Automation and Programming with Stata

Christopher F Baum

Boston College and DIW Berlin

IMF Institute for Capacity Development, October 2018

Overview

This talk focuses on several ways in which you can use Stata as a programming language to automate your data management and statistics tasks and perform them more efficiently. We first discuss Stata's capabilities, augmented by several user-written packages, that allow the automated production of tables, draft and publication-quality estimation output, and graphics.

We then consider how "a little bit of Stata programming goes a long way" in terms of using the do-file language effectively; developing simple ado-files for repetitive tasks and various estimation and forecasting techniques; and by using Mata, Stata's matrix programming language, in conjunction with ado-file programming.

Production of summary statistics

A number of Stata commands can produce summary tables. They differ in their ease of use of producing tables that may be readily inserted into other programs, or generated as publication quality. Various user-written commands, available from SSC, have provided the requisite flexibility in this area.

To illustrate the problem, we might want to tabulate the number of years in which various countries in a panel data set experienced negative GDP growth. We can produce a frequency table with `tabulate`:

```
. bcuse pwt6_3, clear nodesc
(Penn World Tables 6.3, August 2009)
. keep if inlist(isocode, "ITA", "ESP", "GRC", "PRT", "TUR", "USA")
(10672 observations deleted)
. // indicator for negative GDP growth
. g neggrowth = (grgdpch < 0)
. label define tf 0 F 1 T
. label values neggrowth tf
. tab isocode neggrowth
```

ISO country code	neggrowth F	T	Total
ESP	51	7	58
GRC	48	10	58
ITA	53	5	58
PRT	50	8	58
TUR	44	14	58
USA	48	10	58
Total	294	54	348

A useful table, but there is no option to export it. The `tabulate` command does support export of the table contents as a matrix, but that requires additional effort to attach the appropriate row and column labels.

One solution which I have found very useful is Ian Watson's `tabout` command, available from SSC. This program provides a great deal of flexibility in constructing tables, and can export them as tab-delimited text, CSV, or as LaTeX. For example:

```
. tabout isocode neggrowth using imfs5_2b.csv, f(0c) replace
Table output written to: imfs5_2b.csv
```

ISO country code	neggrowth F No.	neggrowth T No.	Total No.
ESP	51	7	58
GRC	48	10	58
ITA	53	5	58
PRT	50	8	58
TUR	44	14	58
USA	48	10	58
Total	294	54	348

Which, when opened in MS Word or OpenOffice, yields

Sheet1

ISO country code	neggrowth F No.	T No.	Total No.
ESP	51	7	58
GRC	48	10	58
ITA	53	5	58
PRT	50	8	58
TUR	44	14	58
USA	48	10	58
Total	294	54	348

By using its style option, `tabout` can also produce the body of a LaTeX table, to which you can add features:

```
 . tabout isocode neggrowth using imfs5_2b.tex, style(tex) f(0c) replace
Table output written to: imfs5_2b.tex
 & \multicolumn{3}{c}{neggrowth} \\
ISO country code&F&T&Total \\
&No.&No.&No. \\
\hline
ESP&51&7&58 \\
GRC&48&10&58 \\
ITA&53&5&58 \\
PRT&50&8&58 \\
TUR&44&14&58 \\
USA&48&10&58 \\
Total&294&54&348 \\
```

TABLE 1. Years with negative GDP growth, 1960–2007

ISO country code	neggrowth F No.	neggrowth T No.	neggrowth Total No.
ESP	51	7	58
GRC	48	10	58
ITA	53	5	58
PRT	50	8	58
TUR	44	14	58
USA	48	10	58
Total	294	54	348

We can also use `tabout` to produce statistical tables, presenting one of the summary statistics for a given series:

```
 . g decade = int(year/10) * 10
. tabout decade isocode using imfs5_2d.tex, c(mean grgdpch) ///
> clab(_) style(tex) sum replace f(2) ptotal(none)
Table output written to: imfs5_2d.tex
 & \multicolumn{7}{c}{ISO country code} \\
decade&ESP&GRC&ITA&PRT&TUR&USA&Total \\
& & & & & & &  \\
\hline
1950&4.90&4.65&5.21&4.20&5.42&1.39&4.29 \\
1960&7.81&6.84&5.52&6.34&2.60&3.18&5.38 \\
1970&2.63&4.35&3.28&4.49&2.53&2.34&3.27 \\
1980&2.44&0.15&2.56&3.12&1.55&2.12&1.99 \\
1990&2.59&1.47&1.46&3.01&2.24&2.16&2.16 \\
2000&3.57&4.27&1.20&0.73&3.19&1.48&2.41 \\
```

