Contents of this issue

an44.1.  StataQuest disk enclosed (really)  
2
an46.  Stata and Stage now available for IBM PowerPC  
2
crc36. Clarification on analytic weights with linear regression  
2
dm19.  Merging raw data and dictionary files  
3
dm20.  Date functions  
6
ip6. Storing variables in vectors and matrices  
8
ip6.1.  Data and matrices  
10
os14.  A program to format raw data files  
10
sg25.  Interaction expansion  
12
ss6.1.  Simplied Monte Carlo simulations  
22
sts7.3.  A library of time series programs for Stata (Update)  
25
sts8.  Hansen’s test for parameter instability  
26
zz3.4. Computerized index for the STB (Update)  
32
an1.1  STB categories and insert codes

Inserts in the STB are presently categorized as follows:

**General Categories:**
- *an* announcements
- *cc* communications & letters
- *dm* data management
- *dt* data sets
- *gr* graphics
- *in* instruction

**Statistical Categories:**
- *sbe* biostatistics & epidemiology
- *sed* exploratory data analysis
- *smv* multivariate analysis
- *snp* nonparametric methods
- *snc* quality control
- *sqv* analysis of qualitative variables

### General Categories:
- *an* announcements
- *cc* communications & letters
- *dm* data management
- *dt* data sets
- *gr* graphics
- *in* instruction

### Statistical Categories:
- *sbe* biostatistics & epidemiology
- *sed* exploratory data analysis
- *smv* multivariate analysis
- *snp* nonparametric methods
- *snc* quality control
- *sqv* analysis of qualitative variables

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**


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.

**Reference**


an44  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.

**Reference**


an40  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.

**Reference**

of $\text{Var}(u_j \sqrt{n_j})$. The \texttt{aweight}ed regression reports $s^2_a$, an estimate of $\text{Var}(u_j \sqrt{n_j} \sqrt{N/\sum_k n_k})$, where $N$ is the number of observations. Thus,

$$s^2_a = \frac{N}{\sum_k n_k} s_t^2 = \frac{s_t^2}{n}$$

(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, $\text{Var}(u) = \sigma^2_u$, a constant. Assume that individual data is not available; what is available are averages $(\bar{g}_j, \bar{x}_{1j}, \bar{x}_{2j})$ for $j = 1, \ldots, N$, and that each average is calculated over $n_j$ observations. Then it is still true that

$$g_j = \beta_0 + \beta_1 \bar{x}_{1j} + \beta_2 \bar{x}_{2j} + a_j$$

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

$$g_j \sqrt{n_j} = \beta_0 \sqrt{n_j} + \beta_1 \bar{x}_{1j} \sqrt{n_j} + \beta_2 \bar{x}_{2j} \sqrt{n_j} + a_j \sqrt{n_j}$$

and $\text{Var}(\bar{g}_j \sqrt{n_j}) = \sigma^2_a$. The mean square error $s^2_t$ reported by estimating this transformed regression is an estimate of $\sigma^2_u$. Alternatively, the coefficients and covariance matrix could be obtained by \texttt{aweight}ed \texttt{regress}. The only difference would be in the reported mean square error, which per equation 1, is $\sigma^2_a / \bar{n}$. 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 $\sigma^2_u$ by multiplying the reported mean square error by $\bar{n}$.

More generally, \texttt{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 $\alpha a_j$, where $\alpha$ 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 $\text{Var}(u_j) = k \alpha a_j \sigma^2$, 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 $\sqrt{n_j}$,

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

produces a model with $\text{Var}(y_j / \sqrt{n_j}) = k \alpha \sigma^2$, which is the constant part of $\text{Var}(u_j)$. Notice in particular that this variance is a function of $\alpha$, 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

\texttt{. regress y x1 x2 [aweights=1/a]}

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

---

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 \texttt{merge} command merges a \texttt{.dta} file into the current data set but it will not handle raw data or dictionary files. My \texttt{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(#) no label 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

```
.print 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 1.273875
  7 .4236486
  8 -1.741924
  9 .4869484
.print order using indict1
.list
```
mergedct can also merge the same data stored in a raw, rather than a dictionary, file:

```
use dta/, clear
type inraw/1/.raw
mergedct order using inraw/1/, 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  -.6571347   3
6.   6  -1.067126   1.272875   3
7.   7   1.471047  -.4236496   3
8.   8  -1.971935  -.7149242   3
9.   9   1.805707  -.4869484   3
10.  10  -.5496167  .7764114   3
```

mergedct can handle match merging as well:

```
use dta, clear
type indict/2/.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 collapsed 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 $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
.list lastbday = 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= 5068)
   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 $S_1$
$S_1$: 29
.disp $S_2$
$S_2$: Saturday
.disp $S_3$
$S_3$: Sat
.disp $S_4$
$S_4$: Sat.
.today
July 1, 1994
```
Note that today did not mistakenly put “Sat.” in S₄. today happens to use only S₁–S₃, leaving S₄ 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 \( N = J \), 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
```
The variables can be restricted using if and in subcommands as well.

