STATA TECHNICAL BULLETIN

May 1997
STB-37

A publication to promote communication among Stata users

Editor

H. Joseph Newton
Department of Statistics
Texas A & M University
College Station, Texas 77843
409-845-3142
409-845-3144 FAX
stb@stata.com EMAIL

Associate Editors

Francis X. Diebold, University of Pennsylvania
Joanne M. Garrett, University of North Carolina
Marcello Pagano, Harvard School of Public Health
James L. Powell, UC Berkeley and Princeton University
J. Patrick Royston, Royal Postgraduate Medical School

Subscriptions are available from Stata Corporation, email stata@stata.com, telephone 979-696-4600 or 800-STATAPC, fax 979-696-4601. Current subscription prices are posted at www.stata.com/bookstore/stb.html.

Previous Issues are available individually from StataCorp. See www.stata.com/bookstore/stbj.html for details.

Submissions to the STB, including submissions to the supporting files (programs, datasets, and help files), are on a nonexclusive, free-use basis. In particular, the author grants to StataCorp the nonexclusive right to copyright and distribute the material in accordance with the Copyright Statement below. The author also grants to StataCorp the right to freely use the ideas, including communication of the ideas to other parties, even if the material is never published in the STB. Submissions should be addressed to the Editor. Submission guidelines can be obtained from either the editor or StataCorp.

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

Contents of this issue

		page
an64.	STB-31–STB-36 available in bound format	2
an65.	Memorium for Stewart West	2
dm44.	Sequences of integers	2
dm45.	Changing string variables to numeric	4
dm46.	Enhancement to the sample command	6
dm47.	Verifying value label mappings	7
gr26.	Bin smoothing and summary on scatter plots	9
ip17.	While loops from the command line	12
ip18.	A command for randomly resampling a dataset	17
smv7.	Inference on principal components	22
ssa9.	Cox proportional hazards model with the exact calculation for ties	24
ssa9.1.	Survival analysis subroutine for programmers	26
zz7.	Cumulative index for STB-31–STB-36	30

an64	STB-31–STB-36 available in bound format

Patricia Branton, Stata Corporation, stata@stata.com

The sixth year of the *Stata Technical Bulletin* (issues 31–36) has been reprinted in a bound book called *The Stata Technical Bulletin Reprints, Volume 6*. The volume of reprints is available from StataCorp for $25, plus shipping. Or, you may purchase all six reprint volumes for $115, plus shipping. Authors of inserts in STB-31–STB-36 will automatically receive the book at no charge and need not order.

This book of reprints includes everything that appeared in issues 31–36 of the STB. As a consequence, you do not need to purchase the reprints if you saved your STBs. However, many subscribers find the reprints useful since they are bound in a convenient volume. Our primary reason for reprinting the STB, though, is to make it easier and cheaper for new users to obtain back issues. For those not purchasing the *Reprints*, note that *zz7* in this issue provides a cumulative index for the sixth year of the original STBs.

an65	Memorium for Stewart West

Joseph Hilbe, Arizona State University, atjmh@asuvm.inre.asu.edu

Stewart West, a member of the first editorial board of the *Stata Technical Bulletin*, died May 3, 1997 at Methodist Hospital in Houston, Texas. In addition to helping formulate the initial direction of the STB, Dr. West was instrumental in the creation of the well-used `epitab` commands and reviewed many other commands of biostatistical import.

Stewart was born June 27, 1947. He attended Rice University and received his Ph.D. from the University of Texas School of Public Health. While living in Chicago from 1978–1985, he was Assistant Director of the Coordinating Center for the Study of Sickle Cell Disease, a multi-center epidemiologic study. He returned to Houston as an Associate Professor of Biostatistics at the Baylor College of Medicine in 1985. Much of his subsequent research involved cardiovascular disease. He also consulted to a number of pharmaceutical and software development firms. His favorite hobbies included baseball and traveling.

I, together with all others in the Stata community, will miss Stewart's wit and statistical insight. Stewart made many positive contributions to life and work. He will be missed. We give our best regards and heartfelt condolences to Leslie, his wife of many years, and to the remainder of his family. A Memorial Scholarship Fund has been established in the name of M. Stewart West. Contributions may be sent to: Dr. L. Whitehead, U. T. Houston, P.O. Box 20186, Houston, TX 77225.

dm44	Sequences of integers

Nicholas J. Cox, University of Durham, UK, FAX (011) 44-91-374-2456, n.j.cox@durham.ac.uk

Syntax

The syntax for the `seq` command is

`seq` *newvar* [`if` *exp*] [`in` *range*] [, `by(`*byvar*`)` `block(#)` `from(#)` `to(#)`]

Options

`by(`*byvar*`)` specifies that `seq` is to generate the sequence within each group defined by the `by` variables.

`block(#)` specifies that each integer is repeated in a block of #. The default is 1.

`from(#)` specifies that integers start at #. The default is 1.

`to(#)` specifies that integers stop at #. The default is `_N`, or the number of the last value reached as determined by whatever combination of `by`, `if`, and `in` has been issued. If there are more values, numbering restarts at the value set by `from`.

Remarks

`seq` creates a new variable containing one or more sequences of integers. It is principally useful for quick creation of observation identifiers or automatic numbering of levels of factors or categorical variables.

`seq` revisits territory covered by Jacobs (1992) and Lachenbruch (1992). They provided short programs for generating sequences of integers that restart (e.g. $1, 2, 3, 1, 2, 3, 1, 2, 3, \ldots$) or are blocked (e.g. $1, 1, 1, 2, 2, 2, 3, 3, 3, \ldots$). `seq` generalizes such ideas modestly to allow the use of initial integers other than 1 and the production of decreasing sequences, and to support `by`, `if`, and `in`.

`seq` falls short of the full functionality offered by (e.g.) combinations of `seq( )` and `rep( )` in S and S-Plus, but includes the features offered by `%gl` in GLIM.

`seq` may be compared with `range`. See [R] **range**. Note that `range` allows direct creation of decreasing sequences, which is not obvious from the manual. However, it cannot produce sequences of integers that restart or are blocked.

If the user specifies `if` or `in`, values of *newvar* are calculated only for those observations specified. If the user specifies `by`, values of *newvar* are calculated separately for each value of *byvar*.

Naturally, it is the user's responsibility to ensure that observations are in the appropriate order when `seq` is issued.

Examples

In the simplest case

```
. seq a
```

is just equivalent to the common idiom

```
. gen a = _n
```

The variable `a` may also be obtained from

```
. range a 1 _N
```

(the actual value of `_N` may also be used).

In more complicated cases, `seq` with option calls is equivalent to nimble fingerwork with those versatile functions `int` and `mod`.

```
. seq b, b(2)
```

produces integers in blocks of 2, while

```
. seq c, t(6)
```

restarts the sequence after 6 is reached.

The command

```
. seq d, f(10) t(12)
```

shows that sequences may start with integers other than 1, and

```
. seq e, f(3) t(1)
```

shows that they may decrease.

Suppose we have 12 observations in memory. The results of these commands are shown by

```
. list a b c d e

            a          b          c          d          e
 1.         1          1          1         10          3
 2.         2          1          2         11          2
 3.         3          2          3         12          1
 4.         4          2          4         10          3
 5.         5          3          5         11          2
 6.         6          3          6         12          1
 7.         7          4          1         10          3
 8.         8          4          2         11          2
 9.         9          5          3         12          1
10.        10          5          4         10          3
11.        11          6          5         11          2
12.        12          6          6         12          1
```

Each of these sequences could have been generated in one line with `gen` and use of the `int` and `mod` functions. The variables `b` through `e` are obtainable by

```
. gen b = 1 + int((_n - 1)/2)
. gen c = 1 + mod(_n - 1, 6)
. gen d = 10 + mod(_n - 1, 3)
. gen e = 3 - mod(_n - 1, 3)
```

Nevertheless, `seq` may save users from puzzling out such solutions, or indeed from typing in the needed values.

References

Jacobs, M. 1992. dm5: Creating a grouping variable for data sets. *Stata Technical Bulletin* 5: 3. Reprinted in *Stata Technical Bulletin Reprints*, vol. 1, p. 30.

Lachenbruch, P. A. 1992. dm9: An ANOVA blocking utility. *Stata Technical Bulletin* 7: 4–5. Reprinted in *Stata Technical Bulletin Reprints*, vol. 2, p. 28.

dm45	Changing string variables to numeric

Nicholas J. Cox, University of Durham, UK, FAX (011) 44-91-374-2456, n.j.cox@durham.ac.uk
William Gould, Stata Corporation, wgould@stata.com

Syntax

`destring` [*varlist*] [`, noconvert noencode float`]

`destring` changes string to numeric variables if such a change is possible:

1. It changes a string variable containing `"1"`, `"2"`, `"3.14"` to a numeric variable containing 1, 2, and 3.14, but would leave a variable containing `"1"`, `"2"`, `"3.14"`, and `"fifty"` unchanged because `"fifty"` is not a number. This is called changing by conversion or retyping.
2. It changes a string variable containing `"no"` and `"yes"` to a numeric variable containing 1 and 2 but would leave a variable containing `"no"`, `"yes"`, and `"50"` unchanged because `"50"` looks like a number. This is called changing by encoding. A value label labeling 1 and 2 as no and yes is also created.

`destring` is designed to be a safer and more convenient alternative to

1. `generate` *newvar* `= real(`*strvar*`)`
2. `encode` *strvar*`, gen(`*newvar*`)`

Options

`noconvert` and `noencode` forbid `destring` from changing certain kinds of strings. You might specify one or the other option but never should you specify both, because then `destring` would have nothing to do.

`noconvert` forbids conversion or retyping changes such as changing a variable containing `"1"`, `"2"`, and `"3.14"` to contain 1, 2, and 3.14.

`noencode` forbids encoding changes such as changing a variable containing `"female"` and `"male"` to contain 1 and 2.

Specifying either of these options does not affect how `destring` changes variables it does change: the options merely prevent `destring` from changing certain kinds of variables.

`float` indicates that in converting or retyping, storing the result as `float` rather than `double` is adequate when results contain digits to the right of the decimal point.

Remarks

The great divide among data types in Stata is between numeric data types (`byte`, `int`, `float`, `long` and `double`) and string data types. Conversions between different numeric types and between different string data types are essentially a matter of using data types large enough to hold values accurately, yet small enough to economize on storage. Quite often these changes are carried out automatically by Stata or easily achieved using `recast` or `compress`.

Yet there are also problems needing more drastic changes of data types. This insert describes a utility, `destring`, for changing string variables to numeric, whenever that can be achieved without loss of information. Suppose you have a string variable *strvar*. You can always take whatever can be interpreted as numeric and put it into a numeric variable *numvar* by

```
. gen numvar = real(strvar)
```

but a side-effect of this is that whatever cannot be interpreted as numeric will be converted to numeric missing values. This is also true of the `conv2num` utility written by Farmer (1996), except that `conv2num` is a shade more indulgent than the `real()` function: it recognizes and handles suffixes such as those in `12.5%` and in `72d`, which are commonly attached to numbers by some other programs, notably various spreadsheets.

Note incidentally that (for example)

```
. recast int strvar
```

is not allowed in Stata, even if *strvar* contains values that could all be interpreted as `int`s.

The attitude underlying destring is rather different from that implicit in the use of real() or conv2num. It is assumed that you want any changes to be safe, so that there is no loss of information in the change, and that no change is made otherwise. However, we take it that whatever, when trimmed of any leading and trailing blanks, reduces either to an empty string "" or to a single period "." can be treated as missing data.

Modes and moods

destring works in two modes and in two moods, depending on how it is invoked, so we need to explain the possibilities. Let us enter a little dataset, also available as destring.dta.

```
. input n str1 (s1-s5)
            n         s1         s2         s3         s4         s5
  1. 1 1 1 1 I a
  2. 2 2 2 2 2 b
  3. 3 3 3 3 3 c
  4. 4 4 "" "." 4 d
  5. end
```

n is a numeric variable. s1, s2, s3, s4, and s5 are all string variables. s1 allows the same numeric interpretation as n. s2 and s3 may also be interpreted as numeric if the empty string value of s2 and the period of s3 in observation 4 may be interpreted as numeric missing values. s4 contains one non-number, I in observation 1. s5 is in a sense opposite to n: none of its values is a number. However, we can set up a mapping between the integers in n and the strings in s5, and in that sense they convey precisely the same information.

The first mode of destring is conversion or retyping. In this mode, s1, s2 and s3 would all be converted to numeric variables, but nothing would be done to s4 or s5.