TABLE 2. Average GDP per capita growth by decade, 1960–2007

	ISO country code						
decade	ESP	GRC	ITA	PRT	TUR	USA	Total
1950	4.90	4.65	5.21	4.20	5.42	1.39	4.29
1960	7.81	6.84	5.52	6.34	2.60	3.18	5.38
1970	2.63	4.35	3.28	4.49	2.53	2.34	3.27
1980	2.44	0.15	2.56	3.12	1.55	2.12	1.99
1990	2.59	1.47	1.46	3.01	2.24	2.16	2.16
2000	3.57	4.27	1.20	0.73	3.19	1.48	2.41

The primary focus of Ben Jann's `estout` suite is the production of estimation tables. However, it can also be used to produce tables of multiple summary statistics. For instance, let's calculate the average shares of consumption, investment and government spending (`kc, ki, kg` respectively) by decade using his `estpost` routine, a wrapper for `tabstat`, and feed the result to his `esttab`:

```
. qui estpost tabstat kc ki kg, by(decade) statistics(mean sd) ///
> columns(statistics) listwise nototal
. esttab using imfs5_2e.tex, replace main(mean) aux(sd) nostar ///
> unstack noobs nonote nomtitle nonumber
(output written to imfs5_2e.tex)
```

For more details, see the Examples->Advanced section of the `estout` website, `http://repec.sowi.unibe.ch/stata/estout/`.

TABLE 3. Average shares of consumption, investment, and government spending

	1950	1960	1970	1980	1990	2000
kc	67.15	62.49	61.69	62.64	62.24	61.49
	(8.440)	(8.226)	(7.247)	(5.512)	(5.205)	(5.350)
ki	21.60	27.41	28.68	24.96	27.81	31.24
	(6.234)	(7.458)	(7.205)	(4.443)	(4.104)	(4.358)
kg	12.38	11.26	11.04	12.78	12.78	12.56
	(4.058)	(2.604)	(2.011)	(1.701)	(1.903)	(2.367)

Note: Standard errors in parentheses.

Production of estimates tables

Stata has a suite of commands, `estimates`, that allow you to store sets of estimation results (and optionally save them to disk) so that they may be accessed later in either a statistical command (such as `hausman`) or, more commonly, to produce tables of estimates.

After any estimation (e-class) command, you may use `estimates store` *setname* to store that set of estimates for the duration of your Stata session. The *setnames* may then be used later in your do-file to access the stored estimates.

The `estimates table` command, which we have seen in earlier slides, can be used to produce a readable table of estimation results from several different models, with a number of options to control what is presented (e.g., point estimates only, standard errors, t- or z-statistics, significance stars) and their format. The command can also `keep` or `drop` certain coefficients (e.g., a set of time dummies) from the tabular output, and add a set of scalars to the table, including AIC and BIC values.

Although this command was enhanced in recent versions of Stata, it is still limited to producing a table in the results window and the logfile (if open). It does not support table export to other formats.

The estout command suite

To overcome these limitations, 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 freely downloadable *Stata Journal* articles, 5:3 (2005) and 7:2 (2007), and `estout` has its own website:

`http://repec.sowi.unibe.ch/stata/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 clear
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 MS Word, use `.rtf`. You may also use extension `.html` for HTML or `.smcl` for a table in 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 `margins`) so that they can be accessed by `esttab`.

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.

For instance, consider an example from the Penn World Tables (v6.3) dataset where we run the same regression on three Mediterranean countries, and would like to present a summary table of results:

```
. eststo clear
. foreach c in ESP GRC ITA {
  2.          eststo: qui reg grgdpch`c´ grgdpchUSA  openc`c´ L.cgnp`c´
  3. }
(est1 stored)
(est2 stored)
(est3 stored)
```

We use `eststo clear` to remove all prior sets of estimates named by `eststo`.

Now, merely giving the `esttab` command produces a readable table, and allows us to change some aspects of the table with simple options:

```
. esttab, drop(_cons) stat(r2 rmse)
```

	(1) grgdpchESP	(2) grgdpchGRC	(3) grgdpchITA
grgdpchUSA	0.279	0.358	0.149
	(1.42)	(1.71)	(1.02)
opencESP	-0.0207		
	(-0.50)		
L.cgnpESP	2.058		
	(1.62)		
opencGRC		-0.211***	
		(-4.56)	
L.cgnpGRC		-1.351**	
		(-3.26)	
opencITA			-0.0672
			(-1.60)
L.cgnpITA			1.353*
			(2.53)
r2	0.152	0.425	0.296
rmse	3.051	3.173	2.236

t statistics in parentheses
* p<0.05, ** p<0.01, *** p<0.001

By providing variable labels and using a few additional `esttab` options, we can make the table more readable:

```
. esttab, drop(_cons) se stat(r2 rmse) lab nonum ti("GDP growth regressions")
GDP growth regressions
```

