STATA
TECHNICAL
BULLETIN

November 1992
STB-10

A publication to promote communication among Stata users

Editor

Joseph Hilbe
Stata Technical Bulletin
10952 North 128th Place
Scottsdale, Arizona 85259-4464
602-860-1446 FAX
stb@stata.com EMAIL

Associate Editors

J. Theodore Anagnoson, Cal. State Univ., LA
Richard DeLeon, San Francisco State Univ.
Paul Geiger, USC School of Medicine
Lawrence C. Hamilton, Univ. of New Hampshire
Stewart West, Baylor College of Medicine

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

an1.1 STB categories and insert codes

Inserts in the STB are presently categorized as follows:

General Categories:

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

Statistical Categories:

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

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

an24 Stata for the Macintosh ships

G. Edward Anderson, CRC, 800-STATAPC

Stata 3.0 is now available for the Macintosh. As with the DOS version of Stata, it comes in two flavors: regular and Intercooled. Regular Stata will run on any Macintosh except the original 128K Mac and MacPlus, under either System 6 or System 7, and requires 1 megabyte of free memory. Intercooled Stata will also run on any Macintosh under System 6 or 7, but requires 2 megabytes of free memory and a floating-point coprocessor.

Pricing is the same as for the respective DOS versions of Stata.

In timings performed on an SE/30, Intercooled Stata ran 4 to 12 times faster than the regular version. The smaller difference applies to data management tasks (such as sorting, for which the factor was 6.7) and the larger difference applies to heavy-duty statistical calculations (such as a probit model, for which the factor was 12.07).

More information is available by calling our 800 telephone number. Also see *os7* in this issue of the STB for details on how Stata and the Macintosh windowed environment were integrated.

an25 Complementary STB subscription for school libraries

Patricia Branton, CRC, 800-782-8272, FAX 310-393-7551

CRC will donate a subscription to the *Stata Technical Bulletin* to your school library. We will also donate past issues of the STB, subject to availability. All you need to do is have your library request it. We do require that the request come from the library so that we can verify the mailing address.

The library may phone, mail, or fax the request. Please have the library clearly indicate the mailing address for the subscription, and include a contact person and phone number, so we may call if we have questions.

an26 Request for data management problems

J. Theodore Anagnoson, Political Science, California State University, Los Angeles, 213-343-2230

At the Stata Workshop held at California State University, Fullerton, in the Summer of 1992, we found that there was considerable interest in solving data management problems with Stata. We should like to learn about your most interesting, most boring, easiest, hardest, etc., data management problems. This could be everything from a data set you could (couldn't) easily read into Stata, to memory management problems you could or couldn't solve, data manipulation and transformation problems (you needed to transpose this data set but when you used the transpose command, it didn't work . . .), or whatever. We will then compile a list of the ten most difficult problems in data management, publish them in the STB, credit you as the source if you wish, and use them in future workshops.

Send submissions to Ted Anagnoson, Department of Political Science, California State University, Los Angeles, CA 90032-8226. Call 213-343-2230 if there are questions. Thank you in advance for your help.

crc20 Stata expression-processing bug

When an expression has too many operators for Stata to process, it should issue the error message "`expression too long`" with a nonzero return code. On all except DOS computers, it does this. On DOS, however, it says not a word and (worse) ignores the part of the expression that was too long for it. (Actually, it engages in this awful behavior only if hacking off the expression leaves an expression that is potentially interpretable.)

Do not panic; unless you type exceedingly long expressions, this will not happen. At a minimum, an expression must contain in excess of 60 operators and would occupy four full lines. This bug will be fixed in the next release of Stata; in the meantime, we warn against typing expressions in excess of three lines.

crc21 True Kaplan–Meier estimates for ltable

The `ltable` command (see [5s] ltable) claims to produce Kaplan–Meier estimates of the survival function. If you study the formulas at the end, however, you will discover that it produces Kaplan–Meier style estimates with an actuarial adjustment. In the strict Kaplan–Meier approach, deaths and censorings are assumed to occur at the end of the period. In the actuarial adjustment, deaths and censorings are assumed to occur at the midpoint of the period. This is, it should be noted, a fine point.

In Stata, `ltable` was implemented using the actuarial adjustment while the related `kapmeier` command was implemented using the strict Kaplan–Meier approach. The new `noadjust` option to `ltable` makes calculations following the strict Kaplan–Meier approach and thus that are equivalent to those produced by the `kapmeier` command.

crc22 Error fixed in rchart

The formula printed in [5s] qc, vol. 3, p. 127 is incorrect. It says that when σ is unknown,

$$\text{central line} = d_2\bar{R}$$

when it should say:

$$\text{central line} = \bar{R}$$

The `rchart` command itself was implemented with this error, as shown most clearly by the first graph on page 123. It is fixed.

crc23 Minor bug fixes

`boxcox` did not converge in some circumstances. More sophisticated logic for interpolating between previous iterations and even introduction of random starting values when difficulty is encountered has been added. It is all automatic; there are no changes to the options or command syntax.

`qnorm` did not correctly process the `in` *range* qualifier.

`ranksum` treated a grouping variable containing missing values as though the missing values identified a third group and so refused to run the test.

`ttest`, when performing a paired test on, say, 30 and 31 observations, would report the mean for the 30 and 31 observations although it would then (correctly) continue to report the difference and statistical test based on the 30 pairable observations.

ip3 Stata programming

William Gould, CRC, FAX 310-393-7551

It has long been a goal of ours to make Stata programming accessible to a broad range of users, but despite this goal, we have done what could at best be termed a poor job of explaining how to write Stata programs. Everything you need to know is in the Stata manual, but it is scattered and nowhere does there appear a single, coherent discussion from start to finish. We want to correct this problem. To begin, I and others at CRC will start by writing a series of inserts in the STB explaining Stata programming. In my first attempt, I will outline the broad goals of Stata programming, showing how you can get started and, in the process, I hope, demonstrate that everyone can benefit from a knowledge of some amount of Stata programming, even those who do not consider themselves programmers.

In order to program in Stata, you will need two things: Stata and a word processor or editor capable of producing ASCII (DOS or text) files. Mechanically, Stata provides three ways of "programming" (do-files, interactive use of `program define`, and ado-files), and you will have to choose among them. Each has a purpose:

1. Do-files. These are stata commands to be executed sequentially; no looping is allowed and many people would not call this programming at all. Think of a do-file as a non-interactive session. Typical use: perform a particular analysis.
2. Interactively user-defined programs. These are Stata commands to be executed when the program name is typed. Looping is allowed. These programs are either typed directly into the keyboard or loaded via do-files. Typical use: reduce typing.
3. Ado-files. These are Stata's formal programs, commands to be executed when the ado-file name is typed. Looping is allowed and the program need not be previously loaded, so no thought goes into using them. Commands implemented this way appear to be a part of Stata. Typical use: general tools.

Examples

Problem 1a: Read the variables from base.raw (a data set you supposedly know all about) and prepare the data for analysis.

Solution: You could do this interactively, but you want to mechanize the process because (1) the data might be updated later and (2) you want to have proof that you prepared the data correctly. In determining whether to mechanically approach this problem with do-files, user-defined programs, or ado-files, the key is that you could do this interactively and that the solution to the problem is not a general tool; it is not something you might want to do to any data set. The right solution is a do-file. The do-file simply contains the lines you might have typed interactively:

```
-------------------------------- File crbase.do ---------------------------------
log using crbase, replace
clear
input long id str20 name byte(age sex) ... using base.raw
label data "Employee data"
label var id "Employee id"
label var name "Employee name"
label var age "Age of employee"
label var sex "Sex of employee"
label values sex sex
label define sex 0 male 1 female
assert sex==0 | sex==1
compress
save base, replace
log close
-------------------------------- end of file ------------------------------------
```

You create `crbase.do` outside of Stata, using an editor or word processor. You execute `crbase.do` by entering Stata and typing '`do crbase`'.

> *Aside:* Note the use of the `assert` command. `assert` verifies the truth of a claim and issues an error message if a claim is not true. Error messages, in turn, stop programs, do-files, and ado-files. In `crbase.do`, we claim that the variable `sex` contains the numeric values 0 and 1 and only 0 and 1; not only are there no third sexes, there is no one with unknown sex, either. We make this assertion *before* saving the data. If the assertion is false, the data will not be saved. This way, even if we do not carefully examine the output, we cannot possibly continue to analyze mistaken data. We include `asserts` not only to guard against the possibility of mistaken data; we ourselves might be mistaken in thinking that it is the third variable in the data or that it is coded 0 and 1 and not, say 1 and 2. Other things, too, could be asserted about this data and should be. We include only a single `assert` as an example of the concept, not an illustration of complete usage.

Problem 1b: Same as problem 1a, except there are now 15 different data sets, one for each plant, called plant1.raw, plant2.raw, and so on.

Solution: We modify `crbase.do` to read the files for a single plant, the plant being determined by an argument we supply:

```
-------------------------------- File crplant.do --------------------------------
log using cr`1´, replace
clear
input long id str20 name byte(age sex) ... using `1´.raw
label data "Employee data"
label var id "Employee id"
label var name "Employee name"
label var age "Age of employee"
label var sex "Sex of employee"
label values sex sex
label define sex 0 male 1 female
assert sex==0 | sex==1
compress
save `1´, replace
log close
---------------------------------- end of file ----------------------------------
```

`crplant.do` is the same as `crbase.do` except that we have replaced the word `base` with `` `1´ ``. `` `1´ `` to Stata means the first thing typed after `do` *filename*, so we can now read and prepare plant1.raw by typing '`do crplant plant1`'; plant2.raw by typing '`do crplant plant2`'; and so on. We might also create the do-file:

```
-------------------------------- File crall.do ---------------------------------
do crplant plant1
do crplant plant2
do crplant plant3
.
.
do crplant plant15
---------------------------------- end of file ----------------------------------
```

Typing '`do crall`' will prepare all 15 data sets.

Problem 1c: Same as problem 1b, except there are now 100 rather than 15 plants.

Solution: You could extend `crall.do`, but here is a better solution:

```
-------------------------------- File crall.do ---------------------------------
capture program drop doit
program define doit
        local i=1
        while `i´<=100 {
                do crplant plant`i´
                local i=`i´+1
        }
end
doit
---------------------------------- end of file ----------------------------------
```

Here we defined a program called `doit`; it is nothing fancy, but it does loop and so has to be a program. Now we can type '`do crall`' and read plant1.raw, plant2.raw, . . ., and plant100.raw. Note that the last line of our do-file is "`doit`". If we did not have that line, the program `doit` would be defined, but it would not be executed.

If you look at this program carefully, your first question is probably "What does `` `i´ `` mean?" Remember in `crplant.do`, above, we learned that `` `1´ `` means the first thing typed by the user after `do` *filename*. It literally means "the contents of `1`," where `1` is understood to mean a place where the first thing the user typed is stored. Similarly, `` `i´ `` means "the contents of `i`." Before using `` `i´ ``, we said '`local i=1`'. In the next statement, we say '`while `i´<=100`', meaning while the contents of `i` is less than or equal to 100, or while 1 is less than or equal to 100. Later in our program, we say '`local i=`i´+1`', meaning that `i` is to be redefined as the contents of `i` plus 1, or 2. Now going back to our `while` statement, it reads '`while 2<=100`' because the contents of `i` is 2. And so on. In the midst of our loop, we say '`do crplant`i´`'. If the contents of `i` is 2, then after substitution, this line reads '`do crplant2`'. The next time through the loop, the contents of `i` is 3, and so after substitution, the line reads '`do crplant3`'.

