Automation and Programming with Stata

Christopher F Baum

Boston College and DIW Berlin

NCER, Queensland University of Technology, March 2014

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 readily produce a frequency table with `tabulate`:

```
 . use pwt6_3, clear
(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	T	Total
	No.	No.	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

As we will soon discuss, Ben Jann's `estout` suite is exceedingly useful for the production of estimation tables. But 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.org/bocode/e/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 *Stata Journal* articles, 5:3 (2005) and 7:2 (2007), and `estout` has its own website:

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

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

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

```
eststo 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 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)`.

As an aside, `margins` 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

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

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

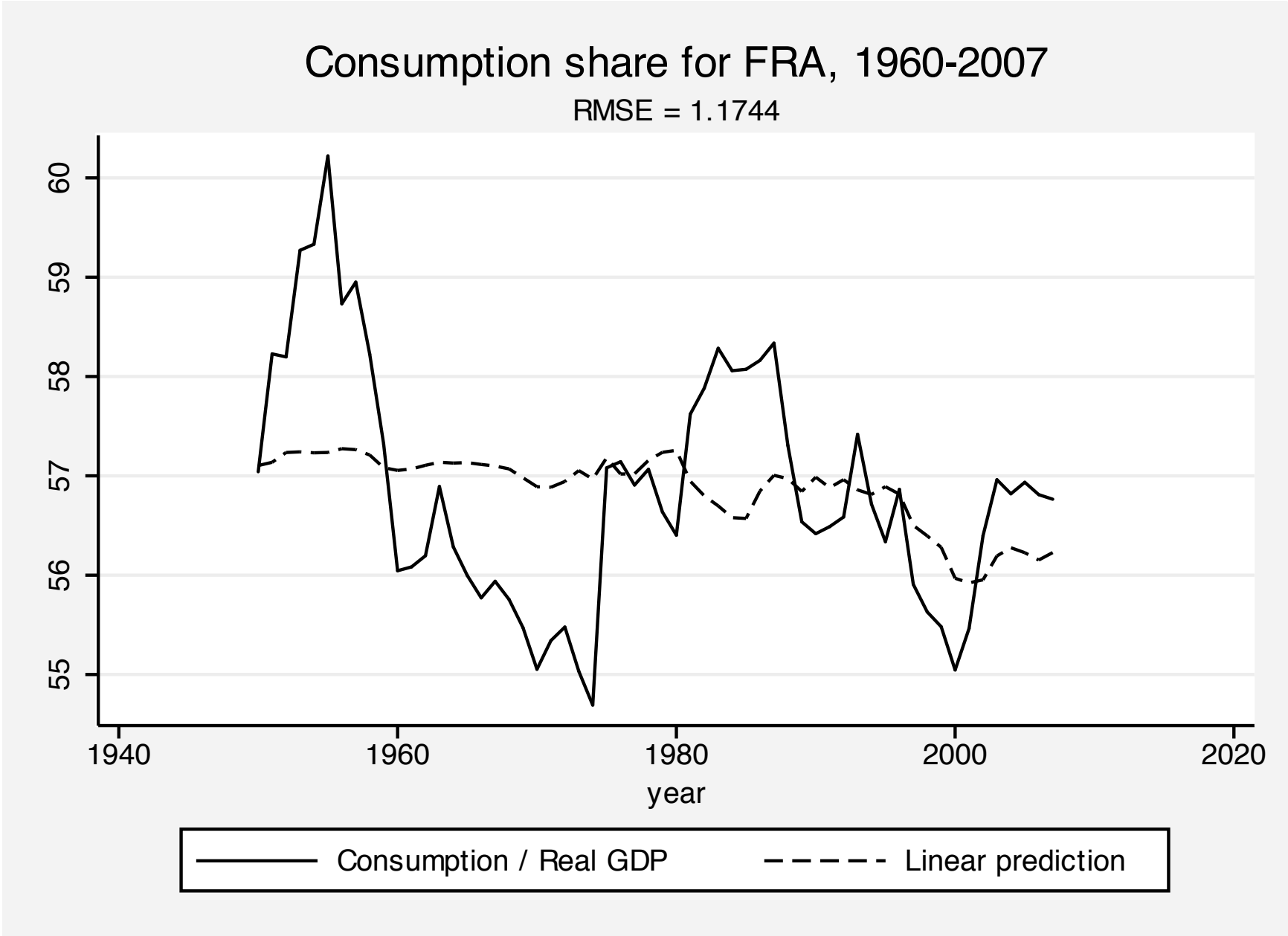
t statistics in parentheses

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

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.