In conversion mode, the original string variable is replaced by a numeric variable, and compressed to the most compact numeric data type possible. In fact, it starts out in life as a variable of type double. If you are short on memory, and you are confident that type float will be sufficient, you may use the float option.

The second mode of destring is encoding. In this mode, s5 would be replaced by a numeric variable with its original string values as value labels, but nothing would be done to s1, s2, s3 or s4. Note that this differs from the existing encode command (see [R] **encode**), in that encode creates new variables, while destring changes existing variables.

In encoding mode, the original string variable will also be replaced by a numeric variable of the most compact data type possible.

By default, destring is allowed both to convert and to encode. It is a two-armed robot allowed to use both arms, unless forbidden to use one arm or the other by the option noconvert or the option noencode. (If forbidden to use both, it can only express puzzlement over why it has been invoked.)

Further, destring has two moods, one terse and one talkative. First, issuing just

```
. destring
```

without specifying a *varlist* puts destring in terse mood. destring assumes that you want to change whatever variables in your dataset can be taken as numeric into numeric. Often you will know that many, if not virtually all, of your variables are already numeric, and so require no action. destring behaves like compress (see [R] **compress**), and reports only on those variables whose data types were changed. If even this terse mood is too loquacious for you, preface destring with quietly (see [R] **quietly**).

Second, specifying a *varlist* puts destring in talkative mood. It assumes that you are interested specifically in those variables named, and would welcome a report of any difficulty, just in case you have some misapprehension about the data. Difficulties are of various kinds:

1. You try to convert a variable in the *varlist* that is not a string variable.
2. You try to convert a string variable in the *varlist* which contains one or more characters that can not be regarded as numeric (for example, there remains a lurking I for 1 that escaped your scrutiny).
3. A string variable cannot be encoded properly, because it contains too many distinct string values, or it contains strings longer than 8 characters.
4. A string variable cannot be encoded properly, because it contains one or more strings that have purely numeric meaning. We do not want to encode a variable that in some observation has "2", and have that mapped, say, to 5. There is likely to be some problem with the data in this case that should be fixed in some other way.

Examples

We will use the data from `destring.dta`.

`destring` by itself allows both conversion and encoding and evokes terse mood:

```
. destring
s1 was str1 now converted to byte
s2 was str1 now converted to byte
s3 was str1 now converted to byte
s5 was str1 now encoded into byte

. _
```

However, if you were to name `n` and/or `s4` explicitly, `destring` would be more talkative and issue a warning:

```
. destring n
n not string
r(109);

. destring s4
s4 contains values that cannot be converted to numeric
s4 contains some numeric values
r(109);
```

The two messages in the last example arise because both conversion and encoding were unsuccessful.

In the case of `s4`, if `I` in observation 1 were an error for `1`, you would need to fix it first.

```
. replace s4 = "1" in 1

. destring s4
s4 was str1 now converted to byte
```

Reasons for needing destring

There could be several reasons for needing `destring`. Most seem to arise from what were (in retrospect) simple mistakes by users or those providing the data. We will mention a couple of oddities which we have encountered.

Suppose you are entering data for categorical variables into Stata's editor. Given data for `gender`, you are going to code males as `1` and females as `2`, and then to label those codes with appropriate value labels. Now Stata's editor attempts to be as smart as possible and to divine your intentions from what you type. You need have only one lapse of absent-mindedness and type `male` in the first cell of a column of the editor and it instantly thinks "Aha! This user wants this variable to be a string", and it promptly and silently makes that variable a string variable. This also arises if you type the letter `I` at first, even if you overwrite it later and leave the editor with just `1`s and `2`s in that column: these are perfectly acceptable characters in a string variable, and the editor has no reason to change its mind.

At some point you will notice that there is a problem. When you define labels, and attempt to link them to `gender`, Stata will refuse, because `gender` is a string variable, so you will need to apply `destring`.

Another example arises if you record two-digit codes `01`, `02`, `03`, ..., `99` in a `str2` variable. At some point, you will find that you cannot label such a variable. In fact, the leading zeros are superfluous for analysis, and you would be better off `destring`ing the variable and working with a numeric variable, which you can label. If you really needed the codes for some presentation purpose—say people in your field recognize these codes and would feel deprived in some way without them—just keep a copy somewhere.

References

Farmer, R. M. 1996. dm40: Converting string variables to numeric variables. *Stata Technical Bulletin* 29: 8–9. Reprinted in *Stata Technical Bulletin Reprints*, vol. 5, pp. 48–49.

dm46	Enhancement to the sample command

Jeroen Weesie, Utrecht University, Netherlands, weesie@weesie.fsw.ruu.nl

This insert briefly describes `sample2`, an extension of the standard command `sample` that adds the possibility to sample clusters of observations (e.g., households, i.e., all individuals within the selected households) rather than individual observations (e.g., individuals). Using the `if` restriction in combination with the new `any` or `if` options, it becomes possible to draw samples from multi-record observations in which any or all of the record satisfy some logical condition.

`sample2` is backward compatible with `sample`; if no `cluster` variable is specified, `sample2` should behave the same as `sample`.

Syntax

```
sample2 # [if exp] [in range] [, by(groupvars) cluster(varname) any all keep(varname) ]
```

Description

`sample2` draws a # percent pseudo-random sample of the data in memory, thus discarding 100−# percent of the observations. Observations not meeting the optional `if` and `in` criteria are kept (sampled at 100%).

Sampling here is defined as drawing observations without replacement; see online help for `bsample` for sampling with replacement.

If you are serious about drawing random samples, you must first set the random number seed; see online help for `generate`.

Options

`by(`*groupvars*`)` specifies a # percent sample is to be drawn within each set of values of *groupvars*, thus maintaining the proportion of each group.

`cluster(`*varname*`)` specifies that an observation should be interpreted as a collection of records with the same value of the cluster variable. The cluster variable should be nonmissing. If `if` or `in` clauses are defined without specifying `any` or `all`, `if/in` should select all records associated with clusters. If `by` is combined with `cluster`, the *groupvars* should be constant within clusters.

`any` specifies that the sampling frame is comprised of all clusters for which at least one record was selected via `if/in`.

`all` specifies that the sampling frame is comprised of all clusters for which all records were selected via `if/in`.

`keep(`*varname*`)` specifies the name of a variable that specifies which observations are to be kept (value 1) or dropped (value 0). If `keep` is not specified, all obs with `keep==0` are dropped automatically.

Examples

```
. sample2 10                          (draw 10 percent sample)
. sample2 10 if race==0               (keep all the race~=0, but sample 10 percent of race==0)
. sample2 10, by(race)                (sample 10% within race)
```

Suppose you analyze a dataset of individuals in households. You want to subsample households rather than individuals. Thus, in the subsample all individuals from the sampled households should be included. This can be obtained as

```
. sample2 20, c(hrespnr)
```

To sample 50% from all households in which at least one person has income higher than 100000,

```
. sample2 50 if inc>100000, c(hrespnr) any
```

If the sample frame should consist of all households in which all persons are at least 40 years of age, we would issue

```
. sample2 50 if age>40, c(hrespnr) all
```

Finally, `cluster` and `by` can be combined. To sample 50% of households while maintaining the regional distribution,

```
. sample2 50 if age>40, c(hrespnr) all by(region)
```

dm47 Verifying value label mappings

Jeroen Weesie, Utrecht University, Netherlands, weesie@weesie.fsw.ruu.nl

Value labels are attached to categorical data to improve readability of descriptive or statistical reports on data. In addition, I find value labels useful to track unexpected values of variables—these will appear as unlabeled values in, e.g., `tabulate` output. Unlabeled values may well indicate that something is wrong about the data. However, visual inspection of a large number of `tabulate` tables is tiresome. The Stata command `codebook` produces similar output as `unlabeld`, but embedded in much other summary output.

Syntax

`unlabeld` [*varlist*] [`if` *exp*] [`in` *range*] [`,` `any` `incompl`]

Options

`any` specifies that the names of variables without associated value labels are listed.

`incompl` specifies that only reports on variables with incomplete value label mappings are to be displayed.

Example

```
. unlabeld
Variable | Label    | #val   unlabeled values
---------+----------+-------------------------------------------------------------
x1       | x1       |    5
x1m      | x1m      |    5
x2       | x2       |    5   2
x3       | x3       |    5   2 4
x4       | x4       |    5   0 1 2 3 4
```

According to this table, all values of the variables `x1` and `x1m` are value labeled, while variable `x2` has one unlabeled value (2) and `x3` has 2 unlabeled values (2 and 4). Note that `x4` takes 5 distinct nonmissing values, while 5 values are unlabeled; this suggests that the wrong value label was attached to variable `x4`. To find the variables with incomplete value labels, one can specify the `incompl` option.

```
. unlabeld, incompl
Variable | Label    | #val   unlabeled values
---------+----------+-------------------------------------------------------------
x2       | x2       |    5   2
x3       | x3       |    5   2 4
x4       | x4       |    5   0 1 2 3 4
```

`unlabeld` stores the number of variables with incomplete mappings in the global macro `S_1`. Thus, a data verification file may contain the command

```
. unlabeld, incompl
. assert $S_1==0
```

I find it useful to include variables without value labels in the table. Variables that take a small number of distinct values "should" often be value labeled.

```
. unlabeld, any
Variable | Label    | #val   unlabeled values
---------+----------+-------------------------------------------------------------
x1       | x1       |    5
x1m      | x1m      |    5
x2       | x2       |    5   2
x3       | x3       |    5   2 4
x4       | x4       |    5   0 1 2 3 4
x5       |          |    5
```

Remark

I would like to add in the table the value label values that are not actually used by any particular variable or any of the variables using the value label. This may well indicate an error. For instance, when I `recode` a categorical variable to fewer categories, I tend to forget that the value label may no longer be appropriate. (Actually, I would like Stata's `recode` to be enhanced so that value labels can be assigned in the `recode` statement.) Listing unused value labels may capture this common mistake. Unfortunately, I can not think of an effective way to obtain the value labels mapping inside my program.

gr26	Bin smoothing and summary on scatter plots

Nicholas J. Cox, University of Durham, UK, FAX (011) 44-91-374-2456, n.j.cox@durham.ac.uk

Syntax

```
binsm yvar xvar [if exp] [in range] [, width(#) bin(#) starts(string)
        quantile(#) sf(string) sonly graph_options ]
```

Options

`width(#)` means that *xvar* is to be rounded in bins of width # before calculating a summary for values that round to the same value. The default is 1, which would mean that bins are centered on multiples of 1 plus 0.5.

`bin(#)` means that the range of *xvar* is to be divided into # bins of equal width. This overrides any setting of `width`.

`starts(`*string*`)` means that bins start at the values specified. If just one value is given, that is the first start and all bins have a `width` as specified. If two or more values are given, they define the starts of all the bins and `width` is ignored. However, if the lowest start is greater than the minimum value of *xvar*, that minimum will be used to start the lowest bin. `starts` overrides any setting of `bin`. `starts` must be in increasing order, but may be separated by spaces or commas.

`quantile(#)` means that bins have as far as possible equal frequencies as determined by # quantiles of *xvar*. For example `q(4)` uses the quartiles of *xvar* to define the bins. This overrides any setting of `bin` or `width`.

`sf(`*string*`)` defines the smoothing function. The default is the mean but `count`, `iqr`, `max`, `median`, `min`, `p#`, `sd`, and `sum`, and any similar user-written function may be used.

`sonly` means that only summaries are to be plotted and that the data for *yvar* are to be suppressed. This may be useful if the numerical range (and possibly the units) of the summaries differ markedly from that of *yvar* or if the dataset is very large.

graph_options are options allowed with `graph, twoway`. The default values include `xlabel ylabel c(.J) s(oi) sort gap(6)` where `c(J)` and `s(i)` are repeated for each summary; `l1title` describes *yvar*; `b2title` describes *xvar*.

Note especially that a sensible *y*-axis title is the responsibility of the user.

Remarks

`binsm` is a development of an idea long incorporated in `graph, twoway`, where

```
. graph yvar xvar , c(m) bands(#)
```

or

```
. graph yvar xvar , c(s) bands(#)
```

connects medians of *yvar* calculated for vertical bands of *xvar*. The medians are plotted as points. You can tune the number of bands between 3 and 200.

The aim of `binsm` is to allow any reasonable summary of *yvar* to be plotted on a scatter or twoway plot for any reasonable bins of *xvar*. The default is to make bin boundaries explicit and show summaries histogram-style.

