some context, and developing a general-purpose tool to implement that repetition is likely to be a very good investment in terms of time and effort. Many of the modules available from the SSC Archive were first conceived by individuals looking to ease the burden of their own work. Stata's unique extensibility makes it trivial to incorporate user-written additions—including those which you author—into your copy of Stata, and to share it with collaborators or the Stata user community if desired.

As should be evident from this programming example, the `program` `define` command is used to declare a program. The program name must match the name of the ado-file in which it is stored. Most user-written programs are `r-class`. This program could be modified to return its parameters to the calling program with the `return` statement:

```
return local vol 'vol'  
return local win 'win'  
return local ar 'ar'  
return local first 'start'  
return local last 'last'
```

With these statements added to the end of the routine, the local macros are defined, and their values stored.

The second element to be noted is the `syntax` statement, which defines the allowable syntax for a user-written command. One may specify that the command allows a single variable, with `varname`; a set of variables, with `varlist`, optionally specifying how many are allowed. For instance, a statistical technique that operates on a pair of variables could specify that exactly two existing variables are to be provided. Likewise, one may specify that a new variable (or set of variables) are the `newvarlist` of the command, and `syntax` will check that they are indeed new variables.

Although not illustrated above, the `syntax` command will often specify that `if` and `in` clauses are optional elements. Optional elements of `syntax` (such as the options `Win` and `AR` above) are placed in brackets (`[]`).

This programming example illustrates a “required option”—the `vo1` option, which must be used on this command to specify the output of the command. The other two options are indeed optional, and take on default values if they are not specified. The argument of the `vo1` option is meant to be a new variable name; that will be trapped when the `generate` statement attempts to create the variable if it is already in use, or is not a valid variable name.

Most user–written programs could be improved by adding code to trap errors in users’ input. If the program is primarily for your own use, you may eschew extensive development of error trapping: for instance, checking the options for sensibility (although one test is applied here to prevent nonsensical results).

Local macros are exactly that: objects with local scope, defined within the program in which they are used, disappearing when that program terminates. This is generally the desired outcome, preventing a clutter of objects from being retained when a program calls numerous others in the course of execution. At times, though, it is necessary to have objects that can be passed from one subprogram to another. The `return` logic above would not really serve, since although it passes local macros from a program to its caller, they would then have to be passed as arguments to a second program.

To deal with the need for persistent objects, Stata contains *global macros*. These objects, once defined, live for the duration of your Stata session, and may be read or written within any Stata program. They are defined with the `global` command, rather than `local`, and referred to as `$macroname`. Global macros should only be used where they are required.

We now present an example of a Stata program that operates on panel, or longitudinal data. When you use panel data, you must use the panel data form of `tsset` in which both a unit variable and a time variable are specified.

Assume that you have a panel data set, properly identified as such, containing several time series for each unit in the panel: for instance, investment or population measures for several countries. We would like to generate a new series containing the deviations from a constant growth path (exponential trend) or, alternatively, the constant growth values themselves (the predicted values from the exponential trend line).

This program, `pangrodev`, performs this task for each unit of a panel, automatically identifying the observations belonging to each unit, taking the logarithm of the specified variable, running the appropriate regression and prediction commands, and assembling the results in the specified new variable.

The program makes use of Stata's `tempname` and `tempvar` commands to create non-scalar objects (in this case the matrix `VV` and variables `lvar` and `pvar` which, like local macros, will exist only for the duration of the ado-file). These temporary facilities, like the associated `tempfile` which allows temporary files to be specified, help reduce clutter and guarantee that objects' names will not conflict with other items in the user's namespace.

```
#! pangrodev 1.0.0  CFBaum 28May2001
* generate deviations from constant growth in panel context
program define pangrodev, rclass
version 7.0
syntax varname, Gen(string) [xb]
local togens "deviations from constant growth"
if "'xb'" != "" {
local togens "predicted growth"
}
qui tsset
local ivar = r(panelvar)
local timevar = r(timevar)
tempname VV
tempvar lvar pvar

(continues...)
```

```
qui gen 'lvar' = log('varlist')
qui tab 'ivar',matrow('VV')
local nvals=r(r)
local j = 1
while 'j'<='nvals' {
    local val = 'VV'['j',1]
    local vals "'vals' 'val'"
    local j='j'+1
}
qui gen double 'gen'=.
local xc 0
local tbar 0
local rsqr 0
```

(continues...)

```

foreach v of local vals {
summ 'lvar' if 'ivar'=='v',meanonly
if r(N)>2 {
qui regress 'lvar' 'timevar' if 'ivar'=='v'
capt drop 'pvar'
qui predict double 'pvar' if e(sample),xb
qui replace 'gen' = exp('pvar') if e(sample)
if "'xb'" =="" {
qui replace 'gen' = 'varlist'-'gen' if e(sample)
}
local xc = 'xc' + 1
local tbar = 'tbar' + e(N)
local rsqr = 'rsqr' + e(r2)
}
}

```

(continues...)

```
local tbar = int(100*'tbar' / 'xc')/100.0
local rsqr = int(1000*'rsqr' / 'xc')/1000.0
di in gr _n "'gen' : 'togens' for 'xc' of 'nvals' units"
di in gr "tbar = 'tbar'      rsq-bar = 'rsqr'"
exit
end
```

This program defines the `pangrodev` command, which will appear like any other Stata command on your machine. It may be executed as

```
. use http://fmwww.bc.edu/ec-p/data/macro/cap797wa  
(World Bank Database for Sectoral Investment, 1948-1992)  
. pangrodev TotSECap, g(totcapdev)
```

