STATA TECHNICAL BULLETIN

July 1994
STB-20

A publication to promote communication among Stata users

The *Stata Technical Bulletin* (ISSN 1097-8879) is published six times per year by Stata Corporation. Stata is a registered trademark of Stata Corporation.

Contents of this issue

an1.1 STB categories and insert codes

Inserts in the STB are presently categorized as follows:

General Categories:			
an	announcements	*ip*	instruction on programming
cc	communications & letters	*os*	operating system, hardware, & interprogram communication
dm	data management		
dt	data sets	*qs*	questions and suggestions
gr	graphics	*tt*	teaching
in	instruction	*zz*	not elsewhere classified
Statistical Categories:			
sbe	biostatistics & epidemiology	*srd*	robust methods & statistical diagnostics
sed	exploratory data analysis	*ssa*	survival analysis
sg	general statistics	*ssi*	simulation & random numbers
smv	multivariate analysis	*sss*	social science & psychometrics
snp	nonparametric methods	*sts*	time-series, econometrics
sqc	quality control	*sxd*	experimental design
sqv	analysis of qualitative variables	*szz*	not elsewhere classified

In addition, we have granted one other prefix, *crc*, to the manufacturers of Stata for their exclusive use.

an44.1 StataQuest disk enclosed (really)

Patricia Branton, Stata Corporation, FAX 409-696-4601

Although *an44* stated that a StataQuest disk would be included with STB-19 for those who subscribe with magnetic media, we forgot. The disk is included with this issue.

To enter StataQuest, put the diskette into the drive, select that drive, and type `go`.

There have been a few enhancements to the original StataQuest software. After entering StataQuest, if you select "Read StataQuest release 2 notes", you will see a list of the new features.

Reference

Loll, S. 1994. an44: StataQuest: Stata for teaching. *Stata Technical Bulletin* 19: 3–4.

an46 Stata and Stage now available for IBM PowerPC

Tim McGuire, Stata Corporation, FAX 409-696-4601

Stata 3.1 and the Stata Graphics Editor (Stage) are now available for the IBM PowerPC running AIX (IBM's version of Unix). The PowerPC chip is the result of a joint venture of Motorola, IBM, and Apple. IBM's version is available in both desktop and notebook computers, and supports Unix (in the form of AIX and the X Window standard (in the form of Motif.)

Stata 3.1 on the PowerPC is like Stata 3.1 on all other platforms; thus version 3.1 data sets, graphs, and ado-files from other computers can be used without translation. Pricing is the same as for all other Stata/Unix systems.

crc36 Clarification on analytic weights with linear regression

A popular request on the help line is to describe the effect of specifying `[aweight=`*exp*`]` with `regress` or `fit` in terms of transformation of the dependent and independent variables. The mechanical answer is that typing

```
. regress y x1 x2 [aweight=n]
```

is equivalent to estimating the model:

$$y_j\sqrt{n_j} = \beta_0\sqrt{n_j} + \beta_1 x_{1j}\sqrt{n_j} + \beta_2 x_{2j}\sqrt{n_j} + u_j\sqrt{n_j}$$

This regression will reproduce the coefficients and covariance matrix produced by the `aweight`ed regression. The mean square errors (estimate of the variance of the residuals) will, however, be different. The transformed regression reports s_t^2, an estimate of Var$(u_j\sqrt{n_j})$. The `aweighted` regression reports s_a^2, an estimate of Var$(u_j\sqrt{n_j}\sqrt{N/\sum_k n_k})$, where N is the number of observations. Thus,

$$s_a^2 = \frac{N}{\sum_k n_k} s_t^2 = \frac{s_t^2}{\bar{n}} \tag{1}$$

The logic for this adjustment is as follows: Consider the model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + u$$

Assume that, were this model estimated on individuals, Var$(u) = \sigma_u^2$, a constant. Assume that individual data is not available; what is available are averages $(\bar{y}_j, \bar{x}_{1j}, \bar{x}_{2j})$ for $j = 1, \dots, N$, and that each average is calculated over n_j observations. Then it is still true that

$$\bar{y}_j = \beta_0 + \beta_1 \bar{x}_{1j} + \beta_2 \bar{x}_{2j} + \bar{u}_j$$

where $\bar{u}_j$ is the average of n_j mean 0 variance σ_u^2 deviates and so itself has variance $\sigma_{\bar{u}}^2 = \sigma_u^2/n_j$. Thus, multiplying through by $\sqrt{n_j}$ produces

$$\bar{y}_j\sqrt{n_j} = \beta_0\sqrt{n_j} + \beta_1\bar{x}_{1j}\sqrt{n_j} + \beta_2\bar{x}_{2j}\sqrt{n_j} + \bar{u}_j\sqrt{n_j}$$

and Var$(\bar{u}_j\sqrt{n_j}) = \sigma_u^2$. The mean square error s_t^2 reported by estimating this transformed regression is an estimate of σ_u^2. Alternatively, the coefficients and covariance matrix could be obtained by `aweighted` `regress`. The only difference would be in the reported mean square error, which per equation 1, is $\sigma_u^2/\bar{n}$. On average, each observation in the data reflects the averages calculated over $\bar{n} = \sum_k n_k/N$ individuals, and thus this reported mean square error is the average variance of an observation in the data set. One can retrieve the estimate of σ_u^2 by multiplying the reported mean square error by $\bar{n}$.

More generally, `aweights` are used to solve general heteroskedasticity problems. In these cases, one has the model

$$y_j = \beta_0 + \beta_1 x_{1j} + \beta_2 x_{2j} + u_j$$

and the variance of u_j is thought to be proportional to a_j. If the variance is proportional to a_j, it is also proportional to αa_j, where α is any positive constant. Not quite arbitrarily, but with no loss of generality, let us choose $\alpha = \sum_k(1/a_k)/N$, the average value of the inverse of a_j. We can then write Var$(u_j) = k\alpha a_j\sigma^2$, where k is the constant of proportionality that is no longer a function of the scale of the weights.

Dividing this regression through by the $\sqrt{a_j}$,

$$y_j/\sqrt{a_j} = \beta_0/\sqrt{a_j} + \beta_1 x_{1j}/\sqrt{a_j} + \beta_2 x_{2j}/\sqrt{a_j} + u_j/\sqrt{a_j}$$

produces a model with Var$(u_j/\sqrt{a_j}) = k\alpha\sigma^2$, which is the constant part of Var(u_j). Notice in particular that this variance is a function of α, the average of the reciprocal weights; if the weights are scaled arbitrarily, then so is this variance.

We can also estimate this model by typing

```
. regress y x1 x2 [aweight=1/a]
```

This will produce the same estimates of the coefficients and covariance matrix; the reported mean square error is, per equation 1, $\left[N/\sum_k(1/a_k)\right]k\alpha\sigma^2 = k\sigma^2$. Note that this variance is independent of the scale of a_j.

dm19 | Merging raw data and dictionary files

Jonathan Nash, CS First Boston, FAX 212-318-0748

I maintain several Stata data sets of selected financial market data. These data are updated regularly by tapping the massive financial market data sets maintained by CS First Boston. The data from the First Boston data sets are converted to dictionary files and then merged (by date) into my existing Stata data sets.

Stata's `merge` command merges a `.dta` file into the current data set but it will not handle raw data or dictionary files. My `do-files` for updating my data sets contained code for infiling my dictionary files, sorting them by date, saving them to temporary data sets, using my existing data set, and—finally—merging the new data into the existing data.

I wrote `mergedct.ado` to avoid these steps. `mergedct` is just like Stata's `merge` command except `mergedct` combines the current data set with a raw or dictionary file rather than a `.dta`-file. The syntax for `mergedct` is

`mergedct` [*varlist*] `using` *filename* [, `automatic` `byvariable(#)` `nolabel` `u2(`*filename2*`)` `vlist(`*varlist*`)`]

Options

`mergedct` combines the options of `infile` and `merge`. The `automatic`, `byvariable(#)`, and `u2(`*filename2*`)` options have the same effect as the `infile` options `automatic`, `byvariable(#)`, and `using(`*filename2*`)`, respectively. The `nolabel` option has the same effect as the `merge` command's `nolabel` option.

The `vlist(`*varlist*`)` option is new. It is used to identify the variables when merging raw (non-dictionary) data files. The command

```
. mergedct using rawdata, vlist(x y z)
```

indicates that the raw data can be read using the following `infile` command:

```
. infile x y z using rawdata
```

Examples

Here's an example of merging a dictionary file. Note that `mergedct` does not require the disk data set to be sorted in advance.

```
. use dta, clear
. describe
Contains data from dta.dta
  Obs:    10 (max= 32766)
 Vars:     2 (max=    99)
Width:     6 (max=   200)
  1. order        int    %8.0g
  2. x            float  %9.0g
Sorted by:  order
. list
        order          x
  1.        1   1.605509
  2.        2  -1.363875
  3.        3   1.255479
  4.        4   .6902485
  5.        5   .7910749
  6.        6  -1.067126
  7.        7   1.471047
  8.        8  -1.971935
  9.        9   1.805707
 10.       10  -.5496167
. type indict1.dct
dictionary {
        int     order
        float   y
}
      10   .7764114
       1  -1.849471
       2   .4429037
       3   2.122414
       4   .3309246
       5  -.6871347
       6   1.272875
       7  -.4236486
       8  -1.741924
       9  -.4869484

. mergedct order using indict1
. list
```

```
          order          x          y    _merge
  1.          1   1.605509  -1.849471         3
  2.          2  -1.363875   .4429037         3
  3.          3   1.255479   2.122414         3
  4.          4   .6902485   .3309246         3
  5.          5   .7910749  -.6871347         3
  6.          6  -1.067126   1.272875         3
  7.          7   1.471047  -.4236486         3
  8.          8  -1.971935  -1.741924         3
  9.          9   1.805707  -.4869484         3
 10.         10  -.5496167   .7764114         3
```

mergedct can also merge the same data stored in a raw, rather than a dictionary, file:

```
. use dta, clear

. type inraw1.raw
      10   .7764114
       1  -1.849471
       2   .4429037
       3   2.122414
       4   .3309246
       5  -.6871347
       6   1.272875
       7  -.4236486
       8  -1.741924
       9  -.4869484

. mergedct order using inraw1, vlist(order y)

. list
          order          x          y    _merge
  1.          1   1.605509  -1.849471         3
  2.          2  -1.363875   .4429037         3
  3.          3   1.255479   2.122414         3
  4.          4   .6902485   .3309246         3
  5.          5   .7910749  -.6871347         3
  6.          6  -1.067126   1.272875         3
  7.          7   1.471047  -.4236486         3
  8.          8  -1.971935  -1.741924         3
  9.          9   1.805707  -.4869484         3
 10.         10  -.5496167   .7764114         3
```

mergedct can handle match merging as well:

```
. use dta, clear

. type indict2.dct
dictionary {
        int     z
}
     101
     102
     103
     104
     105
     106
     107
     108
     109
     110

. mergedct using indict2

. list
          order          x         z    _merge
  1.          1   1.605509       101         3
  2.          2  -1.363875       102         3
  3.          3   1.255479       103         3
  4.          4   .6902485       104         3
  5.          5   .7910749       105         3
  6.          6  -1.067126       106         3
  7.          7   1.471047       107         3
  8.          8  -1.971935       108         3
  9.          9   1.805707       109         3
 10.         10  -.5496167       110         3
```

dm20 Date functions

Alan Riley, Stata Corporation, FAX 409-696-4601

Since the publication in the STB of *A library of time series programs* (see sts7.3 below), Stata Corp. has received numerous requests for additional commands to manipulate dates. One request in particular piqued our interest and led to the development of the commands described below. The user in question maintains Stata data sets of daily financial data which are `collapse`d to create monthly averages and then used to generate financial forecasts. The Stata commands used to calculate these averages and forecasts are stored in `do`-files that are executed automatically (they are executed as `cron` jobs on a Unix system). The user wanted to make sure only full months of daily data were used; if the most recent month was not yet over, the user wanted to discard its data.