```
. mkmat x if z<9 in 7/1
. mat l x
   r[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]
   x[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.
Heinecke’s *makmat* 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 × 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 × 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 $(X'X)^{-1}X'y$, $X$ 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, $X$ is limited in the way stated but note that $X'X$ is a mere $k \times k$ matrix and, similarly, $X'y$ 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 $X$, it would still be more efficient to use *matrix accum* to form $X'X$. $X'X$, 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 $k \times k$, $n \times n$, and $n \times n$, with $k$ small and $n$ large. Terms dealing with the data are of the generic form $X'_{k_1 \times n} W_{n \times n} Z_{n \times k_2}$. ($X'X$ fits the generic form with $X = X$, $W = I$, and $Z = X$.) Matrix programming languages are not capable of dealing with the deceivingly simple calculation $X'WZ$ because of the staggering size of the $W$ matrix. For $n = 12,000$, storing $W$ requires a little more than a gigabyte of memory. In statistical formulas, however, $W$ 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 $X'WZ$ 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.

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

\[ \text{xi} \quad \text{term(s)} \]
\[ \text{xi}: \quad \text{any stata command} \quad \text{varlist with terms . . .} \]

where a term is of the form:

\[ i/\text{varname} \quad \text{or} \quad I/\text{varname} \]
\[ i/\text{varname}_1 \times i/\text{varname}_2 \]
\[ i/\text{varname}_1 \times \text{varname}_3 \]
\[ i/\text{varname}_1 | \text{varname}_3 \]

\text{varname, varname}_1, \text{and varname}_2 \text{ denote categorical variables and may be numeric or string. varname}_3 \text{ denotes a continuous, numeric variable.} \]

\text{xi} \text{ 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.} \]

\textbf{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.”

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

For instance, assume the categorical variable \texttt{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, \texttt{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,

\[ y = \beta_0 + \beta_1 \text{agegrp} + X\beta_2 \]

the effect of being in the first age group is \( \beta_1 \), 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 \( \beta_1 \), \( 4\beta_1 \), and \( 9\beta_1 \).

The solution to this arbitrariness is to convert the categorical variable \texttt{agegrp} to a set of indicator variables \( a_1, a_2, \text{and} a_3 \), where \( a_i \) is 1 if the individual is a member of the \( i \)-th age group and 0 otherwise. We can then estimate the model:

\[ y = \beta_0 + \beta_1 a_1 + \beta_2 a_2 + \beta_3 a_3 + X\beta_2 \]
The effect of being in age group 1 is now $\beta_{11}$; 2, $\beta_{12}$; and 3, $\beta_{13}$; 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

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

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

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

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,

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

but what does it mean that $\beta_{13}$ 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 $\beta_{13}$ 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 $\beta_{13}$ 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 $\beta_{13} = \beta_{13} - \beta_{11} = 1 - (-2) = 3$, exactly the same result we got in the $\beta_{12} = 0$ 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 \texttt{xi} automates.

\textbf{Using \texttt{xi}: Overview}

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

\begin{verbatim}
.xi: logistic outcome weight i.agegrp bp
\end{verbatim}

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

\begin{verbatim}
.xi: logistic outcome weight bp i.agegrp i.race
\end{verbatim}

would include indicator variables for the race group as well.

The \texttt{i.} indicator variables \texttt{xi} expands may appear anywhere in the \texttt{varlist}, so

\begin{verbatim}
.xi: logistic outcome i.agegrp weight i.race bp
\end{verbatim}

would estimate the same model.

You can also create interactions of categorical variables; typing

\begin{verbatim}
.xi: logistic outcome weight bp i.agegrp* i.race
\end{verbatim}

estimates a model including indicator variables for all \texttt{agegrp} and \texttt{race} combinations.

You can interact dummy variables with continuous variables:

\begin{verbatim}
.xi: logistic outcome bp i.agegrp*weight i.race
\end{verbatim}

And, of course, you can include multiple interactions:

\begin{verbatim}
.xi: logistic outcome bp i.agegrp*weight i.agegrp*i.race
\end{verbatim}

We will now back up and consider each of \texttt{xi}’s features in detail.