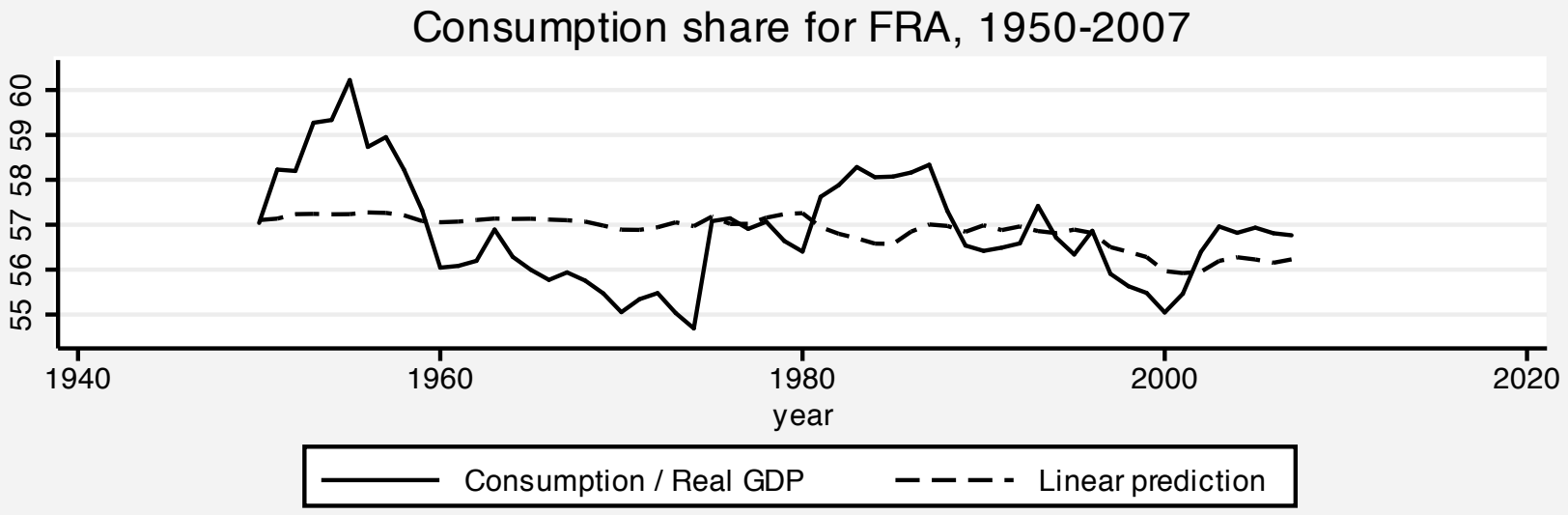
As an example, consider automating a plot of the actual and predicted values for time-series regressions for each country in this sample:

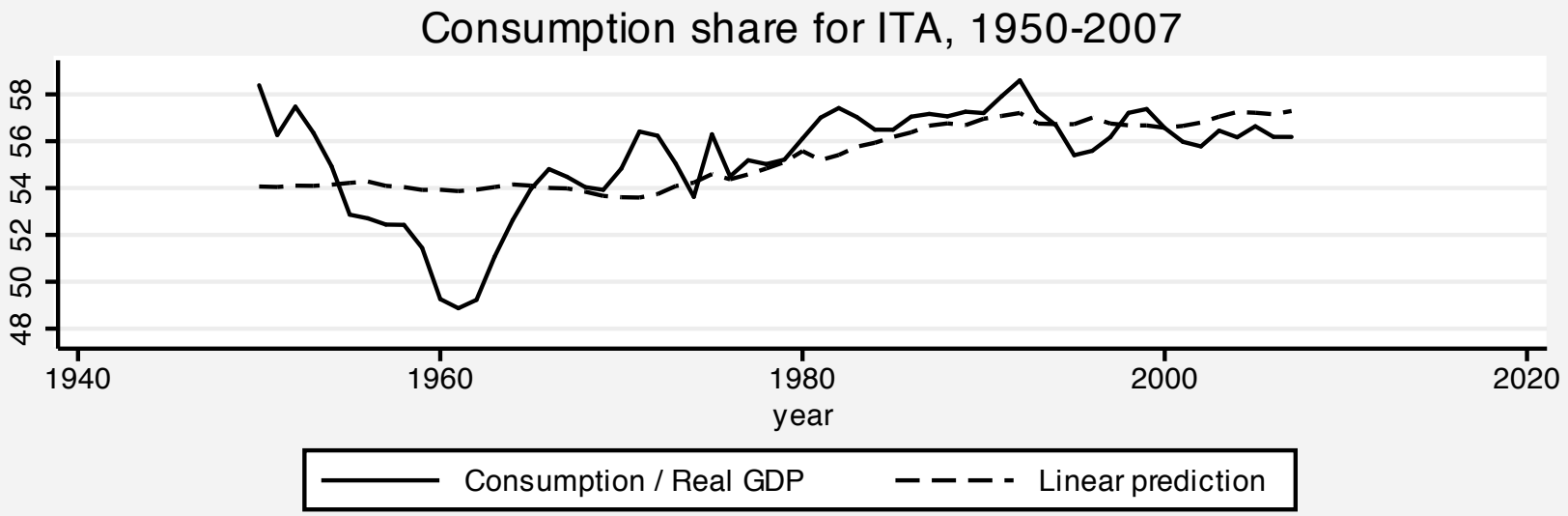
```
 . levelsof isocode, local(ctys)
`"BEL"´ `"DNK"´ `"ESP"´ `"FRA"´ `"GER"´ `"GRC"´ `"ITA"´ `"PRT"´
. foreach c of local ctys {
  2.                 qui regress kc openk pc xrat if isocode == "`c´"
  3.                 local rmse = string(`e(rmse)´, "%7.4f")
  4.                 qui predict double kchat`c´ if e(sample), xb
  5.                 tsline kc kchat`c´ if e(sample), scheme(s2mono) ///
>                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 FRA, 1960-2007
RMSE = 1.1744
55
56
57
58
59
60
1940
1960
1980
2000
2020
year
Consumption / Real GDP
Linear prediction

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(s2mono) ///
>                ti("Consumption share for `c´, 1950-2007")  ///
>                name(gr`c´, replace)
  3. }
. graph combine grFRA grITA, cols(1) saving(grFRA_ITA, replace)
(file grFRA_ITA.gph saved)
```