This is a more subtle problem than it first appears. How can we tell in a `do`-file if the last observation comes from the last day of the month? We wrote `lastday` to answer this question. But the problem isn't solved yet. The last day of the month may fall on a weekend. How can we tell if the last observation falls on the last business day of the month? `lastbday` addresses this question. Now, when a forecast is produced, how can we distinguish the historical observations from the projections? One way is to compare the observation date to today's date. Hence, the `today` command.

I hope the following commands will satisfy many of your requests. I am particularly interested in hearing from users about other date-related problems for which Stata may not yet provide a ready solution. Any other comments or suggestions that you have concerning these date-conversion programs are also appreciated.

New date commands

`downame` *dowvar*`,` `generate(`*dayvar*`)`

`namedow` *dayvar*`,` `generate(`*dowvar*`)`

`mnthname` *mvar*`,` `generate(`*mthvar*`)`

`namemnth` *mthvar*`,` `generate(`*mvar*`)`

`mdytodow` *mvar dvar yvar*`,` `generate(`*dowvar*`)`

`lastday` *mvar yvar*`,` `generate(`*dvar*`)`

`lastbday` *mvar yvar*`,` `generate(`*dvar*`)`

`ystrday` *mvar dvar yvar*`,` `generate(`*mvar dvar yvar*`)`

`today,` `generate(`*mvar dvar yvar*`)`

where

dayvar	string variable containing day (Sunday–Saturday)
mthvar	string variable containing month (January–December)

dowvar, dvar, mvar, and *yvar* are defined as in [5d] dates.

These commands perform various date conversions. `downame` converts a numeric day of week to the corresponding name. `namedow` performs the opposite conversion. Likewise, `mnthname` converts a numeric month to the corresponding name, and `namemnth` performs the opposite conversion. `mdytodow` calculates the numeric day of the week from the month, day, and year. `lastday` calculates the last calendar day for a given month and year, while `lastbday` calculates the last business day for a particular month and year. ("Business day" is defined as a day from Monday through Friday. Holidays are not taken into account).

`ystrday` calculates the previous month, day, and year given any month, day, and year. `today` simply reads the `S_DATE` global macro (see [2] macros) and generates variables containing the current month, day, and year.

Example

The following example shows the use of a few of the commands on a short artificial data set.

```
. list

             m          y
  1.         2       1992
  2.         2       1994
  3.         4       1994
  4.         7       1994
  5.        12       1994

. lastbday m y, generate(lbday)

. list

             m          y     lbday
  1.         2       1992        28
  2.         2       1994        28
  3.         4       1994        29
  4.         7       1994        29
  5.        12       1994        30

. mdytodow m lbday y, gen(dow)

. list

             m          y     lbday       dow
  1.         2       1992        28      Fri.
  2.         2       1994        28      Mon.
  3.         4       1994        29      Fri.
  4.         7       1994        29      Fri.
  5.        12       1994        30      Fri.

. list, nolabel

             m          y     lbday       dow
  1.         2       1992        28         5
  2.         2       1994        28         1
  3.         4       1994        29         5
  4.         7       1994        29         5
  5.        12       1994        30         5

. describe

Contains data
  Obs:     5 (max=  5088)
 Vars:     4 (max=    99)
Width:    12 (max=   200)
  1. m           float  %9.0g
  2. y           float  %9.0g
  3. lbday       int    %8.0g
  4. dow         int    %8.0g    Dayslab
Sorted by:
```

Note that the new variable `dow` is an integer, but has been given the label `Dayslab`.

Each of these date commands may also be used in immediate form. To use the command in immediate form, simply type the name of the command along with the numbers or strings for the day, month, or year you want converted. The command will display its results to the screen as well as saving them in the global macros `S_1`–`S_4`.

Example

```
. lastday 2 1992
29
Saturday, February 29, 1992

. disp_s
S_1:       29
S_2:       Saturday
S_3:       Sat
S_4:       Sat.

. today
July 1, 1994
```

```
. disp_s
S_1:      7
S_2:      1
S_3:      1994
S_4:      Sat.
```

Note that `today` did not mistakenly put “Sat.” in `S_4`. `today` happens to use only `S_1`–`S_3`, leaving `S_4` from the `lastday` command.

ip6 Storing variables in vectors and matrices

Ken Heinecke, Federal Reserve Bank of Kansas City, FAX 816-881-2199

`mkmat` takes the variables listed in *varlist* and stores them in column vectors; that is, $N \times 1$ matrices where $N =$ `_N`, the number of observations in the data set. The syntax of `mkmat` is

`mkmat` *varlist* [`if` *exp*] [`in` *range*] [, `matrix(`*matname*`)`]

If the `matrix()` option is specified, the vectors are also combined in a matrix.

Discussion

Although it is possible to load variables into a matrix using the `matrix accum` command, programmers may find it more convenient to work with the variables in their data sets as vectors instead of as cross products. `mkmat` allows the user a simple way to load specific variables into matrices in Stata’s memory.

Example

```
. describe
Contains data from test.dta
  Obs:    10 (max=  2562)
 Vars:     3 (max=   200)
Width:    12 (max=   402)
  1. x           float  %9.0g
  2. y           float  %9.0g
  3. z           float  %9.0g
Sorted by:
. list
             x          y          z
  1.         1         10          2
  2.         2          9          4
  3.         3          8          3
  4.         4          7          5
  5.         5          6          7
  6.         6          5          6
  7.         7          4          8
  8.         8          3         10
  9.         9          2          1
 10.        10          1          9
. mkmat y z
. matrix list y
y[10,1]
      y
 r1  10
 r2   9
 r3   8
 r4   7
 r5   6
 r6   5
 r7   4
 r8   3
 r9   2
r10   1
. matrix list z
```

```
z[10,1]
      z
 r1   2
 r2   4
 r3   3
 r4   5
 r5   7
 r6   6
 r7   8
 r8  10
 r9   1
r10   9
```

The variables can be restricted using `if` and `in` subcommands as well.

```
. mkmat x if z<9 in 7/l

. mat l x

x[2,1]
     x
r1   7
r2   9
```

Note that `mkmat` uses the variable name to name the single column in the vector. This feature guarantees that the variable name will be carried along in any additional matrix calculations. This feature is also useful when vectors are combined in a general matrix.

```
. matrix drop _all

. mkmat x y z, matrix(xyzmat)

. matrix dir
       xyzmat[10,3]
            z[10,1]
            y[10,1]
            x[10,1]

. matrix list xyzmat

xyzmat[10,3]
      x   y   z
 r1   1  10   2
 r2   2   9   4
 r3   3   8   3
 r4   4   7   5
 r5   5   6   7
 r6   6   5   6
 r7   7   4   8
 r8   8   3  10
 r9   9   2   1
r10  10   1   9
```

Caveats

The size of any matrix will be restricted by your `matsize` specification. A variable can have a maximum of 399 observations under Unix and Intercooled versions of Stata and 40 for regular versions of Stata. Variables containing more data points will not fit into a single vector.

Finally, if one of the variables has missing values, you will receive an error message and no matrices will be created.

```
. matrix drop _all

. replace y = . in 5
(1 real change made, 1 to missing)

. mkmat x y z
matrix y would have missing values
r(504);

. matrix dir

.
```

This problem can be taken care of by restricting the matrix to nonmissing values.

ip6.1 Data and matrices

William Gould, Stata Corporation, FAX 409-696-4601

Heinecke's `mkmat` program (*ip6*) provides a useful addition to Stata's matrix commands. The addition is so useful, in fact, that you may wonder how it was ever omitted from Stata. Indeed, we must admit that a popular question among Stata's matrix-programming language users is how to create data matrices.

Heinecke's program provides a solution, but it is a solution that will work only with small data set sizes. Stata limits matrices to being no more than matsize $\times$ matsize which, by default, means 40×40 and, even with Intercooled Stata, means no more than 400×400. Such limits appear to contradict Stata's claims of being able to process large data sets. By limiting Stata's matrix capabilities to matsize $\times$ matsize, has not Stata's matrix language itself been limited to data sets no larger than matsize? It would certainly appear so; in the simple matrix calculation for regression coefficients $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, $\mathbf{X}$ is an $n \times k$ matrix (n being the number of observations and k the number of variables) and, given the matsize constraint, n must certainly be less than 400.

Our answer is as follows: Yes, $\mathbf{X}$ is limited in the way stated but note that $\mathbf{X}'\mathbf{X}$ is a mere $k \times k$ matrix and, similarly, $\mathbf{X}'\mathbf{y}$ only $k \times 1$. Both these matrices are well within Stata's matrix-handling capabilities and Stata's `matrix accum` command (see [6m] accum) can directly create both of them.

Moreover, even if Stata could hold the $n \times k$ matrix $\mathbf{X}$, it would still be more efficient to use `matrix accum` to form $\mathbf{X}'\mathbf{X}$. $\mathbf{X}'\mathbf{X}$, interpreted literally, says to load a copy of the data, transpose it, load a second copy of the data, and then form the matrix product. Thus, two copies of the data occupy memory in addition to the original copy Stata already had available (and from which `matrix accum` could directly form the result with no additional memory use). For small n, the inefficiency is not important but, for large n, the inefficiency can be such as to actually make the calculation infeasible. (For instance, with $n = 12{,}000$ and $k = 6$, the additional memory use is 1,125K bytes.)

More generally, matrices in statistical applications tend to have dimension $k \times k$, $n \times k$, and $n \times n$, with k small and n large. Terms dealing with the data are of the generic form $\mathbf{X}'_{k_1 \times n}\mathbf{W}_{n\times n}\mathbf{Z}_{n\times k_2}$. ($\mathbf{X}'\mathbf{X}$ fits the generic form with $\mathbf{X} = \mathbf{X}$, $\mathbf{W} = \mathbf{I}$, and $\mathbf{Z} = \mathbf{X}$.) Matrix programming languages are not capable of dealing with the deceivingly simple calculation $\mathbf{X}'\mathbf{W}\mathbf{Z}$ because of the staggering size of the $\mathbf{W}$ matrix. For $n = 12{,}000$, storing $\mathbf{W}$ requires a little more than a gigabyte of memory. In statistical formulas, however, $\mathbf{W}$ is given by formula and, in fact, never needs to be stored in its entirety. Exploitation of this fact is all that is needed to resurrect the use of a matrix programming language in statistical applications. Matrix programming languages may be inefficient because of copious memory use, but in statistical applications, the inefficiency is minor for matrices of size $k \times k$ or smaller. Our design of the various `matrix accum` commands allow calculating terms of the form $\mathbf{X}'\mathbf{W}\mathbf{Z}$ and this one feature, we have found, is all that is necessary to allow efficient and robust use of matrix languages.