	ESP	GRC	ITA
US gdp gr	0.279	0.358	0.149
	(0.196)	(0.209)	(0.146)
ESP openness	-0.0207		
	(0.0411)		
L.ESP rgdp per cap.	2.058		
	(1.267)		
GRC openness		-0.211***	
		(0.0463)	
L.GRC rgdp per cap.		-1.351**	
		(0.415)	
ITA openness			-0.0672
			(0.0419)
L.ITA rgdp per cap.			1.353*
			(0.534)
r2	0.152	0.425	0.296
rmse	3.051	3.173	2.236

```
Standard errors in parentheses
* p<0.05, ** p<0.01, *** p<0.001
```

Still, this is merely a SMCL-format table in Stata's results window, and something we could have probably produced with `estimates table`. The usefulness of the `estout` suite comes from its ability to produce the tables in other output formats. For example:

```
. esttab using imfs5_1d.rtf, replace drop(_cons) se stat(r2 rmse) ///
> lab nonum ti("GDP growth regressions, 1960-2007")
(note: file imfs5_1d.rtf not found)
(output written to imfs5_1d.rtf)
```

Which, when opened in MS Word or OpenOffice, yields

GDP growth regressions, 1960-2007

	ESP	GRC	ITA
US gdp gr	0.279	0.358	0.149
	(0.196)	(0.209)	(0.146)
ESP openness	-0.0207		
	(0.0411)		
L.ESP rgdp per cap.	2.058		
	(1.267)		
GRC openness		-0.211***	
		(0.0463)	
L.GRC rgdp per cap.		-1.351**	
		(0.415)	
ITA openness			-0.0672
			(0.0419)
L.ITA rgdp per cap.			1.353*
			(0.534)
r2	0.152	0.425	0.296
rmse	3.051	3.173	2.236

Standard errors in parentheses
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

Let us illustrate how additional statistics may be added to a table. Consider the prior regressions (dropping the openness measure, and adding two additional countries) where we use `margins` to compute the elasticity of each country's GDP growth with respect to US GDP growth. By default, `margins` is a r-class command, so it returns the elasticity in matrix `r(b)` and its estimated variance in `r(V)`.

The `margins` command can also be used as an e-class command by invoking the `post` option. This example would be somewhat more complicated in that case, as we would have two e-class commands from which various results are to be combined.

```
. eststo clear
. foreach c in ESP GRC ITA PRT TUR {
  2.          eststo: qui reg grgdpch`c´ grgdpchUSA L.cgnp`c´
  3.          qui margins, eyex(grgdpchUSA)
  4.          matrix tmp = r(b)
  5.          scalar eta = tmp[1,1]
  6.          matrix tmp = r(V)
  7.          scalar etase = sqrt(tmp[1,1])
  8.          qui estadd scalar eta
  9.          qui estadd scalar etase
 10. }
(est1 stored)
(est2 stored)
(est3 stored)
(est4 stored)
(est5 stored)
```

The greatest degree of automation, using `estout`, arises when using it to produce LaTeX tables. As LaTeX is a programming language as well, `estout` can be instructed to include, for instance, Greek symbols, sub- and superscripts, and the like in its output, which will then produce a beautifully formatted table, ready for inclusion in a publication. In fact, camera-ready copy for Stata Press books, such as those I have authored, is produced in that manner.

```
 . esttab using imfs5_1f.tex, replace drop(_cons) se lab nonum ///
> ti("GDP growth regressions, 1960-2007") stat(eta etase r2 rmse, ///
> labels("\$\hat{\eta}\$" "s.e." "\$R^2\$" "\$RMSE\$")) ///
> note("Note: \$\eta\$: elasticity of GDP growth w.r.t. US GDP growth")
(output written to imfs5_1f.tex)
```

In this example, I have inserted LaTeX typesetting commands to label statistics as you might choose to label them in a journal submission.

TABLE 2. GDP growth regressions, 1960-2007

	ESP	GRC	ITA	PRT	TUR
US GDP growth	0.291	0.577*	0.193	0.439	0.331
	(0.193)	(0.245)	(0.146)	(0.284)	(0.309)
L.ESP RGDP p/c	2.400*				
	(1.061)				
L.GRC RGDP p/c		-0.770			
		(0.475)			
L.ITA RGDP p/c			1.716**		
			(0.492)		
L.PRT RGDP p/c				0.499	
				(0.366)	
L.TUR RGDP p/c					-0.577
					(1.036)
$\hat{\eta}$	0.151	0.869	0.386	0.380	0.150
s.e.	0.0397	43.04	0.636	0.939	0.180
R^2	0.147	0.146	0.254	0.0881	0.0390
$RMSE$	3.025	3.822	2.276	4.452	4.382

Note: η: elasticity of GDP growth w.r.t. US GDP growth

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

In a slightly more elaborate example, consider modelling the probability that GDP growth will exceed its historical median value, using a binomial probit model. In such a model, we do not want to report the original coefficients, which are marginal effects on the latent variable, but rather their transformations as measures of the effects on the probability of high GDP growth.