Consumption share for FRA, 1950-2007
55 56 57 58 59 60
1940
1960
1980
2000
2020
year
Consumption / Real GDP
Linear prediction


Consumption share for ITA, 1950-2007
48 50 52 54 56 58
1940
1960
1980
2000
2020
year
Consumption / Real GDP
Linear prediction

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 `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 13
syntax varname [if] [in]
return local varname `varlist'
tempvar new
marksample touse
quietly {
  count if `varlist' !=. & `touse'
  return scalar N = r(N)
  gen double `new' = sum(`varlist') if `touse'
  return scalar sum = `new'[_N]
  return scalar mean = return(sum) / return(N)
}
end
```

Examples of ado-file programming

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

For instance, the regression

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

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

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

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

```
 . type myregress.ado
*! myregress v1.0.0  CFBaum 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
```

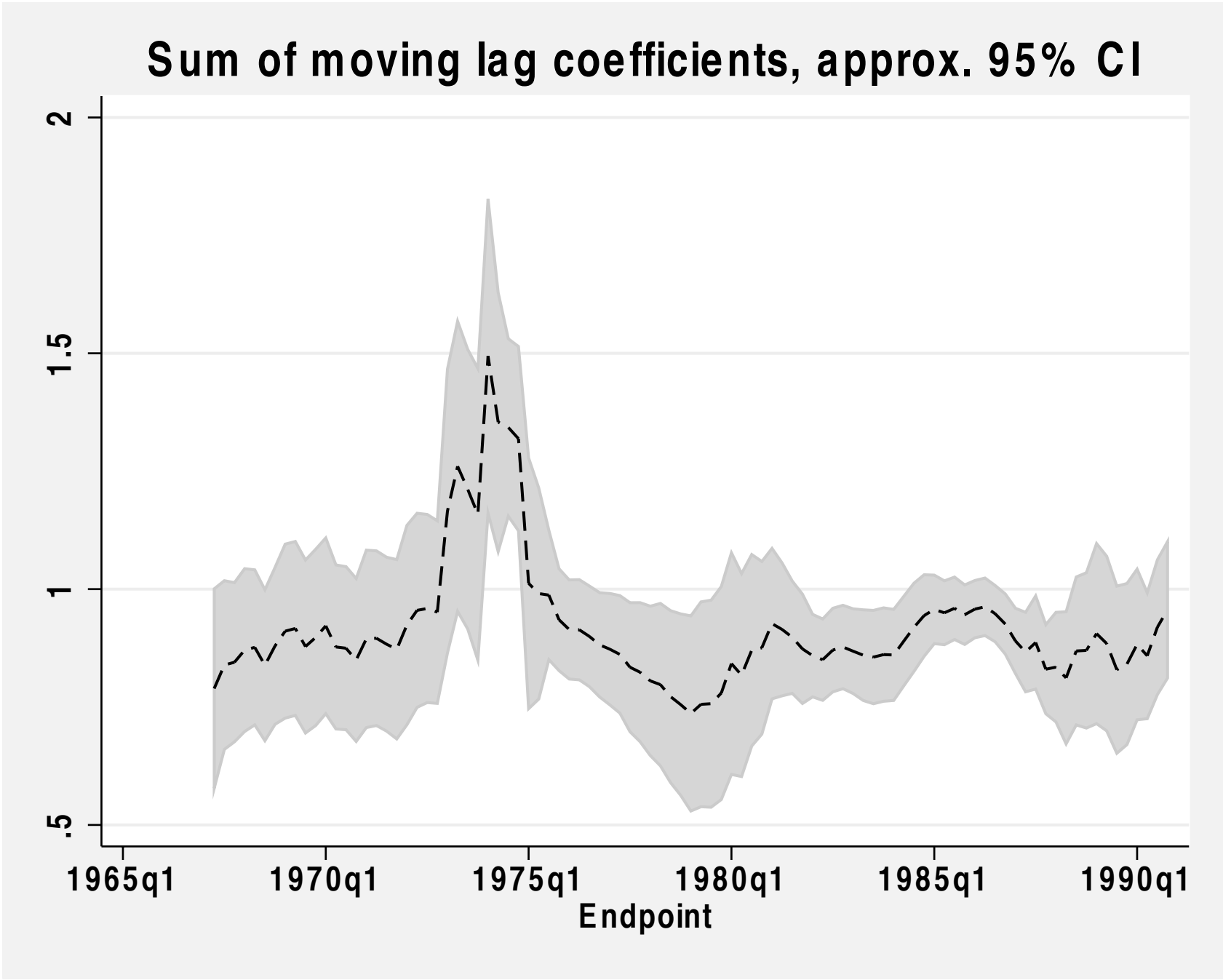
wpi	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
(1)	.9809968	.0082232	119.30	0.000	.9647067	.9972869

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

Maximum likelihood estimation

For many limited dependent models, Stata contains commands with "canned" likelihood functions which are as easy to use as `regress`. However, you may have to write your own likelihood evaluation routine if you are trying to solve a non-standard maximum likelihood estimation problem.

In Stata 13, the `mlexp` command allows you to specify some maximum likelihood problems in the 'substitutable expression' syntax. As this approach is somewhat restrictive in its applicability, we will not discuss it further.

A key resource for ML estimation is the book *Maximum Likelihood Estimation in Stata*, Gould, Pitblado and Poi, Stata Press: 4th ed., 2010. A good deal of this presentation is adapted from their excellent treatment of the subject, which I recommend that you buy if you are going to work with MLE in Stata.

To perform maximum likelihood estimation (MLE) in Stata using `ml`, you must write a short Stata program defining the likelihood function for your problem. In most cases, that program can be quite general and may be applied to a number of different model specifications without the need for modifying the program.

Let's consider the simplest use of MLE: a model that estimates a binomial probit equation, as implemented in official Stata by the `probit` command. We code our probit ML program as:

```
program myprobit_lf
  version 13
  args lnf xb
  quietly replace `lnf´ = ln(normal( `xb´ ))  if $ML_y1 == 1
  quietly replace `lnf´ = ln(normal( -`xb´ )) if $ML_y1 == 0
end
```

This program is suitable for ML estimation in the *linear form* or `lf` context. The local macro `lnf` contains the contribution to log-likelihood of each observation in the defined sample. As is generally the case with Stata's `generate` and `replace`, it is not necessary to loop over the observations. In the linear form context, the program need not sum up the log-likelihood.

Several programming constructs show up in this example. The `args` statement defines the program's *arguments*: `lnf`, the variable that will contain the value of log-likelihood for each observation, and `xb`, the linear form: a single variable that is the product of the "X matrix" and the current vector **b**. The arguments are local macros within the program.

The program replaces the values of `lnf` with the appropriate log-likelihood values, conditional on the value of `$ML_y1`: the first dependent variable or "y"-variable. Thus, the program may be applied to any 0–1 variable as a function of any set of X variables without modification.

Given the program—stored in the file `myprobit_lf.ado` on the `ADOPATH`—how do we execute it?

```
sysuse auto, clear
gen gpm = 1/mpg
ml model lf myprobit_lf  (foreign = price gpm displacement)
ml maximize
```

The `ml model` statement defines the context to be the linear form (`lf`), the likelihood evaluator to be `myprobit_lf`, and then specifies the model. The binary variable `foreign` is to be explained by the factors `price, gpm, displacement`, by default including a constant term in the relationship. The `ml model` command only defines the model: it does not estimate it. That is performed with the `ml maximize` command.

You can verify that this routine duplicates the results of applying `probit` to the same model. Note that our ML program produces estimation results in the same format as an official Stata command. It also stores its results in the `ereturn list`, so that postestimation commands such as `test` and `lincom` are available.

Of course, we need not write our own binomial probit. To understand how we might apply Stata's ML commands to a likelihood function of our own, we must establish some notation, and explain what the linear form context implies.

The log-likelihood function can be written as a function of variables and parameters:

$$\begin{aligned} \ell &= \ln L\{(\theta_{1j}, \theta_{2j}, \ldots, \theta_{Ej}; y_{1j}, y_{2j}, \ldots, y_{Dj}),\ j = 1, N\} \\ \theta_{ij} &= \mathbf{x}_{ij}\beta_i = \beta_{i0} + x_{ij1}\beta_{i1} + \cdots + x_{ijk}\beta_{ik} \end{aligned}$$

or in terms of the whole sample:

$$\begin{aligned} \ell &= \ln L(\theta_1, \theta_2, \ldots, \theta_E; \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D) \\ \theta_i &= \mathbf{X}_i\beta_i \end{aligned}$$

where we have D dependent variables, E equations (indexed by i) and the data matrix X_i for the i^{th} equation, containing N observations indexed by j.

In the special case where the log-likelihood contribution can be calculated separately for each observation and the sum of those contributions is the overall log-likelihood, the model is said to meet the *linear form restrictions*:

$$\begin{aligned} \ln \ell_j &= \ln \ell(\theta_{1j}, \theta_{2j}, \ldots, \theta_{Ej}; y_{1j}, y_{2j}, \ldots, y_{Dj}) \\ \ell &= \sum_{j=1}^{N} \ln \ell_j \end{aligned}$$

which greatly simplify the task of specifying the model. Nevertheless, when the linear form restrictions are not met, Stata provides three other contexts in which the full likelihood function (and possibly its derivatives) can be specified.

One of the more difficult concepts in Stata's MLE approach is the notion of ML *equations*. In the example above, we only specified a single equation:

```
(foreign = price gpm displacement)
```

which served to identify the dependent variable (`$ML_y1` to Stata) and the **X** variables in our binomial probit model.

Let's consider how we can implement estimation of a linear regression model via ML. In regression we seek to estimate not only the coefficient vector **b** but the error variance σ^2. The log-likelihood function for the linear regression model with normally distributed errors is:

$$\ln L = \sum_{j=1}^{N} [\ln \phi\{(y_j - x_j\beta)/\sigma\} - \ln \sigma]$$

with parameters β, σ to be estimated.

Writing the conditional mean of y for the j^{th} observation as μ_j,

$$\mu_j = E(y_j) = x_j\beta$$

we can rewrite the log-likelihood function as

$$\begin{aligned}
\theta_{1j} &= \mu_j = x_{1j}\beta_1 \\
\theta_{2j} &= \sigma_j = x_{2j}\beta_2 \\
\ln L &= \sum_{j=1}^{N} [\ln \phi\{(y_j - \theta_{1j})/\theta_{2j}\} - \ln \theta_{2j}]
\end{aligned}$$

This may seem like a lot of unneeded notation, but it makes clear the flexibility of the approach. By defining the linear regression problem as a two-equation ML problem, we may readily specify equations for both β and σ. In OLS regression with homoskedastic errors, we do not need to specify an equation for σ, a constant parameter; but the approach allows us to readily relax that assumption and consider an equation in which σ itself is modeled as varying over the data.

Given a program `mynormal_lf` to evaluate the likelihood of each observation—the individual terms within the summation—we can specify the model to be estimated with