Programs for creating data matrices such as that offered by Heinecke are useful for pedagogical purposes and, in addition, I can imagine myself using it in some specific application where Stata's matsize constraint is not binding; it seems so natural. On the other hand, it is important that general tools not be implemented by forming data matrices because such tools will be drastically limited in terms of the data set size. Coding the problem in terms of the various `matrix accum` commands is admittedly more tedious but, by abolishing data matrices from your programs, you will produce tools suitable for use on large data sets.

os14 A program to format raw data files

Phillip Swagel, Department of Economics, Northwestern University

Stata can easily read raw data from ASCII files as long as the data are stored rectangularly. For example, the file

```
11 12 13
21 22 23
31 32 33
```

can be read by typing `infile x1 x2 x3 using` *filename*. In fact, this `infile` command will work even if the data are stored in the following arrangement:

```
11 12 13
21 22

23 31 32 33
```

Stata would have problems, however, if the same data arrangement appeared in a dictionary file:

```
. type in1.dct
dictionary {
     int      x1
     int      x2
     int      x3
}
11 12 13
21 22

23 31 32 33

. infile using in1

dictionary
        int     x1
        int     x2
        int     x3

(4 observations read)

. list

           x1        x2        x3
  1.       11        12        13
  2.       21        22         .
  3.        .         .         .
  4.       23        31        32
```

Stata's dictionary files are the preferred form for storing and documenting raw data. The dictionary subcommands can handle most kinds of formatted data including multi-line records and data sets without carriage returns ([5d] infile). Nonetheless, Murphy's law guarantees that you will occasionally confront data sets that confound Stata's dictionary capabilities. More commonly, you will have a data set that Stata's dictionary features can handle but only with difficulty. Clearly, life would be simpler if all raw data sets were rectangular, as in the first example.

I have written a C program called `block` that makes my life simpler. `block` takes an arbitrary ASCII file as input and produces as output the same information arrayed rectangularly. The following example illustrates how to use `block`.

```
C:> type in1
11 12 13
21 22

23 31 32 33

C:> block

Name of the input file: in1

Name of the output file: out1

Number of columns: 3
...
Read in   9  fields from in1
Wrote out 3  rows of 3 columns to out1

C:> type out1
11 12 13
21 22 23
31 32 33
```

`block` handles non-rectangular data gracefully:

```
C:> type in2
11 12 13
21 22

23 31 32 33 44

C:> block

Name of the input file: in2

Name of the output file: out2

Number of columns: 3
...
Read in   10  fields from in2
Wrote out 3  rows of 3 columns to out2

WARNING:  last row not complete.

C:> type out2
11 12 13
21 22 23
31 32 33
44
```

Both the source (`block.c`) and a DOS executable (`block.exe`) are available on the STB-20 diskette. Unix users can modify `block.c` and recompile it if they wish, although there are already several tools in Unix that provide the same service as `block`. DOS users who have purchased DOS versions of Unix utilities may also have tools that replicate `block`. For me, `block` is a simple, special-purpose tool. It does one job easily and well; it's nice to have when you need it.

sg25 Interaction expansion

William Gould, Stata Corporation, FAX 409-696-4601

The syntax of `xi` is

`xi` *term(s)*

`xi:` *any_stata_command varlist_with_terms* ...

where a *term* is of the form:

`i.`*varname*	or	`I.`*varname*
`i.`$varname_1$`*i.`$varname_2$		`I.`$varname_1$`*I.`$varname_2$
`i.`$varname_1$`*`$varname_3$		`I.`$varname_1$`*`$varname_3$
`i.`$varname_1$`|`$varname_3$		`I.`$varname_1$`|`$varname_3$

varname, $varname_1$, and $varname_2$ denote categorical variables and may be numeric or string. $varname_3$ denotes a continuous, numeric variable.

`xi` expands terms containing categorical variables into dummy variable sets by creating new variables and, in the second syntax (`xi:` *any_stata_command*) executes the specified command with the expanded terms.

Background

The terms continuous, categorical, and indicator or dummy variables are used below. Continuous variables are variables that measure something—such as height or weight—and at least conceptually can take on any real number over some range. Categorical variables, on the other hand, take on a finite number of values each denoting membership in a subclass, for example excellent, good, and poor—which might be coded 0, 1, 2 or 1, 2, 3 or even "Exc," "Good," and "Poor." An indicator or dummy variable—the terms are used interchangeably—is a special type of two-valued categorical variable that contains values 0, denoting false, and 1, denoting true. The information contained in any k-valued categorical variable can be equally well represented by k indicator variables. Instead of a single variable recording values representing excellent, good, and poor, one can have three indicator variables, the first indicating the truth or falseness of "result is excellent," the second "result is good," and the third "result is poor."

`xi` provides a convenient way to convert categorical variables to dummy or indicator variables when estimating a model (say with `regress`, `logistic`, etc.).

For instance, assume the categorical variable `agegrp` contains 1 for ages 20–24, 2 for ages 25–39, and 3 for ages 40–44. (There is no one over 44 in our data.) As it stands, `agegrp` would be a poor candidate for inclusion in a model even if one thought age affected the outcome. It would be poor because the coding would force the restriction that the effect of being in the second age group must be twice the effect of being in the first and, similarly, the effect of being in the third must be three times the first. That is, if one estimated the model,

$$y = \beta_0 + \beta_1 \texttt{agegrp} + X\beta_2$$

the effect of being in the first age group is β_1, the second $2\beta_1$, and the third $3\beta_1$. If the coding 1, 2, 3 is arbitrary, we could just as well have coded the age groups 1, 4, and 9, and the effects would now be β_1, $4\beta_1$, and $9\beta_1$.

The solution to this arbitrariness is to convert the categorical variable `agegrp` to a set of indicator variables a_1, a_2, and a_3, where a_i is 1 if the individual is a member of the ith age group and 0 otherwise. We can then estimate the model:

$$y = \beta_0 + \beta_{11}a_1 + \beta_{12}a_2 + \beta_{13}a_3 + X\beta_2$$

The effect of being in age group 1 is now β_{11}; 2, β_{12}; and 3, β_{13}; and these results are independent of our (arbitrary) coding. The only difficulty at this point is that the model is unidentified in the sense that there are an infinite number of $(\beta_0, \beta_{11}, \beta_{12}, \beta_{13})$ that fit the data equally well.

To see this, pretend $(\beta_0, \beta_{11}, \beta_{12}, \beta_{13}) = (1, 1, 3, 4)$. Then the predicted values of y for the various age groups are

$$y = \begin{cases} 1 + 1 + X\beta_2 = 2 + X\beta_2 & \text{(age group 1)} \\ 1 + 3 + X\beta_2 = 4 + X\beta_2 & \text{(age group 2)} \\ 1 + 4 + X\beta_2 = 5 + X\beta_2 & \text{(age group 3)} \end{cases}$$

Now pretend $(\beta_0, \beta_{11}, \beta_{12}, \beta_{13}) = (2, 0, 2, 3)$. Then the predicted values of y are

$$y = \begin{cases} 2 + 0 + X\beta_2 = 2 + X\beta_2 & \text{(age group 1)} \\ 2 + 2 + X\beta_2 = 4 + X\beta_2 & \text{(age group 2)} \\ 2 + 3 + X\beta_2 = 5 + X\beta_2 & \text{(age group 3)} \end{cases}$$

These two sets of predictions are indistinguishable: for age group 1, $y = 2 + X\beta_2$ regardless of which coefficient vector is used, and similarly for age groups 2 and 3. This arises because we have 3 equations and 4 unknowns. Any solution is as good as any other and, for our purposes, we merely need to choose one of them. The popular selection method is to set the coefficient on the first indicator variable to 0 (as we have done in our second coefficient vector). This is equivalent to estimating the model:

$$y = \beta_0 + \beta_{12}a_2 + \beta_{13}a_3 + X\beta_2$$

How one selects a particular coefficient vector (identifies the model) does not matter. It does, however, affect the *interpretation* of the coefficients.

For instance, we could just as well choose to omit the second group. In our artificial example, this would yield $(\beta_0, \beta_{11}, \beta_{12}, \beta_{13}) = (4, -2, 0, 1)$ instead of $(2, 0, 2, 3)$. These coefficient vectors are the same in the sense that,

$$y = \begin{cases} 2 + 0 + X\beta_2 = 2 + X\beta_2 = 4 - 2 + X\beta_2 & \text{(age group 1)} \\ 2 + 2 + X\beta_2 = 4 + X\beta_2 = 4 + 0 + X\beta_2 & \text{(age group 2)} \\ 2 + 3 + X\beta_2 = 5 + X\beta_2 = 4 + 1 + X\beta_2 & \text{(age group 3)} \end{cases}$$

but what does it mean that β_{13} can just as well be 3 or 1? We obtain $\beta_{13} = 3$ when we set $\beta_{11} = 0$, and so $\beta_{13} = \beta_{13} - \beta_{11}$ and β_{13} measures the difference between age groups 3 and 1.

In the second case, we obtain $\beta_{13} = 1$ when we set $\beta_{12} = 0$, so $\beta_{13} - \beta_{12} = 1$ and β_{13} measures the difference between age groups 3 and 2. There is no inconsistency. According to our $\beta_{12} = 0$ model, the difference between age groups 3 and 1 is $\beta_{13} - \beta_{11} = 1 - (-2) = 3$, exactly the same result we got in the $\beta_{11} = 0$ model.

The issue of interpretation, however, is important because it can affect the way one discusses results. Imagine you are studying recovery after a coronary bypass operation. Assume the age groups are (1) children under 13 (you have 2 of them), (2) young adults under 25 (you have a handful of them), (3) adults under 46 (of which you have more yet), (4) mature adults under 56, (5) older adults under 65, and (6) elder adults. You follow the prescription of omitting the first group, so all of your results are reported relative to children under 13. While there is nothing statistically wrong with this, readers will be suspicious when you make statements like, "compared to young children, older and elder adults ...". Moreover, it is likely that you will have to end each statement with "although results are not statistically significant" because you have only 2 children in your comparison group. Of course, even with results reported in this way, you can do reasonable comparisons (say to mature adults), but you will have to do extra work to perform the appropriate linear hypothesis test using Stata's `test` command.

In this case, it would be better if you forced the omitted group to be more reasonable, such as mature adults. There is, however, a generic rule for automatic comparison group selection that, while less popular, tends to work better than the omit-the-first-group rule. That rule is to omit the most prevalent group. The most prevalent is usually a reasonable baseline.

In any case, the prescription for categorical variables is

1. Convert each k-valued categorical variable to k indicator variables.
2. Drop one of the k indicator variables; any one will do but dropping the first is popular, dropping the most prevalent is probably better in terms of having the computer guess at a reasonable interpretation, and dropping a specified one often eases interpretation the most.

3. Estimate the model on the remaining $k-1$ indicator variables.

It is this procedure that xi automates.

Using xi: Overview

xi provides a convenient way to include dummy or indicator variables when estimating a model (say with regress, logistic, etc.). For instance, assume the categorical variable agegrp contains 1 for ages 20–24, 2 for ages 25–39, 3 for ages 40–44, etc. Typing

```
. xi: logistic outcome weight i.agegrp bp
```

estimates a logistic regression of outcome on weight, dummies for each agegrp category, and bp. That is, xi searches out and expands terms starting with "i." but leaves the other variables alone. xi will expand both numeric and string categorical variables, so if you had a string variable race containing "white," "black," and "other," typing

```
. xi: logistic outcome weight bp i.agegrp i.race
```

would include indicator variables for the race group as well.