In this context, we estimate the model for each country, use `margins` to produce its default `dydx` values of $\partial Pr[\cdot]/\partial X$, and use the `post` option to store those as e-returns, to be captured by `eststo`. We also store the median growth rate so that it can be reported in the table.

```
. eststo clear
. foreach c in ESP GRC ITA PRT TUR {
  2.         qui summ grgdpch`c´, detail
  3.         scalar medgro`c´ = r(p50)
  4.         g higrowth`c´ = (grgdpch`c´ > medgro`c´)
  5.         lab var higrowth`c´ "`c´"
  6.         qui probit higrowth`c´ grgdpchUSA L.cgnp`c´, nolog
  7.         qui eststo: margins, dydx(*) post
  8.         qui estadd scalar medgro = medgro`c´
  9. }

. esttab using imfs5_1h.tex, replace se lab nonum ///
> ti("Pr[GDP growth \$>\$ median], 1960-2007") stat(medgro, ///
> labels("Median growth rate")) mti("ESP" "GRC" "ITA" "PRT" "TUR") ///
> note("Note: Marginal effects (\$\partial Pr[\cdot]/\partial X\$ displayed")
(output written to imfs5_1h.tex)
```

TABLE 1. Pr[GDP growth > median], 1960-2007

	ESP	GRC	ITA	PRT	TUR
US GDP growth	0.0566*	0.0309	0.0239	0.0482	0.0566
	(0.0278)	(0.0292)	(0.0288)	(0.0291)	(0.0321)
L.ESP RGDP p/c	0.299*				
	(0.151)				
L.GRC RGDP p/c		-0.150**			
		(0.0529)			
L.ITA RGDP p/c			0.292***		
			(0.0775)		
L.PRT RGDP p/c				0.0507	
				(0.0380)	
L.TUR RGDP p/c					-0.124
					(0.109)
Median growth rate	3.553	3.495	2.473	3.671	3.156

Note: Marginal effects ($\partial Pr[\cdot]/\partial X$ displayed

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

Producing documents within Stata

In Stata 15, there are two sets of commands that enable production of documents within Stata:

- `putdocx` writes paragraphs, images, and tables to an Office Open XML file (.docx). It may also be used to format each object added. This allows you to automate exporting and formatting of, for example, Stata estimation results and also to generate various reports based on those results. The generated file is compatible with Microsoft Word 2007 and later.
- `putpdf` writes paragraphs, images, and tables to a PDF file. It may also be used to format each object added. This allows you to automate exporting and formatting of, for example, Stata estimation results and also generate various reports based on those results.

Production of sets of tables and graphs

You may often have the need to produce a sizable number of very similar tables or graphs: one per country, sector or industry, or one per year, quinquennium or decade. We first illustrate how that might be automated for a set of regression tables: in this case, cross-country regressions over several decades, one table per decade.

```
. use pwt6_3, clear
(Penn World Tables 6.3, August 2009)
. keep if inlist(isocode, "ITA", "ESP", "GRC", "PRT", "BEL", ///
>                 "FRA", "ITA", "GER", "DNK")
(10556 observations deleted)
. g decade = int(year/10) * 10
```

```
. forvalues y = 1960(10)2000 {
  2.                 eststo clear
  3.                 qui regress kc openk pc  if decade == `y´
  4.                 scalar r2 = e(r2)
  5.                 qui eststo: margins, eyex(*) post
  6.                 qui estadd scalar r2 = r2
  7.                 qui regress kc openk pc ppp if decade == `y´
  8.                 scalar r2 = e(r2)
  9.                 qui eststo: margins, eyex(*) post
 10.                 qui estadd scalar r2 = r2
 11.                 qui regress kc openk pc xrat if decade == `y´
 12.                 scalar r2 = e(r2)
 13.                 qui eststo: margins, eyex(*) post
 14.                 qui estadd scalar r2 = r2
 15.
.                esttab using imfs5_3_`y´.tex, replace stat(N r2) ///
>                ti("Cross-country elasticities of Consumption/GDP for decade:
>  `y´s") ///
>                substitute("r2" "\$R^2\$") lab
 16. }
(output written to imfs5_3_1960.tex)
(output written to imfs5_3_1970.tex)
(output written to imfs5_3_1980.tex)
(output written to imfs5_3_1990.tex)
(output written to imfs5_3_2000.tex)
```

We can then include the separate LaTeX tables produced by the do-file in a research paper with the commands:

```
\input{imfs5_3_1960}
\input{imfs5_3_1970}
```

etc.