More precisely, `binsm` produces a scatter plot of *yvar* against *xvar* with one or more summaries of *yvar* for bins of *xvar*. The smoothing or summary function `sf` is by default just the mean, but in general may be any space-separated list of functions acceptable to `egen` that have syntax of the form

```
. egen newvar = mean( yvar), by( binned-version-of-xvar)
```

`count`, `iqr`, `max`, `median`, `min`, `p#`, `sd`, and `sum` are all legitimate, as is any similar user-written function. Note that `p#`, where # can be any integer from 1 to 99, is interpreted as a call to the `pctile` function of `egen` with option `p(#)`. `sf(pctile)` is equivalent to `sf(median)`, just as in `egen`.

The term *bin smoother* appears in the monograph of Hastie and Tibshirani (1990), but the method is much older. Tukey (1961) introduced the term *regressogram* to emphasize the similarity to histograms. The idea without such terminology goes back at least as far as Pearson and Lee (1903).

The results of bin smoothing are smooth only in the weak sense that bin means or medians (say) are less variable than the data from which they are computed. The results will typically be discontinuous at each bin boundary, and only exceptionally could the summaries (or their first or second derivatives) be continuous across such boundaries. Hence the method is not very attractive for smoothing if the aim is to show some gently-changing continuous curve as an estimate of the functional dependence

of *yvar* upon *xvar*. Cubic smoothing splines, for example, possess many more desirable properties. Nevertheless, bin smoothing is very easy to understand and to explain, especially to nonstatisticians or to beginning students, and it has some value as an exploratory or informal tool in examining bivariate data. For its theoretical role as a nonparametric regression estimator, see Schlee (1988) and the references there cited. Hastie and Tibshirani (1990) and Scott (1992) include examples of bin smoothing (regressograms) in recent reviews of various smoothing techniques.

Any reasonable summary of *yvar* is defined technically as any single-number summary computable by an `_g*.ado` that is written to go with `egen`. In addition to the possibilities provided with Stata as distributed, programmers looking at such `_g*.ado` files will see that it is easy to take one as a template for programming the calculation of their own favorite summary. For anyone mad enough to use it, a sample `_gmad.ado` is included with this insert that calculates the median absolute deviation from the median, named the MAD by Andrews *et al.* in 1972, but known already to K. F. Gauss in 1816, according to Hampel *et al.* (1986). With `binsm`, this would be plotted by using `sf(mad)`.

Any reasonable bins are those obtainable in one of the following ways:

1. A bin `width` (default 1) is specified and the bins start at multiples of that `width`. If `width` is 100, bins would start at 100, 200, 300, and so forth (or whatever such bins are needed to include the data).
2. If a single start is specified for the bins, that is used together with the `width`. If a start of 50 and a `width` of 100 were specified, bins would start at 50, 150, 250, and so forth. This permits the user to override the default start as determined under 1.
3. If several `starts` are specified for the bins, these define the bin boundaries, together with the maximum value and (if necessary) the minimum value of *xvar*. This permits the user to use idiosyncratic bin intervals when so desired.
4. If a number of bins is specified, the range of *xvar* is divided into that number of bins. This resembles the default in `graph, histogram`.
5. If the `quantile` option is specified, quantiles of *xvar* are used to define bins with, as far as possible, equal frequencies. The `_pctile` command is used to calculate the quantiles, which limits the number of quantiles to 20.

Examples

Here are some brief ideas on how `binsm` might be used:

1. The default is just to show the mean of *yvar* in bins. Plotting median as well as mean is a useful way of checking for well-behaved variability in the response *yvar* given the predictor *xvar*.
2. By using `min p25 median p75 max` as smoothing function, users can hybridize the scatter plot and the box plot. By using any desired percentiles (quantiles), they can customize scatter plots to show changes in conditional distribution.
3. Plotting measures of variability such as `sd`, `iqr`, and `mad` provides a handle on the amount of heteroscedasticity. In this case it might be advisable to suppress the data on the scatter plot by using the `sonly` option.

A first example is with data on 158 glacial cirques from the English Lake District (Evans and Cox 1995), found in the accompanying file `cirques.dct`. Glacial cirques are hollows excavated by glaciers that are open downstream, bounded upstream by the crest of a steep slope, and arcuate in plan around a more gently sloping floor. (More informally, they are sometimes described as "armchair-shaped".) They are common in mountain areas that have or have had glaciers present.

A scatter plot of length against width reveals a wedge-shaped scatter with marked heteroscedasticity. This is confirmed by a call to `binsm` with `sf(sd)`:

(*Graph on next page.*)

```
. binsm length width, q(6) sf(sd)
```

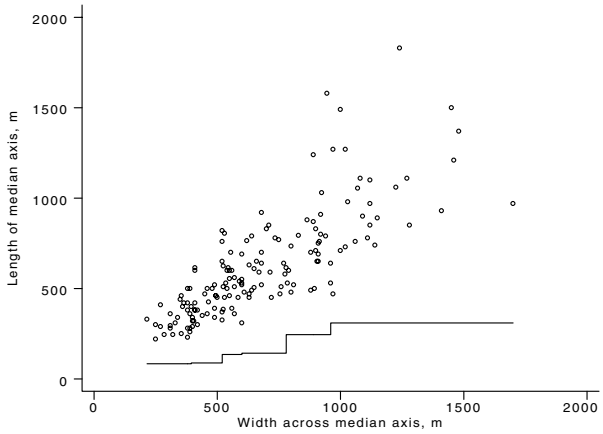

Figure 1.

As shown in Figure 1, the heteroscedasticity is systematic, with a steady increasing trend. `q(6)` was used so that each standard deviation is based on approximately the same number of values.

Logarithmic transformation seems an obvious possibility:

```
. gen logl = log10(length)
. gen logw = log10(width)
. label var logl "log 10 of length in m"
. label var logw "log 10 of width in m"
. binsm logl logw , q(6) sf(mean median) xlabel(2.4,2.6,2.8,3,3.2)
  ylabel(2.4,2.6,2.8,3,3.2)
```

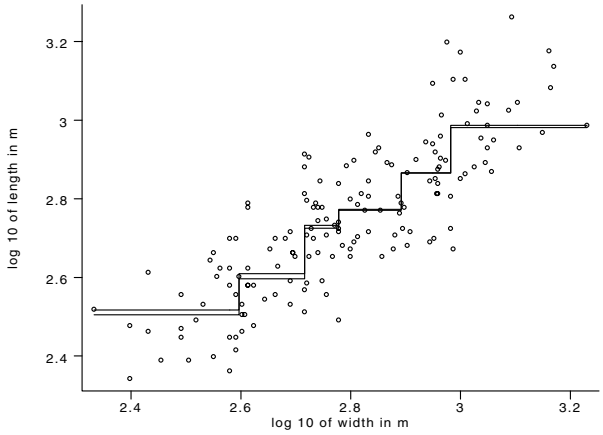

Figure 2.

The second `binsm` graph is shown in Figure 2. The scatter is much more nearly homoscedastic, and the basic similarity of bin means and medians reassuring. (On a color monitor, the lines can be distinguished more easily.)

A second example is with data on daily rainfall measured at Changshou, Sichuan, China, for 1957–1987. Rainfall was recorded on 4439 out of 11322 days. Particular interest attaches to any trends towards higher daily rainfalls, which are most problematic for erosion or flooding.

(*Graph on next page.*)

```
. binsm r y if r , xlabel(1957,1962,1967,1972,1977,1982,1987) ylabel(3,10,30,100)
  sf(median p75 p90 max) r2(median p75 p90 max) ylog sonly
```

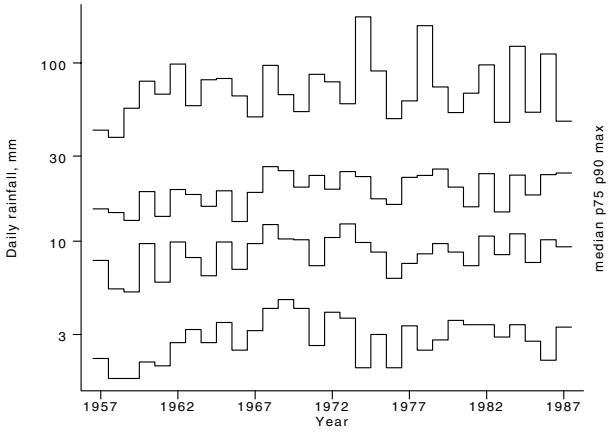

Figure 3.

The resulting graph, shown in Figure 3, suggests approximate stability over time, although four high rainfalls can be identified in the second half of the period.

Warning

As it stands, `binsm` has one notable limitation. It uses `c(J)` as the basic plotting style. This means that any empty bins will be shown with values pertaining to the last bin occupied with lower values of *xvar*. This probably will not cause great difficulties so long as the data that give rise to the plot are kept in mind. However, users trying some other kind of `connect()` are warned that `binsm` creates pseudo-observations with values equal to the bin boundaries, so that each bin can be demarcated graphically.

Acknowledgments

Ian S. Evans made helpful comments during the development of `binsm`. Lu Xixi kindly provided a copy of rainfall data for Changshou.

References

Andrews, D. F., P. J. Bickel, F. R. Hampel, P. J. Huber, W. H. Rogers, and J. W. Tukey. 1972. *Robust Estimates of Location: Survey and Advances*. Princeton, NJ: Princeton University Press.

Evans, I. S. and N. J. Cox. 1995. The form of glacial cirques in the English Lake District, Cumbria. *Zeitschrift für Geomorphologie* 39: 175–202.

Hampel, F. R., E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel. 1986. *Robust Statistics: The Approach Based on Influence Functions*. New York: John Wiley & Sons.

Hastie, T. J. and R. J. Tibshirani. 1990. *Generalized Additive Models*. London: Chapman and Hall.

Pearson, K. and A. Lee. 1903. On the laws of inheritance in man I. Inheritance of physical characters. *Biometrika* 2: 357–462.

Schlee, W. 1988. Regressograms. In *Encyclopedia of Statistical Sciences*, ed. S. Kotz and N. L. Johnson, vol. 8, pp. 1–3. New York: John Wiley & Sons.

Scott, D. W. 1992. *Multivariate Density Estimation: Theory, Practice, and Visualization*. New York: John Wiley & Sons.

Tukey, J. W. 1961. Curves as parameters, and touch estimation. In *Proceedings, Fourth Berkeley Symposium on Mathematical Statistics and Probability*, ed. J. Neymann, vol. 1, pp. 681–694. Berkeley: University of California Press.

ip17	While loops from the command line

John R. Gleason, Syracuse University, 73241.717@compuserve.com

It is often necessary to perform a series of calculations repeatedly, perhaps capturing results along the way; simple simulations and the iterative solution of equations are familiar examples. Stata's `while` loop provides exactly this capability, but it is not directly available from the command line. So, to build even a very simple loop, a novice must learn the rudiments of Stata programming, including macro manipulation, and probably also the details of creating and storing text files where Stata can find them. This insert offers two commands, `loop` and `loopdef`, that permit `while` loops to be built and used interactively, from the command line.

`loop` builds a `while` loop, then fetches and executes commands until some condition (the *guard*) is satisfied. The user need only specify the guard and the commands that form the body of the loop. The command sequence (or, loop body) can be included in a `loop` option or created beforehand, perhaps with the accessory program `loopdef`. In the latter case, the loop body can be built either by typing on the command line or by recalling previously issued commands. (In windowed versions of Stata, clicking in the Review window does the trick.) In this way, worthwhile computations can be designed and run by users with little or no knowledge of Stata programming, and without using a text editor. In fact, `loop` and `loopdef` were designed to enable undergraduates using StataQuest to explore the bootstrap and other simulation-based methods. (Each of the examples below runs nicely in StataQuest.)

We begin with a simple example and defer, briefly, presentation of the formal syntax of `loop` and `loopdef`.