Next, notice the odd command '`capture program drop doit`' at the start of our do-file, just before we define the program `doit`. `doit` is an excellent meaningless name, a name that can be used over and over again. We want to be able to run this do-file without worrying that a program named `doit` is already defined. Imagine running this do-file twice in a row without the '`capture program drop doit`' line. The first time, it would work. The second time, we would be told that the program `doit` is already defined and everything would stop right there. Now imagine our file said '`program drop doit`' at the top—without the `capture`. The second time, our do-file would work; `doit` would be dropped and redefined. The first time, however, it would

not work because we would be told that the program `doit` does not exist! Putting `capture` in front of any Stata command eats the error message, if any. If the command works, fine, and if it does not, that is fine, too. Thus, our do-file works the first time and the second time.

Problem 1d: The data is now processed. Combine the 100 small data sets into a single data set.

Solution: One solution is to interactively type

```
. use plant1, clear
. append using plant2
. append using plant3
...
. append using plant100
. save combined, replace
```

That is a lot of typing and, since we will probably want to create a new variable indicating the plant to which an observation belongs, we really need to type

```
. use plant1, clear
. gen plant=1
. append using plant2
. replace plant=2 if plant==.
. append using plant3
. replace plant=3 if plant==.
. ...
. append using plant100
. replace plant=100 if plant==.
. save combined, replace
```

There is a better way, but all better ways start by understanding how we could do it by hand. Never begin writing a program until you understand the mechanical solution to the problem. A program is nothing more than a shorthand for the more lengthy solution.

```
-------------------------------- File combine.do --------------------------------
capture program drop doit               /* throw away the old doit, if any */
program define doit
        use plant1, clear
        gen byte plant=1
        local i=1
        while `i´<=100 {
                append using plant`i´
                replace plant=`i´ if plant==.
                local i=`i´+1
        }
        save combined, replace
end
doit
-------------------------------- end of file --------------------------------
```

Note that our program is nothing more than a line-by-line translation of the more lengthy, mechanical solution.

Problem 2: You have some data and among the variables are `thisvar`, `thatvar`, and so on. Tabulate each against the variable `plant`.

Solution: We could type

```
. tab plant thisvar
. tab plant thatvar
```

and continue this way for however many variables we have. If we have a lot of variables, however, it would be easier to type

```
. program define x
  1.        tab plant `1´
  2. end
. x thisvar
. x thatvar
```

If we had, say, eight variables, this would be a perfectly good solution.

If we had, say, twenty variables, it would be easier if we could type the names of all the variables after `x`, e.g., type '`x thisvar thatvar` ... `whatvar`' rather than '`x thisvar`', '`x thatvar`', ..., '`x whatvar`'. An improved implementation of our program is

```
. program define x
  1.        while "`1´"!="" {
  2.                tab plant `1´
  3.                mac shift
  4.        }
  5.  end
```

This program is worth some study. We have learned that `` `1´ `` means the first thing typed by the user after the program name. You might suspect that the second thing is `` `2´ ``, the third thing `` `3´ ``, and so on. You are right. The `mac shift` command (an abbreviation for `macro shift`) shifts the contents of the `` `1´ ``, `` `2´ ``, ..., macros. That is, whatever was in `` `1´ `` is thrown away, what was in `` `2´ `` is placed in `` `1´ ``, what was in `` `3´ `` is placed in `` `2´ ``, and so on. Thus, we tabulate the first variable, shift the variables, tabulate the new first variable, and continue like this until there are no more variables.

Now, why do we type '`while "`1´"!=""`' and not simply '`while `1´!=""`'? Imagine the first variable typed by the user is `thisvar`. Substitute for the meaning of `` `1´ ``. Without the double quotes, we have '`while thisvar!=""`'. With the double quotes, we have '`while "thisvar"!=""`'. These statements mean two very different things and it is the second interpretation we want.

To see this, trace the loop. The user types, say, '`x thisvar`' and the first time through, we tabulate `thisvar`. Making the substitution into the `while` statement, it says '`while "thisvar"!=""`'. The question being asked is whether the sequence of characters `t-h-i-s-v-a-r` is the same as the empty sequence of characters. It is not, so the loop executes. We then shift the macros. There is no second thing typed, so `` `2´ `` contains nothing and that nothing is copied into `` `1´ ``; `` `1´ `` now means nothing at all. Our while statement reads '`while ""!=""`', which is not true, so the loop ends. Without the double quotes, the first time through, the statement would have read '`while thisvar!=""`', which is a question about the contents of the variable `thisvar`. At this point, we do not care about the contents of the variable `` `1´ ``, we just want to know whether `` `1´ `` exists at all. The easiest way to do this is to treat `` `1´ `` as a string, i.e., to phrase our question in terms of `` "`1´" ``, which might be `""` or might be, say, `"thisvar"`.

In any case, our four-line program is probably too much to type interactively, so we might define the do-file `x.do`:

```
-------------------------------- File x.do -------------------------------------
capture program drop x
program define x
        while "`1´"!="" {
                tab plant `1´
                mac shift
        }
end
-------------------------------- end of file -----------------------------------
```

Now we can type

```
. do x
. x thisvar thatvar ... whatvar
```

We cannot, however, type '`x thisvar-whatvar`' or '`x t* w*`'. A better version of `x` would be:

```
-------------------------------- File x.do -------------------------------------
capture program drop x
program define x
        local varlist "req ex"
        parse "`*´"
        parse "`varlist´", parse(" ")
        while "`1´"!="" {
                tab plant `1´
                mac shift
        }
end
-------------------------------- end of file -----------------------------------
```

Stata's `parse` command is viewed by new programmers as terribly difficult, but they are wrong. Before trying to understand the example above, note that it took only three extra lines to make our command understand a varlist and that, thereafter, it required no change in the logic of our program.

Aside: You do not have to read this aside; it will explain how those three lines succeeded in making our program understand a varlist, all of which has to do with the `parse` command. `parse` is very useful for persons who wish to write commands that obey standard Stata syntax, but for casual programming, you never have to use, or understand, `parse`.

The first new line defined what our program was willing to tolerate of standard Stata syntax. We declare the varlist is required and is to be composed solely of existing (as opposed to new) variables.

That declared, the second line told Stata to go ahead and parse what the user typed. Stata examined this and determined whether it fit our specifications. If the user typed, say, '`x wrongvar`' and `wrongvar` does not exist, Stata issues an error message and stops our program. If the user typed, say, '`x this-that if region==3`', Stata issues an error message that the `if` is not allowed. But if the user typed '`x this-what`', Stata accepted it. (Stata would also accept '`x thisvar`' or '`x thisvar thatvar`', or anything else that is a valid varlist following our command). Stata also expanded the varlist, unabbreviating and filling in for the dash (if there is a dash), and put back the expanded varlist in `varlist`. Thus, `varlist` might now contain "`thisvar myvar othvar whatvar`".

The third line—'`parse "`varlist´", parse(" ")`'—is called a low-level parse. It told Stata to take what was now in varlist—say "`thisvar myvar othvar whatvar`"—and to reset `` `1´ ``, `` `2´ ``, `` `3´ ``, etc., to reflect the words in this list. Thus, at this point, things are just as if the user had tediously typed out the names of all the variables he or she specified with Stata's shorthand like `thisvar-whatvar` or `this*`. Of course, the user might not have used Stata's shorthand at all and really have typed everything out and then all of this work was unnecessary, but it did not hurt.

You can learn more about `parse` by reading [5u] parse.

With our new, improved `x`, we can type

```
. do x
. x thisvar-whatvar
```

Our program, however, is still not a tool. It is specific to the purpose of tabulating a bunch of variables by `plant`. Let's make it into a tool. Step 1 is to improve the program by allowing the user to specify the variable to be tabulated against (`plant` in our previous examples). We will also give our program a better name:

```
-------------------------------- File mytab.do ---------------------------------
capture program drop mytab
program define mytab
        local varlist "req ex min(2)"
        parse "`*´"
        parse "`varlist´", parse(" ")
        local lhs "`1´"
        mac shift
        while "`1´"!="" {
                tab `lhs´ `1´
                mac shift
        }
end
-------------------------------- end of file -----------------------------------
```

Now we can type

```
. do mytab
. mytab plant thisvar-whatvar
```

Our program is general enough to be promoted to an ado-file in our personal ado directory (`c:\ado` under DOS and `~/ado` under Unix).

```
. !copy mytab.do C:\ADO\mytab.ado
. program drop _all               /* no programs loaded now          */
. mytab plant myvar-thisvar       /* it works, loading automatically! */
```

We could further improve `mytab.ado` by allowing `if` *exp*, `in` *range*, and any of the options allowed by `tabulate`:

```
------------------------------- File mytab.ado --------------------------------
                                                /* see note #1        */
program define mytab
        version 3.0                             /* new, see #2        */
        local varlist "req ex min(2)"
        local if "opt"                          /* new, see #3        */
        local in "opt"                          /* new, see #3        */
        local options "*"                       /* new, see #4        */
        parse "`*'"
        parse "`varlist'", parse(" ")
        local lhs "`1'"
        mac shift
        while "`1'"!="" {
                tab `lhs' `1' `if' `in', `options'          /* see #5 */
                mac shift
        }
end
------------------------------- end of file -----------------------------------
```

Notes:

1. We removed the `capture program drop mytab`; it is unnecessary (but wouldn't hurt). `mytab.ado` will only be loaded when program `mytab` does not exist.
2. Our program is general and we intend to keep it around. To make sure it works in the future, we state the version of Stata under which we wrote it. Future versions of Stata will automatically "backdate" themselves, if necessary, to keep our program working.
3. We now allow `if` *exp* and `in` *range*.
4. We now allow any options.
5. We pass the `if` *exp*, `in` *range*, and any options along to `tabulate`.

We have written our first ado-file, a genuine tool for future use. We should also write `mytab.hlp` (stored in the same directory as `mytab.ado`); then we will be able to type '`help mytab`'.

Programming in Stata

Real Stata users write programs—lots of them. Most of the programs, however, are only a few lines long. My favorite program name is `x`:

```
. program drop x
. program define x
  1.  tabulate plant `1'
  2.  end
. x thisvar
. x thatvar
```

I am forever redefining the program `x` as I work interactively.

The genesis of the program `mytab` (outlined above) is typical. It is seldom that I sit down to write some general tool; rather, I run into a problem and need a specific solution. On occasion, as I refine my program, it becomes more and more general and, the next thing I know, I have a general tool, which I document and save for the next time I face the problem.

Aside on macros

What distinguishes a specific solution from a general tool? It is that the identity of the variables, the data, and so on, are "macro'd out"; rather than coding particular names, one codes symbols that stand in for the particular names.

For instance, I write a program that says, in effect, "tabulate `plant` by BOX", where BOX stands for something I will fill in when I tell you to run the program. Then, when I run the program, I somehow manage to communicate that BOX is "`thisvar`".

The symbols that stand in for other things (like BOX) are called macros. For instance, in

```
program define x
        tabulate plant `1'
end
```

`` `1' `` is a macro. Actually, `1` in this case is called the macro name and `` `1' `` refers to the contents of the macro named `1` and, to be even more precise about it, `` `1' `` is the contents of the local macro named `1`. The (local) macros named `1`, `2`, `3`, etc., are automatically defined by Stata when a program is invoked. Their contents are the first thing typed by the user, the second thing,

and so on. (They are "local" in the sense that their definition is known only to that program. If the program should call another program, the contents of 1, 2, ... are redefined as the arguments for that program and when that program concludes, the contents of 1, 2, ... are restored to their original contents. This technical detail plays no role, for now.)