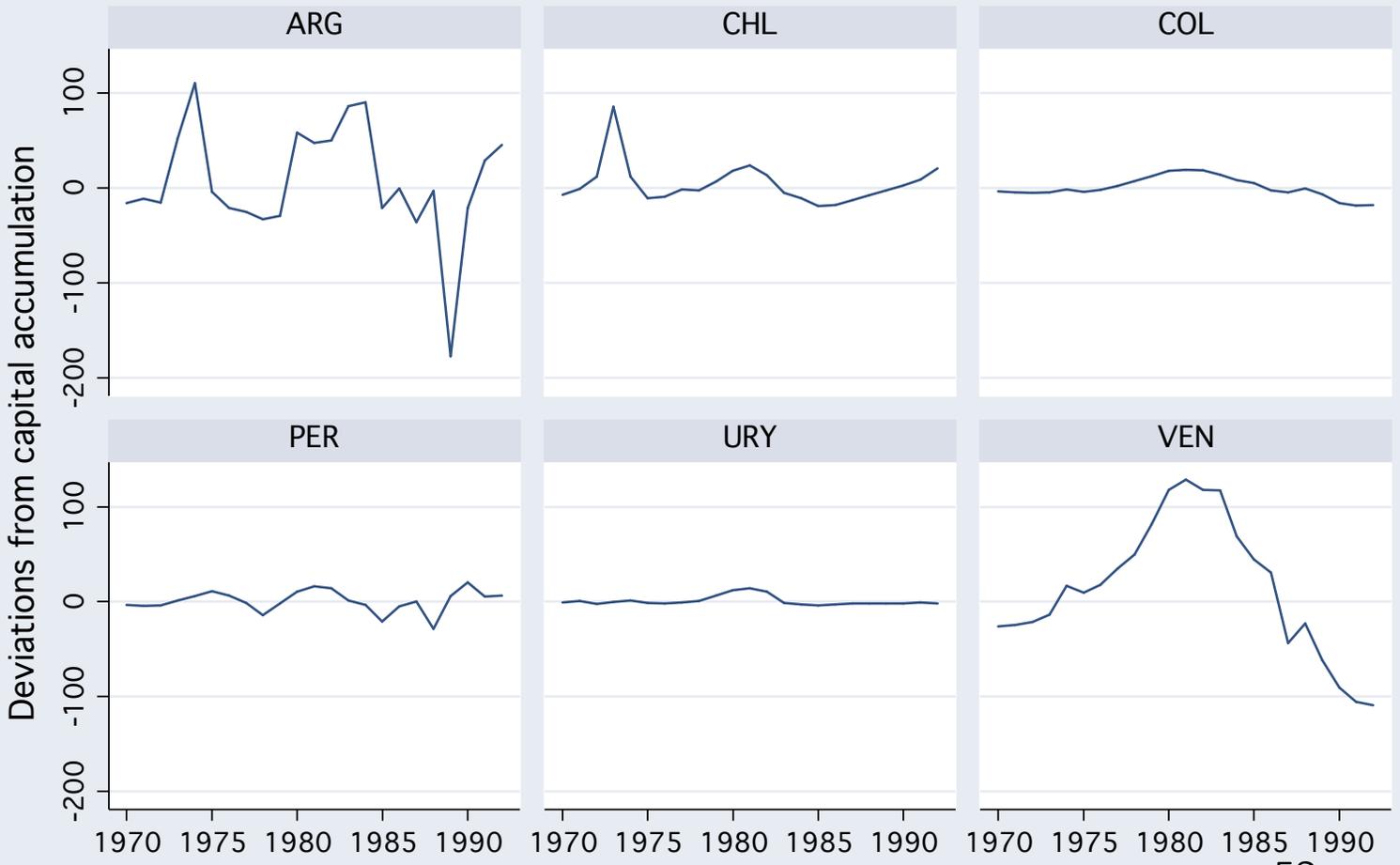
```
totcapdev : deviations from constant growth for 57 of 63 units  
tbar = 25.94    rsq-bar = .673
```

```
. pangrodev TotSECap, g(totcaphat) xb  
(output omitted)
```

Selected series computed by `pangrodev` can now be graphed by the `tsline` command, which accepts a `by(varlist)` option:

```
replace totcapdev=totcapdev/10^9
keep if (ccode=="ARG" | ccode=="CHL" | ccode=="COL" | ///
ccode=="PER" | ccode=="URY" | ccode=="VEN")
label var totcapdev "Deviations from capital accumulation"
label var ccode "South American country"
tsline totcapdev if year>1969, by(ccode)
```

will demonstrate how many countries followed the same pattern of below–trend growth of the capital stock (curtailed investment) during the 1980s.



Graphs by South American country

Whether or not you use Stata's programming facilities to write your own ado-files, a "reading knowledge" of the programming language is very useful in case you want to adapt an existing Stata command (official or user-contributed) in a do-file you are writing. For instance, the program listed above has a useful code fragment that places the unique *i*-values into a local macro. There may be many instances where you want to assemble a list of the unique IDs in a panel that satisfy a certain condition. This code fragment (which builds the local macro `vals`) can be extracted from `pangrodev.ado` and reused in your own do-files.

Since the code for all Stata commands that are implemented as ado-files (as the command `which...` will show) are available on your hard disk, Stata itself is a fertile source of programming techniques that may be adapted to solve any programming problem.