\textbf{Indicator variables for simple effects}

When you type ‘\texttt{i.varname}, \texttt{xi} internally tabulates \texttt{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 \texttt{agegrp} takes on the values 1, 2, 3, and 4. Typing

\begin{verbatim}
.xi: logistic outcome i.agegrp
\end{verbatim}

creates indicator variables named \texttt{Iagegr2}, \texttt{Iagegr3}, and \texttt{Iagegr4}. (\texttt{xi} chooses the names and tries to make them readable; \texttt{xi} guarantees that the names are unique.) The expanded logistic model then is

\begin{verbatim}
.logistic outcome Iagegr2 Iagegr3 Iagegr4
\end{verbatim}

Afterwards, you can drop the new variables \texttt{xi} leaves behind by typing ‘\texttt{drop I*}’ (note capitalization).

\texttt{xi} provides the following features when you type ‘\texttt{i.varname}’:

1. \texttt{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 ‘\texttt{drop I*}’. You do not have to do this; each time you use the \texttt{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 ‘\texttt{describe}’ or ‘\texttt{describe I*}’.
6. \texttt{xi} may be used with any Stata command (not just \texttt{logistic}).
Controlling the omitted dummy

By default, `.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`, `.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:

\[
y = b_0 + b_1 Iagegr.1 + b_2 Iagegr.3 + b_4 Iagegr.4 + u
\]

Then,

- Predicted \( y \) for `agegr` = 1 = \( b_0 + b_1 \)
- Predicted \( y \) for `agegr` = 3 = \( b_0 + b_3 \)
- Predicted \( y \) for `agegr` = 2 = \( b_0 \)
- Predicted \( y \) for `agegr` = 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 `agegr` = 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:

\[
y = b'_0 + b'_1 Iagegr.1 + b'_2 Iagegr.2 + b'_4 Iagegr.4 + u
\]

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

    global agegrp

thus clearing the macro.

In summary, `.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 `#`"

doing, if `varname` is a string,

    global varname /"xi omit string-literal"

then the specified value will be omitted.

Examples:

    global agegrp /"xi omit 3/"  (for `agegr` a string variable)
    global race /"xi omit White/"  (to restore default for `agegrp`)
    global agegrp
Categorical variable interactions

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

\begin{verbatim}
.xi: regress y i.agegrp*i.race
\end{verbatim}

results in the model:

\[
\begin{align*}
y &= a + b_2 \text{Iagegr}_2 + b_3 \text{Iagegr}_3 + b_4 \text{Iagegr}_4 + c_2 \text{Irace}_2 + c_3 \text{Irace}_3 + d_2 \text{IaXr}_2.2 + d_3 \text{IaXr}_3.2 + d_4 \text{IaXr}_3.3 + d_4 \text{IaXr}_3.4 + u \\
\end{align*}
\]

That is,

\begin{verbatim}
.xi: regress y i.agegrp*i.race
\end{verbatim}

results in the same model as typing:

\begin{verbatim}
.xi: regress y i.agegrp i.race i.agegrp*i.race
\end{verbatim}

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:

\begin{verbatim}
.testparm IaXr*
\end{verbatim}

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

\begin{verbatim}
.xi: regress y i.race*i.agegrp
\end{verbatim}

estimates the same model.

You may also include multiple interactions simultaneously:

\begin{verbatim}
.xi: regress y i.agegrp*i.race i.agegrp*i.sex
\end{verbatim}

The model estimated is

\[
\begin{align*}
y &= a + b_2 \text{Iagegr}_2 + b_3 \text{Iagegr}_3 + b_4 \text{Iagegr}_4 + c_2 \text{Irace}_2 + c_3 \text{Irace}_3 + d_2 \text{IaXr}_2.2 + d_3 \text{IaXr}_3.2 + d_4 \text{IaXr}_3.3 + d_4 \text{IaXr}_3.4 + e_2 \text{Isx}_2 + f_2 \text{IaXs}_2.2 + f_3 \text{IaXs}_3.2 + f_4 \text{IaXs}_4.2 + u \\
\end{align*}
\]

Note that the \textit{agegrp} dummies are (correctly) included only once.