Example 1

Think of the dataset `readpoor.dta` (included with this insert) as a simple population:

```
. use readpoor, replace
(Poverty and 3rd Grade Reading)
. describe
Contains data from readpoor.dta
 obs:            124                          Poverty and 3rd Grade Reading
 vars:             4                          22 Feb 1997 13:22
 size:         1,364 (99.8% of memory free)   * _dta has notes
-------------------------------------------------------------------------------
   1. school      byte   %8.0g                 Elementary school
   2. district    byte   %8.0g                 School district
   3. reading     byte   %9.0g                * % reading mastery, 3rd grade
   4. eligible    float  %9.0g                * % students free lunch eligible
                                              * indicated variables have notes
-------------------------------------------------------------------------------
Sorted by:  school
```

Imagine that you wanted to demonstrate the process of sampling without replacement and calculating the mean of the variable `eligible` for samples of size $n = 10$. Rather than using Stata's `sample` command, you might type

```
. gen U = .
. replace U = uniform()
. sort U
. summ eligible in 1/10
```
(*responses from Stata suppressed*)

and repeat the last three commands a few times. If you were accustomed to programming in Stata, you might even write a simple program to repeat those three commands, something resembling

```
program define x
   local i = 0
   while `i´ < `1´ {
      local i = `i´ + 1
      replace U = uniform()
      sort U
      summ eligible in 1/10
    }
end
```

so that typing `x 10` repeats the `replace/sort/summarize` command sequence 10 times.

`loop` provides a reusable form of the program `x`, one that permits the loop body `replace/sort/summarize` to be replaced with other sequences, transparently—e.g., no `.ado` file is ever created or modified. In fact, a one-liner is possible:

```
. loop, cmd(replace U = uniform\[\]; sort U; summ eligible in 1/10) iter(10) noi
(124 real changes made)
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
eligible |      10       30.47   25.63626        3.4       83.2
 (124 real changes made)
```
(*output omitted*)

In this usage, the loop body is supplied as the argument of an option, `cmd()`, with individual commands separated by the character "`;`" (rather as though the directive "`#delimit ;`" were in effect). The option `iter(10)` requests 10 iterations of the loop, and `noi` specifies that the loop is to be run `noisily`, so that output of commands within the loop body is visible.

Note the treatment of the parentheses for the `uniform()` function. Stata's parser will not permit the characters "`(`" and "`)`" within the enclosing parentheses of the option `cmd()`. Thus, any parentheses in the commands of the loop body are entered using escape sequences, "`\[`" in place of "`(`" and "`\]`" in place of "`)`". After translating such escape sequences, `loop` saves the commands given in the `cmd()` option in the global macro `LOOP_CMD`—which is also its default source for the loop body. So, `loop, iter(5)` requests another 5 iterations of the commands currently stored in `LOOP_CMD`. Similarly, the option `review` causes `loop` to display a command sequence but without executing the commands. Hence,

```
. loop, rev
-> Iteration 1:
>. replace U = uniform()
>. sort U
>. summ eligible in 1/10
```

shows the commands that would be executed five times by `loop, iter(5)`.

Entering parentheses as "`\[`" and "`\]`" may strike some as unwieldy, and a loop body can be built directly by typing

`global LOOP_CMD "`*command1*`;` *command2*`; ... "`

This avoids the difficulty with parentheses, but `loopdef` provides a more flexible approach. `loopdef` creates, replaces, or appends to an existing loop body by prompting for and accepting command line input. For our example, the process resembles

```
. loopdef
Replacing LOOP_CMD
Enter: command <Return> command <Return> ... command <Return><Return>
>. replace U = uniform()
>. sort U
>. summ eligible in 1/10
>.
```

In response to each prompt (`>.`), the user types a command, or a null line to exit. Alternatively, the user can insert recently issued commands by using PrevLine (*PgUp* in Windows) or, in windowed versions of Stata, by clicking in the Review window.

Syntax of loopdef

The syntax of `loopdef` is

`loopdef [, append macro(`*macname*`) ]`

`loopdef` accepts a series of Stata commands from the command line and stores them, separated by the character "`;`", in a global macro.

Options for loopdef

`append` requests that commands be appended to the current contents of the selected macro.

If the option `macro(`*macname*`)` is present, the commands are stored in the macro *macname*; otherwise, the default macro `LOOP_CMD` is used. By default, `loopdef` creates the selected global macro, or replaces its contents if it already exists.

Syntax of loop

The syntax of `loop` is

`loop [, cmd(`*cmdseq*`) iter(#) more noisily review step verbose while(`*wguard*`) ]`

Options for loop

The option `cmd(`*cmdseq*`)` determines the source of the commands to be executed by `loop`. If `cmd(`*cmdseq*`)` is absent, `loop` tries to read commands from the global macro `LOOP_CMD`. If `cmd(`*cmdseq*`)` is present, *cmdseq* is either a sequence of Stata

commands separated by ";", or the name of a global macro containing such a command sequence, preceded by the keyword "MACRO". (Note the capitalization.) In the former case, the character "(" must be entered as "\[" and the character ")" as "\]"; after those pairs are translated to the appropriate parenthesis characters, the command sequence is saved in the global macro LOOP_CMD.

iter(#) controls the maximum number of iterations performed by loop; the default # is 1.

more causes loop to insert a more command after executing each command in the loop body, so that any displayed results can be examined.

noisily controls loop's overall noisiness. If present, the loop is defined as "noisily while *some condition* {...": Unless overridden, loop body commands display their output in the Results window. If the option noisily is absent, the loop is defined as "quietly while *some condition* {...".

review invokes the verbose option and then steps through the loop body until the iteration limit is reached, displaying but never executing commands.

step inserts a single more at the end of the loop body, so that execution pauses after each iteration is complete.

verbose performs a service similar to noisily for the commands that make up the loop body. By default, only command output is displayed; verbose causes each command to be echoed to the Results window before it is executed, and also displays an iteration counter.

while(*wguard*) allows entry to the loop as long as the condition *wguard* remains true *and*, when the iter(#) option is present, if the iteration limit has not expired. That is, if both the iter(#) and while(*wguard*) options are given, the loop is defined as while (*iterations* $<$ #) & (*wguard is true*) {...

Only the relevant logical condition is present if only one of iter() or while() is given; exactly one iteration occurs if neither of the iter() and while() options is present.

Example 2

There is another feature of the loop command, one that requires a demonstration. loop makes it possible to construct worthwhile simulations from the command line; we illustrate this point with the command resample (article *ip18* in this issue). Briefly, resample draws a random sample with replacement of a *varlist* from the current data, adding the resampled variables to the current data. Placing resample inside a loop body generates a stream of bootstrap samples.

To illustrate, suppose we wish to simulate the bootstrap distribution of the correlation r between reading and eligible in readpoor.dta, using $B = 200$ random resamples of the $n = 124$ data points. First, we need to make room for $B = 200$ values of r, create a variable to hold the simulated r's, and set the seed for uniform():

```
. use readpoor, replace
(Poverty and 3rd Grade Reading)
. set obs 200
obs was 124, now 200
. gen r_boot = .
(200 missing values generated)
. set seed 970211
```

A bootstrap distribution, and bootstrap standard error for r are then but one and two commands away, respectively:

```
. loop, cmd(resample reading eligible in 1/124; corr reading_ eligibl_ in 1/124;
           replace r_boot=_result\[4\] in I_) i(200)
. summ r_boot
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
  r_boot |     200   -.7530506   .0345884  -.8444534  -.6398583
```

In this example, the variables reading_ and eligibl_ are a bivariate random resample of the variables reading and eligible, created by the command resample; see article *ip18* in this issue for details. Following the correlate command, _result(4) holds the value of r and the trick is to deposit that value, on iteration i, in the ith observation of the variable r_boot. loop's iteration counter is the global macro I_, so in a program one would write replace r_boot=_result(4) in $I_, but saving that command in another macro would cause premature evaluation of the macro I_. loop avoids this by accepting instead the range "in I_", which it translates into "in $I_" at the last moment. Note that this trick works only for the phrase "in I_", and only at the end of a command. See Remarks (below) for comments on more general usage of loop's iteration counter.

Example 3

readpoor.dta contains data from 124 elementary schools contained within 46 geographically-defined districts. The dataset sch_dist.dta holds the distribution of the number of schools per district:

```
. use sch_dist, replace
(124 schools within 46 districts)
. list
       schools       freq
  1.         1         26
  2.         2          7
  3.         3          3
  4.         4          2
  5.         5          3
  6.         6          3
  7.        11          1
  8.        23          1
. summ schools [fw=freq]
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
 schools |      46    2.695652   3.687032          1         23
```

Thus, 26 districts consist of a single school, 7 districts have 2 schools, etc. The mean number of schools per district is $\bar{x} = 2.696$, and the proportion of single-school districts is $\hat{p}_1 = 26/46$.

Suppose we wish to fit a logarithmic series distribution to this dataset, i.e., a model in which the probability of exactly k schools per district is

$$p_k = \tau^{-1}\theta^k/k, \quad k = 1, 2, \ldots, \quad 0 < \theta < 1, \quad \tau = -\log(1-\theta)$$

The maximum-likelihood estimator (MLE) $\hat{\tau}$ of τ satisfies $\hat{\tau} = \log(1 + \bar{x}\hat{\tau})$, and an easy initial estimate of τ is $\hat{\tau}_1 = -\log(\hat{p}_1/\bar{x})$. The MLE can be found by repeatedly setting $\hat{\tau}_0 = \hat{\tau}_1$, and $\hat{\tau}_1 = \log(1 + \bar{x}\hat{\tau}_0)$, after which $\hat{\theta} = 1 - \exp(-\hat{\tau}_1)$ is the MLE of θ. Plainly, we need to issue two commands not a fixed number of times, but until the two scalars are nearly equal. The while() option provides just that ability.

In our example, following the summarize command above, we can find the MLE $\hat{\tau}$ to within 0.0001 by

```
. scalar XB = _result(3)
. scalar T0 = .
. scalar T1 = -log((26/46)/_result(3))
. global LOOP_CMD "scalar T0 = scalar(T1); scalar T1 = log(1 + scalar(XB)*scalar(T0))"
. loop, while(abs\[scalar\[T0\] - scalar\[T1\]\] >= 0.0001)
```

At exit, the global macro LOOP_0 holds the number of iterations performed, so

```
. di "LOOP_0", scalar(T1), 1 - exp(-scalar(T1))
11 1.7376413 .82406512
```

shows the number of iterations, and the MLEs $\hat{\tau} = 1.738$ and $\hat{\theta} = 0.824$. The listing below displays the observed frequencies (dist) and the expected frequencies (exp_dist) from the logarithmic series distribution with $\hat{\theta} = 0.824$:

```
schools       dist   exp_dist
     1         26      21.82
     2          7       8.99
     3          3       4.94
     4          2       3.05
     5          3       2.01
     6+         3       5.19
```

Note that (a) as with cmd(), all parentheses in the argument of while() must be entered as escape sequences, and (b) since the while() condition is evaluated at the top of the loop, both the scalars T0 and T1 must be defined in advance.

Remarks

1. When the loop body is extracted from the `cmd()` option of `loop`, built with `loopdef`, or created directly as in Example 3, the result is saved in a global macro. The command sequence is then limited by the properties of Stata's macros; see [R] **limits** for length restrictions. Also, no command thus entered can contain "`"`", "`{`", or "`{`", because these characters cannot be embedded in a macro. This precludes the use of certain `display` commands, and of `if-else` constructs.
2. The limitations mentioned in (1) can be circumvented by placing commands in a do-file. Any number of the commands in the loop body can be of the form `do` *dofilename*; the commands in *dofilename* are executed directly, and do not suffer the effects of Stata's macros. This does, however, require a bit more sophistication from the user. In addition, the commands in the do-file are treated as a block so that some of `loop`'s options are applicable only to the `do` command itself. For example, `review` will display only "`>. do` *dofilename*", not the individual commands in *dofilename*.
3. Regardless of the source of the loop body, `loop`'s iteration counter is the global macro `I_`. Translation of the phrase `in I_` into `in $I_`, described in Example 2, occurs only for individual loop body commands, not for commands in a do-file. But do-file commands do not suffer from premature evaluation of macros, and so any do-file command can refer to the value of the iteration counter as `$I_` wherever it is legal to do so in a Stata command. For example, `if $I_ == 8` tests whether the current iteration is the eighth—a phrase that cannot be used in any command that is saved in a macro.
4. Nested `loop`s are possible, provided the macros containing the loop bodies do not conflict; so, there can be at most one instance of the global macro `LOOP_CMD` in any nest of `loop`s. The iteration counter is always `I_`, regardless of the depth of nesting—`loop` handles the details of managing multiple instances of `I_`. At exit, each instance of `loop` saves the number of iterations performed in global macro `LOOP_0`. If an error or *Break* causes a premature exit from any level, global macros `LOOP__0` and `LOOP__D` may be left behind; this is harmless.
5. Individual commands can be prefixed with `noisily` or `quietly`, and followed with the command `more`. This provides finer control over the display of output and over pauses than do the options `noisily`, `more`, and `step`; however, these options can be imposed or removed without editing the loop body.