With just this much knowledge, we can create useful programs. For instance, Stata has no copy command, but DOS does. In DOS, I can type '`copy` *filename1 filename2*' to make a copy of *filename1* called *filename2*. Stata does have a `shell` (synonym `!`) command, however, that allows it to pass commands through to DOS (see [5u] shell). Thus, I could type

```
. !copy filename1 filename2
```

Say I want to create a Stata `copy` command so I do not have to remember to type the explanation point:

```
-------------------------------- File copy.ado ---------------------------------
program define copy
        !copy `1´ `2´
end
-------------------------------- end of file -----------------------------------
```

So that I do not have to define this program in the future, I would store `copy.ado` in an appropriate directory (such as `c:\ado`). A better version of this program would read

```
-------------------------------- File copy.ado ---------------------------------
program define copy
        !copy `1´ `2´ `3´ `4´ `5´ `6´ `7´ `8´
end
-------------------------------- end of file -----------------------------------
```

and the best version would read

```
-------------------------------- File copy.ado ---------------------------------
program define copy
        !copy `*´
end
-------------------------------- end of file -----------------------------------
```

In the better version, we allow more than two arguments; most times `copy` takes two arguments, but there are exceptions. Including undefined extra arguments does not hurt. If the user types only two arguments, `` `3´ ``, `` `4´ ``, and so on contain nothing and will add nothing to the line. In the best version, we make use of the Stata synonym `` `*´ `` for `` `1´ `2´ `3´ ... ``. (By the way, the number of arguments does not end at 8, or 9, or any other arbitrary number. There are as many macros as their are arguments specified by the user. If that requires 1,000, then the macros `` `1´ `` through `` `1000´ `` are defined and `` `*´ `` stands for `` `1´ `2´ ... `1000´ ``).

Another ado-file you should write is one to invoke your favorite editor or word processor. Let's pretend your editor is called `vi`, and its syntax is "`vi` *filename*" typed at the DOS (or Unix) prompt:

```
-------------------------------- File vi.ado -----------------------------------
program define vi
        !vi `*´
end
-------------------------------- end of file -----------------------------------
```

Let us now move from one-line programs to multi-line programs, but before I do, I want to emphasize a rule:

The test of a program is how useful it is,
not how complicated or elegant it is internally.

Most of the programs I write are simple. Some I use so often that I have canned them (like `vi.ado`). Others are so specific to a particular task that I define them only when I need them and let them die when their usefulness expires.

So, let's now explore some multi-line, but still simple, programs.

Problem 3: I have four data sets to merge (match variable: `pid`). Every observation should merge. Match them.

Solution: I could type

```
. use ds1, clear
. merge pid using ds2
. tabulate _merge
. drop _merge
. sort pid
. merge pid using ds3
. (etc.)
```

I would type

```
. program drop x
. program define x
  1.        sort pid
  2.        merge pid using `1´
  3.        assert _merge==3
  4.        drop _merge
  5.        end
. use ds1, clear
. x ds2
. x ds3
. x ds4
```

Notice my use of `assert` instead of `tabulate`.

Problem 4: I want to play with the specification of a regression of `y` on `x1`, `x2`, `x3`, `x4`, `x8`, `x9` (which I know belong in the model) and `z1`, `z2`, `z3`, `z4` (of which I am not so sure). (The names are really more complicated than that.)

Solution:

```
. program drop x
. program define x
  1.        fit y x1 x2 x3 x4 x8 x9 `*´
  2.        test `*´
  3.        end
. x z1 z2 z3 z4
. x z1 z2
```

Programming endeavors

Let us now consider multi-line, complicated programs. Such programs arise in two circumstances:

1. Specific tasks—the problem is not general, but it is complicated;
2. Creation of tools.

In case 1, the program may get longer and more complicated, but the methods we use are unchanged. In short, we are willing to be sloppy in the sense of tolerating inelegant command syntaxes. We are willing to be restrictive in that we will write code that handles our problem, but does not generalize to handle all such problems correctly.

Problem 5: You have two data sets, each containing 30-character strings identifying the name of a company. You want to merge the two data sets on company. The company names were typed by different people and so may not exactly match. For purposes of this example, you may assume that:

1. co1.dta has two variables, `company` and `x`;
2. co2.dta has two variables, `company` and `y`.

You may assume that both co1.dta and co2.dta are sorted by `company`.

Solution: The obvious solution is simply to `merge` the two data sets. We have already been warned, however, that many observations will not merge because of slight spelling differences. Our overall plan will be:

1. Merge in the obvious way, remove whatever does merge—saying that they merged for reason 1—and look at whatever did not merge.
2. Attempt to develop a rule that would allow more of the unmerged observations to merge. Apply the rule to the remaining data, remove whatever now merges—saying that they merged for reason 2—and look at whatever did not merge.
3. Attempt to develop another rule . . .

Our solution will be contained in a do-file called `merco.do`—the problem is not general and therefore not deserving an ado-file. Nevertheless, when we type '`do merco`', the merge will be performed. Our do-file will grow as we make it more and more sophisticated, but the first draft is not sophisticated at all:

```
-------------------------------- File merco.do ---------------------------------
use co1, clear
merge company using co2
tabulate _merge
-------------------------------- end of file -----------------------------------
```

We type '`do merco`' and see the results. Some of the data merged, but some did not. We look at what did not merge and discover capitalization problems, e.g., "ABC Company" and "ABC company". A simple fix requires what seems an unreasonable amount of code:

```
------------------------- File merco.do, 2nd version -------------------------
use co1, clear
merge company using co2
tabulate _merge

save t_all, replace
keep if _merge==3
generate byte reason=1
drop _merge
save reason1, replace                   /* what merged so far           */

use t_all
keep if _merge!=3                       /* what has not merged yet      */
replace company=lower(company)          /* ignore capitalization        */

save t_all, replace
keep if _merge==2
keep company y                          /* from ds2                     */
sort company
save t_2, replace                      /* what has not merged from ds2  */
use t_all
keep if _merge==1                       /* from ds1                     */
keep company x                          /* what has not merged from ds1  */
sort company
merge company using t_2
(The do-file must now continue to save the merges in reason2.dta and set
things up to look at the nonmerged observations, just as we did before)
--------------------------------- end of file ---------------------------------
```

I stopped here because the do-file is getting too long and I am starting to repeat myself. A reorganization using some helper programs will improve this do-file:

```
------------------- File merco.do, 2nd version, reorganized --------------------
/* Programs I am going to use:        */

capture program drop splitout /* reason_# */
program define splitout
        capture erase reason`1´.dta
        save t_all, replace
        keep if _merge==3
        if _N>0 {
                drop _merge
                gen byte reason=`1´
                save reason`1´, replace
        }
        use t_all, clear
        drop if _merge==3
        erase t_all.dta
end

capture program drop tryagain
program define tryagain
        save t_all, replace
        keep if _merge==2
        keep company y                  /* from ds2                     */
        sort company
        save t_2, replace               /* what has not merged from ds2 */
        use t_all, clear
        keep if _merge==1
        keep company x                  /* from ds1                     */
        sort company
        merge company using t_2
        erase t_2.dta
        erase t_all.dta
end

/* execution begins here        */

use co1, clear
merge company using co2
tabulate _merge
splitout 1                              /* save merges as reason 1      */

replace company=lower(company)          /* ignore capitalization        */
tryagain
splitout 2                              /* save merges as reason 2      */
--------------------------------- end of file ---------------------------------
```

The program `splitout` is for use after a merge. '`splitout #`' saves the merged observations (`_merge==3`) in `reason#.dta` and leaves in memory all the unmerged observations (`_merge==1` or `_merge==2`).

The program `tryagain` takes unmerged observations left behind by `splitout` and attempts to merge them. (The idea is that `company` will somehow be "improved" (changed) so that what did not merge last time might now merge.)

The important point, however, is that I did not "design" this program. I merely started to type out sequentially what I needed to do. When I observed that I was beginning to repeat myself, I went back, copied lines into programs, and "macro'd out" the specifics of the step.

When we type '`do merco`', it produces reason1.dta and reason2.dta, exact matches and exact matches after ignoring capitalization. It leaves behind any observations that are still not merged. When we list the data set, we see spacing differences, e.g., "abc company" and "a b c company". Let's remove all spaces in the remaining names (to form "abccompany") and call that reason 3:

```
------------------------- File merco.do, 3rd version -------------------------
/* Programs I am going to use:        */
capture program drop splitout /* reason_# */
program define splitout
        (same as above)
end
capture program drop tryagain
program define tryagain
        (same as above)
end
capture program drop rmspace
program define rmspace
        tempvar l
        quietly {
                replace company=trim(company)
                gen `l´ = index(company," ")
                capture assert `l´==0
                while _rc {
                        replace company=substr(company,1,l-1) + /*
                        */ substr(company,l+1,.) if l!=0
                        replace `l´ = index(company," ")
                        capture assert `l´==0
                }
        }
end
/* execution begins here        */
use co1, clear
merge company using co2
tabulate _merge
splitout 1                              /* save merges as reason 1        */
replace company=lower(company)          /* ignore capitalization          */
tryagain
splitout 2                              /* save merges as reason 2        */
rmspace
tryagain
splitout 3                              /* save merges as reason 3        */
-------------------------------- end of file ---------------------------------
```

The new `rmspace` program goes through a string variable and removes all the blanks in it.

> (Comment: this might be a useful tool someday. How would you go about converting `rmspace` into a tool? What is wrong with it from the tool perspective? Answer: Most obviously, it has hardcoded the name of the variable, namely `company`. Secondly, it destroys the original contents of the variable; that might be okay in this case, but does not sound desirable in general. Maybe the syntax should be '`rmspace` *newvar*=*oldvar*'. We'll discuss writing tools, later, but it is surprising how often particular solutions suggest general tools.)

Now on version 3 of our program, we match exactly (reason 1), after ignoring capitalization (reason 2), and after ignoring spacing (reason 3). Typing '`do merco`' again, we find that there are sometimes commas in the remaining names and sometimes not, as in "abc,inc" and "abcinc" (remember, we have removed all the blanks). Since we do not care about elegance, we'll make a `rmcomma` program based on `rmspace`, substituting comma for blank.

> (Comment: If we were writing a general tool, we might think of generalizing our unwritten `rmspace` *newvar*=*oldvar* to allow an option to specify whether it is space, comma, or some other character we want removed. At that point, we'd probably want to change its name to a more general `rmchar`).

```
------------------------- File merco.do, 4th version -------------------------
/* Programs I am going to use:        */
capture program drop splitout /* reason_# */
program define splitout
        (same as above)
end
capture program drop tryagain
program define tryagain
        (same as above)
end
capture program drop rmspace
        (same as above)
end
capture program drop rmcomma
program define rmcomma                /* copied from rmspace        */
        tempvar l
        quietly {
                replace company=trim(company)
                gen `l´ = index(company,",")                  /* change here */
                capture assert `l´==0
                while _rc {
                        replace company=substr(company,1,l-1) + /*
                        */ substr(company,l+1,.) if l!=0
                        replace `l´ = index(company,",")      /* and here */
                        capture assert `l´==0
                }
        }
end
/* execution begins here        */
use co1, clear
merge company using co2
tabulate _merge
splitout 1                            /* save merges as reason 1        */
replace company=lower(company)        /* ignore capitalization          */
tryagain
splitout 2                            /* save merges as reason 2        */
rmspace
tryagain
splitout 3                            /* save merges as reason 3        */
rmcomma
tryagain
splitout 4                            /* save merges as reason 4        */
--------------------------------- end of file ---------------------------------
```

We type '`do merco`' and discover everything has merged (which was not true in the real application on which this example is based; it took about 8 rules, but by this time, you get the idea.) To complete our program, we need only combine the results:

```
------------------------- File merco.do, 5th version -------------------------
/* Programs I am going to use:        */
capture program drop splitout /* reason_# */
program define splitout
        (same as above)
end
capture program drop tryagain
program define tryagain
        (same as above)
end
capture program drop rmspace
        (same as above)
end
capture program drop rmcomma
program define rmcomma                /* copied from rmspace          */
        (same as above)
end
/* execution begins here        */
use co1, clear
merge company using co2
tabulate _merge
```

```
splitout 1                          /* save merges as reason 1        */
replace company=lower(company)      /* ignore capitalization          */
tryagain
splitout 2                          /* save merges as reason 2        */
rmspace
tryagain
splitout 3                          /* save merges as reason 3        */
rmcomma
tryagain
splitout 4                          /* save merges as reason 4        */
assert _N==0                        /* IMPORTANT!                     */
use reason1, clear
append using reason2
append using reason3
append using reason4
sort company
                                    /* We are done                     */
-------------------------------- end of file ---------------------------------
```

Note the line marked `IMPORTANT!` Why is it important? In this example, we completely merged the data with these four rules. If the data should change in the future, however, and we run this program again, it may or may not completely merge the data. Perhaps there will be a new twist, such as sometimes spelling out Incorporated and sometimes abbreviating it Inc. Our specific program will stop if it does not achieve its goal and we will know that we have problems instead of blithely continuing on our way.