The i. indicator variables xi expands may appear anywhere in the varlist, so

```
. xi: logistic outcome i.agegrp weight i.race bp
```

would estimate the same model.

You can also create interactions of categorical variables; typing

```
xi: logistic outcome weight bp i.agegrp*i.race
```

estimates a model including indicator variables for all agegrp and race combinations.

You can interact dummy variables with continuous variables:

```
xi: logistic outcome bp i.agegrp*weight i.race
```

And, of course, you can include multiple interactions:

```
xi: logistic outcome bp i.agegrp*weight i.agegrp*i.race
```

We will now back up and consider each of xi's features in detail.

Indicator variables for simple effects

When you type 'i.*varname*', xi internally tabulates varname (which may be a string or a numeric variable) and creates indicator (dummy) variables for each observed value, omitting the indicator for the smallest value. For instance, say agegrp takes on the values 1, 2, 3, and 4. Typing

```
xi: logistic outcome i.agegrp
```

creates indicator variables named Iagegr_2, Iagegr_3, and Iagegr_4. (xi chooses the names and tries to make them readable; xi guarantees that the names are unique.) The expanded logistic model then is

```
. logistic outcome Iagegr_2 Iagegr_3 Iagegr_4
```

Afterwards, you can drop the new variables xi leaves behind by typing 'drop I*' (note capitalization).

xi provides the following features when you type 'i.*varname*':

1. *varname* may be string or numeric.
2. Dummy variables are created automatically.
3. By default, the dummy-variable set is identified by dropping the dummy corresponding to the smallest value of the variable (how to specify otherwise is discussed below).
4. The new dummy variables are left in your data set. You can drop them by typing 'drop I*'. You do not have to do this; each time you use the xi prefix or command, any previously created automatically generated dummies are dropped and new ones created.
5. The new dummy variables have variable labels so you can determine to what they correspond by typing 'describe' or 'describe I*'.
6. xi may be used with any Stata command (not just logistic).

Controlling the omitted dummy

By default, `i.`*varname* omits the dummy corresponding to the smallest value of *varname*; in the case of a string variable, this is interpreted as dropping the first in an alphabetical, case-sensitive sort. `xi` provides two alternatives to dropping the first: `xi` will drop the dummy corresponding to the most prevalent value of *varname* or `xi` will let you choose the particular dummy to be dropped.

To change `xi`'s behavior to dropping the most prevalent, you type,

```
. global S_XIMODE "prevalent"
```

although whether you type "prevalent" inside the quotes or "yes" or anything else does not matter. You need type this command only once per session and, once typed, it affects the expansion of all categorical variables. If, during a session, you want to change the behavior back to the default drop-the-first rule, you type

```
. global S_XIMODE
```

Once you set `S_XIMODE`, `i.`*varname* omits the dummy corresponding to the most prevalent value of *varname*. Thus, the coefficients on the dummies have the interpretation of change from the most prevalent group. For example,

```
. global S_XIMODE "prevalent"
. xi: regress y i.agegrp
```

might create `Iagegr_1` through `Iagegr_4` and would result in `Iagegr_2` being omitted if `agegr` = 2 is most common (as opposed to the default dropping of `Iagegr_1`). The model is then:

$$\text{y} = b_0 + b_1\,\texttt{Iagegr\_1} + b_3\,\texttt{Iagegr\_3} + b_4\,\texttt{Iagegr\_4} + u$$

Then,

Predicted `y` for `agegrp` $1 = b_0 + b_1$	Predicted `y` for `agegrp` $3 = b_0 + b_3$
Predicted `y` for `agegrp` $2 = b_0$	Predicted `y` for `agegrp` $4 = b_0 + b_4$

Thus, the model's reported t or z statistics are for a test of whether each group is different from the most prevalent group.

Perhaps you wish to omit the dummy for `agegrp` 3 instead. Whether you have set the global macro `S_XIMODE` or not, you do this by creating a global macro with the same name of the variable containing "`xi omit` *value*". In this case:

```
. global agegrp "xi omit 3"
```

Now when you type

```
 . xi: regress y i.agegrp
```

`Iagegr_3` will be omitted and you will estimate the model:

$$\text{y} = b_0' + b_1'\,\texttt{Iagegr\_1} + b_2'\,\texttt{Iagegr\_2} + b_4'\,\texttt{Iagegr\_4} + u$$

Later, if you want to return to the default omission, you type

```
. global agegrp
```

thus clearing the macro.

In summary, `i.`*varname* omits the first group by default but if you define

```
. global S_XIMODE "prevalent"
```

then the default behavior changes to that of dropping the most prevalent group. Either way, if you define a macro of the form

```
. global varname "xi omit #"
```

of, if *varname* is a string,

```
. global varname "xi omit string-literal"
```

then the specified value will be omitted.

Examples:	`. global agegrp "xi omit 3"`	
	`. global race "xi omit White"`	(for `race` a string variable)
	`. global agegrp`	(to restore default for `agegrp`)

Categorical variable interactions

`i.`*varname*$_1$`*i.`*varname*$_2$ creates the dummy variables associated with the interaction of the categorical variables *varname*$_1$ and *varname*$_2$. The identification rules—which categories are omitted—are the same as for `i.`*varname*. For instance, assume `agegrp` takes on four values and `race` takes on three values. Typing,

```
. xi: regress y i.agegrp*i.race
```

results in the model:

$$\begin{array}{ll}
y = a + b_2\,\texttt{Iagegr\_2} + b_3\,\texttt{Iagegr\_3} + b_4\,\texttt{Iagegr\_4} & (\texttt{agegrp}\text{ dummies}) \\
\quad +c_2\,\texttt{Irace\_2} + c_3\,\texttt{Irace\_3} & (\texttt{race}\text{ dummies}) \\
\quad +d_{22}\,\texttt{IaXr\_2\_2} + d_{23}\,\texttt{IaXr\_2\_3} + d_{32}\,\texttt{IaXr\_3\_2} + d_{33}\,\texttt{IaXr\_3\_3} & (\texttt{agegrp*race}\text{ dummies}) \\
\quad +d_{42}\,\texttt{IaXr\_4\_2} + d_{43}\,\texttt{IaXr\_4\_3} & \\
\quad +u &
\end{array}$$

That is,

```
. xi: regress y i.agegrp*i.race
```

results in the same model as typing:

```
. xi: regress y i.agegrp i.race i.agegrp*i.race
```

While there are lots of other ways the interaction could have been parameterized, this method has the advantage that one can test the joint significance of the interactions by typing:

```
. testparm IaXr*
```

Returning to the estimation step, whether you specify `i.agegrp*i.race` or `i.race*i.agegrp` makes no difference (other than in the names given to the interaction terms; in the first case, the names will begin with `IaXr`; in the second, `IrXa`). Thus,

```
. xi: regress y i.race*i.agegrp
```

estimates the same model.

You may also include multiple interactions simultaneously:

```
. xi: regress y i.agegrp*i.race i.agegrp*i.sex
```

The model estimated is

$$\begin{array}{ll}
y = a + b_2\,\texttt{Iagegr\_2} + b_3\,\texttt{Iagegr\_3} + b_4\,\texttt{Iagegr\_4} & (\texttt{agegrp}\text{ dummies}) \\
\quad +c_2\,\texttt{Irace\_2} + c_3\,\texttt{Irace\_3} & (\texttt{race}\text{ dummies}) \\
\quad +d_{22}\,\texttt{IaXr\_2\_2} + d_{23}\,\texttt{IaXr\_2\_3} + d_{32}\,\texttt{IaXr\_3\_2} + d_{33}\,\texttt{IaXr\_3\_3} & (\texttt{agegrp*race}\text{ dummies}) \\
\quad +d_{42}\,\texttt{IaXr\_4\_2} + d_{43}\,\texttt{IaXr\_4\_3} & \\
\quad +e_2\,\texttt{Isex\_2} & (\texttt{sex}\text{ dummy}) \\
\quad +f_{22}\,\texttt{IaXs\_2\_2} + f_{23}\,\texttt{IaXs\_2\_3} + f_{24}\,\texttt{IaXs\_2\_4} & (\texttt{agegrp*sex}\text{ dummies}) \\
\quad +u &
\end{array}$$

Note that the `agegrp` dummies are (correctly) included only once.

Interactions with continuous variables

`i.`*varname*$_1$`*`*varname*$_2$ (as distinguished from `i.`*varname*$_1$`*i.`*varname*$_2$, note the second `i.`) specifies an interaction of a categorical variable with a continuous variable. For instance,

```
. xi: regress y i.agegr*wgt
```

results in the model:

$$\begin{array}{ll}
y = a + b_2\,\texttt{Iagegr\_2} + b_3\,\texttt{Iagegr\_3} + b_4\,\texttt{Iagegr\_4} & (\texttt{agegrp}\text{ dummies}) \\
\quad +c\,\texttt{wgt} & (\text{continuous }\texttt{wgt}\text{ effect}) \\
\quad +d_2\,\texttt{IaXwgt\_2} + d_3\,\texttt{IaXwgt\_3} + d_4\,\texttt{IaXwgt\_4} & (\texttt{agegrp*wgt}\text{ interactions}) \\
\quad +u &
\end{array}$$

A variation on this notation, using `|` rather than `*` omits the `agegrp` dummies. Typing

```
. xi: regress y i.agegr|wgt
```

estimates the model:

$$\begin{aligned} y = a' \quad & +c'\,\texttt{wgt} && \text{(continuous \texttt{wgt} effect)} \\ & +d'_2\,\texttt{IaXwgt\_2} + d'_3\,\texttt{IaXwgt\_3} + d'_4\,\texttt{IaXwgt\_4} && \text{(\texttt{agegrp*wgt} interactions)} \\ & +u' \end{aligned}$$

The predicted values of `y` are

agegrp*wgt model	agegrp|wgt model	
$y = a + c\,\texttt{wgt}$	$a' + c'\,\texttt{wgt}$	if $\texttt{agegrp} = 1$
$a + c\,\texttt{wgt} + b_2 + d_2\,\texttt{wgt}$	$a' + c'\texttt{wgt} + d'_2\,\texttt{wgt}$	if $\texttt{agegrp} = 2$
$a + c\,\texttt{wgt} + b_3 + d_3\,\texttt{wgt}$	$a' + c'\texttt{wgt} + d'_3\,\texttt{wgt}$	if $\texttt{agegrp} = 3$
$a + c\,\texttt{wgt} + b_4 + d_4\,\texttt{wgt}$	$a' + c'\texttt{wgt} + d'_4\,\texttt{wgt}$	if $\texttt{agegrp} = 4$

That is, typing

```
. xi: regress y i.agegr*wgt
```

is equivalent to typing:

```
. xi: regress y i.agegr i.agegr|wgt
```

Also note that in either case, it is not necessary to specify separately the continuous variable `wgt`; it is included automatically.

Using ci: Interpreting output

```
. xi: regress mpg i.rep78
i.rep78              Irep78_1-5   (naturally coded; Irep78_1 omitted)
(output from regress appears)
```

Interpretation: `i.rep78` expanded to the dummies `Irep78_1`, `Irep78_2`, ..., `Irep78_5`. The numbers on the end are "natural" in the sense that `Irep78_1` corresponds to `rep78` = 1, `Irep78_2` to `rep78` = 2, and so on. Finally, the dummy for `rep78` = 1 was omitted.

```
 . xi: regress mpg i.make
 i.make               Imake_1-74   (Imake_1 for make==AMC Concord omitted)
(output from regress appears)
```

Interpretation: `i.make` expanded to `Imake_1`, `Imake_2`, ..., `Imake_74`. The coding is not natural because `make` is a string variable. `Imake_1` corresponds to one make, `Imake_2` another, and so on. We can find out the coding by typing '`describe`'. `Imake_1` for the AMC Concord was chosen to be omitted.