Acknowledgment

This project was supported by a grant R01-MH54929 from the National Institute on Mental Health to Michael P. Carey.

References

Gleason, J. R. 1997. ip18: A command for randomly resampling a dataset. *Stata Technical Bulletin* 37: 17–22.

ip18	A command for randomly resampling a dataset

John R. Gleason, Syracuse University, 73241.717@compuserve.com

Stata offers a generous selection of commands to aid in resampling-based inference. Some examples are `bs`, `bsample`, `bstat`, `bstrap`, and `simul`, a collection that provides great power and flexibility for designing simulations, but at some cost. First, to make good use of these commands requires some knowledge of Stata programming. That, combined with an unfamiliar design (`bs` *is* different) is enough to intimidate novice users. Second, these commands overwrite the current data with a simulated dataset, which is not desirable in some problems. Third, the commands in this collection are not presently available in StataQuest, which prevents their use in some teaching situations.

This insert presents `resample`, another command for generating random samples with replacement—a less formidable sampling command in two useful senses: It requires no knowledge of Stata programming, and it uses only very basic features of Stata. In fact, `resample` and its companion program `loop` (Gleason 1997) were designed for use by undergraduates using StataQuest. (Each of the examples below runs nicely in StataQuest; see the file `sqdemo.do` that accompanies this insert.) Together, `loop` and `resample` help to bring resampling methods within the computing skills of novice Stata users.

Syntax

`resample` *varlist* [`if` *exp*] [`in` *range*] [`,` `names(`*namelist*`)`]

Description

`resample` draws a random sample with replacement from each of the *varlist* variables, sampling in casewise fashion observations where the `if` and `in` clauses are true. Thus, if there are p *varlist* variables, `resample` produces a p variate random resample, not p unrelated univariate resamples.

Options

`names(`*namelist*`)` provides a list of names for the variables created by resampling *varlist*; names not so provided are fabricated by `resample` (see Remark 2, below).

Remarks

Stata's `bsample` command also draws random resamples with replacement, but `resample` is different in several ways. `bsample` samples from *all* variables and *all* observations in memory, and the resample replaces the current data. `resample` samples from selected variables, and inserts the resample in the current data as a set of new variables named (by default) to resemble their parent variables. `resample` permits `if` *exp* and `in` *range* qualifiers so that, unlike `bsample`, it can sample from subsets of observations. The resample is stored in the observations selected by the `if` and `in` clauses; observations not selected receive missing values in the resample.

`resample` aims at simplicity and ease of use. The current data need not be sorted in any particular way, and the current sort order is restored when `resample` exits. The naming and labeling of variables in the resample is automatic, as is the replacement of an old resample by a new one each time `resample` is invoked. In effect, typing `resample varlist` leaves the data unchanged, except that a random resample has just been added to the current data (or has just replaced an earlier resample). The entire process resembles commands that are familiar to novice users, such as `generate x = y+z` or `replace a = c/d`. But, the user need not think about whether to use `generate` or `replace`, about how to name the variables for the resample, and other similar matters.

Example 1

Consider the familiar law school dataset that appears in Efron and Tibshirani (1993), and many other places:

```
. use law_sch, replace
(15 Law Schools in 1973)
. describe
Contains data from law_sch.dta
  obs:            15                          15 Law Schools in 1973
 vars:             2                          12 Feb 1997 21:00
 size:           150 (99.9% of memory free)   * _dta has notes
-------------------------------------------------------------------------------
   1. lsat        int    %8.0g                 Average LSAT score
   2. gpa         float  %9.0g                 Average undergrad GPA
-------------------------------------------------------------------------------
Sorted by:
```

The correlation between `lsat` and `gpa` is $r = 0.776$:

```
. corr lsat gpa
(obs=15)
        |     lsat      gpa
--------+------------------
    lsat|   1.0000
     gpa|   0.7764   1.0000
```

Imagine that we wish to simulate (yet again!) the bootstrap distribution of this well-studied correlation. This is, of course, a matter of resampling the variables `lsat` and `gpa` and applying the `correlate` command to the resample, some largish number of times. The first of these steps is just

```
. resample lsat gpa
```

which, without further ado, adds two new variables to the data in memory:

```
. describe
Contains data from law_sch.dta
  obs:            15                          15 Law Schools in 1973
 vars:             4                          12 Feb 1997 21:00
 size:           240 (99.9% of memory free)   * _dta has notes
-------------------------------------------------------------------------------
   1. lsat        int    %8.0g                 Average LSAT score
   2. gpa         float  %9.0g                 Average undergrad GPA
   3. lsat_       int    %8.0g                 Random resample of lsat
   4. gpa_        float  %9.0g                 Random resample of gpa
-------------------------------------------------------------------------------
Sorted by:
     Note:  data has changed since last save
```

The variables lsat_ and gpa_ are, as their labels assert, a random sample with replacement from the parent variables lsat and gpa. Correlating the resampled variables is as simple as using Prevline (*PgUp* in Windows) a time or two, and appending underscores in the earlier command corr lsat gpa. (After which, _result(4) holds the value of r.) The next replication is just resample lsat gpa, which places a new resample in lsat_ and gpa_, and corr lsat_ gpa_—a few more uses of Prevline.

To perform a real bootstrap simulation, the resample/correlate sequence must be performed many times. For that, we need a faster version of repeatedly using Prevline; the command loop (Gleason 1997) provides just that. Briefly, loop makes it possible to build and run simple while loops—otherwise available only in programs—from the command line. To illustrate, here is a complete bootstrap simulation of r for the law school data, with $B = 600$ replications:

```
. set seed 970211
. set obs 600
obs was 15, now 600
. gen r_boot = .
(600 missing values generated)
. loop, c(resample lsat gpa in 1/15; corr lsat_ gpa_ in 1/15;
          replace r_boot = _result\[4\] in I_) i(600)
```

The first three commands select a seed for uniform(), make room for, and create a variable in which to store 600 bootstrapped values of r. The loop command repeats the resample/corr commands 600 times, saving r from the ith resample in the ith observation of r_boot; see Gleason (1997) for additional details. The standard deviation of the 600 values in r_boot is the bootstrap estimate of the standard error of r, about 0.137 for these 600 replications:

```
. summ r_boot
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
  r_boot |     600    .7683948    .137143   .1929524   .9924148
```

Replacing the command resample lsat gpa in 1/15 with the two commands resample lsat in 1/15 and resample gpa in 1/15 creates resamples of lsat_ and gpa_ that are independent of each other. This would permit a bootstrap test of the hypothesis that $\rho = 0$; Example 2 provides a more interesting example of such a test.

Example 2

Consider now a bootstrap test of a hypothesis that cannot be tested by traditional methods. First, the data:

```
. use mammal42, replace
(Sleep Correlates in 42 Mammals)
. describe
Contains data from mammal42.dta
  obs:            42                          Sleep Correlates in 42 Mammals
 vars:            11                          3 Mar 1997 17:25
 size:         2,520 (99.4% of memory free)   * _dta has notes
-------------------------------------------------------------------------------
   1. species     str25  %25s                  Mammal species
   2. swav_sl     float  %9.0g                 Slow wave sleep (hrs/day)
   3. pdox_sl     float  %9.0g                 Paradoxical sleep (hrs/day)
   4. totsleep    float  %9.0g                 Total sleep (hrs/day)
   5. lifespan    float  %9.0g                 Maximum life span (yrs)
   6. is_prey     byte   %8.0g       gradelbl  Likely to be preyed upon?
   7. expos_sl    byte   %8.0g       gradelbl  Exposed during sleep?
   8. indanger    byte   %8.0g       gradelbl  In danger from other animals?
   9. lbod        float  %9.0g                 log(Body Weight, in kg)
  10. lbr         float  %9.0g                 log(Brain Weight, in g)
  11. lgest       float  %9.0g                 log(Gestation Time, in days)
-------------------------------------------------------------------------------
Sorted by:  species
```

The observations are 42 species of mammals, a subset of the data originally published by Allison and Cicchetti (1976) as part of a study of correlates of sleep in mammals. (The larger dataset is also included with this insert as mammals.dta.) Suppose that we take pdox_sl as our response variable (y), and that we wish to select a linear regression model using forward stepwise regression, choosing among $k = 8$ candidate carriers (X):

```
. sw reg pdox_sl lbod lbr lgest swav life is_p expo ind, forward pr(.4) pe(.2)
```

(*output omitted*)

```
  Source |       SS       df       MS                  Number of obs =      42
---------+------------------------------                F(  6,    35) =   12.96
   Model |  54.5372384     6  9.08953973                Prob > F      =  0.0000
Residual |  24.5427598    35  .701221708                R-squared     =  0.6896
---------+------------------------------                Adj R-squared =  0.6364
   Total |  79.0799982    41  1.92878044                Root MSE      =  .83739
------------------------------------------------------------------------------
 pdox_sl |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
   lgest |  -.4995744   .2121019     -2.355   0.024      -.9301641   -.0689848
indanger |  -1.127007   .3452461     -3.264   0.002      -1.827894   -.4261201
    lbod |   .5592599   .1487169      3.761   0.001       .2573486    .8611712
 is_prey |   .6291256     .30274      2.078   0.045       .0145307     1.24372
     lbr |  -.4629427   .1973462     -2.346   0.025      -.8635768   -.0623086
 swav_sl |   .0785652   .0519718      1.512   0.140      -.0269432    .1840737
   _cons |   5.412631   1.079146      5.016   0.000       3.221848    7.603413
------------------------------------------------------------------------------
```

The selected model has $k' = 6$ of the $k = 8$ candidate X's, with $R^2 = 0.69$ and "adjusted R-squared" $R^{2-} = 0.64$.

We may view the fit of this model with suspicion: might not stepwise selection "find" equally good models when y is unrelated to all k X's? After all, we have $n = 42$ and $k = 8$, and it is known that $E(R^2) = k/(n-1) = 0.195$ in the null case where $E(y) = \beta_0$, a constant; R^{2-} adjusts R^2 so that $E(R^{2-}) = 0$ in the same null case. But these results are for a single, fixed k-carrier model, and under the usual normal theory assumptions. What values of R^2 and R^{2-} would be likely given stepwise selection applied to *real* data where y is quite unrelated to the k X's?

A bootstrap test can answer such questions; the approach here follows Efron and Tibshirani (1993, Chapter 16). Suppose we randomly sample the response variable `pdox_sl`, and use the resample `pdox_sl_` rather than its parent in the regression, leaving the values of the carriers (X) unperturbed. Let T_0 be the value of some statistic (say R^2 or R^{2-}) from the stepwise regression of the parent variable `pdox_sl` on the 8 X's, and T_i be the same statistic when the ith resample `pdox_sl_` serves as $y, i = 1, 2, \ldots$. Calculating T_i from each of B resamples creates a bootstrap sampling distribution for T under the null hypothesis that the distribution of y is not a function of the X's. The bootstrap p value for testing that hypothesis is just $\hat{p}_B = \#(T_i \geq T_0)/B$. (Note: this is the familiar conditional model for inference in regression, where the X's are held fixed and the response y is sampled at those fixed values.)

Implementing this test is a matter of a few preliminaries and then, repeatedly: a call to `resample` and one to the stepwise regression command, followed by saving the value of T; `loop` manages the details of repeating those steps B times. We illustrate the test with $B = 100$ replications, and two versions of T. First, the preliminaries (with responses from Stata suppressed):

```
. global LOOP_CMD "resample pdox_sl in 1/42;
      sw reg pdox_sl_ lbod lbr lgest swav life is_p expo ind in 1/42, forward pr(.4) pe(.2);
      replace r2 = _result(8) in I_;
      reg pdox_sl_ lbod lbr lgest swav life is_p expo ind in 1/42;
      replace R2 = _result(8) in I_"
. gen r2 = .
. gen R2 = .
. set seed 970222
. set obs 100
```

By default, `loop` repeats commands stored in the global macro `LOOP_CMD`. In practice, `LOOP_CMD` would likely be created with the command `loopdef`, not assigned directly as in the display above; see Gleason (1997). The commands in `LOOP_CMD` generate a random resample of `pdox_sl`, use that resample as y in a stepwise regression on the 8 carriers, and save $T = R^{2-}$ in the ith observation on variable `r2`. Then, the same resample serves as y in an ordinary regression on the same carriers, and $T = R^{2-}$ from that regression is saved in the ith observation on variable `R2`.