Interactions with continuous variables

1.\textit{varname}_1*\textit{varname}_2 (as distinguished from \textit{i.varname}_1*\textit{i.varname}_2, note the second \textit{i}. ) specifies an interaction of a categorical variable with a continuous variable. For instance,

\begin{verbatim}
.xi: regress y i.agegrp*wgt
\end{verbatim}

results in the model:

\[
\begin{align*}
y &= a + b_2 \text{Iagegr}_2 + b_3 \text{Iagegr}_3 + b_4 \text{Iagegr}_4 + c_2 \text{Iwgt} + d_2 \text{IaXwgt}_2.2 + d_3 \text{IaXwgt}_3.2 + d_4 \text{IaXwgt}_4.2 + u \\
\end{align*}
\]

A variation on this notation, using \textbar{} rather than \textasteriskcentered{} omits the \textit{agegrp} dummies. Typing

\begin{verbatim}
.xi: regress y i.agegrp\textbar{wgt}
\end{verbatim}
estimates the model:

\[ y = a' + c' wgt + d_2 I_{agegr\times wgt} + d_3 I_{agegr\times wgt} + d_4 I_{agegr\times wgt} \]  
(continuous \( wgt \) effect)  
\[ + u' \]  
(agegrp*\( wgt \) interactions)

The predicted values of \( y \) are:

<table>
<thead>
<tr>
<th>agegrp*( wgt ) model</th>
<th>agegrp</th>
<th>( wgt ) model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = a + c wgt )</td>
<td>( a' + c' wgt ) if agegrp = 1</td>
<td></td>
</tr>
<tr>
<td>( a + c wgt + b_2 + d_2 wgt )</td>
<td>( a' + c' wgt + d_2 wgt ) if agegrp = 2</td>
<td></td>
</tr>
<tr>
<td>( a + c wgt + b_3 + d_3 wgt )</td>
<td>( a' + c' wgt + d_3 wgt ) if agegrp = 3</td>
<td></td>
</tr>
<tr>
<td>( a + c wgt + b_4 + d_4 wgt )</td>
<td>( a' + c' wgt + d_4 wgt ) if agegrp = 4</td>
<td></td>
</tr>
</tbody>
</table>

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 \texttt{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 \( I_{rep78.1}, I_{rep78.2}, \ldots, I_{rep78.5} \). The numbers on the end are “natural” in the sense that \( I_{rep78.1} \) corresponds to \( rep78 = 1 \), \( I_{rep78.2} \) 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 \( Imake.1, Imake.2, \ldots, Imake.74 \). 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 ‘\texttt{describe}’. \( Imake.1 \) for the AMC Concord was chosen to be omitted.

**How \( \texttt{xi} \) names variables**

The names \( \texttt{xi} \) 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 ‘\texttt{Istub*}’. For example:

<table>
<thead>
<tr>
<th>name</th>
<th>I + subst + _+ groupid</th>
<th>Entire set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iagegr.1</td>
<td>I agegr _ 1</td>
<td>Iagegr*</td>
</tr>
<tr>
<td>Iagegr.2</td>
<td>I agegr _ 2</td>
<td>Iagegr*</td>
</tr>
<tr>
<td>IaXwgt.1</td>
<td>I aXwgt _ 1</td>
<td>IaXwgt*</td>
</tr>
<tr>
<td>IaXr.1_2</td>
<td>I aXr _ 1_2</td>
<td>IaXr*</td>
</tr>
<tr>
<td>IaXr.2_1</td>
<td>I aXr _ 2_1</td>
<td>IaXr*</td>
</tr>
</tbody>
</table>