Programming endeavors—creation of tools

The creation of tools is different from our previous programming endeavor in three ways:

1. (Beginning.) The syntax of a tool—the way the user specifies his or her desires—is important.
2. (Middle.) The tool must be general and work in a wide variety of situations.
3. (End.) The tool, if it calculates anything, must leave behind the result in a way that other programs can use it so that we can build new tools based on existing tools.

Tools can be divided into two categories:

1. Reporting tools. Tools that report a result (e.g., `summarize`). Such tools must never change the user's data.
2. Data manipulation tools. Tools that purposefully change the data. Such tools must either change the data in a way that is easily reversed or must be difficult to use unintentionally (`merge` is easily undone; `use` requires `clear` be spelled out; and `collapse` is just plain hard to use.)

If you determine the type of tool you are writing and follow the recommendations, you will produce tools that are safe to use. The most common error made by beginning tool makers (i.e., advanced Stata programmers) is not to determine the type of tool they are writing; to mix reporting and manipulation. The second most common error is to change the user's data when writing what is supposed to be a reporting tool.

> Stata Criticism: What does "change the data" mean? We can all agree that changing the contents of a variable qualifies as changing the data, but what about, say, changing the order of the observations? Stata says that is not a change, but many users disagree. It is most certainly a change if the user has no way to get the data back in the original order.

Programming endeavors—reporting tools

Problem: Write a tool to display the mean, median, and "trimmed" mean of a variable. The trimmed mean is defined as the mean of the variable after discarding observations below the 10th and above the 90th percentiles.

Solution: Determine how you would solve the problem with a hand calculator from Stata's output. We discover `summarize, detail` will supply the necessary information. Determine where `summarize` saves its results; looking in *Saved Results* in [5s] summarize, we find

mean = `_result(3)`

median = `_result(10)`

10th percentile = `_result(8)`

90th percentile = `_result(12)`

trimmed mean = `_result(3)` from `summ x if x>_result(8) & x<_result(12)`

Program development:

The first version of this program is designed only to achieve the result, little thought is given to command syntax or neat output.

```
------------------------------- File trimean.ado -------------------------------
*! version 1.1 -- 1st draft
program define trimean        /* varname */
        version 3.0
        quietly summarize `1´, detail
        display "        mean = " _result(3)
        display "      median = " _result(10)
        quietly summarize `1´ if `1´>_result(8) & `1´<_result(12)
        display "trimmed mean = " _result(3)
end
-------------------------------- end of file -----------------------------------
```

Testing our program, we discover that it does indeed work. We now set to making the output look neater, although I will cheat and leave filling in that part to you:

```
------------------------------- File trimean.ado -------------------------------
*! version 1.1 -- 1st draft
*! version 1.2 -- neat output
program define trimean        /* varname */
        version 3.0
        quietly summarize `1´, detail
        local mean=_result(3)
        local median=_result(10)
        quietly summarize `1´ if `1´>_result(8) & `1´<_result(12)
        display "..." (make neat output from `mean´, `median´, _result(3))
end
-------------------------------- end of file -----------------------------------
```

We now have a program that works and produces pretty output. Our program allows us to specify only a single variable, however, and most reporting programs in Stata allow one or more variables:

```
------------------------------- File trimean.ado -------------------------------
*! version 1.1 -- 1st draft
*! version 1.2 -- neat output
*! version 1.3 -- allow more than one variable name to be specified
program define trimean         /* varname [varname [varname [...]]]        */
        version 3.0
        while "`1´" != "" {
                quietly summarize `1´, detail
                local mean=_result(3)
                local median=_result(10)
                quietly summarize `1´ if `1´>_result(8) & `1´<_result(12)
                display "..." (make neat output from `mean´, `median´, _result(3))
                mac shift
        }
end
-------------------------------- end of file -----------------------------------
```

In all honesty, I would never have written version 1.3; my next draft would have been version 1.4, which allows a Stata varlist rather than merely a list of spelled-out variable names, but most programmers find Stata's `parse` command difficult to grasp, at least at first. Use of `parse`, however, requires adding only three lines to our program:

```
------------------------------- File trimean.ado -------------------------------
*! version 1.1 -- 1st draft
*! version 1.2 -- neat output
*! version 1.3 -- allow more than one variable name to be specified
*! version 1.4 -- allow varlist
program define trimean        /* varlist */
        version 3.0
        local varlist "req ex"          /* <-- new this version         */
        parse "`*´"                     /* <-- new this version         */
        parse "`varlist´", parse(" ")   /* <-- new this version         */
        while "`1´" != "" {
                quietly summarize `1´, detail
                local mean=_result(3)
                local median=_result(10)
                quietly summarize `1´ if `1´>_result(8) & `1´<_result(12)
                display "..." (make neat output from `mean´, `median´, _result(3))
                mac shift
        }
end
```

Most Stata commands also allow `if` *exp* and `in` *range*, and our `trimean` command will, too:

```
------------------------------- File trimean.ado ------------------------------
*! version 1.1 -- 1st draft
*! version 1.2 -- neat output
*! version 1.3 -- allow more than one variable name to be specified
*! version 1.4 -- allow varlist
*! version 1.5 -- allow if and in; tricky!
program define trimean        /* varlist [if exp] [in range]        */
        version 3.0
        local varlist "req ex"
        local if "opt"
        local in "opt"
        parse "`*'"
        parse "`varlist'", parse(" ")
        tempvar touse
        quietly gen `touse'=1 `if' `in'
        quietly replace `touse'=0 if `touse'==.
        while "`1'" != "" {
                quietly summarize `1' if `touse', detail
                local mean=_result(3)
                local median=_result(10)
                quietly summarize `1' if `1'>_result(8) & `1'<_result(12) & `touse'
                display "..." (make neat output from `mean', `median', _result(3))
                mac shift
        }
end
-------------------------------- end of file --------------------------------
```

This change was tricky because at one point in our program we must `summarize `1' if `1'>_result(8) & `1'<_result(12)` and we must further restrict the `summarize` to include any user-supplied `if`. Stata commands do not allow multiple `if`s, so we solve the problem by introducing a temporary variable `touse` that marks the observations the user said we are to use.

In our final draft, we save the results in global `S_#` macros so that the answers produced by our program can be used by other programs:

```
------------------------------- File trimean.ado ------------------------------
*! version 1.1 -- 1st draft
*! version 1.2 -- neat output
*! version 1.3 -- allow more than one variable name to be specified
*! version 1.4 -- allow varlist
*! version 1.5 -- allow if and in; tricky!
*! version 1.6 -- save results so can be used as utility
program define trimean        /* varlist [if exp] [in range]        */
        version 3.0
        local varlist "req ex"
        local if "opt"
        local in "opt"
        parse "`*'"
        parse "`varlist'", parse(" ")
        tempvar touse
        quietly gen `touse'=1 `if' `in'
        quietly replace `touse'=0 if `touse'==.
        while "`1'" != "" {
                quietly summarize `1' if `touse', detail
                local mean=_result(3)
                local median=_result(10)
                quietly summarize `1' if `1'>_result(8) & `1'<_result(12) & `touse'
                display "..." (make neat output from `mean', `median', _result(3))
                mac def S_1 = _result(1)         /* # of obs      */
                mac def S_2 = `mean'             /* mean          */
                mac def S_3 = `median'           /* median        */
                mac def S_4 = _result(3)         /* trimmed mean  */
                mac shift
        }
end
-------------------------------- end of file --------------------------------
```

How would another program make use of our program? If it wanted the trimmed mean of some variable, it might include the lines:

```
quietly trimean `1´                        /* say */
local tmean=$S_4
```

Programming tricks

Most programming tricks in Stata have to do with tricky use of macros. There are two kinds of macros in Stata: local macros and global macros. Local macros are local to the program; other programs may have local macros of the same name, but it does not matter; they are different and private. Global macros, on the other hand, are shared across all programs.

Local macros are defined with the `local` command. Global macros are defined with the `macro define` command. The contents of local macros are obtained by quoting, as in "`local i=`i´+1`". The contents of global macros are obtained by preceding their names with a dollar sign, as in "`mac def i=$i+1`".

Every program has defined for it a special set of local macros, `` `1´ ``, `` `2´ ``, `` `3´ ``, . . ., which are the arguments typed by the user. In addition, the macro `` `*´ `` is defined as the arguments typed by the user with a single blank between.

Thus, `` `2´ `` is the second thing typed by the user (or nothing if no second thing was typed). Now consider defining '`local i=2`'. What does `` ``i´´ `` mean? Start inside: `` `i´ `` means 2, so `` ``i´´ `` means `` `2´ ``, or the second thing typed by the user.

Thus, in looping across varlists, you can either

```
parse "`varlist´", parse(" ")
while "`1´"!="" {
        (code using `1´)
        mac shift
}
```

or

```
parse "`varlist´", parse(" ")
local i=1
while "``i´´"!="" {
        (code using `1´)
        local i=`i´+1
}
```

In fact, the second form can be executed more quickly by Stata than the first.

The second trick in dealing with macros is to omit the equal sign whenever possible. '`local i 1`' is better than '`local i=1`'. The presence of the equal sign flags Stata that an expression which must be evaluated is coming up. Stata then goes through hoops, parsing and compiling the expression and then later, executing it. All of this takes time. With no equal sign, Stata knows it can just copy the definition into the macro. (In the programs above, I have always written '`local i=1`' because many users think '`local i 1`' is a mistake. Were I writing these programs for myself, however, they would have said '`local i 1`'.)

There are times, however, when the equal sign is required, as in "`local i=`i´+1`". We want the expression evaluated, not copied. If we typed '`local i `i´+1`', and `` `i´ `` contained, say, 1, the new contents of '`i`' would be "`1+1`", not "`2`".

os7 Stata and windowed operating systems

William Gould, CRC, FAX 310-393-7551

As Stata users know, Stata is command driven and it is Stata's language that, as much as anything, defines Stata. Stata has, until now, been available under DOS and Unix, two operating systems for which a command-language interface is natural. Stata has not been available on the Macintosh, OS/2's Presentation Manager, and Windows (although in the case of Windows, this is only in terms of exploiting the interface, not in terms of being usable). These operating systems and environments are qualitatively different than the language-based environments of DOS and Unix and users of these systems expect something different from their software.