This approach has the advantage that the tables themselves need not be included in the document, so if we revise the tables we need not copy and paste the tables. There may be a similar capability available using RTF tables. To illustrate, here is one of the tables produced by this do-file:

TABLE 4. Cross-country elasticities of Consumption/GDP for decade: 1970s

	(1)	(2)	(3)
Openness in constant prices	-0.0113	-0.0137	-0.0134
	(-0.86)	(-1.03)	(-1.02)
Price level of consumption	-0.0430	-0.0162	-0.0168
	(-1.44)	(-0.41)	(-0.47)
Purchasing power parity		-0.00679	
		(-1.03)	
Exchange rate vs USD			-0.00885
			(-1.31)
N	80	80	80
R^2	0.0550	0.0685	0.0765

t statistics in parentheses

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

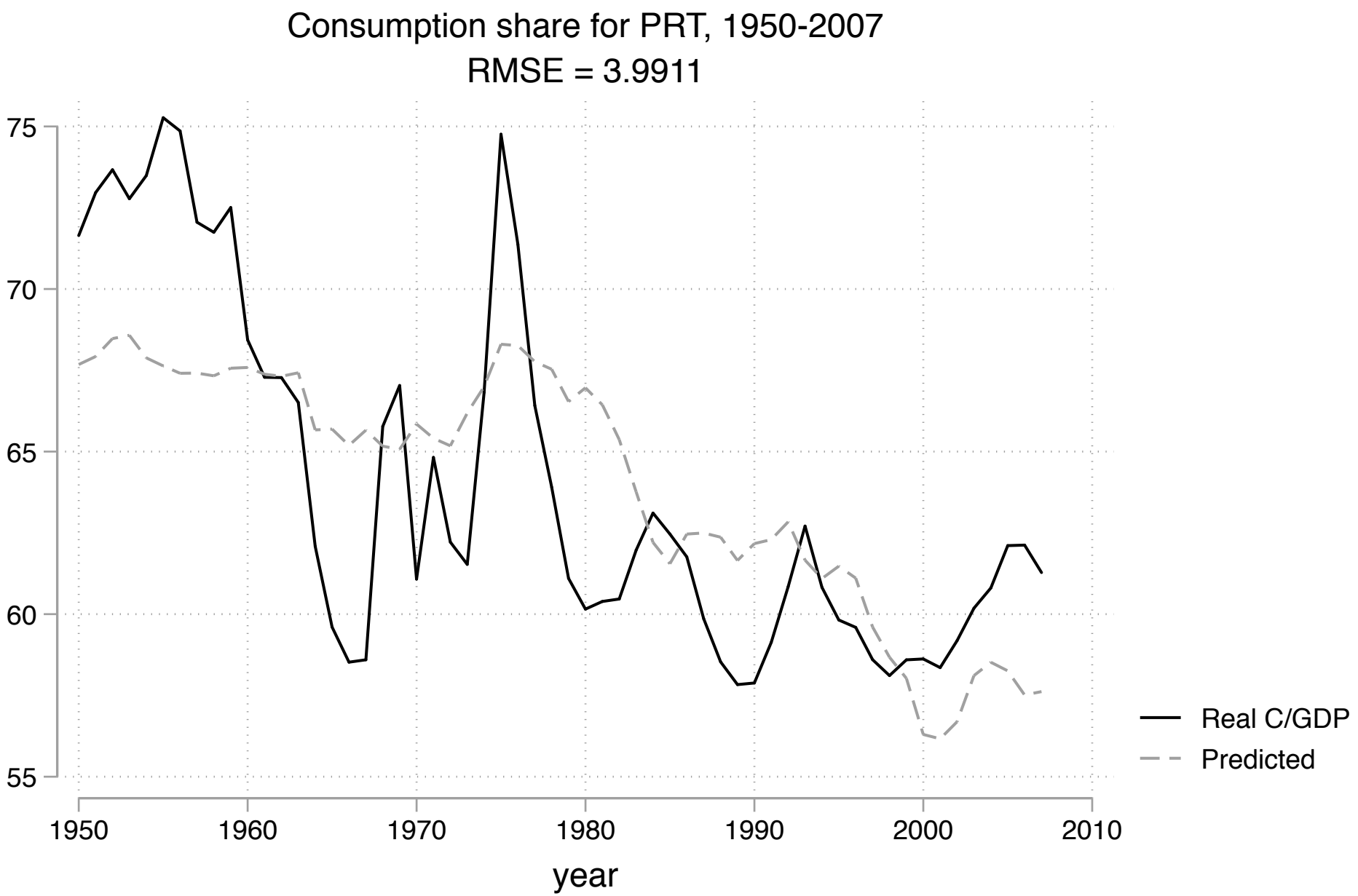
Graphics automation

Likewise, we could automate the production of a set of very similar graphs. Graphics automation is very valuable, as it avoids the manual tweaking of graphs produced by other software by making it a purely programmable function. Although the Stata graphics language is complex, the desired graph can be built up with the options needed to produce exactly the right appearance.