How xi names variables

The names `xi` assigns to the dummy variables it creates are of the form:

I*stub_groupid*

You may subsequently refer to the entire set of variables by typing 'I*stub**'. For example:

name	= I +	*stub*	+ _ +	*groupid*	Entire set
Iagegr_1	I	agegr	_	1	Iagegr*
Iagegr_2	I	agegr	_	2	Iagegr*
IaXwgt_1	I	aXwgt	_	1	IaXwgt*
IaXr_1_2	I	aXr	_	1_2	IaXr*
IaXr_2_1	I	aXr	_	2_1	IaXr*

xi as a command rather than a command prefix

`xi` can be used as a command prefix or as a command by itself. In the latter form, `xi` merely creates the indicator and interaction variables. Equivalent to typing,

```
. xi: regress y i.agegrp*wgt
 i.agegrp            Iagegr_1-4   (naturally coded; Irep78_1 omitted)
 i.agegrp*wgt        IaXwgt_1-4   (coded as above)
```
(output from regress appears)

is

```
 . xi i.agegrp*wgt
 i.agegrp            Iagegr_1-4   (naturally coded; Irep78_1 omitted)
 i.agegrp*wgt        IaXwgt_1-4   (coded as above)

 . regress y Iagegr* IaXwgt*
```
(output from regress appears)

Warnings

1. When you use `xi`, either as a prefix or a command by itself, `xi` first drops all previously created interaction variables—variables starting with capital `I`. Do not name your variables starting with this letter.
2. `xi` creates new variables in your data; most are `byte`s but interactions with continuous variables will have the storage type of the underlying continuous variable. You may get the message "no room to add more variables". If so, you must repartition memory; see [4] memory.
3. When using `xi` with an estimation command, you may get the message "matsize too small". If so, see [5u] matsize.

ssi6 Routines to speed Monte Carlo experiments

William Gould, Stata Corporation, FAX 409-696-4601

The syntax of the `post` commands is

postfile *varlist* **using** *filename* [, **every(**#**)** **replace**]

post *exp exp* ... *exp*

postclos

These commands are utilities to assist Stata programmers in performing Monte Carlo type experiments. `postfile` declares the variable and file names of a (new) Stata data set for containing results. `post` adds a new observation to the declared data set. `postclos` declares an end to the posting of observations. All three commands manipulate the new results data without disturbing the data in memory. After `postclos`, the new data set contains the posted results and may be loaded with `use`; see [5d] save.

Options

`every(#)` specifies how often posted results are to be written to disk. `post` attempts to be efficient by buffering results and writing to disk only occasionally. `every()` should not be specified; the default value `every(32)`—23 for non-Intercooled versions of Stata—is believed to be fastest. If `every()` is specified, it is taken as merely a suggestion; values that are too large (larger than 208 for Intercooled and 23 for regular Stata) are treated as meaning `every(32)` and `every(23)`.

`replace` indicates that the file specified (may) already exist and, if it does, `postfile` may erase the file and create a new one.

Remarks

Persons performing Monte Carlo experiments in Stata have, in the past, employed one of two programming approaches. The first, originally suggested by Hamilton (1991), might be called the display-and-infile method. The basic idea is to start a session log, display the results of the calculations on the screen (and therefore into the log), close the log, and read back the results:

```
set more 1
log using filename
repeat {
    draw a sample
    make a calculation
    display the results of the calculation
}
log close
infile the results previously displayed
```

This method suffers from two disadvantages. First, it requires that the results of the individual simulations be displayed on the screen which is, at best, inelegant. Second, the simulation cannot itself be logged, meaning that those of us who keep notebooks of printed logs backing up important results are prevented from doing so.

An alternative programming approach does not have those problems and is therefore widely used in the ado-files we at Stata Corp. write. It might be called the append method because the approach amounts to adding observations, one at a time, to a data set being maintained on disk:

```
create a temporary data set
repeat {
    draw a sample
    make a calculation
    use the temporary data set
    append the calculated result(s) to the end of the data
    resave the temporary data set
}
erase the temporary data set
```

This approach is used in Stata's `boot` and `bsqreg` commands; see [5s] boot and [5s] qreg. While not suffering from the disadvantages of the display-and-infile method, it has its own disadvantage—it is slow.

There is a third way simulations could be programmed in Stata. It could be called the buffered-append method because, while it is basically the append method, rather than adding observations one at a time to the data, results are temporarily buffered in memory and then, periodically, the buffers are used to update the data:

```
create a temporary data set
repeat {
    draw a sample
    make a calculation
    save the results in memory somewhere
    when memory is full {
        use the temporary data set
        append the buffered results to the data
        resave the temporary data set
    }
}
use the temporary data set
```

This method has the potential to be faster because the costly use and resave occurs less often. The `post` commands do this. In outline, their use is

```
postfile ... using ...
repeat {
    draw a sample
    make a calculation
    post ...
}
postclos
use the data set
```

Example

Let us consider the coverage of the 95%, t-based confidence interval for the mean applied to log-normal populations. To explain, the central limit theorem assures us that, asymptotically, distributions of means are normally distributed regardless of the underlying distribution of the population. In finite samples, less can be said, but if the underlying population follows a normal distribution and if one uses estimates of the mean and standard deviation, the mean will follow a t distribution with $n-1$ degrees of freedom. (Note that as $n \to \infty$, the t approaches the normal, so the finite-sample result is consistent with the central limit theorem.)

In real life, people often apply confidence intervals calculated on the basis of t distributions to means calculated on data that are far from normal. Do they, on average, nevertheless generate correct predictions? That is, a 95% confidence interval should include the true mean 95% of the time. If the calculation results in an interval that is too wide, however, that too-wide interval will include the mean more than 95% of the time. If it is too narrow, that interval will include the mean less than 95% of the time.

Thus, we could take some distribution—we will use the log normal—and draw samples from it. We could calculate the mean and perform the classic t test, recording whether the true mean (which we know) lies in the interval. If we do this enough times, we can answer the question, at least with respect to the log-normal distribution. (A variable z is log-normally distributed if $z = e^u$ and u is normally distributed. If u has mean μ and variance σ^2, then z has median, not mean, e^μ; the mean of z is $e^\mu e^{\sigma^2/2}$.)

Let us begin by constructing a data set of means and variances for 100-observation samples of a log-normal distribution:

```
program define lnsim
        version 3.1
        postfile mean var using results, replace
        quietly {
                local i = 1
                while `i´ <= 10000 {
                        drop _all
                        set obs 100
                        gen z = exp(invnorm(uniform()))
                        summarize z
                        post _result(3) _result(4)
                        local i=`i´+1
                }
        }
        postclos
end
```

The heart of this program are the three lines in the middle, the first two of which are

```
set obs 100
gen z = exp(invnorm(uniform()))
```

and correspond to drawing our sample. The third line

```
summarize z
```

calculates our results. `summarize`, in addition to displaying summary statistics (which we suppress with the `quietly { }` surrounding the code), stores the sample mean in `_result(3)` and variance in `_result(4)`. The new `post` command allows us to save those results. Prior to using `post`, however, the `postfile` must be declared. This we did at the outset of our program, declaring that we would be saving two results which we would call `mean` and `var` (for mean and variance) in a data set called `results.dta`. Then, when we are all done, we must inform `post` with the `postclos` command.

The rest of the program was merely concerned with performing the experiment 10,000 times:

```
local i = 1
while `i´ <= 10000 {
        ...
        local i=`i´+1
}
```

The results of running our program are

```
. lnsim

. describe

Contains data from results.dta
  Obs: 10000 (max= 19997)
 Vars:     2 (max=    99)
Width:     8 (max=   200)
  1. mean        float  %9.0g
  2. var         float  %9.0g
Sorted by:

. summarize

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
    mean |   10000    1.648349    .2165937   1.022719   4.280587
     var |   10000    4.720659    6.208903   .6215334   450.1076
```

We now have 10,000 means and variances from independent 100-observation log-normal data sets. On a 25MHz 486, this took about 14 minutes.

Our log-normal population was based on $z = e^u$ with $u \sim \mathrm{N}(0,1)$, so the true mean of z is $e^{1/2} \approx 1.6487213$. Let $\bar{x}_j$ and s_j^2 represent the calculated mean and variance of the jth sample. Then the 95% confidence bounds that would be calculated by a standard t test are $\bar{x}_j \pm t_{.95}\sqrt{s_j^2/100}$. Making these calculations, we can mark each sample as rejecting or not rejecting that the mean is $e^{1/2}$:

```
. gen se = sqrt(var/100)
. gen lower = mean - invt(100-1, .95)*se
. gen upper = mean + invt(100-1, .95)*se
. gen accept = lower<exp(1/2) & exp(1/2)<upper
. count if accept
 9198
```

Thus, the coverage of our 95% test is only 92%—the confidence intervals are too narrow. We performed this experiment "only" 10,000 times, so we should verify that the observed 92% differs from 95% due to more than chance:

```
. cii 10000 9198

                                                  -- Binomial Exact --
Variable |     Obs        Mean    Std. Err.       [95% Conf. Interval]
---------+-------------------------------------------------------------
         |   10000       .9198     .002716        .9142983    .9250475
```

A 95% confidence interval for the coverage is .914 to .925. (Moreover, given a probability of .95, the chances of observing 9198 or fewer successes in 10,000 trials is virtually 0, as you can verify for yourself by typing '`bitest 10000 9188 .95`'.

So, if the standard t test performs poorly, what about the central-limit-theorem result? Rather than using $t_{.95}$, what if we use $Z_{.95}$? The result will be worse: t intervals are wider than normal intervals and we have already determined that the intervals are too narrow. It will not, however, make much difference since $t_{.95} \approx 1.97$ for 99 degrees of freedom whereas $Z_{.95} \approx 1.96$. For the record:

```
. drop lower upper accept
. gen lower = mean - 1.96*se
. gen upper = mean + 1.96*se
. gen accept = lower<exp(1/2) & exp(1/2)<upper
. count if accept
 9169
```

Performance

As I find myself running simulations more and more these days, I went to the effort of timing the display-and-infile, append, and buffered-append (`post`) alternatives. The good news is that buffered-append is substantially faster than the append method. The bad news is that display-and-infile is still the fastest way to run simulations in Stata:

replications	display and infile (seconds)	append (seconds)	buffered append (seconds)
100	5.22	11.81	8.57
500	25.71	73.77	40.76
1000	51.13	155.99	81.62

The timings above were performed on a 25MHz 486 running Intercooled Stata under DOS.

`postfile` also provides an `every()` option which controls how often buffers are flushed. The documentation above recommends you never specify this option. Using the same simulation with 500 replications, I performed timings for different values of `every()`:

`every()`	time (sec.)	`every()`	time (sec.)
2	64.87	32	40.81
4	49.49	64	41.30
8	43.55	128	43.77
16	41.96	200	46.41

Between `every(16)` and `every(64)` the function is virtually flat.