The question then becomes: can Stata be successfully ported to windowed operating environments? One solution, of course, would be to develop a new product unrelated to Stata, call it Stata anyway, and market it, but there would no justification for this except financial profit. (Such an approach would not only be intellectually and morally dishonest, it would serve only to dilute the name Stata and confuse users.)

Let me start with two premises:

1. Stata's command language is desirable. Language allows communication of more complicated ideas than pointing and clicking and data analysis requires such communication between man and machine.
2. Point-and-click interfaces are the wave of the future.

The question is, then, whether (1) and (2) can be successfully combined. Let me first justify—not prove—my premises with respect to data analysis. I take the complexity-of-thought point as obvious: Were I speaking verbally, imagine that I now stopped speaking and restricted myself to pointing at things around the room to communicate to you. I would obviously be limited in what I could communicate. The language humans use to communicate is, of course, inordinately rich compared to Stata's simple language, but still, Stata's grammar is composed of phrases that can be combined in new and surprising ways. By surprising, I mean that sentences (instructions) can be constructed in Stata's language that were not considered at the time we wrote Stata's internal code. In point-and-click interfaces, the user is limited to selecting among the alternatives the programmer conceived of at the time the program was written.

Command languages have other advantages, too. Most important among these advantages is that sequences of commands given by the user can be recorded, or logged, for later review and execution. In data analysis work, this is of great importance.

On the other hand, communication with computers, at least these days, is via a keyboard; we do not talk to the computer. Typing out, and remembering, variable names (say when estimating a regression model) is one place where command languages are burdensome; the point-and-click interface is better.

Our "solution" is to view the point-and-click interface as an assistant for constructing commands. In the windowed environment, the user still issues commands, but can, say, click from a list of variable names to copy the names into the command. Viewed this way, the entire point-and-click interface is demoted to the level of the backspace and other editing keys—useful but not inherent.

The more conventional view holds that the entire application is designed around the interface, summarized by the term "look and feel."

> The central advantage of graphics-based user interfaces such as PM (and Microsoft Windows and the Macintosh) is that you don't need to remember the commands used to operate the system and your applications. [. . .] The "look and feel" will be familiar, whether you're using a spreadsheet or a word processor. [Dror and Lafore 1990, 13–14]
>
> One of the greatest advantages of using a Macintosh computer is that most Macintosh programs [. . .] are designed to look similar and work in similar ways. [Apple Computer Inc. 1990, 43]

The conventional view tacitly assumes that the problem of man communicating with machine has been solved—we now have a consistent look and feel—and it is now only a matter of implementing the various tools we need with the certified look and feel. I, however, do not think the problem has been solved, either by any of the windowed operating systems or by Stata. In particular, I do not think the look and feel of a wordprocessor is the appropriate model for a statistical system or vice-versa. I do think that the look and feel of the windowed operating systems is a positive development and, speaking as a systems designer, I am interested in finding ways of exploiting that development. At a generic level, however, I worry that the concerns with look and feel could stifle development of alternatives.

Turning back to Stata, our demoted point-and-click interface is referred to as helper keys. To estimate a regression, you still type '`regress`' or '`fit`' to Stata's command prompt, but following that you can use a pull-down menu or, on the Macintosh, press Command-N, to pop up a window containing a list of variable names. You can click on the variable names to copy them into the command. A similar facility is available for data set names. It is our intention to make this a standard feature of Stata in all windowed environments.

We have taken standard advantage of the Macintosh on issues of interfacing to the operating system: Stata can be invoked by double-clicking on a data set and graphs and logs can be examined and printed by double-clicking on their file icon. From a system design point of view, this should be viewed as no more than allowing DOS users to specify command options using '`/`' while Unix users use '`-`'. There are no grand implications, but such touches are important to a user.

Returning to Stata itself, one of our goals is to establish a definition of Stata independent of operating system. Internally, we imagine that we could take a user from a particular platform, place him or her in front of a different platform, invoke Stata for them however is appropriate in that environment, and then walk away. Without any further assistance, they could use Stata just as they do on their own platform. This means that researchers from different operating environments can still work together and that any single researcher can decide to change operating environments without losing his or her investment in data sets, programs, and Stata-specific knowledge.

An important part of this across-platform compatibility is that Stata data sets, graphs, programs, and logs can be used on any other platform without translation of any kind. Data sets created on the Macintosh can be used under DOS or Unix, and vice-versa. Ado-files written under DOS and Unix can also be used on the Macintosh.

I would be interested in any comments users of the Macintosh, or any windowed operating system, have. The question of how to exploit windowed operating systems in Stata is not settled and it is a topic of hot debate at CRC. Bill Rogers of CRC, for instance, is far more positive on the value of point-and-click interfaces than am I and, thankfully for windowed-operating-system users, it was he who performed the port. What Bill and I do agree on, however, is that Stata's principles must not be sacrificed and that Stata should work the same way in all operating environments. This means that the features of Stata on the Macintosh will be ported to other environments and that new features will be ported to all environments as Stata continues to develop.

References

Apple Computer, Inc. 1990. *Getting Started with your Macintosh.* Cupertino, CA.

Dror, A. and R. Lafore. 1990. *OS/2 Presentation Manager and Programming Primer.* New York: Osborne McGraw–Hill.

sbe9	Brier score decomposition

William Rogers, CRC, FAX 310-393-7551

The syntax of `brier` is

`brier` *outcome forecast* [`if` *exp*] [`in` *range*] [`, group(#)`]

`brier` computes the Yates, Sanders, and Murphy decompositions of the Brier Mean Probability Score. `outcome` must be a variable containing 0-1 values and `forecast` contains the corresponding predicted probabilities as produced by, say, logit, probit, or a human forecaster. `group(#)` specifies the number of groups that will be used to compute the decomposition and defaults to 10.

Discussion

You have a binary (0-1) response and a formula that predicts this response from some covariates. If the binary response were calculated by logistic regression, there is a plethora of methods by which you can assess goodness of fit (see `lfit` in [5s] logistic). However, the formula might be computed from a published formula or another sample, completely unrelated to the data at hand. For example, if a weatherman predicts the "probability of rain" over a course of 365 days, and you measure the subsequent event "rain" or "no rain," you have such a situation.

The Brier score is an aggregate measure of disagreement between a prediction and a binary variable—the average squared error difference. The Brier score decomposition is a partition of the Brier score into components that suggest reasons for discrepancy. These reasons fall roughly into three groups: (1) Lack of overall calibration between the average predicted probability and the actual probability of the event in your data; (2) Misfit of the data in groups defined within your sample; and (3) Inability to match actual 0 and 1 responses.

Using logit or probit analysis to fit your data will guarantee that there is no lack of fit due to (1), and a good model fitter will be able to avoid problem (2). Problem (3) is inherent in any prediction exercise.

"Brier score" measures the total difference between an event and the forecast probability of that event.

"Sanders Brier score" measures the difference between a grouped forecast measure and the event. Grouping is done by sorting the sample on the forecast and dividing it into groups with a similar forecast. The difference between this and the Brier score is therefore minimal.

"Sanders decomposition" measures error that arises from statistical considerations in making a forecast. For example, the error that arises because a prediction of $p = .4$ does not predict the zeros and ones, is part of this term.

"Reliability-in-the-small" measures the error that comes from the average forecast within group not measuring the average outcome within group. This term measures lack of modeling quality.

"Murphy decomposition" measures the tendency of outcome differences in forecast groups to differ from the overall outcome. The better the "information" in the forecast, the larger this term will be.

"Outcome index variance" is just the variance due to the outcome variable.

"Forecast variance" measures the amount of forecast discrimination being attempted. In a logit model, this would tend to go up with the amount of information available.

"Reliability-in-the-large" is the ability of the mean forecast to match the mean probability. This should be zero for statistical forecasting methods applied to the training dataset, since they always get the overall probability right.

"Forecast-Outcome Covariance" is a measure of how accurately the forecast responds to the outcome. It is similar in concept to R-squared.

"Grouping forecast error" describes the amount of error in the resolution process due to use of groups.

Example

You have data on the outcomes of 20 basketball games (`win`) and the probability of victory predicted by a local pundit (`for`).

```
. brier win for, group(5)
Brier score                   0.1828
Sanders-modified Brier score  0.1887
Sanders resolution            0.1375
Outcome index variance        0.2275
Murphy resolution             0.0900
Reliability-in-the-small      0.0512
Forecast variance             0.0438
Excess forecast variance      0.0285
Minimum forecast variance     0.0153
Reliability-in-the-large      0.0294
2*Forecast-Outcome-Covar      0.1179
Grouping forecast error       0.0038
. gen group = int((_n-1)/4) + 1
. sort group
. qui by group: gen formean = sum(for)
. qui by group: replace formean = formean[_N]/4
. qui by group: gen winmean = sum(win)
. qui by group: replace winmean = winmean[_N]/4
. list
          for        win      group    formean    winmean
  1.      .15          0          1      .2175        .25
  2.       .2          1          1      .2175        .25
  3.      .25          0          1      .2175        .25
  4.      .27          0          1      .2175        .25
  5.      .27          0          2       .345         .5
  6.      .33          0          2       .345         .5
  7.      .38          1          2       .345         .5
  8.       .4          1          2       .345         .5
  9.      .45          1          3      .4625         .5
 10.      .45          0          3      .4625         .5
 11.      .45          0          3      .4625         .5
 12.       .5          1          3      .4625         .5
 13.       .5          1          4      .5625          1
 14.       .5          1          4      .5625          1
 15.       .6          1          4      .5625          1
 16.      .65          1          4      .5625          1
 17.      .67          1          5       .805          1
 18.      .75          1          5       .805          1
 19.       .9          1          5       .805          1
 20.       .9          1          5       .805          1
. summ for win
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+--------------------------------------------------------
     for |      20       .4785   .2147526        .15         .9
     win |      20         .65   .4893605          0          1
. ci win, binomial
                                                    -- Binomial Exact --
Variable |     Obs         Mean    Std. Err.      [95% Conf. Interval]
---------+--------------------------------------------------------------
     win |      20          .65    .1066536       .4075195     .8461914
```

The listing shows the original data as well as the means of `for` and `win` within groups used by `brier`.

If this dataset were larger, the most critical problem in it would be the difference between the average forecast (.4785) and the probability of victory (.65). This is measured by "Reliability-in-the-large." However, that difference was not statistically significant. In terms of contribution to the Brier score, the largest term is due to forecast outcome variance, the variance that would be observed if the forecast probabilities were correct. In other words, this pundit may be able to guess probabilities, but not the outcome of the game itself.

References

Brier, G. W. 1950. Verification of forecasts expressed in terms of probability. *Monthly Weather Review* 75: 1–3.

Hadorn, D., E. B. Keeler, W. H. Rogers, and R. Brook. 1993. *Assessing the Performance of Mortality Prediction Models.* N-3599-HCFA. Santa Monica, CA: The Rand Corporation.

Yates, J. F. 1982. External correspondence: Decompositions of the mean probability score. *Organizational Behavior and Human Performance* 30: 132–156.

sg9.1 Additional statistics to similari output

Joseph Hilbe, Editor, STB, FAX 602-860-4331

I have been asked by users to add the odds ratio and relative risk statistics to the `similari` output as discussed in STB-9. An updated program by the same name is included on the STB-10 diskette. The help file has also been changed to reflect the additional statistical output.