**\( \texttt{xi} \) as a command rather than a command prefix**

\( \texttt{xi} \) can be used as a command prefix or as a command by itself. In the latter form, \( \texttt{xi} \) merely creates the indicator and interaction variables. Equivalent to typing,
`st: regress y i.agegrp*wgt  
i.agegrp         Iagegr_1-4  (naturally coded; Irepl8_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; Irepl8_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 bytes 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 ]  
  postexp 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:

```stata
  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 \texttt{boot} and \texttt{bsqreg} commands; see \cite{[5s] boot} and \cite{[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 \texttt{post} commands do this. In outline, their use is

\begin{verbatim}
postfile ... using ...
repeat {
    draw a sample
    make a calculation
    post ...
}
postclos
use the data set
\end{verbatim}

\textbf{Example}

Let us consider the coverage of the 95\%, \textit{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 \textit{t} distribution with \(n - 1\) degrees of freedom. (Note that as \(n \to \infty\), the \textit{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 \textit{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 $\mu$ and variance $\sigma^2$, then $z$ has median, not mean, $e^\mu$; the mean of $z$ is $e^{\mu + \sigma^2/2}$.)

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
        }
    }
    postclose
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 `postclose` 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.  Min    Max
-------------+----------------------
     mean | 10000 1.648349 2165937  1.022719 4.260587
    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 \( u \sim N(0,1) \), so the true mean of \( z \) is \( e^{1/2} \approx 1.6487213 \). Let \( \bar{x}_j \) and \( s^2_j \) represent the calculated mean and variance of the \( j \)th sample. Then the 95% confidence bounds that would be calculated by a standard \( t \) test are \( \bar{x}_j \pm t_{.95} s^2_j / 100 \). Making these calculations, we can mark each sample as rejecting or not rejecting that the mean is \( e^{1/2} \):

\[
\begin{align*}
\text{. gen } & \text{ se } = \text{ sqrt(var/100)} \\
\text{. gen } & \text{ lower } = \text{ mean } - \text{ invt(100-1,.95)*se} \\
\text{. gen } & \text{ upper } = \text{ mean } + \text{ invt(100-1,.95)*se} \\
\text{. gen } & \text{ accept } = \text{ lower<exp(1/2) & exp(1/2)<upper} \\
\text{. count if accept} \\
9198
\end{align*}
\]

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:

\[
\text{. cii 10000 9198}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. Err.</th>
<th>Binomial Exact</th>
<th>[95% Conf. Interval]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10000</td>
<td>.9198</td>
<td>.002716</td>
<td>.9142983</td>
<td>.9260475</td>
</tr>
</tbody>
</table>

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 \texttt{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:

\[
\begin{align*}
\text{. drop lower upper accept} \\
\text{. gen lower } = \text{ mean } - 1.96*se \\
\text{. gen upper } = \text{ mean } + 1.96*se \\
\text{. gen accept } = \text{ lower<exp(1/2) & exp(1/2)<upper} \\
\text{. count if accept} \\
9169
\end{align*}
\]

**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 (\texttt{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:

<table>
<thead>
<tr>
<th>replications</th>
<th>display and infile (seconds)</th>
<th>append (seconds)</th>
<th>buffered append (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5.22</td>
<td>11.81</td>
<td>8.57</td>
</tr>
<tr>
<td>500</td>
<td>25.71</td>
<td>73.77</td>
<td>40.76</td>
</tr>
<tr>
<td>1000</td>
<td>51.13</td>
<td>155.99</td>
<td>81.62</td>
</tr>
</tbody>
</table>

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

\texttt{postfile} also provides an \texttt{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 \texttt{every()}:

\[
\begin{align*}
\text{every()} & \quad \text{time (sec.)} & \quad \text{every()} & \quad \text{time (sec.)} \\
2 & 64.87 & 32 & 40.81 \\
4 & 49.49 & 64 & 41.30 \\
8 & 43.55 & 128 & 43.77 \\
16 & 41.96 & 200 & 46.41
\end{align*}
\]

Between \texttt{every(16)} and \texttt{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 ≈ 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


`ssi6.1` Simplified Monte Carlo simulations

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

The syntax of the `simul` command is

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

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

`simul` calls `proname` two ways. At the outset, `simul` issues “proname ?” and expects `proname` 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 “proname” 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 `proname` on invocation. The query call is then of the form “proname ? whatever” and subsequent calls of the form “proname whatever”.

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

Remarks

`proname` must have the following outline:

```
program define proname
if ""_""=="?" {
    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:

```stata
program define lnsim
  version 3.1
  if "i"=="?" {
    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 1.646349  .2165937 1.022719 4.280567
    var   | 10000 4.700659 6.206903 .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:

```
  . postclose
  . 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 \( N(0,1) \). 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 \( N(0,1) \).

We begin by interactively making the true data:
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
        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
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:

<table>
<thead>
<tr>
<th>N</th>
<th>post (sec.)</th>
<th>simul (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>8.13</td>
<td>8.90</td>
</tr>
<tr>
<td>500</td>
<td>49.21</td>
<td>43.83</td>
</tr>
<tr>
<td>1000</td>
<td>89.19</td>
<td>87.16</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>Command</th>
<th>Status</th>
<th>Documentation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>A</td>
<td>sts1</td>
<td>display autocorrelation plot</td>
</tr>
<tr>
<td>chow</td>
<td>C</td>
<td>—</td>
<td>perform Chow test for a shift in regression coefficients</td>
</tr>
<tr>
<td>coint</td>
<td>B</td>
<td>sts2</td>
<td>perform Engle–Granger cointegration test</td>
</tr>
<tr>
<td>csum</td>
<td>B</td>
<td>—</td>
<td>perform CUSUM test of regression stability. (Note: this name conflicts with Stata’s csum command for binary variables.)</td>
</tr>
<tr>
<td>datevars</td>
<td>A</td>
<td>sts4</td>
<td>specify date variables</td>
</tr>
<tr>
<td>dickey</td>
<td>B</td>
<td>sts2</td>
<td>perform unit root tests</td>
</tr>
<tr>
<td>dif</td>
<td>A</td>
<td>sts2</td>
<td>generate differences</td>
</tr>
<tr>
<td>downame</td>
<td>A</td>
<td>dn20</td>
<td>convert code to day-of-week name</td>
</tr>
<tr>
<td>dropoper</td>
<td>A</td>
<td>sts2</td>
<td>drop operator variables</td>
</tr>
<tr>
<td>findlag</td>
<td>B</td>
<td>sts2</td>
<td>find optimal lag length</td>
</tr>
<tr>
<td>findexpl</td>
<td>B</td>
<td>sts4</td>
<td>display sample coverage</td>
</tr>
<tr>
<td>growth</td>
<td>A</td>
<td>sts2</td>
<td>generate growth rates</td>
</tr>
<tr>
<td>growthi</td>
<td>A</td>
<td>sts2</td>
<td>immediate form of growth</td>
</tr>
<tr>
<td>lag</td>
<td>A</td>
<td>sts2</td>
<td>generate lags</td>
</tr>
<tr>
<td>lastday</td>
<td>A</td>
<td>dm20</td>
<td>calculate last day of month</td>
</tr>
<tr>
<td>lead</td>
<td>A</td>
<td>sts2</td>
<td>generate leads</td>
</tr>
<tr>
<td>lstbdoy</td>
<td>A</td>
<td>dm20</td>
<td>calculate last business day of month</td>
</tr>
<tr>
<td>mdytdow</td>
<td>A</td>
<td>dm20</td>
<td>calculate day of week from month/day/year</td>
</tr>
<tr>
<td>mnthname</td>
<td>A</td>
<td>dm20</td>
<td>convert code to month name</td>
</tr>
<tr>
<td>namedow</td>
<td>A</td>
<td>dm20</td>
<td>convert name to day of week code</td>
</tr>
<tr>
<td>namemnth</td>
<td>A</td>
<td>dm20</td>
<td>convert name to month code</td>
</tr>
<tr>
<td>pac</td>
<td>A</td>
<td>sts1</td>
<td>display partial autocorrelation plot</td>
</tr>
<tr>
<td>pearson</td>
<td>A</td>
<td>sg5.1</td>
<td>calculate Pearson correlation with p-value</td>
</tr>
<tr>
<td>period</td>
<td>A</td>
<td>sts2</td>
<td>specify period (frequency) of data</td>
</tr>
<tr>
<td>quandt</td>
<td>B</td>
<td>—</td>
<td>calculate Quandt statistics for a break in a regression</td>
</tr>
<tr>
<td>regdiag</td>
<td>B</td>
<td>sg20</td>
<td>calculate regression diagnostics</td>
</tr>
<tr>
<td>spear</td>
<td>A</td>
<td>sg5.1</td>
<td>Spearman correlation with p-value</td>
</tr>
<tr>
<td>tauprob</td>
<td>A</td>
<td>sts6</td>
<td>approximate p-values for unit root and cointegration tests</td>
</tr>
<tr>
<td>testsum</td>
<td>B</td>
<td>—</td>
<td>test whether the sum of a set of regression coefficients is zero</td>
</tr>
<tr>
<td>today</td>
<td>A</td>
<td>dm20</td>
<td>calculate today</td>
</tr>
<tr>
<td>tsfit</td>
<td>A</td>
<td>sts4</td>
<td>estimate a time series regression</td>
</tr>
<tr>
<td>tsload</td>
<td>B</td>
<td>—</td>
<td>load an ad hoc time series equation into memory</td>
</tr>
<tr>
<td>tsmult</td>
<td>A</td>
<td>st4</td>
<td>display information about lag polynomials</td>
</tr>
<tr>
<td>tspred</td>
<td>B</td>
<td>—</td>
<td>dynamically forecast or simulate a time series regression</td>
</tr>
<tr>
<td>tsreg</td>
<td>A</td>
<td>sts4</td>
<td>combined tsfit, tsmult, and regdiag</td>
</tr>
<tr>
<td>xcorr</td>
<td>A</td>
<td>sts3</td>
<td>calculate cross correlations</td>
</tr>
<tr>
<td>ystrday</td>
<td>A</td>
<td>dm20</td>
<td>calculate yesterday from today</td>
</tr>
</tbody>
</table>

For more information on these programs, type ‘help ts’ or ‘help command-name’.