Simulating the effects of stepwise selection is then just `loop, i(100)`, after which

```
. summ r2
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
      r2 |     100    .0690929    .085126          0   .3901406
```

reveals that the largest simulated value of R^{2-} is well below the observed value $R^{2-} = 0.64$. So the model picked by stepwise selection fits better than we would expect under the null hypothesis that `pdox_sl` is unrelated to the 8 carriers ($\hat{p}_B = 0/100$). And R^{2-} does display the expected upward bias from stepwise selection, having mean near 0.07, but a closer examination reveals more interesting effects. For example,

```
. count if r2 == 0
  34
```

shows that stepwise selection has about a 1/3 chance of choosing the correct model, i.e., of choosing $k' = 0$ of the 8 X's so that $R^{2-} = 0$. But in the remaining 2/3 of the cases, R^{2-} is typically larger than it would have been with all $k = 8$ X's in the model; recall that variable `R2` has exactly that latter value of R^{2-}:

```
. reg r2 R2 if r2
(output omitted)
------------------------------------------------------------------------------
      r2 |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
      R2 |   .6899984   .0540483     12.766   0.000       .5820245    .7979722
   _cons |   .0811955   .0059041     13.752   0.000       .0694008    .0929902
------------------------------------------------------------------------------
```

Roughly, when stepwise selection fails to eliminate all of the X's, it produces $R^{2-} \approx 0.08 + 0.7t$, where t is the value of R^{2-} from the model with all 8 of the X's included. Figures 1 and 2 show histograms of the bootstrap distributions contained in variables `r2` and `R2`; the vertical line in each plot is at the observed $R^{2-} = 0.64$.

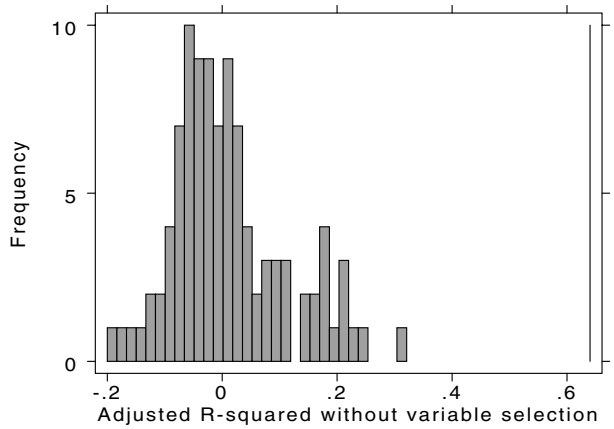

Figure 1.

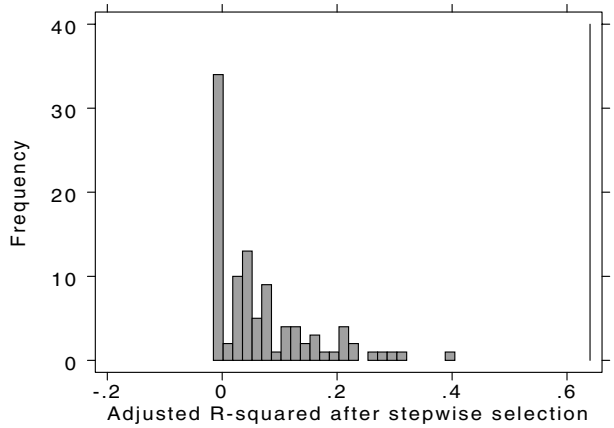

Figure 2.

Remarks

1. `resample` imposes some discipline, in the name of simplicity. First, there must be enough memory to store a copy of each *varlist* variable. Also, the resample size is always the same as the number of observations selected by the `if` and `in` clauses. Typically that's as you would want it, but there's no alternative.

2. Then, there is the matter of resampled variable names. Here is the rule that `resample` follows: (a) Name the resample of the ith *varlist* variable with the ith word of *namelist*, if present; (b) Else, append an underscore to the ith *varlist* name or, if ith name has eight characters, change the eighth character to an underscore. (c) If the result at step (b) is the same as

the ith *varlist* name, change the resampled variable name to upper case; if the result is still the ith *varlist* name, change the resampled variable name to lowercase.

3. Once it has a name, `resample` overwrites any existing variable so named with a resampled variable, and with no warning. Certainly, it is possible to name variables so that steps (b) and (c) in Remark 2 will lead `resample` to mistakenly overwrite variables that are not previous resamples of its own creation. However, some unusual habits in variable naming are required to create this situation; the `names` option exists to avoid this problem, when necessary. Beyond this, the `names` option need never be used.
4. Note that the default naming precludes use of abbreviated variable names, and that use of `resample _all` to resample all current variables is unwise. In the latter case, use instead `resample` *A* `-` *Z* where *A* and *Z* are the first and last *varlist* variables in the current ordering.
5. Missing values in the *varlist* variables are resampled, exactly as though they were not missing. But care should be taken to avoid resampling "artificial" missing values. As in Examples 1 and 2, it may be necessary to store results from $B > N$ replications, where N is the number of observations in the data. `set obs` *B* creates the necessary space, but it is then important to avoid resampling and using the missing values in the observations thus created. The `resample` command (and others) should then include a phrase resembling `in 1/`*N*.
6. The data need not be sorted in any special way, e.g., the n observations selected by the `if` and `in` clauses can be scattered throughout the current data. However, `resample` will be faster if the data are sorted so that, as in Examples 1 and 2, the selected observations are the first n observations in the dataset.
7. `resample` can be either slower or faster than `bsample`; its main virtue is that it permits resampling of selected observations and/or variables without otherwise altering the current data.

Acknowledgment

This project was supported by a grant R01-MH54929 from the National Institute on Mental Health to Michael P. Carey.

References

Allison, T. and D. V. Cicchetti. 1976. *Science* 194: 732–734.

Efron, B. and R. J. Tibshirani. 1993. *An Introduction to the Bootstrap.* New York: Chapman & Hall.

Gleason, J. R. 1997. ip17: While loops from the command line. *Stata Technical Bulletin* 37: 12–17.

smv7	Inference on principal components

Jeroen Weesie, Utrecht University, Netherlands, email: weesie@weesie.fsw.ruu.nl

The Stata command `factor` computes a principal components analysis (or, in linear algebra terms, a spectral decomposition) of a covariance or correlation matrix. Like all statistical software that I am familiar with, Stata does not compute standard errors of the principal components, or the percentage of variance explained. The required statistical theory has been available for quite a while now. Anderson (1963) and James (1964) considered the eigenvectors ("principal components") and eigenvalues of a Wishart distribution, i.e., the distribution of the sample covariance matrix S from multivariate normally distributed $N(\mu, \Sigma)$ observations. They showed that sample eigenvectors/values are consistent and are asymptotically normally distributed with an asymptotic variance matrix that can be expressed in terms of the eigenvalues and eigenvectors of Σ, and hence can be consistently estimated in terms of the spectral decomposition of S (actually, we have to impose the condition that all eigenvalues are distinct; otherwise eigenvalues are not continuous functions of a matrix, and so sample eigenvectors are inconsistent estimates of the "true" eigenvector.) Moreover, the eigenvectors and eigenvalues are asymptotically independent. Various authors have shown, however, that these asymptotic results are not very robust for violations of the normality assumption. Tyler (1981) generalizes James's results to general elliptic distributions.

Note that Stata does not compute the "correct" asymptotic standard errors for other multivariate models as well. For instance, while Anderson (1963) discusses asymptotic unconditional standard errors for canonical correlations and canonical loadings, Stata computes standard errors that are conditional on the "other" loadings without indication why the standard results were not used.

I would welcome enhancements of Stata in this direction. To stimulate this development, I submit a new command `pca` that computes the asymptotic VCE of the k largest eigenvalues and the associated eigenvectors for multivariate normally distributed observations. While the implementation allows standardization (i.e., computation of results on the correlation matrix, rather than the covariance matrix), standard errors are not adapted. Due to sensitivity to normality, the asymptotic results should be interpret cautiously. In a future version, I plan to implement Tyler's results for multivariate elliptic distributions.

Syntax

pca *varlist* [if *exp*] [in *range*] [*weight*] [, factor(#) std level(#)]

Options

factor(#) specifies the number of factors (components) to be extracted.

std specifies that the pca analysis should be performed on the correlation matrix rather than on the covariance matrix. The reported standard errors are not fully asymptotically correct for correlation matrices.

level(#) specifies the confidence level, in percent, for confidence intervals. The default is level(95) or as set by set level.

To illustrate pca, we use the standard automobile data distributed with Stata. The command

```
. pca price-gratio, factor(2) std
```

requests the 2-factor solution for the standardized variables and gives as output

```
. pca price-gratio, factor(2) std
#factors    EigenValue    %Explained     Std. Err
-------------------------------------------------
     1        6.3125        0.6312        0.0434
     2        1.0763        0.7389        0.0325
     3        1.0140        0.8403        0.0204
     4        0.4303        0.8833        0.0155
     5        0.3959        0.9229        0.0107
     6        0.2792        0.9508        0.0073
     7        0.2563        0.9764        0.0036
     8        0.1257        0.9890        0.0019
     9        0.0815        0.9972        0.0006
    10        0.0284        1.0000        0.0000
Principal components of correlation matrix         Number of obs     =      69
(Std. errors based on multivariate normality)      Number of factors =       2
                                                   Rho (%var expl.)  =  0.7389
                                                   Std Err Rho       =  0.0325
------------------------------------------------------------------------------
         |      Coef.   Std. Err.       z     P>|z|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
fac1     |
   price |   .2073618   .0479062      4.328   0.000       .1134674    .3012562
     mpg |  -.3394075   .0265159    -12.800   0.000      -.3913778   -.2874373
   rep78 |  -.1829803   .0502463     -3.642   0.000      -.2814611   -.0844994
  hdroom |   .2303667   .0455058      5.062   0.000        .141177    .3195564
   trunk |   .3003215   .0357232      8.407   0.000       .2303054    .3703377
  weight |   .3847707   .0129657     29.676   0.000       .3593584     .410183
  length |    .377097   .0159992     23.570   0.000       .3457392    .4084548
    turn |   .3542234   .0234499     15.106   0.000       .3082626    .4001843
   displ |   .3742455   .0172583     21.685   0.000       .3404199    .4080712
  gratio |  -.3305683   .0287349    -11.504   0.000      -.3868876    -.274249
---------+--------------------------------------------------------------------
fac2     |
   price |     .38764   1.303601      0.297   0.766       -2.16737     2.94265
     mpg |   .0520395   .2204905      0.236   0.813       -.380114    .4841931
   rep78 |   .7638665   .2149283      3.554   0.000       .3426147    1.185118
  hdroom |   .3048525    1.28677      0.237   0.813      -2.217171    2.826876
   trunk |   .3401359   .7019222      0.485   0.628      -1.035606    1.715878
  weight |   .0094606   .2239531      0.042   0.966      -.4294795    .4484007
  length |   .0432423   .0757097      0.571   0.568       -.105146    .1916306
    turn |  -.1831128   .0992953     -1.844   0.065       -.377728    .0115025
   displ |  -.0120712   .2630603     -0.046   0.963      -.5276599    .5035175
  gratio |   .1388243   .2278293      0.609   0.542       -.307713    .5853615
------------------------------------------------------------------------------
```

pca interfaces fully to the internal Stata code for estimation. Thus, one can use test to test hypotheses, predict to obtain factor scores etc.

References

Anderson, T. W. 1963. Asymptotic theory for principal components analysis. *Annals of Mathematical Statistics* 34: 468–488.

James, A. T. 1964. Distributions of matrix variates and latent roots derived from normal samples. *Annals of Mathematical Statistics* 35: 475–501.

Tyler, D. T. 1981. Asymptotic inference for eigenvectors. *Annals of Statistics* 9: 725–736.

ssa9	Cox proportional hazards model with the exact calculation for ties

William Gould, Stata Corporation, wgould@stata.com

The new command provided here—`stcoxe`—estimates Cox proportional hazards (CPH) models just as does Stata's `stcox` command. It differs from `stcox` in that it uses the exact calculation for failures occurring at the same time; `stcox`, on the other hand, uses the Peto–Breslow approximation (Peto 1972, Breslow 1974).