Reference

Hilbe, J. 1992. sg9: Similarity coefficients for 2 x 2 binary data. *Stata Technical Bulletin* 9: 14–15.

sg12 Extended tabulate utilities

Dean H. Judson, 1013 James St., Newberg, OR 97132, 503-537-0660

The syntax for the basic extended tabulate utility `etab` is

`etab` *depvar varlist* [*weight*] [`if` *exp*] [`in` *range*] [`,` *tabulate_options*]

`etab` tabulates the first variable in its variable list, here designated *depvar*, against all the other variables in *varlist*, applying the desired *tabulate_options* to each cross-tabulation.

An example:

`etab` is useful when you have a categorical result (or dependent) variable, against which you wish to compare a variety of categorical predictor (or independent) variables.

Suppose you have a set of categorical variables from a survey: `vote` (GOP or democrat), `race` (white or nonwhite), `sex` (male or female), `marstat` (marital status: single, married, divorced, separated, or widowed), and `housing` (urban or rural). You wish to compare all of the variables with `vote`. Using `etab`, one way to perform this is

```
etab vote race sex marstat housing, col chi2
```

This command would cross-tabulate `vote` by `race`, showing column percent and chi-square test of independence, then `vote` by `sex`, showing the same options, then `vote` by `marstat`, showing the same options, then `vote` by `housing`, showing the same options. You would use this command in a situation where you need to quickly assess the bivariate relationship between one dependent variable and a wide variety of potential independent variables.

All of the options available for two-way tabulations are available with this command. Also see [5s] tabulate and the on-line help for `tab` and `xtab`.

Extended cross-tabulations:

The syntax for the cross-tabulation command, `xtab`, is

`xtab` *varlist*$_1$ [*weight*] [`if` *exp*] [`in` *range*], `by(`*varlist*$_2$`)` [*tabulate_options*]

`xtab` cross-tabulates each of the variables in *varlist*$_1$ against each of the variables in *varlist*$_2$, applying the *tabulate_options* to each cross-tabulation.

An example:

Suppose you have 6 variables, $var1, var2, \ldots, var6$. You wish to see how the first three ($var1$—$var3$) are each related to each of the last three ($var4$—$var6$). Using `xtab`, you type

`xtab var1 var2 var3, by(var4 var5 var6)` *options*

This generates the following cross-tabulations:

var1 by var4	var1 by var5	var1 by var6
var2 by var4	var2 by var5	var2 by var6
var3 by var4	var3 by var5	var3 by var6

In each case, the *options* you specified would also be presented with the cross-tabulation.

Consider this command to be a multi-way extension of the `tab2` command, designed for different purposes. For example, if you had five variables in *varlist*$_1$ and two variables in *varlist*$_2$, this command would generate ten tables. `tab2` with all seven variables would generate 21 tables.

In general, if you have n_1 variables in the first varlist and n_2 variables in the second varlist, `xtab` will generate $n_1 \times n_2$ tables. `tab2` applied to the combination of each varlist will generate $(n_1 + n_2)(n_1 + N_2 - 1)/2$ tables, which will always be larger for $n_1 > 1$ or $n_2 > 1$.

Thus, use `tab2` if you really wish to see *all* of the two-way cross-tabulations among a set of variables. Use `xtab` if you wish to see a specific set of variables individually cross-tabulated with another specific set of variables. Also see [5s] tabulate and the on-line help for `tab` and `etab`.

sg13 Is a transformation of the dependent variable necessary?

Richard Goldstein, Qualitas, Brighton, MA, EMAIL goldst@harvarda.bitnet

Stata now provides at least two easy ways to help users attempt to determine whether a transformation of the left-hand-side variable in a regression is necessary: `linktest` and `boxcox` (see also Goldstein 1992). Here we add another procedure to this arsenal: Atkinson's score test. The syntax is

`atkinson` *depvar varlist* [`if` *exp*] [`in` *range*]

This temporarily adds a new variable to your data set called `lambda1`; note that the dependent variable should be the raw, untransformed, version. Of course, you could also use this on the transformed version to test its adequacy. Your model is then fit, adding the new variable. If the t-test for the new variable is statistically significant, then you should transform the left-hand-side variable. The t-test and its p-value are shown, but not the regression.

A rough guess of the appropriate transformation is (1 − coefficient of `lambda1`). This value, and a rounded version, are shown. Atkinson (1985) also suggests a graph, but I have not implemented this.

`atkinson` can give quite different results from `linktest`, as shown below. Two examples appear below: one from Weisberg's book and one using the Stata `auto.dta` data set. Note that even though `linktest` says the auto model is okay, as shown on page 5 of volume 3 of the Stata manual, both Atkinson's test and `boxcox` suggest there is still a problem. This provides some statistical support to the Stata manual which suggests a different model than the one on page 5 of volume 3.

In addition to agreeing with Weisberg for the example shown below, this has also been tested against other data sets in the literature and agreement was found for these also.

```
. use weisberg
(Weisberg, 1985, p. 149)
```

```
. fit area perim
  Source |       SS       df       MS              Number of obs =      25
---------+------------------------------           F(  1,    23) =  597.04
   Model |  5657.86954     1  5657.86954            Prob > F      =  0.0000
Residual |  217.960733    23   9.4765536           R-square      =  0.9629
---------+------------------------------           Adj R-square  =  0.9613
   Total |  5875.83027    24  244.826261            Root MSE      =  3.0784
------------------------------------------------------------------------------
    area |      Coef.   Std. Err.       t     P>|t|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
perimetr |   12.08766   .4946988     24.434   0.000     11.06429    13.11102
   _cons |  -6.745398   1.154252     -5.844   0.000     -9.13315   -4.357646
------------------------------------------------------------------------------


. linktest
  Source |       SS       df       MS              Number of obs =      25
---------+------------------------------           F(  2,    22) =  304.41
   Model |  5670.90653     2  2835.45327            Prob > F      =  0.0000
Residual |  204.923742    22  9.31471555           R-square      =  0.9651
---------+------------------------------           Adj R-square  =  0.9620
   Total |  5875.83027    24  244.826261            Root MSE      =   3.052
------------------------------------------------------------------------------
    area |      Coef.   Std. Err.       t     P>|t|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
    _hat |    .863869   .1220117      7.080   0.000     .6108323    1.116906
  _hatsq |   .0032477   .0027452      1.183   0.249    -.0024455    .0089408
   _cons |   .6434524   1.072575      0.600   0.555    -1.580932    2.867837
------------------------------------------------------------------------------


. atkinson area per
score test for whether should transform area: t =   4.868 p-value: 0.0000
if above significant, transform area using  0.684 (round to .5)


. boxcox area perimet
(note: iterations performed using zero =.001)
   Iteration      Lambda         Zero     Variance        LL
   ------------------------------------------------------------
         0        1.0000    -41.14734   9.01019727    -27.47946
         1       -0.5031     31.03189   51.1919266    -49.19477
         2        0.2484     31.28928   7.83862772    -25.73830 (*)
         3        2.6960    -36.71690   1769.14164    -93.47812
         4        1.4722    -40.54610   44.1196614    -47.33632 (*)
         5        6.1441    -39.04709   61685753.9   -224.21954
         0       -0.9356     33.10620   154.460181    -62.99920 (**)
         1        4.3632    -37.92518    254358.23   -155.58124
         2        1.7138    -38.75852   94.7651416    -56.89252 (*)
         3        8.2958    -39.80028   5.4995e+10   -309.13131
         0        0.3422     29.23279   6.24456081    -22.89638 (**)
         1        1.1534    -42.89068   15.1677673    -33.98966
         2        0.7478    -19.31421    4.5950881    -19.06235 (*)
         3        0.6187      2.03013   4.16145803    -17.82332
         4        0.6310     -0.00321   4.16068656    -17.82100
         5        0.6310     -0.00002   4.16068123    -17.82098
   ------------------------------------------------------------
    * (lambda obtained as midpoint of previous two iterations)
   ** (lambda obtained by randomly choosing a new starting value)
      (despite this, convergence was achieved)
Transform:  (area^L-1)/L
                L       [95% Conf. Interval]     Log Likelihood
              ------------------------------------------------
              0.6310      (not calculated)          -17.820985
     Test:  L == -1      chi2(1) =    95.31    Pr>chi2 =  0.0000
            L ==  0      chi2(1) =    32.21    Pr>chi2 =  0.0000
            L ==  1      chi2(1) =    18.49    Pr>chi2 =  0.0000
(type regress without arguments for regression estimates conditional on L)


. use auto, replace
(1978 Automobile Data)
```

```
. fit mpg we w2 displ fore
  Source |       SS       df       MS              Number of obs =      74
---------+------------------------------           F(  4,    69) =   39.37
   Model |  1699.02634     4  424.756584            Prob > F      =  0.0000
Residual |  744.433124    69  10.7888859            R-square      =  0.6953
---------+------------------------------           Adj R-square  =  0.6777
   Total |  2443.45946    73  33.4720474            Root MSE      =  3.2846
------------------------------------------------------------------------------
     mpg |      Coef.   Std. Err.       t     P>|t|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
  weight |  -.0173257   .0040488     -4.279   0.000    -.0254028   -.0092486
      w2 |   1.87e-06   6.89e-07      2.711   0.008     4.93e-07    3.24e-06
   displ |  -.0101625   .0106236     -0.957   0.342     -.031356     .011031
 foreign |  -2.560016   1.123506     -2.279   0.026    -4.801349   -.3186833
   _cons |   58.23575   6.449882      9.029   0.000     45.36859    71.10291
------------------------------------------------------------------------------

. linktest
  Source |       SS       df       MS              Number of obs =      74
---------+------------------------------           F(  2,    71) =   81.08
   Model |  1699.39489     2  849.697445            Prob > F      =  0.0000
Residual |   744.06457    71  10.4797827            R-square      =  0.6955
---------+------------------------------           Adj R-square  =  0.6869
   Total |  2443.45946    73  33.4720474            Root MSE      =  3.2372
------------------------------------------------------------------------------
     mpg |      Coef.   Std. Err.       t     P>|t|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
    _hat |   1.141987   .7612218      1.500   0.138    -.3758456     2.65982
  _hatsq |  -.0031916   .0170194     -0.188   0.852    -.0371272    .0307441
   _cons |   -1.50305   8.196444     -0.183   0.855    -17.84629    14.84019
------------------------------------------------------------------------------

. fit mpg we w2 displ fore lamb
  Source |       SS       df       MS              Number of obs =      74
---------+------------------------------           F(  5,    68) =   76.14
   Model |   2073.1512     5   414.63024            Prob > F      =  0.0000
Residual |   370.30826    68  5.44570971            R-square      =  0.8484
---------+------------------------------           Adj R-square  =  0.8373
   Total |  2443.45946    73  33.4720474            Root MSE      =  2.3336
------------------------------------------------------------------------------
     mpg |      Coef.   Std. Err.       t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
  weight |  -.0042027   .0032834     -1.280   0.205    -.0107547    .0023493
      w2 |  -2.06e-07   5.50e-07     -0.375   0.708    -1.30e-06    8.91e-07
   displ |  -.0068356   .0075583     -0.904   0.369     -.021918    .0082467
 foreign |  -2.778731   .7986408     -3.479   0.000    -4.372394   -1.185068
 lambda1 |   2.178137   .2627872      8.289   0.000     1.653754    2.702521
   _cons |   72.60252    4.89924     14.819   0.000     62.82624     82.3788
------------------------------------------------------------------------------

. atkinson mpg we w2 displ fore
score test for whether should transform mpg: t =   8.289 p-value: 0.0000
if above significant, transform mpg using -1.178 (round to -1)

. boxcox mpg we w2 displ fore
(note: iterations performed using zero =.001)
   Iteration      Lambda         Zero     Variance        LL
   ------------------------------------------------------------
         0        1.0000    -17.11879   10.1065591    -85.58783
         1       -0.9179      1.74992   6.75625007    -70.68732
         2       -0.7271     -0.03498   6.72785909    -70.53151
         3       -0.7308     -0.00005   6.72781566    -70.53127
   ------------------------------------------------------------

Transform:  (mpg^L-1)/L
                L       [95% Conf. Interval]     Log Likelihood
            -------------------------------------------------
              -0.7308      (not calculated)           -70.531269
     Test:  L == -1      chi2(1) =     0.70    Pr>chi2 =  0.4017
            L ==  0      chi2(1) =     5.21    Pr>chi2 =  0.0225
            L ==  1      chi2(1) =    29.77    Pr>chi2 =  0.0000
(type regress without arguments for regression estimates conditional on L)
```

References

Atkinson, A. C. (1985, reprinted with corrections, 1987). *Plots, Transformations and Regression*. Oxford: Oxford University Press.