As an example, consider automating a plot of the actual and predicted values for time-series regressions for each country in this sample. We use the `plotplainblind` scheme from `blindschemes`, available from SSC.

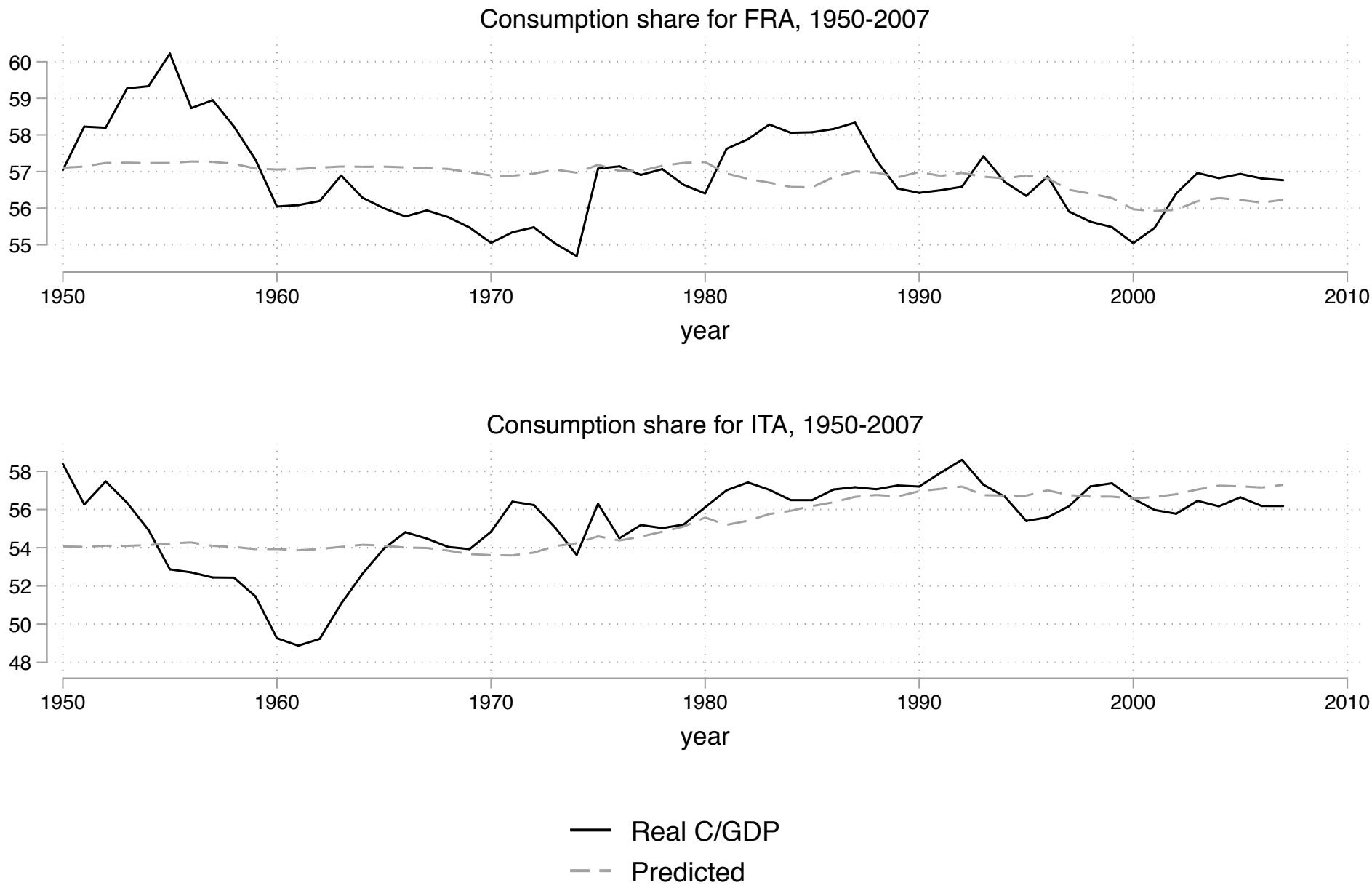
```
. levelsof isocode, local(ctys)
`"BEL"´ `"DNK"´ `"ESP"´ `"FRA"´ `"GER"´ `"GRC"´ `"ITA"´ `"PRT"´

. foreach c of local ctys {
  2.                 qui regress kc openk pc xrat if isocode == "`c´"
  3.                 local rmse = string(`e(rmse)´, "%7.4f")
  4.                 qui predict double kchat`c´ if e(sample), xb
  5.                 tsline kc kchat`c´ if e(sample), scheme(plotplainblind) ///
>               ti("Consumption share for `c´, 1960-2007") t2("RMSE = `rmse´"
> )
  6.                 graph export kchat`c´.pdf, replace
  7. }
(file /Users/baum/Documents/Stata/IMF2011/kchatBEL.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatDNK.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatESP.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatFRA.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatGER.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatGRC.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatITA.pdf written in PDF format)
(file /Users/baum/Documents/Stata/IMF2011/kchatPRT.pdf written in PDF format)
```