That is, each term in the CPH likelihood corresponds to a failure time. Consider a time at which there is one failure. The conditional probability of the particular unit i being the one to fail in the group at risk is

$$\mathrm{P}(i \text{ fails} \,|\, 1 \text{ fails from } R_i) = \frac{\exp(\mathbf{x}_i\mathbf{b})}{\sum_{j\in R_i}\exp(\mathbf{x}_j\mathbf{b})}$$

where R_i is the set of those at risk of failure at the time of the ith failure (which, of course, includes i itself). If each of the failure times in the data had just one failure associated with them, then the overall log-likelihood function would simply be $\sum_i \ln \mathrm{P}(i \text{ fails} \,|\, 1 \text{ fails from } R_i)$.

Assume instead that at some failure time there are two failures, i_1 and i_2. These are called tied failures or simply ties. The exact probability that i_1 and i_2 fail is

$$\mathrm{P}(i_1 \text{ and } i_2 \text{ fail} \,|\, 2 \text{ fail from } R_i) = \frac{\exp(\mathbf{x}_{i_1}\mathbf{b})\exp(\mathbf{x}_{i_2}\mathbf{b})}{\sum_{j,k\in R_i, j<k}\exp(\mathbf{x}_j\mathbf{b})\exp(\mathbf{x}_k\mathbf{b})}$$

That is what `stcoxe` calculates.

`stcox` uses the Peto–Breslow approximation. This starts by sequencing the two failure events,

$$\begin{aligned}\mathrm{P}(i_1 \text{ and } i_2 \text{ fail} \,|\, 2 \text{ fail from } R_i) =& \mathrm{P}(i_1 \text{ and then } i_2 \text{ fail} \,|\, 2 \text{ fail from } R_i) + \mathrm{P}(i_2 \text{ and then } i_1 \text{ fail} \,|\, 2 \text{ fail from } R_i)\\ =& \mathrm{P}(i_1 \text{ fails} \,|\, 1 \text{ fails from } R_i)\mathrm{P}(i_2 \text{ fails} \,|\, 1 \text{ fails from } R_i - i_1)\\ &+ \mathrm{P}(i_2 \text{ fails} \,|\, 1 \text{ fails from } R_i)\mathrm{P}(i_1 \text{ fails} \,|\, 1 \text{ fails from } R_i - i_2)\end{aligned}$$

In the above, R_i is the set of units at risk at the time of failure under consideration which includes i_1 and i_2 along with all the other units failing or censored at later times. $R_i - i_1$ refers to the risk pool with i_1 removed. $R_i - i_2$ refers to the risk pool with i_2 removed. Calculating each of the four probabilities is a straight forward application of the one-failure formula above. Similarly, if at some failure time there were three failures, we would consider all possible permutations of the ordering of the individual failures i_1, i_2, and i_3.

The Peto–Breslow approximation then approximates the above by

$$\begin{aligned}\mathrm{P}(i_1 \text{ and } i_2 \text{ fail} \,|\, 2 \text{ fail from } R_i) =& \mathrm{P}(i_1 \text{ and then } i_2 \text{ fail} \,|\, 2 \text{ fail from } R_i) + \mathrm{P}(i_2 \text{ and then } i_1 \text{ fail} \,|\, 2 \text{ fail from } R_i)\\ \approx& \mathrm{P}(i_1 \text{ fails} \,|\, 1 \text{ fails from } R_i)\mathrm{P}(i_2 \text{ fails} \,|\, 1 \text{ fails from } R_i)\\ &+ \mathrm{P}(i_2 \text{ fails} \,|\, 1 \text{ fails from } R_i)\mathrm{P}(i_1 \text{ fails} \,|\, 1 \text{ fails from } R_i)\end{aligned}$$

That is, in calculating the probability of i_2 failing after i_1 fails, the pool R_i is used rather than $R_i - i_1$ and similarly, in calculating the probability of i_1 failing after i_2, the pool R_i is used rather than $R_i - i_2$. The Peto–Breslow approximation simply fails to remove from the risk pools the failures that have already occurred. If the number of tied failures d_i relative to the size of the risk pool n_i is small, then substitution of the approximation will not affect results much.

Thus, `stcoxe` is for use when d_i/n_i is large. When $d_i = 1$ for all i, `stcox` and `stcoxe` will report the same results.

Syntax

The syntax of `stcoxe` is the same as `stcox` except that there are fewer options:

`stcoxe` *varlist* [`if` *exp*] [`in` *range*] [`, nohr strata(`*varnames*`) level(`#`) noshow`]

As with `stcox`, `stcoxe` is for use with survival-time data and you must have `stset` your data before using this command. `stcoxe` shares the features of all estimation commands. To reset problem-size limits, see [R] **matsize.** `stcoxe` differs from `stcox` in that

1. `stcoxe` uses the exact calculation for failures occurring at the same time.

2. `stcoxe` is slower and requires more memory.
3. `stcoxe` does not provide the robust method of calculating the variance-covariance matrix of the estimators.
4. `stcoxe` does not provide estimates of the baseline hazard or survivor functions nor does it provide martingale or efficient score residuals.

Note that `stcoxe` does provide stratified estimation, as does `stcox`.

`stcoxe` and `stcox` produce identical results when there are no failures occurring at the same time (no ties).

Options

`nohr` specifies that coefficients rather than exponentiated coefficients are to be displayed or, said differently, coefficients rather than hazard ratios. This option affects how results are displayed, not how they are estimated. `nohr` may be specified at estimation or when replaying previously estimated results (which you do by typing `stcoxe`).

`strata(`*varnames*`)` specifies up to 5 strata variables. Observations with equal values of the variables are assumed to be in the same stratum. Stratified estimates (equal coefficients across strata but baseline hazard unique to each stratum) are then estimated.

`level(#)` specifies the confidence level for the confidence intervals of the coefficients.

`noshow` prevents `stcoxe` from displaying the identities of the key st variables above its output.

Examples

Stata comes with a dataset named `cancer.dta` containing fictional data on a drug trial:

```
. use /usr/local/stata/cancer, clear
(Patient Survival in Drug Trial)
. stset studytim died
    dataset name:  /usr/local/stata/cancer.dta
              id:  --                    (meaning each record a unique subject)
      entry time:  --                    (meaning all entered at time 0)
       exit time:  studytim
  failure/censor:  died
. tab drug, gen(dr)
Drug type   |
(1=placebo) |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         20       41.67       41.67
          2 |         14       29.17       70.83
          3 |         14       29.17      100.00
------------+-----------------------------------
      Total |         48      100.00
```

Roughly 43% of failure times record the death of more than one person:

```
. sort studytim
. quietly by studytim: gen deaths = cond(_n==_N,sum(died),.)
. tab deaths if deaths>0
     deaths |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         12       57.14       57.14
          2 |          8       38.10       95.24
          3 |          1        4.76      100.00
------------+-----------------------------------
      Total |         21      100.00
```

Estimating a model with `stcox` (using the Peto–Breslow approximation) and `stcoxe` (using the exact calculation) results in

```
. stcox age dr2 dr3
    failure time:  studytim
  failure/censor:  died
Iteration 0:  Log Likelihood =-99.911448
Iteration 1:  Log Likelihood =-82.331523
Iteration 2:  Log Likelihood =-81.676487
Iteration 3:  Log Likelihood =-81.652584
Iteration 4:  Log Likelihood =-81.652567
Refining estimates:
```

```
Iteration 0:  Log Likelihood =-81.652567

Cox regression -- entry time 0

No. of subjects =           48                  Log likelihood =  -81.652567
No. of failures =           31                  chi2(3)        =       36.52
Time at risk    =          744                  Prob > chi2    =      0.0000
------------------------------------------------------------------------------
studytim |
    died | Haz. Ratio   Std. Err.       z     P>|z|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |   1.118334   .0409074      3.058   0.002       1.040963    1.201455
     dr2 |   .1805839   .0892742     -3.462   0.001       .0685292    .4758636
     dr3 |   .0520066    .034103     -4.508   0.000       .0143843    .1880305
------------------------------------------------------------------------------

. stcoxe age dr2 dr3
    failure time:  studytim
  failure/censor:  died

Cox regression (exact method for ties) -- entry time 0

No. of subjects =           48                  Log likelihood =  -72.972325
No. of failures =           31                  chi2(3)        =       38.40
Time at risk    =          744                  Prob > chi2    =      0.0000
------------------------------------------------------------------------------
studytim | Haz. Ratio   Std. Err.       z     P>|z|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |   1.126355   .0428041      3.131   0.002       1.045509    1.213453
     dr2 |   .1558151   .0805856     -3.595   0.000        .056543    .4293783
     dr3 |   .0405622    .028857     -4.505   0.000        .010059    .1635644
------------------------------------------------------------------------------
```

Results are similar.

	Peto-Breslow Approx.		Exact	
Variable	H.R.	S.E.	H.R.	S.E.
age	1.11	.04	1.13	.04
dr2	.18	.09	.16	.08
dr3	.05	.03	.04	.03

Methods and formulas

For a discussion of the Peto–Breslow approximation and the exact calculation, see Kalbfleisch and Prentice (1980, 72–75).

The conditional logistic regression likelihood function and the CPH likelihood functions are really the same. The difference is that in conditional logistic regression the user specifies the risk pools explicitly whereas in the CPH model the risk pools are derived from the entry, failure, and censoring times.

Stata's `clogit` command for estimating conditional logistic regressions uses the exact calculation for ties. Thus, `stcoxe` temporarily expands the data to contain the risk pools and then uses `clogit` to estimate results.

References

Breslow, N. E. 1974. Covariance analysis of censored survival data. *Biometrics* 30: 89–90.

Kalbfleisch, J. D. and R. L. Prentice. 1980. *The Statistical Analysis of Failure Time Data.* New York: John Wiley & Sons.

Peto, R. 1972. Contribution to the discussion of paper by D. R. Cox. *Journal of the Royal Statistical Society*, Series B 34: 205–207.

ssa9.1	Survival analysis subroutine for programmers

William Gould, Stata Corporation, wgould@stata.com

[*Editor's Note: The software for this insert is obtained by installing ssa9. There is no additional software to install.*]

Syntax

`st_rpool` *newvar* [`if` *exp*] [`in` *range*] [, `strata(`*varnames*`)` `nopreserve`]

`st_rpool` is for use with survival-time data; see [R] **st**. You must have `stset` your data before using this command.

Description

`st_rpool` converts the st data in memory to data on the risk pools. Variable *newvar* is created indexing the pools.

`st_rpool` requires a failure variable be `stset`.

Options

`strata(`*varnames*`)` specifies that separate pools are to be formed for the strata denoted by *varnames*. *varnames* may be string or numeric or any combination.

`nopreserve` specifies the original data is not to be preserved prior to transformation. Note, this preservation does not refer to permanent saving of the data. If `nopreserve` is not specified, the data is temporarily saved so, should anything go wrong during the transformation such as running out of memory, the original data can be brought back. If the transformation completes without problem, the preserved copy of the original data is erased.

If you wish to retain a copy of the original data, it is your responsibility to save the data first regardless of whether you specify this option.

Remarks

The risk pools R_1, R_2, ..., are each defined as the set of observations at risk at first failure time, second failure, ..., in the data. Consider the following simple data:

```
. list
          id       time       dead          x
 1.      101          1          1       6.18
 2.      102          4          1        .61
 3.      103          6          0       5.55
. stset time dead
(output omitted)
```

This data, converted to risk pools, becomes

```
. st_rpool event
. sort event id
. list
          id       time       dead          x      event
 1.      101          1          1       6.18          1
 2.      102          4          0        .61          1
 3.      103          6          0       5.55          1
 4.      102          4          1        .61          2
 5.      103          6          0       5.55          2
```

`event` $= 1$ signifies the risk pool at the time of the first failure in the data. At that time, there were three "persons" who might have potentially failed: ids 101, 102, and 103. Id 101 did fail. (We see this because `dead` $= 1$ for that observation and is 0 for the others in the risk group.)

`event` $= 2$ signifies the risk pool at the time of the second failure. At that time, there were two persons who might have failed, ids 102 and 103. Id 102 did fail.

Risk pools are formed on the basis of failure times, not the failures themselves. If there are multiple failures at a certain time, it is still one risk pool. For instance, in the following data there are two failures at time 7:

```
. list
          id       time       dead          x
 1.      201          1          1        .87
 2.      202          7          0        .26
 3.      203          7          1        .04
 4.      204          8          1        .42
 5.      205          8          1         .9
 6.      206          9          0        .52
. stset time dead
(output omitted)
```

This data, converted to risk pools, becomes

```
. st_rpool event
. sort event id
```

```
. by event: list id time dead x
-> event=        1
            id       time       dead          x
  1.       201          1          1        .87
  2.       202          7          0        .26
  3.       203          7          0        .04
  4.       204          8          0        .42
  5.       205          8          0         .9
  6.       206          9          0        .52
-> event=        2
            id       time       dead          x
  7.       202          7          0        .26
  8.       203          7          1        .04
  9.       204          8          0        .42
 10.       205          8          0         .9
 11.       206          9          0        .52
-> event=        3
            id       time       dead          x
 12.       204          8          1        .42
 13.       205          8          1         .9
 14.       206          9          0        .52
```

Note that there are two failures in the third risk pool (which corresponds to the failure time 8).