Table 2: Utility programs

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>calculate autocorrelations, standard errors, and Q-statistics</td>
</tr>
<tr>
<td>_addl</td>
<td>“add” a lag operator to a variable name</td>
</tr>
<tr>
<td>_addop</td>
<td>“add” an arbitrary operator to a variable name</td>
</tr>
<tr>
<td>_recursive</td>
<td>calculate recursive residuals for a regression model</td>
</tr>
<tr>
<td>_inlist</td>
<td>determine whether a token appears in a token list</td>
</tr>
<tr>
<td>_invarlist</td>
<td>determine whether a varname appears in a varlist</td>
</tr>
<tr>
<td>_opsum</td>
<td>decode the operators (and their powers) in a varname</td>
</tr>
<tr>
<td>_pareyvl</td>
<td>parse a varlist to replace abbreviations</td>
</tr>
<tr>
<td>_subchar</td>
<td>replace one character in a string with another</td>
</tr>
<tr>
<td>_ts_zerom</td>
<td>parse a time series command and generate lags</td>
</tr>
<tr>
<td>_ts_pars</td>
<td>parse a time series command into useful macros</td>
</tr>
<tr>
<td>faketemp</td>
<td>generate temporary variable names that can be lagged</td>
</tr>
</tbody>
</table>

Reference


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 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.

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 \texttt{hansen}, an \texttt{ado}-file that calculates the Hansen test. The syntax of \texttt{hansen} is

\begin{verbatim}
  hansen varlist [if exp] [in range] [, regress tsfit-options]
\end{verbatim}

The test statistics are based on the residuals from the regression model in \texttt{varlist} estimated over the entire sample. \texttt{hansen} uses \texttt{tsfit} (Becketti 1994) to estimate the regression model, hence all the \texttt{tsfit} options can be used. \texttt{hansen} offers one additional option, \texttt{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
\[ y_t = x_t' \beta + \alpha_t, \]
\[ E(\epsilon_t | x_t) = 0, \]
\[ E(\epsilon_t^2) = \sigma_t^2, \]
\[ \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} \sigma_t^2 = \sigma^2, \ t = 1, \ldots, T \]
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, \( \beta \) and \( \sigma^2 \), are constant.

The first-order conditions for the least squares estimates of the parameters are
\[ \sum_{t=1}^{T} f_{it} = 0, \ i = 1, \ldots, K + 1 \]
where
\[ f_{it} = \begin{cases} x_{it} \hat{\epsilon}_i, & i = 1, \ldots, K \\ \epsilon_{it}^2 - \hat{\sigma}^2, & i = K + 1. \end{cases} \]
The first \( K \) first-order conditions determine \( \hat{\beta} \). The \( (K + 1) \)-st equation defines \( \hat{\sigma}^2 \) to be the maximum-likelihood (rather than the unbiased) estimator of the error variance.

Hansen’s test statistic for the \( i \)-th 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 \( i \)-th 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
\[ f_t = (f_{1t}, \ldots, f_{K+1,t})', \]
\[ S_t = (S_{1t}, \ldots, S_{K+1,t})', \] and
\[ V = \sum_{t=1}^{T} f_t f_t'. \]