Consumption share for PRT, 1950-2007
RMSE = 3.9911
75
70
65
60
55
1950
1960
1970
1980
1990
2000
2010
year
Real C/GDP
Predicted

We can also use this technique to produce composite graphs, with more than one panel per graph:

```
. foreach c in FRA ITA {
  2.                 tsline kc kchat`c´ if isocode == "`c´", scheme(plotplainblind) ///
>                 ti("Consumption share for `c´, 1950-2007")  ///
>                 name(gr`c´, replace)
  3. }
. grc1leg grFRA grITA, cols(1) saving(grFRA_ITA, replace)
(file grFRA_ITA.gph saved)
```


Consumption share for FRA, 1950-2007
60
59
58
57
56
55
1950
1960
1970
1980
1990
2000
2010
year
Consumption share for ITA, 1950-2007
58
56
54
52
50
48
1950
1960
1970
1980
1990
2000
2010
year
Real C/GDP
Predicted

Should you be a Stata programmer?

We now turn to the broader question: how advantageous might it be to acquire Stata programming skills? 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.
- You may use Stata's programming language, *Mata*, to write routines in that language that are called by ado-files.

Any of these tasks involve *Stata programming*.

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

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

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

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

Context 1: do-file programming

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

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 of weeks of time over the course of a sizable research project.

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

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

Context 2: ado-file programming

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

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

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

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

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

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

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.

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

- Stata's maximum likelihood (`ml`) capabilities usually involve the construction of an ado-file program 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`) and the GMM command (`gmm`) 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 or GMM task.

Context 3: Mata subroutines for ado-files

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

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

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

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

Tools for do-file authors

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

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

Looping over observations is rarely appropriate

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

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

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

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

The by prefix can often replace a loop

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

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

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

Repeated statements are usually not needed

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

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

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

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

Merge can solve concordance problems

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

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

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

Some simple commands are often overlooked

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

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

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

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

egen functions can solve many programming problems

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

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

Learn how to use return and ereturn

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

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

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

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

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

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

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

extended macro functions, list functions, levelsof

Beyond their use in loop constructs with `foreach`, 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.

Ado-file programming: a primer

Continuing in our trinity of Stata programming roles, let us now discuss the rudiments of ado-file programming.

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 13` 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 `search progname` does not turn up anything, you can use that name. Programs (and Stata commands) are either *r-class* or *e-class*. The latter group of programs are for estimation; the former do everything else. Most programs you write are likely to be r-class.

The syntax statement

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

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

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

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

Option handling

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.

tempvars and tempnames

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

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

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

You may return scalars, local macros, or matrices:

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

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

A sample program from `help return`:

```
program define mysum, rclass
version 13
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
```

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 15
syntax varname [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
```

Examples of ado-file programming

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

For instance, the regression

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

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

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

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