Goldstein, R. 1992. srd9: Box–Cox statistics for help in choosing transformations. *Stata Technical Bulletin* 5: 22–25.

Weisberg, S. 1985. *Applied Linear Regression*. 2d ed. New York: John Wiley & Sons.

sg14	Is a transformation of an independent variable necessary?

Richard Goldstein, Qualitas, Brighton, MA, EMAIL goldst@harvarda.bitnet

It is often easier to interpret a regression model when a right-hand-side variable has been transformed than it is when a left-hand-side variable has been transformed. This ado-file uses an approximation to the Box–Tidwell (Box and Tidwell 1962) technique to help determine if a variable should be transformed and provides an estimate of what the transform should be. The syntax is

`tidwell` *depvar indepvar* [`if` *exp*] [`in` *range*]

This temporarily adds a new variable to your data set called `resp0`; note that the dependent variable should be the raw, untransformed, version. Of course, you could also use this on the transformed version to test its adequacy. Your model is then fit, adding the new variable. If the t-test for the new variable is statistically significant, then you should transform the right-hand-side variable. The t-test and its p-value are shown, but not the regression.

A crude guess of the transform is $\big($(coefficient of `resp0`)/(coefficient of variable from model without including `resp0`)$\big)+1$. This value is also provided, as is a version of this value rounded to the nearest .5. If the suggested transform is outside the range of what you consider reasonable (e.g., -2 to $+2$), then ignore it (absurd results are possible—*be careful*).

Weisberg (1985) also suggests a graph; this is not shown, but the command for this is included in the ado-file; the command is currently commented out with an asterisk ($*$) in column 1 of that line; just erase this asterisk if you want the graph.

The results can be quite different depending on whether the dependent variable is transformed. There are procedures in the literature (e.g., Cook and Weisberg 1982, example 2.4.5; Carroll and Ruppert 1988).

In addition to agreeing with Weisberg for the example shown below, this has also been tested against other data sets in the literature and agreement was found for these also (although only approximate agreement was found with Madansky 1988; however, since I have previously found errors in Madansky, I am not worried about this).

Examples:

```
. use weisberg
(Weisberg, 1985, p. 149)
. tidwell sqrta perim
score test for whether should transform perimetr: t =  -4.454 p-value: 0.0002
if above significant, transform perimetr using  0.564 (round to .5)
. use madan175, replace
(Madansky, 1988, p. 175)
. tidwell y x
score test for whether should transform x: t = -22.769 p-value: 0.0019
if above significant, transform x using  0.480 (round to .5)
```

References

Box, G. E. P. and P. W. Tidwell. 1962. Transformation of the independent variables. *Technometrics* 4: 531–550.

Carroll, R. J. and D. Ruppert. 1988. *Transformation and Weighting in Regression*. New York: Chapman and Hall.

Cook, R. D. and S. Weisberg. 1982. *Residuals and Influence in Regression*, New York: Chapman and Hall.

Madansky, A. 1988. *Prescriptions for Working Statisticians*. New York: Springer-Verlag.

Weisberg, S. 1985. *Applied Linear Regression*. 2d ed. New York: John Wiley & Sons.

sqv6 | Smoothed partial residual plots for logistic regression

Joseph Hilbe, Editor, STB, FAX 602-860-1446

The syntax for the command to produce partial residual plots for logistic regression is

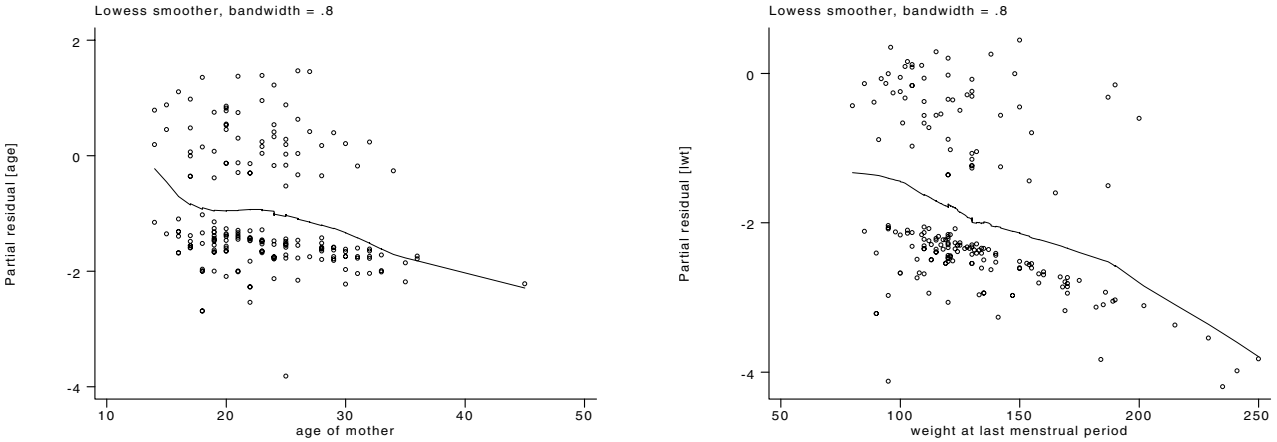
`lpartr` *depvar varlist* [`if` *exp*] [`in` *range*] [`, with(`*varlist*`) smooth(#) lowess bwidth(#) gen`]

`lpartr` produces by-variable smooths using a cubic spline or a lowess smoother. The default is cubic spline with a bandwidth of 4. This may be changed by using the `smooth` option. One may request the alternative lowess smooth, with a default bandwidth of .8, by using the `lowess` option. The lowess `bwidth` option allows the user to change the default; however, the bandwidth should range from between .1 and .9.

Use the `with` option on indicator or categorical variables. The cubic spline default will graph them, but the lowess smooth option will obviously not work. The `with` option allows the indicated variables to be fit in the model, thus contributing to the Pearson residual values, but they will not be smoothed. The consequence of this option is that continuous variables will be smoothed using all predictors in the fit.

Smoothing the partial Pearson residuals assists in evaluating the linearity of a continuous variable in a logistic model. The results produced by `lpartr` are the same as those that can be constructed using the Generalized Linear Model function in S-PLUS.

I shall show an example using the lowess smooth with default bandwidth. Consider variables on low birth weight from Hosmer and Lemeshow (1989); the dependent variable is `lbw` which is explained by the continuous variables `age` and `lwt`, the indicator variables `smoke`, `ht`, and `ui`, and the categorical variable `ptl`. Typing '`lpartr low age lwt, with(smoke ht ui ptl) l`' produces



References

Chambers, J. M. and T. J. Hastie. Editors. 1992. *Statistical Models in S.* Pacific Grove, CA: Wadsworth & Brooks/Cole.

Hosmer, D. W., Jr., and S. Lemeshow. 1989. *Applied Logistic Regression.* New York: John Wiley & Sons.

McCullagh, P. and J. A. Nelder. 1989. *Generalized Linear Models.* 2d ed. New York: Chapman and Hall.

srd14 | Cook–Weisberg test of heteroscedasticity

Richard Goldstein, Qualitas, Brighton, MA, EMAIL goldst@harvarda.bitnet

The syntax of `cwhetero` is

`cwhetero` *varlist*

This command can only be used after `fit`, and no options are available. At the end of the program your data set is automatically reset to the saved data set on disk—*be careful.*

This tests one of the assumptions underlying ordinary least squares regression: constant variance. The varlist can be any, or all, of the right-hand-side variables from your fit.

Because the data set is reset at the end of each run, if you want to test more than one set of RHS variables, you must re-estimate your model. This is most easily done by typing '`qui fit` *varlist*' after using this program. Two test results are

presented after each run along with two graphs. The first test, and the first graph, reference the subset of variables named in the command. The second test, and graph, use the full set of variables to test for variance as a function of the predicted values. If the test is statistically significant (if the graph shows a "wedge" shape), then there is a problem.

This is discussed, with examples, in Cook and Weisberg 1983, and Weisberg 1985, 135–140. The data sets used are included on the disk, under the names of `weisflok.dta`, `weisgas.dta` and `mtb_tree.dta` (this is the well-known Minitab tree data). All examples in both the article and the book are matched.

Note that `qreg` can also be used to test for heteroscedasticity. The `cwhetero.log` file on the disk contains results from the included data sets and also has some `qreg`'s for comparison (though without any tests from the `qreg` results).

The following example uses the Stata `auto.dta` file.

```
. use weisgas
. fit y x1-x4
  Source |       SS       df       MS              Number of obs =      32
---------+------------------------------           F(  4,    27) =   84.54
   Model |  2520.27241     4  630.068101            Prob > F      =  0.0000
Residual |  201.227595    27  7.45287388            R-square      =  0.9261
---------+------------------------------           Adj R-square  =  0.9151
   Total |     2721.50    31  87.7903226            Root MSE      =    2.73
------------------------------------------------------------------------------
       y |      Coef.   Std. Err.     t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
      x1 |  -.0286089   .0906015   -0.316   0.755      -.2145079    .1572901
      x2 |   .2158169    .067718    3.187   0.004        .076871    .3547628
      x3 |  -4.320052   2.850967   -1.515   0.141      -10.16975     1.52965
      x4 |    8.97489   2.772632    3.237   0.003        3.28592    14.66386
   _cons |   1.015018   1.861308    0.545   0.590      -2.804072    4.834107
------------------------------------------------------------------------------
. cwhetero x1
Score test (x1) =    1.395; chi-square p-value (df) =  0.238 (1)
Score test (pred fit) =    0.000; chi-square p-value (df) =  0.985 (1)
. qui fit y x1-x4
. cwhetero x4
Score test (x4) =    0.010; chi-square p-value (df) =  0.922 (1)
Score test (pred fit) =    0.000; chi-square p-value (df) =  0.985 (1)
. qui fit y x1-x4
. cwhetero x1 x4
Score test (x1 x4) =    9.283; chi-square p-value (df) =  0.010 (2)
Score test (pred fit) =    0.000; chi-square p-value (df) =  0.985 (1)
. qui fit y x1-x4
. cwhetero x1-x4
Score test (x1 x2 x3 x4) =   10.299; chi-square p-value (df) =  0.036 (4)
Score test (pred fit) =    0.000; chi-square p-value (df) =  0.985 (1)
```

References

Cook, R. D. and S. Weisberg. 1983. Diagnostics for heteroscedasticity in regression. *Biometrika* 70: 1–10.

Weisberg, S. 1985. *Applied Linear Regression.* 2d ed. New York: John Wiley & Sons.

srd15	Restricted cubic spline functions

Richard Goldstein, Qualitas, Brighton, MA, EMAIL goldst@harvarda.bitnet

The syntax for the `spline` command is

`spline` # *variable* [`if` *exp*] [`in` *range*]

where # must be an integer between 3 and 7 and determines the number of knots. The variable name should refer to a continuous variable.