Summary

There is no question that the display-and-infile approach is the fastest way to run simulations in Stata. The 10,000-replication simulation presented above, I estimate, would have taken only 8.5 minutes rather than the 13.5 minutes actually observed. Nevertheless, I continue to reject using that method because it does not allow me to maintain logs of what I have done. Moreover, it is easy to misinterpret these timings unless one remembers that they are absolute, not relative. For instance, the difference in execution time for 10,000 replications is 5 minutes and that difference remains 5 minutes regardless of the complexity of the simulation. Thus, I recently performed simulations involving bootstrapped quantile regression (10,000 replications of 50 replications, meaning estimation of 500,000 quantile regressions). This simulation took over 8 hours. The difference in execution time between the logable buffered append and display-and-infile is still only 5 minutes.

More importantly, buffered append is substantially faster than the simple append method—in fact, regressions of time on number of replications suggest that the buffered append method is nearly twice as fast once the fixed costs of the routines are eliminated (for buffered append, each added replication is estimated as costing .0847 seconds; for the simple append, .1604 seconds; and the ratio is thus $.1604/.0847 \approx 1.89$). As `boot` and `bsqreg` are currently implemented in terms of the simple-append method, a doubling of performance should be possible by reimplementing these routines in terms of `post`.

References

Hamilton, L. C. 1991. ssi1: Monte Carlo simulation. *Stata Technical Bulletin* 1: 25–28.

ssi6.1 Simplified Monte Carlo simulations

William Gould, Stata Corporation, FAX 409-696-4601

The syntax of the `simul` command is

`simul` *progname*`,` `reps(`*#*`)` [`args(`*whatever*`)` `dots`]

`simul` eases the programming task of performing Monte Carlo type simulations. *progname* is the name of a program that performs a single simulation. Typing '`simul` *progname*`, reps(`*#*`)`' iterates *progname* for # replications and collects the results.

`simul` calls *progname* two ways. At the outset, `simul` issues "*progname* `?`" and expects *progname* to set the global macro `$S_1` to contain a list of variable names under which results are to be stored. Thereafter, `simul` issues straight "*progname*" calls and expects it to perform a single simulation and to store the results using `post`. Details of `post` can be found in the insert above, but enough information is provided below to use `post` successfully.

Options

`reps(`*#*`)` is not optional—it specifies the number of replications to be performed.

`args(`*whatever*`)` specifies any arguments to be passed to *progname* on invocation. The query call is then of the form "*progname* `?` *whatever*" and subsequent calls of the form "*progname whatever*".

`dots` requests a dot be placed on the screen at the beginning of every call to *progname*, thus providing entertainment during a long simulation.

Remarks

progname must have the following outline:

```
program define progname
    if "`1'"=="?" {
        global S_1 "variable names"
        exit
    }
    perform single simulation
    post results
end
```

There must be the same number of results following the `post` command as variable names following the `global S_1` command.

Example

Make a data set containing means and variances of 100-observation samples from a log-normal distribution. Perform the experiment 10,000 times:

```
program define lnsim
        version 3.1
        if "`1'"=="?" {
                global S_1 "mean var"
                exit
        }
        drop _all
        set obs 100
        gen z = exp(invnorm(uniform()))
        summarize z
        post _result(3) _result(4)
end
```

It is instructive to compare this program to the same example in the previous insert. In any case, `lnsim` can then be executed 10,000 times by typing:

```
. simul lnsim, reps(10000)
. describe
Contains data
  Obs: 10000 (max= 19997)
 Vars:     2 (max=    99)
Width:     8 (max=   200)
  1. mean         float  %9.0g
  2. var          float  %9.0g
Sorted by:
. summarize
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
    mean |   10000    1.648349    .2165937   1.022719   4.280587
     var |   10000    4.720659    6.208903   .6215334   450.1076
```

The `simul` command took 14.5 minutes on a 25MHz DOS 486 computer.

Technical note: debugging a simulation

Before executing our `lnsim` simulator, we can verify it works by typing

```
. lnsim
. display $S_1
```

This verifies that the program sets the global macro `$S_1` correctly on a query call. We can then try executing `lnsim`:

```
. postfile $S_1 using myfile
. lnsim
```

The `postfile` command opens a file for the posted results (see insert above). Invoking `lnsim` should perform a single simulation. We could then close the result file and examine it:

```
. postclos
. use myfile, clear
```

Passing arguments to the simulation

Consider a more complicated problem: Let's experiment with estimating

$$y_j = a + bx_j + u_j$$

when the true model has $a = 1$, $b = 1$, and $u_j = 2(z_j + cx_j)$ and z_j is $\mathrm{N}(0,1)$. We will keep the parameter estimates and standard errors and experiment with varying c. x_j will be fixed across the experiments but will originally be generated as $\mathrm{N}(0,1)$. We begin by interactively making the true data:

```
. drop _all
. set obs 100
. gen x = invnorm(uniform())
. gen true_y = 1+2*x
. save truth
```

Our program is

```
program define hetero
        version 3.1
        if "`1´"=="?" {
                global S_1 "a se_a b se_b"
                exit
        }
        use truth, clear
        gen y = true_y + 2*(invnorm(uniform()) + `1´*x)
        regress y x
        post _b[_cons] _se[_cons] _b[x] _se[x]
end
```

Note the use of `` `1´ `` in our statement for generating `y`. `` `1´ `` is an argument. If the argument's value is 2, then the last part of the statement is equivalent to "`2*x`". We can run 10,000 simulations, setting the argument to 2 by typing

```
. simul hetero, args(2) reps(10000)
```

We might then analyze that data and try a different experiment, this time setting the argument to 1.5:

```
. simul hetero, args(1.5) reps(10000)
```

Our program `hetero` could, however, be more efficient because it rereads the file `truth` once every replication. It would be better if we could read the data just once and, in fact, we can because we can use the query call to initialize ourselves. A faster version reads:

```
program define hetero
        version 3.1
        if "`1´"=="?" {
                use truth, clear                (load the data just once)
                global S_1 "a se_a b se_b"
                exit
        }
        gen y = true_y + 2*(invnorm(uniform()) + `1´*x)
        regress y x
        post _b[_cons] _se[_cons] _b[x] _se[x]
        drop y                                  (because we will recreate it next time)
end
```

Assume we plan on replicating the experiment 10,000 times. In our original draft, we used the truth data once per replication, meaning we performed 10,000 uses. The new version does this only once, saving 9,999 unnecessary uses.

Performance

`simul` is implemented in terms of `post` (see insert above) and it must, therefore, be slower. The good news is that it is not much slower:

N	`post` (sec.)	`simul` (sec.)
100	8.13	8.90
500	49.21	43.83
1000	89.19	87.16

These timings were performed on a 25MHz, DOS 486 computer.

Moreover, on the log-normal example, the direct `post` implementation took 13.5 minutes to perform 10,000 replications; the `simul` solution took 14.5 minutes, for a total cost of 1 minute. `post` provides more flexibility than `simul` but, unless that flexibility is necessary, `simul` provides a more convenient way to perform Monte Carlo simulations.

sts7.3 | A library of time series programs for Stata (Update)

Sean Becketti, Stata Technical Bulletin, FAX 914-533-2902

In *sts7*, a library of time series programs for Stata was introduced (Becketti 1994). That insert described an approach to time series analysis that builds on Stata's core commands and on its extensibility. The insert also cataloged the programs in the time series library.

This update describes changes and additions to the time series library. An updated catalog of programs is also included. The updated library is available on the STB diskette. This update will be repeated in each issue of the STB. Consult the original insert for a general discussion of Stata's approach to time series analysis. As always, I actively solicit your comments, complaints, and suggestions.

New features

New date-handling commands: Alan Riley's new date-handling commands (described in *dm20* earlier in this issue) have been included in the time series library. They are listed in Table 1.

A catalog of programs

Table 1 lists the user-level programs in the time series library. Each program's status is indicated by a letter grade. An 'A' indicates a program that is safe for general use. An 'A' program has been documented—*in its current form*—in the STB and follows all Stata guidelines for an estimation command, where relevant (see [4] estimate). A 'B' program produces accurate results, but either is not fully documented, not completely compatible with the standard time series syntax adopted in the library, or not in conformance with the guidelines for an estimation command. Most 'B' programs receive that grade because they have been revised significantly since they were last documented. A 'C' program is incomplete in significant ways but can be used safely by an advanced Stata user. A 'D' program has serious deficiencies, however its code may provide a useful model to advanced Stata users writing their own time series programs. An 'O' program is obsolete, that is, it has been superseded by a newer program. An 'O' program is retained if it is still called by one or two user-level programs. There are currently no 'D' or 'O' programs.

Utilities for time series analysis

Writing programs for time series analysis presents a variety of challenges. In developing this library of programs, I had to write a pool of utility programs to interpret the time series options, to generate lags, to manipulate the list of variables in a lag polynomial, and so on. I recommend that you familiarize yourself with these utilities, if you wish to write your own time series programs. A list of some of the most frequently used utility programs appears in Table 2 below.

Future developments and call for comments

This library of time series programs is under constant revision and extension. Projects under development include programs to estimate rolling regressions, to estimate vector autoregressions, and to perform maximum-likelihood tests for cointegration. Older programs are being revised to bring them up to Stata's standards for estimation programs. A disadvantage of these constant revisions is the likelihood of inadvertently introducing errors into the programs. The advantage of constant revision is the ease and rapidity of fixing these errors and the steady increase in Stata's time series capabilities. I encourage you to alert me to any errors or inconveniences you find.

If you find an error in any of these commands, I will attempt to correct it by the next issue of the STB. To speed the process, please send me a diskette containing a `do`-file that replicates the error. Debugging software is similar to auto mechanics: if I can't reproduce the problem, I can't fix it.

Table 1: User-level programs

Command	Status	Documentation	Description
`ac`	A	*sts1*	display autocorrelation plot
`chow`	C	—	perform Chow test for a shift in regression coefficients
`coint`	B	*sts2*	perform Engle–Granger cointegration test
`cusum`	B	—	perform CUSUM test of regression stability. (**Note:** this name conflicts with Stata's `cusum` command for binary variables.)
`datevars`	A	*sts4*	specify date variables
`dickey`	B	*sts2*	perform unit root tests
`dif`	A	*sts2*	generate differences
`downame`	A	*dm20*	convert code to day-of-week name
`dropoper`	A	*sts2*	drop operator variables
`findlag`	B	*sts2*	find optimal lag length
`findsmpl`	B	*sts4*	display sample coverage
`growth`	A	*sts2*	generate growth rates
`growthi`	A	*sts2*	immediate form of `growth`
`lag`	A	*sts2*	generate lags
`lastday`	A	*dm20*	calculate last day of month
`lead`	A	*sts2*	generate leads
`lstbday`	A	*dm20*	calculate last business day of month
`mdytodow`	A	*dm20*	calculate day of week from month/day/year
`mnthname`	A	*dm20*	convert code to month name
`namedow`	A	*dm20*	convert name to day of week code
`namemnth`	A	*dm20*	convert name to month code
`pac`	A	*sts1*	display partial autocorrelation plot
`pearson`	A	*sg5.1*	calculate Pearson correlation with p-value
`period`	A	*sts2*	specify period (frequency) of data
`quandt`	B	—	calculate Quandt statistics for a break in a regression
`regdiag`	B	*sg20*	calculate regression diagnostics
`spear`	A	*sg5.1*	Spearman correlation with p-value
`tauprob`	A	*sts6*	approximate p-values for unit root and cointegration tests
`testsum`	B	—	test whether the sum of a set of regression coefficients is zero
`today`	A	*dm20*	calculate today
`tsfit`	A	*sts4*	estimate a time series regression
`tsload`	B	—	load an ad hoc time series equation into memory
`tsmult`	A	*sts4*	display information about lag polynomials
`tspred`	B	—	dynamically forecast or simulate a time series regression
`tsreg`	A	*sts4*	combined `tsfit`, `tsmult`, and `regdiag`
`xcorr`	A	*sts3*	calculate cross correlations
`ystrday`	A	*dm20*	calculate yesterday from today

For more information on these programs, type '`help ts`' or '`help` *command-name*'.