Also note that risk pools can contain single observations:

```
. list
            id       time       dead          x
  1.       301          1          1        .84
  2.       302          7          0        .21
  3.       303          7          1        .56
  4.       304          8          1        .26
  5.       305          8          1        .95
  6.       306          9          1        .28
. stset time dead
(output omitted)
. st_rpool event
. sort event id
. by event: list id time dead x
-> event=        1
            id       time       dead          x
  1.       301          1          1        .84
  2.       302          7          0        .21
  3.       303          7          0        .56
  4.       304          8          0        .26
  5.       305          8          0        .95
  6.       306          9          0        .28
-> event=        2
            id       time       dead          x
  7.       302          7          0        .21
  8.       303          7          1        .56
  9.       304          8          0        .26
 10.       305          8          0        .95
 11.       306          9          0        .28
-> event=        3
            id       time       dead          x
 12.       304          8          1        .26
 13.       305          8          1        .95
 14.       306          9          0        .28
-> event=        4
            id       time       dead          x
 15.       306          9          1        .28
```

Note that the final risk pool contains only one observation because only one "person" was left alive at that time.

`st_rpool` handles entry time and will allow stratification, meaning separate pools for separate groups. For instance,

```
. list
            id         t0       time       dead      group
  1.       401          0          4          1          1
  2.       402          0          8          0          1
  3.       403          0         12          1          1
  4.       404          9         16          0          1
  5.       405          0          4          1          2
  6.       406          4          8          0          2
  7.       407          0         12          1          2
. stset time dead, t0(t0)
(output omitted)
. st_rpool event, strata(group)
. sort event id
. by event: list id t0 time dead group
-> event=        1
            id         t0       time       dead      group
  1.       401          0          4          1          1
  2.       402          0          8          0          1
  3.       403          0         12          0          1
-> event=        2
            id         t0       time       dead      group
  4.       403          0         12          1          1
  5.       404          9         16          0          1
-> event=        3
            id         t0       time       dead      group
  6.       405          0          4          1          2
  7.       407          0         12          0          2
-> event=        4
            id         t0       time       dead      group
  8.       407          0         12          1          2
```

Example

The Cox proportional hazard models is the same as the conditional logistic model where the groups are the risk pools. Thus, one way of obtaining an estimate of the Cox model is

```
. st_rpool event
. clogit dead ..., group(event)
```

If there are no multiple failures at the same time in the data (no ties), results estimated this way will be the same as results estimated by `stcox`. The issue of ties is important because `stcox` handles ties using the Peto–Breslow approximation whereas `clogit` handles ties using an exact calculation.

In fact, `stcoxe` uses this method to estimate Cox models, see Gould (1977).

Warnings

Expanding the data to contain the risk pools is convenient for some calculations, but be warned that the resulting dataset may contain a large number of observations because of the replication. A dataset that contained 1,000 observations, each of which failed at times 1, 2, 3, ..., would contain $1{,}000 + 999 + 998 + \ldots + 1 = 500{,}500$ observations after conversion to risk pools.

If, however, there were only ten discrete times at which failures could occur, then the resulting expanded data could not contain more than $1{,}000 + 999 + \ldots + 991 = 9{,}995$ observations.

Unrelatedly, `st_rpool` requires a failure variable be `stset`. In very simple st datasets the user is required to `stset` only a time variable and then all observations are assumed to reflect failures. Before calling `st_rpool`, you must create a failure variable if one does not already exist. The appropriate call is

```
st_is
...
preserve                              /* optional but preferable */
local d : char _dta[st_d]
if "`d´"=="" {
        tempvar d
        qui gen byte `d´ = 1
        char _dta[st_d] "`d´"
}
```

```
tempvar event
st_rpool `event´ ..., nopreserve
```

You are required to make a failure variable because you will need it in interpreting the results produced by `st_rpool`. Within each pool designated by `` `event´ ``, it is the observations `` `d´!=0 `` that failed.

Reference

Gould, W. 1997. ssa9: Cox proportional hazards model with the exact calculation for ties. *Stata Technical Bulletin* 37: 24–26.

zz7 Cumulative Index for STB-31–STB-36

[an] Announcements

STB-31	2	an59	Statement from the new editor
STB-31	2	an60	STB-25–STB-30 available in bound format
STB-32	2	an61	Submission guidelines
STB-33	2	an62	Stata 5.0
STB-33	2	an63	Updates available on the Stata web site

[crc] StataCorp. Provided Support Materials

STB-32	3	crc44	Confidence intervals in a 2×2 table
STB-35	2	crc45	New options for survival-time data
STB-35	3	crc46	Better numerical derivatives and integrals

[dm] Data Management

STB-32	5	dm42	Accrue statistics for a command across a by list
STB-35	6	dm43	Automatic recording of definitions

[dt] Datasets

STB-32	9	dt3	Reading EpiInfo datasets into Stata
STB-34	2	dt3.1	An updated utility to convert EpiInfo datasets

[gr] Graphics

STB-36	2	gr16.1	Convex hull plots
STB-34	3	gr20	Low-level graphics in data coordinates
STB-34	9	gr21	Flexible axis scaling
STB-35	7	gr22	Binomial smoothing plot
STB-35	9	gr23	Graphical assessment of the Cox model proportional hazards assumption
STB-36	4	gr24	Easier bar charts
STB-36	8	gr25	Spike plots for histograms, rootograms, and time-series plots

[ip] Instruction on Programming

STB-34	10	ip13	Maximum-likelihood estimation using the ml command
STB-35	14	ip14	Programming utility: Numeric lists
STB-35	16	ip15	A dialog box layout manager for Stata
STB-36	11	ip16	Using dialog boxes to vary program parameters

[os] Operating System, Hardware, & Interprogram Communication

STB-32	10	os16	Importing Stata graphs into word processors on a Macintosh
STB-34	21	os16.1	Importing Stata graphs into word processors on the Macintosh: Part 2

[sbe] Biostatistics & Epidemiology

STB-34	24	sbe13	Age-specific reference intervals (“normal ranges”)
STB-35	21	sbe13.1	Correction to age-specific reference intervals (“normal ranges”)
STB-36	15	sbe13.2	Correction to age-specific reference intervals (“normal ranges”)
STB-36	15	sbe14	Odds ratios and confidence intervals for logistic regression models with effect modification

[sed] Exploratory Data Analysis

STB-32	12	sed2	Patterns of missing data
STB-33	2	sed2.1	Update to patterns of missing data

[sg] General Statistics

STB-33	3	sg42.1	Extensions to the regpred command
STB-33	6	sg49.1	An improved command for paired t tests: Correction
STB-32	13	sg51	Inferences about correlations using the Fisher z-transform
STB-32	18	sg52	Testing dependent correlation coefficients
STB-32	19	sg53	Maximum-likelihood complementary log-log regression
STB-32	20	sg54	Extended probit regression
STB-32	21	sg55	Extensions to the brier command
STB-32	23	sg56	An ado-file implementation of a simplex-based maximization algorithm
STB-33	7	sg57	An immediate command for two-way tables
STB-33	9	sg58	Mountain plots
STB-33	10	sg59	Index of ordinal variation and Neyman–Barton GOF
STB-33	12	sg60	Enhancements for the display of estimation results
STB-33	15	sg61	Bivariate probit models
STB-33	21	sg62	Hildreth–Houck random coefficients model
STB-35	21	sg63	Logistic regression: Standardized coefficients and partial correlations
STB-35	22	sg64	pwcorrs: An enhanced correlation display
STB-35	25	sg65	Computing intraclass correlations and large ANOVAs
STB-35	32	sg66	Enhancements to the alpha command
STB-36	23	sg67	Univariate summaries with boxplots
STB-36	26	sg68	Goodness-of-fit statistics for multinomial distributions
STB-36	29	sg69	Immediate Mann–Whitney and binomial effect-size display

[smv] Multivariate Analysis

STB-34	34	smv3.1	Discriminant analysis: An enhanced command

[snp] Nonparametric Methods

STB-32	27	snp11	Test for trends across ordered groups revisited
STB-33	24	snp12	Stratified test for trend across ordered groups

[sts] Time-Series and Econometrics

STB-32	30	sts11	Hildreth–Lu regression
STB-34	36	sts12	A periodogram-based test for white noise

[svy] Survey Sample

STB-31	3	svy1	Some basic concepts for design-based analysis of complex survey data
STB-31	6	svy2	Estimation of means, totals, ratios, and proportions for survey data
STB-31	23	svy3	Describing survey data: sampling design and missing data
STB-31	26	svy4	Linear, logistic, and probit regressions for survey data
STB-31	31	svy5	Estimates of linear combinations and hypothesis tests for survey data

[zz] Not Elsewhere Classified

STB-31	42	zz6	Cumulative index for STB-25–STB-30

STB categories and insert codes

Inserts in the STB are presently categorized as follows:

General Categories:

an	announcements	*ip*	instruction on programming
cc	communications & letters	*os*	operating system, hardware, & interprogram communication
dm	data management	*qs*	questions and suggestions
dt	datasets	*tt*	teaching
gr	graphics	*zz*	not elsewhere classified
in	instruction		

Statistical Categories:

sbe	biostatistics & epidemiology	*ssa*	survival analysis
sed	exploratory data analysis	*ssi*	simulation & random numbers
sg	general statistics	*sss*	social science & psychometrics
smv	multivariate analysis	*sts*	time-series, econometrics
snp	nonparametric methods	*svy*	survey sampling
sqc	quality control	*sxd*	experimental design
sqv	analysis of qualitative variables	*szz*	not elsewhere classified
srd	robust methods & statistical diagnostics		

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

International Stata Distributors

International Stata users may also order subscriptions to the *Stata Technical Bulletin* from our International Stata Distributors.

Company:	Applied Statistics & Systems Consultants
Address:	P.O. Box 1169 Nazerath-Ellit 17100, Israel
Phone:	+972 66554254
Fax:	+972 66554254
Email:	sasconsl@actcom.co.il
Countries served:	Israel

Company:	Dittrich & Partner Consulting
Address:	Prinzenstrasse 2 D-42697 Solingen Germany
Phone:	+49 212-3390 99
Fax:	+49 212-3390 90
Email:	evhall@dpc.de
Countries served:	Austria, Germany, Italy

Company:	Metrika Consulting
Address:	Roslagsgatan 15 113 55 Stockholm Sweden
Phone:	+46-708-163128
Fax:	+46-8-6122383
Email:	hedstrom@metrika.se
Countries served:	Baltic States, Denmark, Finland, Iceland, Norway, Sweden

Company:	Ritme Informatique
Address:	34 boulevard Haussmann 75009 Paris France
Phone:	+33 1 42 46 00 42
Fax:	+33 1 42 46 00 33
Email:	info@ritme.com
Countries served:	Belgium, France, Luxembourg, Switzerland

Company:	Smit Consult
Address:	Doormanstraat 19 Postbox 220 5150 AE Drunen Netherlands
Phone:	+31 416-378 125
Fax:	+31 416-378 385
Email:	j.a.c.m.smit@smitcon.nl
Countries served:	Netherlands

Company:	Survey Design & Analysis Services
Address:	249 Eramosa Road West Moorooduc VIC 3933 Australia
Phone:	+61 3 59788329
Fax:	+61 3 59788623
Email:	rosier@survey-design.com.au
Countries served:	Australia

Company:	Timberlake Consultants
Address:	47 Hartfield Crescent West Wickham Kent BR4 9DW U.K.
Phone:	+44 181 462 0495
Fax:	+44 181 462 0493
Email:	timberlake@compuserve.com
Countries served:	Ireland, U.K.

Company:	Timberlake Consultants Satellite Office
Address:	Praceta do Comércio, N°13–9° Dto. Quinta Grande 2720 Alfragide Portugal
Phone:	+351 (01) 4719337
Telemóvel:	0931 62 7255
Email:	timberlake.co@mail.telepac.pt
Countries served:	Portugal