"Spline functions are piecewise polynomials used in curve fitting. That is, they are polynomials within intervals of [the variable] that are connected across different intervals of [the variable]." (Harrell 1992, p. 1—5) Cubic spline functions can be used to help determine the shape of a regression function and/or possible predictor transformations. For these purposes, cubic splines are very flexible and very much less likely to be "fooled" than are quadratics or other normal polynomials or linear piecewise regression.

Cubic splines can be unstable in their tails so these are restricted to make the tail areas linear (see Stone and Koo 1985). The user must choose the number of knots (between 3 and 7, with much literature recommending 4 or 5); the join points are automatically set at specific quantiles following Harrell's recommendations (p. 1—9). If there are fewer than 100 points then the two tail joins are the fifth smallest and fifth largest points rather than a quantile.

`spline` forms a number of new variables (equal to #-2). These new variables should then be added to your model (whether regression, logistic, Cox or some other single-equation model) as new right-hand-side variables in addition to the original form of the variable being examined and any other variables that you think belong in the model. As currently structured, the new variables are given default names (`knot1`, etc.) and you can only `spline` one variable at a time. If you want to spline more, you can rename these variables and start again.

After estimating your model, you can test for problems with a joint test of the significance of your original variable and the new knot variables using `testparm` (simplest) or `test`; e.g., `testparm variable knot*`. If this test is significant, then the assumed linearity of your original variable is false. You can either use these manufactured variables (which I find hard to interpret) or you can search for a possible transformation of your variable; if you obtain the predicted values from this test model and graph these against your original variable you may find this graph helpful in picking a transformation. If you have additional variables in your model, you may want to calculate the value of the spline function (including your original variable) and graph this against the original variable. If you transform, you can, of course, test the adequacy of your transformation by fitting a new cubic spline function. The extension of this to more than one continuous variable is fairly obvious.

The new variables are defined as follows (where X is the original variable, with k knots $t_1,\ldots,t_k$, and j goes from 1 to $k-2$):

$$X_{j+1} = (X-t_j)^3_+ - (X-t_{k-1})^3_+(t_k-t_j)/(t_k-t_{k-1}) + (X-t_k)^3_+(t_{k-1}-t_j)/(t_k-t_{k-1})$$

where the "+" symbol means include if $X >$ join point (i.e., define a new variable that is "1" if $X >$ join point and is "0" if $X <=$ join point). The following example may help: say one wants three knots at 6, 21 and 65 years of age; the new cubic spline function for age is

$$(X-6)^3_+ - 59(X-21)^3_+/44 + 15(X-65)^3_+/44$$

where the numbers come from the spacing between the knots.

Good discussions of this, including discussions of the specific advantages compared with piecewise linear splines or quadratics, can be found in Harrell (1992), or in Harrell, Lee, and Pollock (1988), Herndon and Harrell (1990), and Durrleman and Simon (1989). None of the Harrell papers include data. The Durrleman and Simon paper uses the Stanford Heart Transplant data (157 cases of it), included on the STB disk. The spline program matches the Durrleman and Simon result (except for a very small difference for 4 knots).

The only visible products of this program are the new variable(s) and three lines of "advice." Most of the literature recommends 4 or 5 knots as best, though you might need to stop at 3 for smaller data sets.

The following example uses the Stanford heart data as used in Durrleman and Simon (1989). The data is on the STB diskette. We try both 4 and 5 knots and present the cox model results, the joint test results, and the graphs for these. Clearly there is a problem here. The final part of the example shows that we can also use the Stata command `linktest` to find the problem. Here the graphs are very similar, though some might think that the graph based on 4 knots is easier to interpret (relative to a possible transformation of the right-hand-side variable) than the graph based on 5 knots. However, we might prefer to transform the left-hand-side variable in this case.

```
. use stanford
(Miller & Halperin (ex T5 miss))

. describe
Contains data from stanford.dta
  Obs:   157 (max= 27679)                  Miller & Halperin (ex T5 miss)
 Vars:     3 (max=   254)
Width:    12 (max=   510)
  1. time         float  %9.0g             survival time (days)
  2. dead         float  %9.0g    dead
  3. age          float  %9.0g             at first transplant
Sorted by:

. spline 5 age
for test of functional form use testparm age kn*
for possible help in transforming, obtain the predicted values,
using fpredict, and then gr pred.val by age

. describe
Contains data from stanford.dta
  Obs:   157 (max= 27666)                  Miller & Halperin (ex T5 miss)
 Vars:     6 (max=   254)
Width:    36 (max=   510)
  1. time         float  %9.0g             survival time (days)
  2. dead         float  %9.0g    dead
  3. age          float  %9.0g             at first transplant
  4. knot1        double %10.0g
  5. knot2        double %10.0g
  6. knot3        double %10.0g
Sorted by:
Note:  Data has changed since last save

. cox time age kn*, dead(dead)
Iteration 0:  Log Likelihood =-451.19416
Iteration 1:  Log Likelihood =-443.12036
Iteration 2:  Log Likelihood =-442.23375
Iteration 3:  Log Likelihood =-442.23322
Iteration 4:  Log Likelihood =-442.23322
Cox regression                                        Number of obs =    157
                                                      chi2(4)       =  17.92
                                                      Prob > chi2   = 0.0013
Log Likelihood = -442.23322                           Pseudo R2     = 0.0199
------------------------------------------------------------------------------
    time |
    dead |      Coef.   Std. Err.     t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |  -.0027006   .0432297   -0.062   0.950      -.0881048    .0827037
   knot1 |  -.0000224   .0000853   -0.262   0.794      -.0001908    .0001461
   knot2 |   .0004519   .0004856    0.930   0.354      -.0005075    .0014113
   knot3 |   .0000523   .0007805    0.067   0.947      -.0014896    .0015942
------------------------------------------------------------------------------

. testparm age kn*
 ( 1)  age = 0.0
 ( 2)  knot1 = 0.0
 ( 3)  knot2 = 0.0
 ( 4)  knot3 = 0.0
       F(  4,   153) =    5.46
            Prob > F =    0.0004

. predict yhat, index

. gr yhat age, sav(stanage5)

. use stanford, replace
(Miller & Halperin (ex T5 miss))

. spline 4 age
for test of functional form use testparm age kn*
for possible help in transforming, obtain the predicted values,
using fpredict, and then gr pred.val by age
```

```
. cox time age kn*, dead(dead)
Iteration 0:  Log Likelihood =-451.19416
Iteration 1:  Log Likelihood = -450.1309
Iteration 2:  Log Likelihood =-442.84696
Iteration 3:  Log Likelihood =-442.41776
Iteration 4:  Log Likelihood =-442.41348
Iteration 5:  Log Likelihood =-442.41348
Cox regression                                        Number of obs =    157
                                                      chi2(3)       =  17.56
                                                      Prob > chi2   = 0.0005
Log Likelihood = -442.41348                           Pseudo R2     = 0.0195
------------------------------------------------------------------------------
    time |
    dead |      Coef.   Std. Err.     t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |  -.0196696   .0324776   -0.606   0.546      -.0838287    .0444894
   knot1 |   .0000255   .0000382    0.666   0.506      -.0000501     .000101
   knot2 |   .0003049   .0002086    1.461   0.146      -.0001073     .000717
------------------------------------------------------------------------------

. testparm age kn*
 ( 1)  age = 0.0
 ( 2)  knot1 = 0.0
 ( 3)  knot2 = 0.0
       F(  3,   154) =    7.41
            Prob > F =    0.0001

. predict yhat

. gr yhat age, sav(stanage4)

. use stanford, replace
(Miller & Halperin (ex T5 miss))

. cox time age, dead(dead)
Iteration 0:  Log Likelihood =-451.19416
Iteration 1:  Log Likelihood =-447.44957
Iteration 2:  Log Likelihood =-447.39826
Iteration 3:  Log Likelihood =-447.39825
Cox regression                                        Number of obs =    157
                                                      chi2(1)       =   7.59
                                                      Prob > chi2   = 0.0059
Log Likelihood = -447.39825                           Pseudo R2     = 0.0084
------------------------------------------------------------------------------
    time |
    dead |      Coef.   Std. Err.     t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |   .0298421   .0113515    2.629   0.009       .0074196    .0522647
------------------------------------------------------------------------------

. linktest
Iteration 0:  Log Likelihood =-640.37799
Iteration 1:  Log Likelihood =-635.31318
Iteration 2:  Log Likelihood =-635.18825
Iteration 3:  Log Likelihood =-635.18818

Cox regression                                        Number of obs =    157
                                                      chi2(2)       =  10.38
                                                      Prob > chi2   = 0.0056
Log Likelihood = -635.18818                           Pseudo R2     = 0.0081
------------------------------------------------------------------------------
    time |      Coef.   Std. Err.     t     P>|t|       [95% Conf. Interval]
---------+--------------------------------------------------------------------
    _hat |  -4.135913   1.505569   -2.747   0.007      -7.109994   -1.161832
  _hatsq |   2.062398    .678925    3.038   0.003       .7212579    3.403537
------------------------------------------------------------------------------
```

Figures

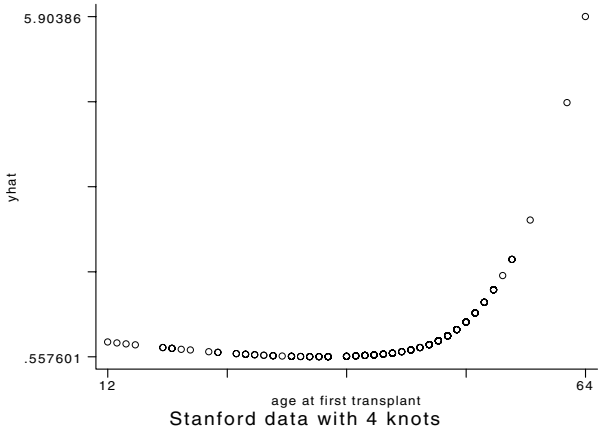


Figure 1

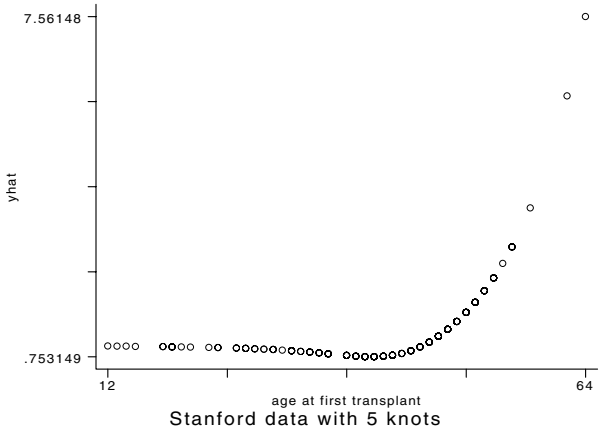


Figure 2

References

Durrleman, S. and R. Simon. 1989. Flexible regression models with cubic splines. *Statistics in Medicine* 8: 551–61.

Harrell, F. E. Jr., 1992. *Survival and Risk Analysis*, unpublished MS.

Harrell, F. E. Jr., K. L. Lee and B. G. Pollock. 1988. Regression models in clinical studies: Determining relationships between predictors and response. *Journal of the National Cancer Institute* 80: 1198–1202.

Herndon, J. E. II, and F. E. Harrell, Jr. 1990. The restricted cubic spline hazard model. *Communications in Statistics–Theory and Methods* 19: 639–663.

Stone C. J. and C-Y Koo. 1985. Additive splines in statistics. *Proceedings of the Statistical Computing Section ASA*, pp. 45–48.