Table 2: Utility programs

Command	Description
`_ac`	calculate autocorrelations, standard errors, and Q-statistics
`_addl`	"add" a lag operator to a variable name
`_addop`	"add" an arbitrary operator to a variable name
`_getrres`	calculate recursive residuals for a regression model
`_inlist`	determine whether a token appears in a token list
`_invlist`	determine whether a varname appears in a varlist
`_opnum`	decode the operators (and their powers) in a varname
`_parsevl`	parse a varlist to replace abbreviations
`_subchar`	replace one character in a string with another
`_ts_meqn`	parse a time series command and generate lags
`_ts_pars`	parse a time series command into useful macros
`faketemp`	generate temporary variable names that can be lagged

Reference

Becketti, S. 1994. sts7: A library of time series programs for Stata. *Stata Technical Bulletin* 17: 28–32.

sts8 Hansen's test for parameter instability

Ken Heinecke and Charles Morris, Federal Reserve Bank of Kansas City, FAX 816-881-2199

In order to conduct statistical inference and prediction with a regression model, the parameters of the model must be stable. A large number of statistics have been developed to test the null hypothesis of parameter stability. Among the most popular of these statistics are the Chow (1960) test, the Quandt (1958, 1960) test, and the CUSUM and CUSUM of squares tests (Brown, Durbin, and Evans 1975).

These tests are distinguished by their alternative hypotheses and their power. Not surprisingly, tests with narrowly defined alternatives have more power, at least against the chosen alternative. Unfortunately, narrowly defined tests can be misleading when confronted with a more general form of parameter instability.

Among the tests mentioned above, the Chow test is the most sharply focused and, thus, the most powerful. The alternative hypothesis is that one or more regression parameters changed values at a single, known break point. This form of instability is frequently not plausible. Moreover, it is rare in observational data to know with certainty and exogenously (that is, without peeking at the data) when the parameters shifted.

The Quandt test is a generalization of the Chow test; it gives up power to broaden the alternative. In the Quandt test, the alternative hypothesis is that one or more regression parameters changed values at a single, *unknown* break point. In essence, the Quandt test performs a Chow test at all potential break points and chooses the statistic that most strongly favors the alternative hypothesis. A drawback to the Quandt test is that choosing the break point endogenously gives the statistic an unknown distribution. Critical values must be developed using Monte Carlo or bootstrap methods each time one wishes to perform the test. In both the Quandt and the Chow tests, the constancy of the error variance is an important part of the maintained hypothesis.

The CUSUM and CUSUM of squares tests are the most general of these tests. They calculate cumulative sums (and sums of squares) of recursive (rolling, one-step-ahead) residuals. Under the null, the distribution of these cumulative sums is known. Any model breakdown can lead the cumulative sums to exceed their critical values. Unfortunately, the extreme generality of the CUSUM and CUSUM of squares tests reduces their power substantially. In practice, the CUSUM and CUSUM of squares tests frequently have scant success in detecting parameter instability, particularly in observational (as opposed to experimental) data.

[The Stata time series library contains `chow`, `quandt`, *and* `cusum`*: commands for the Chow, Quandt, and CUSUM (and CUSUM of squares) tests, respectively.* `cusum` *and* `quandt` *are level B commands, which means they produce accurate results but are either not fully documented, not compatible with the standard time series syntax, or not in conformance with Stata's guidelines for an estimation command.* `quandt` *reports the results of the Quandt test, but it does not calculate Monte Carlo or bootstrap critical values and confidence intervals. These calculations can be very time-consuming.* `chow` *is a level C command, which means it is incomplete in significant ways but can be used safely by an advanced Stata user.* `chow` *also calculates the Farley–Hinich–McGuire test which allows the variable parameters to follow a deterministic trend after the break point. See sts7.3 above for more information on the Stata time series library—Ed.]*

Hansen's test

The tests discussed above illustrate a common dilemma—the desire to find a test that accommodates a very general alternative hypothesis while retaining high power. It is rare that a test statistic with these properties and a known, standard distribution under the null can be developed.

An alternative approach is to derive an asymptotic approximation to the *local power*, that is, to the slope of the power function at the null hypothesis in the direction of interest. This asymptotic approximation can be used to develop tests with maximal local power (Cox and Hinkley 1974). The test statistics will generally follow nonstandard distributions under the null, but critical values also can be derived from the asymptotic local power function.

Hansen (1992) has followed this approach in developing an alternative statistic that is the locally most powerful test of the null hypothesis of constant parameters (both the coefficients and the variance of the error term) against the alternative hypothesis that the parameters follow a martingale. This alternative is very general: it accommodates parameters that change at unknown times and parameters that follow a random walk.

The only constraint on the application of Hansen's test is that the variables in the regression model must be stationary, that is, the variables must follow unconditional distributions that are constant over time. An example of a nonstationary variable is the United States gross domestic product. GDP grows as population, the capital stock, and productivity grow. Its mean and variance are growing over time, thus GDP is not drawn from the same distribution at different points in time. Hansen provides suggestions for treating models with nonstationary variables.

I have written `hansen`, an `ado`-file that calculates the Hansen test. The syntax of `hansen` is

`hansen` *varlist* [`if` *exp*] [`in` *range*] [`, regress tsfit-`*options*]

The test statistics are based on the residuals from the regression model in *varlist* estimated over the entire sample. `hansen` uses `tsfit` (Becketti 1994) to estimate the regression model, hence all the `tsfit` options can be used. `hansen` offers one additional option, `regress`, which causes the regression output to be displayed. The default is to suppress the regression output.

Development of the test

Consider the standard linear regression model

$$\begin{aligned} y_t &= x_t'\beta + \epsilon_t, \\ E(\epsilon_t \mid x_t) &= 0, \\ E(\epsilon_t^2) &= \sigma_t^2, \\ \lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T} \sigma_t^2 &= \sigma^2,\ t = 1,\ldots,T \end{aligned}$$

where x_t is a $K \times 1$ vector containing the t-th observation on K regressors. The null hypothesis is that the model's parameters, β and σ^2, are constant.

The first-order conditions for the least squares estimates of the parameters are

$$\sum_{t=1}^{T} f_{it} = 0,\quad i = 1,\ldots,K+1$$

where

$$f_{it} = \begin{cases} x_{it}\widehat{\epsilon}_t, & i = 1,\ldots,K \\ \widehat{\epsilon}_t^2 - \widehat{\sigma}^2, & i = K+1. \end{cases}$$

The first K first-order conditions determine $\widehat{\beta}$. The $(K+1)$-st equation defines $\widehat{\sigma}^2$ to be the maximum-likelihood (rather than the unbiased) estimator of the error variance.

Hansen's test statistic for the ith parameter is

$$L_i = \frac{1}{TV_i}\sum_{t=1}^{T} S_{it}^2,$$

where S_{it} is the cumulative first-order condition through period t for the ith parameter, that is,

$$S_{it} = \sum_{j=1}^{t} f_{ij},$$

and where

$$V_i = \sum_{t=1}^{T} f_{it}^2.$$

The joint test statistic for a change in the model's parameters is

$$L_c = \frac{1}{T}\sum_{t=1}^{T} S_t' V^{-1} S_t,$$

where

$$\begin{aligned} f_t &= (f_{1t},\ldots,f_{K+1,t})', \\ S_t &= (S_{1t},\ldots,S_{K+1,t})',\ \text{and} \\ V &= \sum_{t=1}^{T} f_t f_t'. \end{aligned}$$

Note that the V_i above are just the diagonal elements of the matrix V.

Parameter stability is rejected if the test statistics L_i and L_c are large. The test statistics are basically averages of the squared cumulative first-order conditions, S_{it}. The cumulative first-order condition for the entire sample, S_{iT}, equals zero by construction. Intuitively, the cumulative first-order conditions for subsamples ending in period $j, j < T$, should wander around zero if the parameters are stable, in which case the test statistics will be small. If the parameters are not stable, the cumulative first-order conditions for the subsamples will wander away from zero, in which case the statistics will be large. Note that the Hansen statistics are based on full-sample estimates only, in contrast to the CUSUM and CUSUM of squares tests which require calculation of the recursive (rolling, one-step-ahead) residuals.

The Hansen statistics follow a nonstandard distribution. However this distribution depends only on the number of parameters being tested. Hansen's Table 1 (1992) presents asymptotic critical values for the test statistics. The 5 and 10 percent critical values for tests of individual parameters (tests with one degree of freedom) are 0.470 and 0.353, respectively.

Example

We use `hansen` to test the stability of the parameters in an error correction model for bank loans. The data and the example are taken from Becketti and Morris (1993).

An important question in monetary economics is whether nonbank sources of short-term business finance have become better substitutes for bank commercial and industrial (C&I) loans. In the traditional view, demand for C&I loans is relatively inelastic. As a consequence, the central bank can exert a powerful influence over economic activity by adjusting the quantity of bank reserves and thereby adjusting the supply of C&I loans. The increasing globalization of financial markets in recent years along with the growth both of finance company business lending and of the commercial paper market have raised questions about the continued relevance of the traditional view. If businesses have come to regard nonbank sources of short-term finance as good substitutes for C&I loans, the central bank's ability to influence economic activity through the quantity of bank reserves may be diminished.

An increase in the substitutability between bank and nonbank loans would be observed as an increase in the own-price-elasticity of demand for C&I loans. It is difficult to estimate this elasticity directly because it is difficult to estimate the structural equation for bank loan demand. However, a change in any of the structural parameters of a model will, in general, change *all* of the parameters of the reduced form model (the transformation of the structural form that eliminates endogenous variables as regressors). In other words, the reduced form equation for C&I loans should exhibit parameter instability if bank and nonbank loans have become better substitutes over time.

Becketti and Morris derive the direction in which some of the reduced form parameters should move if the own-price-elasticity demand of bank loans has increased. They apply a variety of tests for parameter instability to this reduced form equation—including the Chow, Quandt, CUSUM, and CUSUM of squares tests—and find little evidence that bank and nonbank loans have become better substitutes.