```
ml model lf mynormal_lf  (y = equation for y) (equation for sigma)
```

In the homoskedastic linear regression case, this might look like

```
ml model lf mynormal_lf  (mpg = weight displacement) ()
```

where the trailing set of `()` merely indicate that nothing but a constant appears in the "equation" for σ. This `ml model` specification indicates that a regression of `mpg` on `weight` and `displacement` is to be fit, by default with a constant term.

We could also use the notation

```
ml model lf mynormal_lf (mpg = weight displacement) /sigma
```

where there is a constant parameter to be estimated.

But what does the program `mynormal_lf` contain?

```
program mynormal_lf
  version 13
  args lnf mu sigma
  quietly replace `lnf´ = ln(normalden( $ML_y1,  `mu´, `sigma´ ))
end
```

We can use Stata's `normalden(x,m,s)` function in this context. The three-parameter form of this Stata function returns the Normal[`m`, `s`] density associated with `x` divided by `s`. `m`, μ_j in the earlier notation, is the conditional mean, computed as $\mathbf{X}\beta$, while `s`, or σ, is the standard deviation. By specifying an "equation" for σ of `()`, we indicate that a single, constant parameter is to be estimated in that equation.

What if we wanted to estimate a heteroskedastic regression model, in which σ_j is considered a linear function of some variable(s)? We can use the same likelihood evaluator, but specify a non-trivial equation for σ:

```
ml model lf mynormal_lf ///
      (mpg = weight displacement) (price)
```

This would model $\sigma_j = \beta_4$ *price* $+ \beta_5$.

If we wanted σ to be proportional to `price`, we could use

```
ml model lf mynormal_lf ///
      (mu: mpg = weight displacement) (sigma: price, nocons)
```

which also labels the equations as `mu, sigma` rather than the default `eq1, eq2`.

A better approach to this likelihood evaluator program involves modeling σ in log space, allowing it to take on all values on the real line. The likelihood evaluator program becomes

```
program mynormal_lf2
  version 13
  args lnf mu lnsigma
  quietly replace `lnf´ = ///
   ln(normalden( $ML_y1,  `mu´, exp(`lnsigma´ )))
end
```

It may be invoked by

```
ml model lf mynormal_lf2 ///
      (mpg = weight displacement) /lnsigma, ///
      diparm(lnsigma, exp label("sigma"))
```

Where the `diparm( )` option presents the estimate of σ.

We have illustrated the simplest likelihood evaluator method: the linear form (`lf`) context. It should be used whenever possible, as it is not only easier to code (and less likely to code incorrectly) but more accurate. When it cannot be used—when the linear form restrictions are not met—you may use methods `d0`, `d1`, or `d2`.

Method `d0`, like `lf`, requires only that you code the log-likelihood function, but in its entirety rather than for a single observation. It is the least accurate and slowest ML method, but the easiest to use when method `lf` is not available.

Method `d1` requires that you code both the log-likelihood function and the vector of first derivatives, or gradients. It is more difficult than `d0`, as those derivatives must be derived analytically and coded, but is more accurate and faster than `d0` (but less accurate and slower than `lf`.

Method `d2` requires that you code the log-likelihood function, the vector of first derivatives and the matrix of second partial derivatives. It is the most difficult method to use, as those derivatives must be derived analytically and coded, but it is the most accurate and fastest method available. Unless you plan to use a ML program very extensively, you probably do not want to go to the trouble of writing a method `d2` likelihood evaluator.

Many of Stata's standard estimation features are readily available when writing ML programs.

You may estimate over a subsample with the standard `if` *exp* or `in` *range* qualifiers on the `ml model` statement.

The default variance-covariance matrix (`vce(oim)`) estimator is based on the inverse of the estimated Hessian, or information matrix, at convergence. That matrix is available when using the default Newton–Raphson optimization method, which relies upon estimated second derivatives of the log-likelihood function.

If any of the quasi-Newton methods are used, you may select the Outer Product of Gradients (`vce(opg)`) estimator of the variance-covariance matrix, which does not rely on a calculated Hessian. This may be especially helpful if you have a lengthy parameter vector. You can specify that the covariance matrix is based on the information matrix (`vce(oim)`) even with the quasi-Newton methods.

The standard heteroskedasticity-robust `vce` estimate is available by selecting the `vce(robust)` option (unless using method `d0`). Likewise, the cluster-robust covariance matrix may be selected, as in standard estimation, with `cluster(`*varname*`)`.

You may estimate a model subject to linear constraints using the standard `constraint` command and the `constraints( )` option on the `ml model` command.

You may specify weights on the `ml model` command, using the weights syntax applicable to any estimation command. If you specify `pweights` (probability weights) robust estimates of the variance-covariance matrix are implied.

You may use the `svy` option to indicate that the data have been `svyset`: that is, derived from a complex survey design.

A method `lf` likelihood evaluator program will look like:

```
program myprog
   version 13
   args lnf theta1 theta2 ...
   tempvar tmp1 tmp2 ...
   qui gen double `tmp1´ = ...
   qui replace `lnf´ = ...
end
```

`ml` places the name of each dependent variable specified in `ml model` in a global macro: `$ML_y1`, `$ML_y2`, and so on.

`ml` supplies a variable for each equation specified in `ml model` as `theta1`, `theta2`, etc. Those variables contain linear combinations of the explanatory variables and current coefficients of their respective equations. These variables must not be modified within the program.

If you need to compute any intermediate results within the program, use `tempvar`s, and declare them as `double`. If scalars are needed, define them with a `tempname` statement. Computation of components of the LLF is often convenient when it is a complicated expression.

Final results are saved in `` `lnf' ``: a double-precision variable that will contain the contributions to likelihood of each observation.

The linear form restrictions require that the individual observations in the dataset correspond to independent pieces of the log-likelihood function. They will be met for many ML problems, but are violated for problems involving panel data, fixed-effect logit models, and Cox regression.

Just as linear regression may be applied to many nonlinear models (e.g., the Cobb–Douglas production function), Stata's *linear form restrictions* do not hinder our estimation of a nonlinear model. We merely add equations to define components of the model. If we want to estimate

$$y_j = \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_3 x_{3j}^{\beta_4} + \beta_5 + \epsilon_j$$

with $\epsilon \sim N(0, \sigma^2)$, we can express the log-likelihood as

$$\begin{aligned}
\ln \ell_j &= \ln \phi\{(y_j - \theta_{1j} - \theta_{2j} x_{3j}^{\theta_{3j}})/\theta_{4j}\} - \ln \theta_{4j} \\
\theta_{1j} &= \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_5 \\
\theta_{2j} &= \beta_3 \\
\theta_{3j} &= \beta_4 \\
\theta_{4j} &= \sigma
\end{aligned}$$

The likelihood evaluator for this problem then becomes

```
program mynonlin_lf
  version 13
  args lnf theta1 theta2 theta3 sigma
  quietly replace `lnf´ = ln(normalden( $ML_y1, ///
       `theta1´+`theta2´*$X3^`theta3´, `sigma´ ))
end
```

This program evaluates the LLF using a *global macro*, `X3`, which must be defined to identify the Stata variable that is to play the role of x_3.

By making this reference a global macro, we avoid hard-coding the variable name, so that the same model can be fit on different data without altering the program.

We could invoke the program with

```
global X3 bp0
ml model lf mynonlin_lf (bp = age sex) /beta3 /beta4 /sigma
ml maximize
```

Thus, we may readily handle a nonlinear regression model in this context of the linear form restrictions, redefining the ML problem as one of four equations.

If we need to set starting values, we can do so with the `ml init` command. The `ml check` command is very useful in testing the likelihood evaluator program and ensuring that it is working properly. The `ml search` command is recommended to find appropriate starting values. The `ml query` command can be used to evaluate the progress of ML if there are difficulties achieving convergence.

Nonlinear least squares estimation

Besides the capabilities for maximum likelihood estimation of one or several equations via the `ml` suite of commands, Stata provides facilities for single-equation nonlinear least squares estimation with `nl` and the estimation of nonlinear systems of equations with `nlsur`.

In an earlier session, we discussed interactive use of these commands. We now describe how they may be used with a function evaluator program, which is quite similar to the likelihood function evaluators for `ml` that we just discussed.

If you want to use `nl` extensively for a particular problem, it makes sense to develop a *function evaluator program*. That program is quite similar to any Stata `ado`-file or `ml` program. It must be named `nl`*func*`.ado`, where *func* is a name of your choice: e.g., `nlces.ado` for a CES function evaluator.

The stylized function evaluator program contains:

```
program nlfunc
   version 13
   syntax varlist(min=n max=n) if, at(name)
 // extract vars from varlist
 // extract params as scalars from at matrix
 // fill in dependent variable with replace
end
```

As an example, this function evaluator implements estimation of a constant elasticity of substitution (CES) production function:

```
. type nlces.ado
*! nlces v1.0.0  CFBaum 20aug2013
program nlces
   version 13
   syntax varlist(numeric min=3 max=3) if, at(name)
   args logoutput K L
   tempname b0 rho delta
   tempvar kterm lterm
   scalar `b0´ = `at´[1, 1]
   scalar `rho´ = `at´[1, 2]
   scalar `delta´ = `at´[1, 3]
   gen double `kterm´ = `delta´ * `K´^( -(`rho´ )) `if´
   gen double `lterm´ = (1 - `delta´) *`L´^( -(`rho´ )) `if´
   replace `logoutput´ = `b0´ - 1 / `rho´ * ln( `kterm´ + `lterm´ ) `if´
end
```

To use the program `nlces`, call it with the `nl` command, but only include the unique part of its name, followed by `@`:

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

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

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

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

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

You could restrict analysis to a subsample with the *if exp* qualifier:

```
nl ces @ lnQ cap lab if industry==33, ...
```

You can also perform post-estimation commands, such as `nlcom`, to derive point and interval estimates of nonlinear combinations of the estimated parameters. In this case, we want to compute the elasticity of substitution, σ:

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

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

Note that the `nlsur` command estimates systems of *seemingly unrelated* nonlinear equations, just as `sureg` estimates systems of seemingly unrelated linear equations. In that context, `nlsur` cannot be used to estimate a system of simultaneous nonlinear equations. The `gmm` command, as we now discuss, could be used for that purpose, as could Stata's maximum likelihood commands (`ml`).

GMM function-evaluator programs

The GMM function evaluator program, or moment-evaluator program, is passed a *varlist* containing the moments to be evaluated for each observation. Your program replaces the elements of the *varlist* with the 'error part' of the moment conditions. For instance, if we were to solve an OLS regression problem with GMM we might write a moment-evaluator program as:

```
. program gmm_reg
  1.          version 13
  2.          syntax varlist if, at(name)
  3.          qui {
  4.                  tempvar xb
  5.                  gen double `xb´ = x1*`at´[1,1] + x2*`at´[1,2] + ///
>                 x3*`at´[1,3] + `at´[1,4] `if´
  6.                  replace `varlist´ = y - `xb´ `if´
  7.          }
  8. end
```

where we have specified that the regression has three explanatory variables and a constant term, with variable `y` as the dependent variable. The row vector `at()` contains the current values of the estimated parameters. The contents of *varlist* are replaced with the discrepancies, $y - X\beta$, defined by those parameters. A *varlist* is used as `gmm` can handle multiple-equation problems.

To perform the estimation (using the standard `auto` dataset), we specify the parameters and instruments to be used in the `gmm` command:

```
. gen y = price
. gen x1 = weight
. gen x2 = length
. gen x3 = turn
. gmm gmm_reg, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(weight length turn) onestep nolog
Final GMM criterion Q(b) =  2.43e-16
GMM estimation
Number of parameters =   4
Number of moments    =   4
Initial weight matrix: Unadjusted                     Number of obs   =        74
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	5.382135	1.719276	3.13	0.002	2.012415	8.751854
/b2	-66.17856	57.56738	-1.15	0.250	-179.0086	46.65143
/b3	-318.2055	171.6618	-1.85	0.064	-654.6564	18.24543
/b0	14967.64	6012.23	2.49	0.013	3183.881	26751.39

```
Instruments for equation 1: weight length turn _cons
```

This may seem unusual syntax, but we are just stating that we want to use the regressors as instruments for themselves in solving the GMM problem, as under the hypothesis of $E[u|X] = 0$, the appropriate moment conditions can be written as $EX'u = 0$.

Inspection of the parameters and their standard errors shows that these estimates match those from `regress, robust` for the same model. It is quite unnecessary to use GMM in this context, of course, but it illustrates the way in which you may set up a GMM problem.

To perform linear instrumental variables, we can use the same moment-evaluator program and merely alter the instrument list:

```
. webuse hsng2, clear
(1980 Census housing data)
. gen y = rent
. gen x1 = hsngval
. gen x2 = pcturban
. gen x3 = popden
. gmm gmm_reg, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban popden faminc reg2-reg4) onestep nolog
Final GMM criterion Q(b) =  150.8821
GMM estimation
Number of parameters =   4
Number of moments    =   7
Initial weight matrix: Unadjusted                 Number of obs   =       50
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	.0022538	.0006785	3.32	0.001	.000924	.0035836
/b2	.0281637	.5017214	0.06	0.955	-.9551922	1.01152
/b3	.0006083	.0012742	0.48	0.633	-.0018891	.0031057
/b0	122.6632	17.26189	7.11	0.000	88.83052	156.4959

```
Instruments for equation 1: pcturban popden faminc reg2 reg3 reg4 _cons
```

These estimates match those produced by `ivregress 2sls, robust`.

Let us consider solving a nonlinear estimation problem: a binomial probit model, using GMM rather than the usual ML estimator. The moment-evaluator program:

```
. program gmm_probit
  1.          version 13
  2.          syntax varlist if, at(name)
  3.          qui {
  4.                  tempvar xb
  5.                  gen double `xb´ = x1*`at´[1,1] + x2*`at´[1,2] + ///
>                           x3*`at´[1,3] + `at´[1,4] `if´
  6.                  replace `varlist´ = y - normal(`xb´)
  7.          }
  8. end
```

To perform the estimation, we specify the parameters and instruments to be used in the gmm command:

```
. webuse hsng2, clear
(1980 Census housing data)
. gen y = (region >= 3)
. gen x1 = hsngval
. gen x2 = pcturban
. gen x3 = popden
. gmm gmm_probit, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban hsngval popden) onestep nolog
Final GMM criterion Q(b) =  3.18e-21
GMM estimation
Number of parameters =   4
Number of moments    =   4
Initial weight matrix: Unadjusted                 Number of obs   =        50
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	.0000198	.0000146	1.35	0.177	-8.92e-06	.0000484
/b2	.0139055	.0177526	0.78	0.433	-.020889	.0487001
/b3	-.0003561	.0001142	-3.12	0.002	-.0005799	-.0001323
/b0	-1.136154	.9463889	-1.20	0.230	-2.991042	.7187345

```
Instruments for equation 1: pcturban hsngval popden _cons
```

Inspection of the parameters shows that these estimates are quite similar to those from `probit` for the same model. However, whereas `probit` requires the assumption of *i.i.d*. errors, GMM does not; the standard errors produced by `gmm` are robust to arbitrary heteroskedasticity.