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
. describe
Contains data from bankloan.dta
    vars:          11 (max= 99)
    byte:         12
    str8:         36
    int:          8
    float:        1
    date:         1
    value:        1
Sorted by:  year quarter
```

The data are:

- year: integer
- quarter: integer
- Dcc: byte
- cp rate > com bank rate dummy: credit controls of 1980 dummy
- growth rate of cashflow: growth rate of cashflow
- growth rate of C&I loans: growth rate of C&I loans
- gr rate of business fixed inv: gr rate of inventories
- change in federal funds rate: change in federal funds rate
- change in mortgage rate: change in mortgage rate
- change in 3-mo t-bill yield: change in 3-mo t-bill yield
Hansen (1992) recommends a two-step procedure: first estimate the cointegrating vector, then enter the estimated cointegrating error correction term—which is stationary—can be entered as a regressor. When the cointegrating vector is not known, applied to nonstationary regressors, thus the test cannot be applied directly to the model. If the cointegrating vector is known, occur even if the variables were in long-run equilibrium.

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 \( A'(\cdot) \) and \( B'(\cdot) \) are polynomials in the lag operator. (The \texttt{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 \( \Delta \) is the difference operator (\( \Delta x_t \equiv x_t - x_{t-1} \equiv (1 - L)x_t \)). (The \texttt{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, \( \lambda \), is a measure of how rapidly \( y_t \) responds to these deviations. Large values of \( \lambda \) 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.

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

\[
\Delta L_t = \mu \Delta L_{t-1} + \alpha_2 \Delta L_{t-2} + \beta_{1,1} \Delta I_{t-1} + \beta_{1,2} \Delta I_{t-2} + \beta_{2,1} \Delta V_{t-1} + \beta_{3,1} \Delta C_{t-1} + \\
\beta_{4,1} \Delta r_{f,t-1} + \beta_{5,1} \Delta r_{m,t-1} + \beta_{6,2} \Delta r_{m,t-2} + \beta_{6,1} \Delta r_{3,t-1} - \lambda(L_{t-1} - \delta I_{t-1} - \delta_2 V_{t-1} - \delta_3 C_{t-1} - \delta_4 r_{f,t-1} - \delta_5 r_{m,t-1} - \delta_6 r_{3,t-1})
\]

where

\[
L_t = \text{the log of bank C&I loans}, \\
I_t = \text{the log of business fixed investment}, \\
V_t = \text{the log of business inventories}, \\
C_t = \text{the log of corporate cash flow}, \\
r_{f,t} = \text{the federal funds rate}, \\
r_{m,t} = \text{the mortgage interest rate}, \\
r_{3,t} = \text{the secondary market yield on 3-month Treasury bills}.
\]
Stata Technical Bulletin 31

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 \( \delta_i \) 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 rth3, lags(2.1.1.1.2.1) static(L.ghat D731 Dcc) regress
```

Quarterly data: 1955:2 to 1992:3 (150 obs)

<table>
<thead>
<tr>
<th>Source</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
<th>Number of obs = 150</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>.04111377</td>
<td>13</td>
<td>.003162598</td>
<td>F( 13, 136) = 23.02</td>
</tr>
<tr>
<td>Residual</td>
<td>.018666145</td>
<td>136</td>
<td>.000137398</td>
<td>Prob &gt; F = 0.0000</td>
</tr>
<tr>
<td>Total</td>
<td>.059799915</td>
<td>149</td>
<td>.000401342</td>
<td>R² = 0.6875</td>
</tr>
</tbody>
</table>

| ci | Coef. | Std. Err. | t | P>|t| | [95% Conf. Interval] |
|----|-------|-----------|---|-----|-----------------|
| L.c1 | .5962585 | .061186 | 7.344 | 0.000 | .4557012 | 756.159 |
| L2.c1 | -.0936603 | .0791853 | -1.183 | 0.239 | -.2502542 | .6029335 |
| L.finr | .1191324 | .0585116 | 2.036 | 0.044 | .0034221 | .2348428 |
| L2.finr | .0383112 | .0597999 | .672 | 0.502 | -.0746999 | .1509923 |
| L.invb | .1985529 | .1010309 | 1.965 | 0.051 | -.0021221 | .398328 |
| L.cash | -.0066775 | .0339981 | -0.170 | 0.865 | -.0717812 | .0603633 |
| L.rff | .4328981 | .2587728 | 1.635 | 0.099 | -.3899428 | .0355638 |
| L.rmort | .1645535 | .2692366 | 0.635 | 0.527 | -.3480907 | .6771977 |
| L2.rmor | -.1926119 | .2218472 | -0.868 | 0.387 | -.6312383 | .2461045 |
| L.rth3 | .4319183 | .316005 | 1.366 | 0.174 | -.1933273 | 1.056511 |
| L.ghat | .0005253 | .0008624 | 0.297 | 0.767 | -.0001836 | .0011433 |
| D731 | .0770014 | .0120174 | 6.407 | 0.000 | .0532936 | .1007666 |
| Dcc | -.1750361 | .0728567 | 2.428 | 0.016 | -.0556489 | .0258632 |
| _cons | .0033333 | .0026682 | 1.444 | 0.151 | -.0041888 | .0008947 |

Individual Statistics

L.c1 = .20142736
L2.c1 = .15361964
L.finr = .19617365
L2.finr = .2406889
L.invb = .09370907
L.cash = .31049106
L.rff = .05297014
L.rmort = .0243446
L2.rmor = -.040725
L.rth3 = .03389596
L.ghat = .38104943
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 \texttt{tsfit}, \texttt{hansen} also utilizes a program called \texttt{mkmat} which stores a variables as a column vector (Heinecke 1994). Matrices in Stata are constrained by the \texttt{matsize}, a parameter that has a limit of 400 in the Unix and Intercooled versions of Stata ([5u] \texttt{matsize}). Thus, in its current form, \texttt{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

zz3.4 Computerized index for the STB (Update)

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