The following are the data used by Becketti and Morris:

```
. use bankloan, clear
(1955:Q2-1992:Q3)

. describe

Contains data from bankloan.dta
  Obs:   153 (max= 14182)                1955:Q2-1992:Q3
 Vars:    11 (max=    99)
Width:    35 (max=   200)
  1. year         int    %8.0g           Year
  2. quarter      int    %8.0g           Quarter
  3. Dcc          int    %8.0g           Credit controls of 1980 dummy
  4. D731         byte   %8.0g           cp rate > com bank rate dummy
  5. cash         float  %9.0g           growth rate of cashflow
  6. ci           float  %9.0g           growth rate of C&I loans
  7. finr         float  %9.0g           gr rate of business fixed inv
  8. invb         float  %9.0g           gr rate of inventories
  9. rff          float  %9.0g           change in federal funds rate
 10. rmort        float  %9.0g           change in mortgage rate
 11. rtb3         float  %9.0g           change in 3-mo t-bill yield
Sorted by:  year  quarter
```

```
. summarize
Variable |     Obs       Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
    year |     153   1973.124    11.0842       1954       1992
 quarter |     153   2.503268   1.118764          1          4
     Dcc |     153   .0065359   .0808452          0          1
    D731 |     153   .0065359   .0808452          0          1
    cash |     152   .0189988   .0391609  -.1034894   .1195951
      ci |     152   .0209342   .0200359  -.0264533   .1192203
    finr |     152   .0181174   .0262676  -.0856726   .0946808
    invb |     152   .0146167   .0150725   -.015542   .0593133
     rff |     152   .0001468    .010778  -.0399067   .0601867
   rmort |     152    .000232   .0052554  -.0210667   .0157333
    rtb3 |     152   .0001446   .0085612  -.0373367   .0446167
```

The variables in the Becketti and Morris study, like most economic variables, are nonstationary. When nonstationary variables obey a stationary linear relation in the long run, the variables are said to be cointegrated, and the relationship between the variables can conveniently be estimated in error correction form.

Take, as an example, two nonstationary variables, y_t and x_t, that follow the dynamic statistical relationship

$$A^*(L)y_t = B^*(L)x_t + \epsilon_t.$$

where L is the lag operator ($Lx_t \equiv x_{t-1}$) and $A^*()$ and $B^*()$ are polynomials in the lag operator. (The `lag` command in the Stata time series library can be used to mimic the lag operator.) By rearranging terms, this model can be written as

$$A(L)\Delta y_t = B(L)\Delta x_t - \lambda(y_{t-1} - \delta x_{t-1}) + \epsilon_t$$

where Δ is the difference operator ($\Delta x_t \equiv x_t - x_{t-1} \equiv (1-L)x_t$). (The `dif` command in the Stata time series library can be used to mimic the difference operator.) This latter equation is called an error correction model and $(y_{t-1} - \delta x_{t-1})$ is called the error correction term. If y_t and x_t are cointegrated, the error correction term is the stationary linear combination of the variables. The error correction model can be estimated consistently by least squares. The coefficients in the error correction term $(1, -\delta)$ are called the cointegrating vector. The error correction model and the error correction term generalize in a straightforward way to models with many variables.

The error correction model has an intuitively appealing interpretation. The cointegrating vector reveals the equilibrium (long-run) relationship between the variables. The error correction term is a measure of how far the variables have deviated from their equilibrium relationship. The coefficient on the error correction term, λ, is a measure of how rapidly y_t responds to these deviations. Large values of λ correspond to rapid speeds of adjustment back to equilibrium. The other coefficients in the model measure the short-run relationship between the variables, that is, the association between their short-run fluctuations that would occur even if the variables were in long-run equilibrium.

The error correction term contains lagged values of the nonstationary variables as regressors. The Hansen test cannot be applied to nonstationary regressors, thus the test cannot be applied directly to the model. If the cointegrating vector is known, the error correction term—which is stationary—can be entered as a regressor. When the cointegrating vector is not known, Hansen (1992) recommends a two-step procedure: first estimate the cointegrating vector, then enter the estimated cointegrating vector as a generated regressor in the error correction model.

Becketti and Morris estimate the following reduced form equation for C&I loans:

$$\begin{aligned}\Delta L_t = \mu &+ \alpha_1 \Delta L_{t-1} + \alpha_2 \Delta L_{t-2} + \beta_{1,1}\Delta I_{t-1} + \beta_{1,2}\Delta I_{t-2} + \beta_{2,1}\Delta V_{t-1} + \beta_{3,1}\Delta C_{t-1} \\ &+ \beta_{4,1}\Delta r_{f,t-1} + \beta_{5,1}\Delta r_{m,t-1} + \beta_{5,2}\Delta r_{m,t-2} + \beta_{6,1}\Delta r_{3,t-1} \\ &- \lambda(L_{t-1} - \delta_1 I_{t-1} - \delta_2 V_{t-1} - \delta_3 C_{t-1} - \delta_4 r_{f,t-1} - \delta_5 r_{m,t-1} - \delta_6 r_{3,t-1})\end{aligned}$$

where

$$\begin{aligned} L_t &= \text{the log of bank C\&I loans,} \\ I_t &= \text{the log of business fixed investment,} \\ V_t &= \text{the log of business inventories,} \\ C_t &= \text{the log of corporate cash flow,} \\ r_{f,t} &= \text{the federal funds rate,} \\ r_{m,t} &= \text{the mortgage interest rate,} \\ r_{3,t} &= \text{the secondary market yield on 3-month Treasury bills.}\end{aligned}$$

Two dummy variables are also included to account for the credit controls of 1980 and for an episode in 1973 where price controls temporarily held the bank loan rate below the commercial paper rate.

Becketti and Morris find that some of the variables in the model do not enter the error correction term, that is, some of the δ_i are zero. They estimate a constrained cointegrating vector using the constrained maximum likelihood procedure of Johansen and Juselius (1990). In the output for the Hansen test listed below, `L.ghat` is the lagged value of the estimated error correction term.

```
. hansen ci finr invb cash rff rmort rtb3, lags(2,2,1,1,1,2,1) static(L.ghat D731 Dcc) regress
Quarterly data:  1955:2 to 1992:3    (150 obs)
  Source |       SS       df       MS                  Number of obs =     150
---------+------------------------------               F( 13,   136) =   23.02
   Model |   .04111377    13  .003162598               Prob > F      =  0.0000
Residual |  .018686145   136  .000137398               R-square      =  0.6875
---------+------------------------------               Adj R-square  =  0.6577
   Total |  .059799915   149  .000401342               Root MSE      =  .01172
------------------------------------------------------------------------------
      ci |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
    L.ci |   .5962585   .0811896      7.344   0.000       .4357012    .7568159
   L2.ci |  -.0936603   .0791853     -1.183   0.239      -.2502542    .0629335
  L.finr |   .1191324   .0585116      2.036   0.044       .0034221    .2348428
 L2.finr |   .0383112   .0569799      0.672   0.502      -.0743699    .1509923
  L.invb |   .1985529   .1010209      1.965   0.051      -.0012221     .398328
  L.cash |  -.0056775   .0333951     -0.170   0.865      -.0717182    .0603633
   L.rff |  -.4328891   .2358728     -1.835   0.069       -.899342    .0335638
 L.rmort |   .1645535   .2592306      0.635   0.527      -.3480907    .6771977
 L2.rmor |  -.1926119   .2218472     -0.868   0.387      -.6313283    .2461045
  L.rtb3 |   .4315918    .316005      1.366   0.174      -.1933273    1.056511
  L.ghat |   .0002538   .0008543      0.297   0.767      -.0014356    .0019433
    D731 |   .0770014   .0120174      6.407   0.000       .0532362    .1007666
     Dcc |  -.0340361   .0129517     -2.628   0.010      -.0596489   -.0084232
   _cons |    .003838   .0026582      1.444   0.151      -.0014188    .0090947
------------------------------------------------------------------------------

Individual Statistics
      L.ci =  .20142736
     L2.ci =  .15381964
    L.finr =  .19617365
   L2.finr =   .2406588
    L.invb =  .09370907
    L.cash =  .31049186
     L.rff =  .05287014
   L.rmort =   .0242646
   L2.rmor =   .0490725
    L.rtb3 =   .0538896
    L.ghat =  .38104913
      D731 =  .52666667
       Dcc =  .33333333
     _cons =  .33570572
     sigma =   .1079207

Model test statistic with 15 degrees of freedom:
       _Lc =  2.8214405
```

The Hansen statistics for changes in the parameters of the reduced form bank loan equation show virtually no evidence of parameter instability. Among the test statistics for changes in the individual parameters, only the statistic for the error correction term, `L.ghat`, is statistically significant at the 10-percent level and only the statistic for the 1973:Q1 dummy, `D731`, is significant at the 5-percent level. The joint test for a general breakdown in the equation is not statistically significant. The 5 and 10 percent critical values with 15 degrees of freedom are 3.54 and 3.26, respectively. Thus the Hansen test supports the conclusion of Becketti and Morris that the reduced form equation for C&I loans provides little evidence of a change in the elasticity of demand for bank loans.

Caveat

In addition to `tsfit`, `hansen` also utilizes a program called `mkmat` which stores a variables as an `_N` $\times$ 1 matrix, that is, as a column vector (Heinecke 1994). Matrices in Stata are constrained by the `matsize`, a parameter that has a limit of 400 in the Unix and Intercooled versions of Stata ([5u] matsize). Thus, in its current form, `hansen` cannot accommodate a model with more than 399 observations.

Also, as noted above, `hansen` calls `tsfit` to estimate the regression. As a consequence, the Stata time series library must be installed in order to use `hansen`.

[See ip6 earlier in this issue for `mkmat`*. See ip6.1, also in this issue, for a discussion of this approach to matrix calculations. See sts7.3 above for more information on the Stata time series library—Ed.]*

References

Becketti, S. 1994. sts7: A library of time series programs for Stata. *Stata Technical Bulletin* 17: 30–32.

Becketti, S. and C. Morris. 1993. Reduced form evidence on the substitutability between bank and nonbank loans. Research Working Paper RWP 93–18. Federal Reserve Bank of Kansas City.

Brown, R. L., J. Durbin, and J. M. Evans. 1975. Techniques for testing the constancy of regression relationships over time. *Journal of the Royal Statistical Society*, Series B. 37: 149–192.

Chow, G. C. 1960. Tests of equality between sets of coefficients in two linear regressions. *Econometrica* 28: 591–605.

Cox, D. R. and D. V. Hinkley. 1974. *Theoretical Statistics* London: Chapman and Hall.

Hansen, B. E. 1992. Testing for parameter instability in linear models. *Journal of Policy Modeling* 14: 517–533.

Heinecke, K. 1994. ip6: Storing variables in vectors and matrices. *Stata Technical Bulletin* 20: 8–9.

Quandt, R. E. 1958. The estimation of the parameters of a linear regression system obeying two separate regimes. *Journal of the American Statistical Association* 53: 873–880.

——. 1960. Tests of the hypothesis that a linear regression obeys two separate regimes. *Journal of the American Statistical Association* 55: 324–330.

Johansen, S. and K. Juselius. 1990. Maximum likelihood estimation and inference on cointegration–with applications to the demand for money. *Oxford Bulletin of Economics and Statistics* 52: 169–210.

zz3.4 Computerized index for the STB (Update)

William Gould, Stata Corporation, FAX 409-696-4601

The STBinformer is a computerized index to every article and program published in the STB. The command (and entire syntax) to run the STBinformer is `stb`. Once the program is running, you can get complete instructions for searching the index by typing `?` for help or `??` for more detailed help.

The STBinformer appeared for the first time on the STB-16 distribution diskette and included indices for the first fifteen issues of the STB. The STB-20 distribution diskette contains an updated version of the STBinformer that includes indices for the first *nineteen* issues of the STB. As the original insert stated, I intend to include an updated copy of this computerized index on every STB diskette. I encourage you to contact me with suggestions for changes and improvements in the program.

Reference

Gould W. 1993. Computerized index for the STB. *Stata Technical Bulletin* 16: 27–32.