As in the case of our linear regression estimation example, we can use the same moment-evaluator program to estimate an instrumental-variables probit model, similar to that estimated by `ivprobit`. Unlike that ML command, though, we need not make any distributional assumptions about the error process in order to use GMM.

```
. gmm gmm_probit, nequations(1) parameters(b1 b2 b3 b0) ///
>     instruments(pcturban popden rent hsnggrow) onestep nolog
Final GMM criterion Q(b) =  .0470836

GMM estimation

Number of parameters =   4
Number of moments    =   5
Initial weight matrix: Unadjusted                 Number of obs   =       50
```

	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
/b1	-6.25e-06	.0000203	-0.31	0.758	-.000046	.0000335
/b2	.0370542	.0333466	1.11	0.266	-.028304	.1024124
/b3	-.0014897	.0013724	-1.09	0.278	-.0041795	.0012001
/b0	-.538059	1.278787	-0.42	0.674	-3.044435	1.968317

```
Instruments for equation 1: pcturban popden rent hsnggrow _cons
```

Although the examples of gmm moment-evaluator programs we have shown here largely duplicate the functionality of existing Stata commands, they should illustrate that the general-purpose gmm command may be used to solve estimation problems not amenable to any existing commands, or indeed to a maximum-likelihood approach. In that sense, familiarity with gmm capabilities is likely to be quite helpful if you face challenging estimation problems in your research.

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 20aug2013
        program _gpct9010
        version 13
        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 20aug2013
        program _gpct9010
        version 13
        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.

Language syntax: Operators

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

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

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

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

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

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

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

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

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

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

creates a 3×3 matrix.

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

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

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

```
r2 = ( 1 \ 2 \ 3 )´
```

is a row vector.

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

```
r3 = r1, c1´
```

will produce a six-element row vector, and

```
c2 = r1´ \ c1
```

creates a six-element column vector.

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

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

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

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

Element-wise calculations and the colon operator

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

```
k = r1´ :* c1
```

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

Mata's colon operator is very powerful, in that it will work on nonconformable objects, or what Mata considers c-conformable objects. These include cases where two objects have the same number of rows (or the same number of columns), but one is a matrix and the other is a vector or scalar. For example:

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

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

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

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

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

Logical operators

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

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

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

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

Subscripting

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

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

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

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

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

Loop constructs

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

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

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

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

prints the integers 1 to 10 on separate lines.

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

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

do {
    statements
} while (*exp*)

which will execute the *statements* at least once.

Alternatively, you may use `while`:

while(*exp*) {
    statements
}

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

Mata conditional statements

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

if (*exp*) *statement*

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

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

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

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

can be written as

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

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

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

Element and organization types

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

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

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

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

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

Arguments, variables and returns

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

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

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

Data access

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

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

st_view( )

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

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

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

The Mata statements

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

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

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

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

creates matrix `X` containing those variables.

st_data( )

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

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

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

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

Using views to update Stata variables

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

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

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

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

Then compute the new content as:

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

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

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

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

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

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

Access to locals, globals, scalars and matrices

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

For local macros,

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

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

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

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

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

A simple Mata function

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

```
program define mysum3, rclass
   version 13
   syntax varlist(max=1) [if] [in]
   return local varname `varlist´
   marksample touse
   mata: m_mysum("`varlist´", "`touse´")
   return scalar N = N
   return scalar sum = sum
   return scalar mean = mu
   return scalar sd = sigma
end
```

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

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

```
version 13
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 13
   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 13
mata:
void centerv( string scalar varlist, ///
              string scalar newvarlist,
              string scalar touse)
{
  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ ., . ] = X :- mean(X)
}
end
```

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

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

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

Passing a function to Mata

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

The Stata code:

```
program centertrans, rclass
   version 13
   syntax varlist(numeric) [if] [in], ///
              GENerate(string) [TRans(string)] [DOUBLE]
   marksample touse
   quietly count if `touse'
   if `r(N)' == 0  error 2000
   local trops abs exp log sqrt
   if "`trans'" == "" {
     local trfn  "mf_iden"
    }
    else {
     local ntr : list posof "`trans'" in trops
     if !`ntr' {
     display as err "Error: trans must be chosen from `trops'"
     error 198
     }
    local trfn : "mf_`trans'"
    }
    foreach v of local varlist {
       confirm new var `generate'`trans'`v'
   }
   foreach v of local varlist {
       qui generate `double' `generate'`trans'`v' = .
       local newvars "`newvars' `generate'`trans'`v'"
   }
   mata: centertrans( "`varlist'", "`newvars'", &`trfn'(), "`touse'" )
end
```

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

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

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

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

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

To apply the function, we use

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

which applies the function referenced by `f` to the elements of the matrix `X`. The `Z` matrix is then demeaned as before.

The Mata code:

```
version 13
mata:
function mf_abs(x) return(abs(x))
function mf_exp(x) return(exp(x))
function mf_log(x) return(log(x))
function mf_sqrt(x) return(sqrt(x))
function mf_iden(x) return(x)

void centertrans( string scalar varlist, ///
                  string scalar newvarlist,
                  pointer(real scalar function) scalar f,
                  string scalar touse)
{
  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ , ] = (*f)(X)
  Z[ , ] = Z :- mean(Z)
}
end
```

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