```
 . type myregress.ado
*! myregress v1.0.0  CFBaum 20aug2013
program myregress, rclass
version 13
syntax varlist(ts) [if] [in], LAGVar(string) NLAGs(integer)
regress `varlist´ `if´ `in´
local nl1 = `nlags´ - 1
forvalues i = 1/`nl1´ {
        local lv "`lv´ L`i´.`lagvar´ + "
}
local lv "`lv´  L`nlags´.`lagvar´"
lincom `lv´
return scalar sum = `r(estimate)´
return scalar se = `r(se)´
end
```

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

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

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

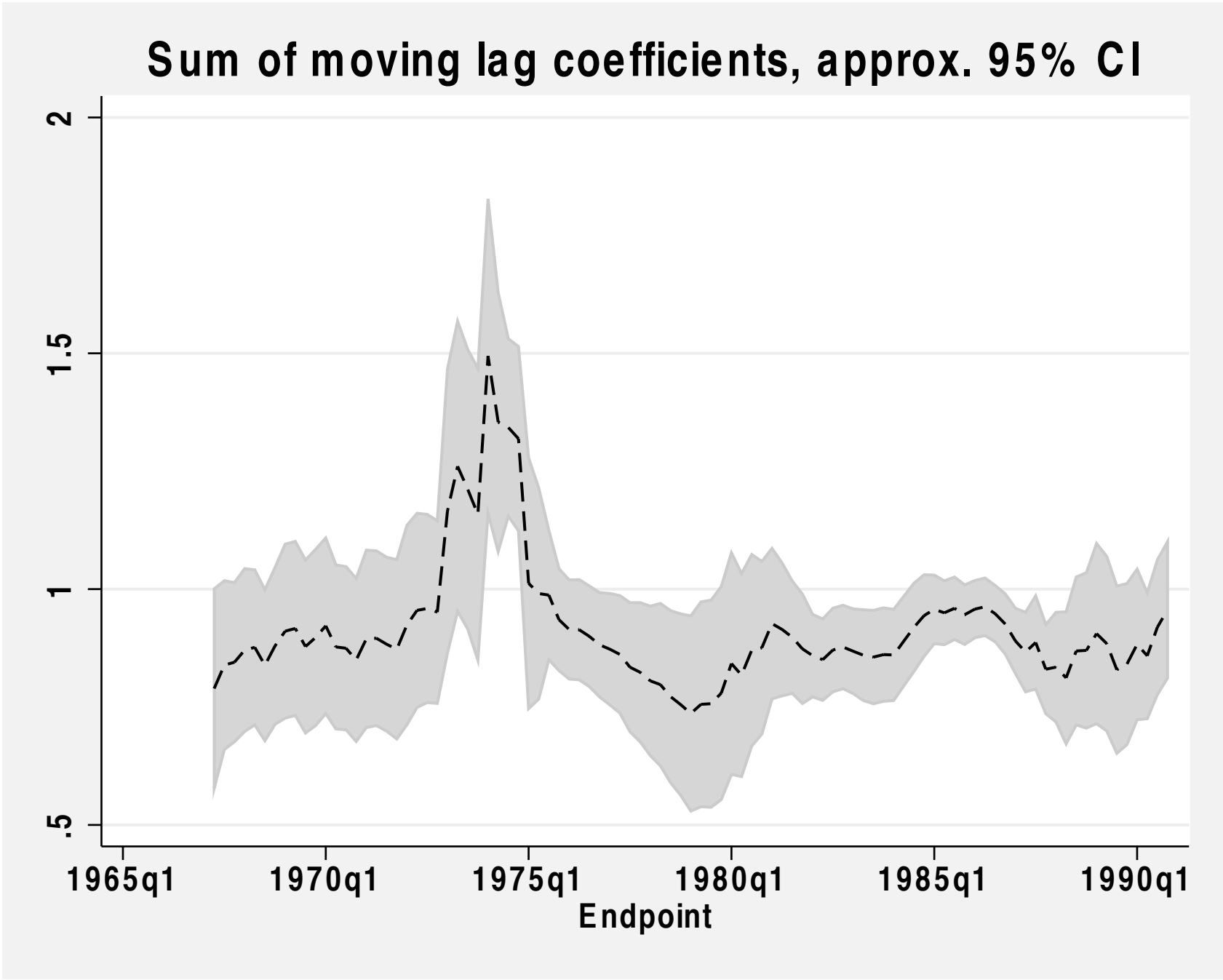
wpi	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
(1)	.9809968	.0082232	119.30	0.000	.9647067	.9972869

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

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

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

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


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

egen functions

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

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

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

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

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

```
. type _gpct9010.ado

*! _gpct9010 v1.0.0  CFBaum
        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.

Introduction to Mata

We now turn to the third way in which you may use Stata programming techniques: by taking advantage of 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/IC. Thus, code relying on Stata matrices is fragile. Stata's matrix language does contain commands such as `matrix accum` which can build a cross-product matrix from variables of any length, but for many applications the limitation of *matsize* is binding.

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

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

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

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

Speed advantages

Last but by no means least, ado-file code written in the matrix language with explicit subscript references is *slow*. Even if such a routine avoids explicit subscripting, its performance may be unacceptable. For instance, David Roodman's `xtabond2` can run in version 7 or 8 without Mata, or in version 9 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.

A simple Mata function

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

```
program define mysum3, rclass
   version 14
   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.

This function is considered a `void` function as it does not return a result when called; rather, it has the side effects of defining several scalars in the Stata environment.

```
version 14
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
```

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 14
   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 14
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.

For more detail on Mata, see Chapters 13–14 of *An Introduction to Stata Programming* and Bill Gould's Stata Conference talk, "Mata: the Missing Manual" at `http://repec.org`.