Title

nl — Nonlinear least-squares estimation

Description	Quick start	Menu	Syntax
Options	Remarks and examples	Stored results	Methods and formulas
Acknowledgments	References	Also see	

Description

nl fits an arbitrary nonlinear regression function by least squares. With the interactive version of the command, you enter the function directly on the command line or in the dialog box by using a *substitutable expression*. If you have a function that you use regularly, you can write a *substitutable expression program* and use the second syntax to avoid having to reenter the function every time. The function evaluator program version gives you the most flexibility in exchange for increased complexity; with this version, your program is given a vector of parameters and a variable list, and your program computes the regression function.

When you write a substitutable expression program or function evaluator program, the first two letters of the name must be nl. *sexp_prog* and *func_prog* refer to the name of the program without the first two letters. For example, if you wrote a function evaluator program named nlregss, you would type nl regss @ ... to estimate the parameters.

Quick start

Linear model of y with parameters b0 for the constant and b1 for x1
    nl (y = {b0} + {b1}*x1)

Add the variables() option to allow for missing values of y and x1
    nl (y = {b0} + {b1}*x1), variables(y x1)

Same as above, but specify starting values
    nl (y = {b0=.5} + {b1=2}*x1), variables(y x1)

Add variables x2 and x3 and parameters b2 and b3
    nl (y = {b0} + {b1}*x1 + {b2}*x2 + {b3}*x3), variables(y x1 x2 x3)

Same as above, but use {xb:} to specify a linear combination of variables
    nl (y = {b0=.5} + {xb:x1 x2 x3}), variables(y x1 x2 x3)

An exponential model
    nl (y = {b0} + {b1}*{b2}^x1), variables(y x1)

Same as above, but use nl's built-in function exp3 to specify the model
    nl exp3: y x1, variables(y x1)

Menu

Statistics > Linear models and related > Nonlinear least-squares estimation

Syntax

Interactive version

nl (*depvar* = *<sexp>*) $\big[$ *if* $\big]$ $\big[$ *in* $\big]$ $\big[$ *weight* $\big]$ $\big[$, *options* $\big]$

Programmed substitutable expression version

nl *sexp_prog* : *depvar* $\big[$ *varlist* $\big]$ $\big[$ *if* $\big]$ $\big[$ *in* $\big]$ $\big[$ *weight* $\big]$ $\big[$, *options* $\big]$

Function evaluator program version

nl *func_prog* @ *depvar* $\big[$ *varlist* $\big]$ $\big[$ *if* $\big]$ $\big[$ *in* $\big]$ $\big[$ *weight* $\big]$,

$\Big\{$ <u>parameters</u>(*namelist*) | <u>nparameters</u>(*#*)$\Big\}$ $\big[$ *options* $\big]$

where

depvar is the dependent variable;

<sexp> is a substitutable expression;

sexp_prog is a substitutable expression program; and

func_prog is a function evaluator program.

options	Description

[Model]

variables(*varlist*)	variables in model
initial(*initial_values*)	initial values for parameters
* parameters(*namelist*)	parameters in model (function evaluator program version only)
* nparameters(*#*)	number of parameters in model (function evaluator program version only)
sexp_options	options for substitutable expression program
func_options	options for function evaluator program

[Model 2]

lnlsq(*#*)	use log least-squares where $\ln(depvar − \#)$ is assumed to be normally distributed
noconstant	the model has no constant term; seldom used
hasconstant(*name*)	use *name* as constant term; seldom used

[SE/Robust]

vce(*vcetype*)	*vcetype* may be gnr, robust, cluster *clustvar*, bootstrap, jackknife, hac *kernel*, hc2, or hc3

[Reporting]

level(*#*)	set confidence level; default is level(95)
leave	create variables containing derivative of $E(y)$
title(*string*)	display *string* as title above the table of parameter estimates
title2(*string*)	display *string* as subtitle
display_options	control column formats and line width

[Optimization]

optimization_options	control the optimization process; seldom used
eps(*#*)	specify *#* for convergence criterion; default is eps(1e-5)
delta(*#*)	specify *#* for computing derivatives; default is delta(4e-7)
coeflegend	display legend instead of statistics

---

* For function evaluator program version, you must specify parameters(*namelist*) or nparameters(*#*), or both.

bootstrap, by, collect, jackknife, rolling, statsby, and svy are allowed; see [U] 11.1.10 Prefix commands.

Weights are not allowed with the bootstrap prefix; see [R] bootstrap.

aweights are not allowed with the jackknife prefix; see [R] jackknife.

vce(), leave, and weights are not allowed with the svy prefix; see [SVY] svy.

aweights, fweights, and iweights are allowed; see [U] 11.1.6 weight.

coeflegend does not appear in the dialog box.

See [U] 20 Estimation and postestimation commands for more capabilities of estimation commands.

Options

[Model]

variables(*varlist*) specifies the variables in the model. nl ignores observations for which any of these variables have missing values. If you do not specify variables(), then nl issues an error message with return code 480 if the estimation sample contains any missing values.

initial(*initial_values*) specifies the initial values to begin the estimation. You can specify a $1 \times k$ matrix, where k is the number of parameters in the model, or you can specify a parameter name, its initial value, another parameter name, its initial value, and so on. For example, to initialize alpha to 1.23 and delta to 4.57, you would type

```
nl ... , initial(alpha 1.23 delta 4.57) ...
```

Initial values declared using this option override any that are declared within substitutable expressions. If you specify a parameter that does not appear in your model, nl exits with error code 480. If you specify a matrix, the values must be in the same order that the parameters are declared in your model. nl ignores the row and column names of the matrix.

parameters(*namelist*) specifies the names of the parameters in the model. The names of the parameters must adhere to the naming conventions of Stata's variables; see [U] **11.3 Naming conventions**. If you specify both parameters() and nparameters(), the number of names in the former must match the number specified in the latter; if not, nl issues an error message with return code 198.

nparameters(*#*) specifies the number of parameters in the model. If you do not specify names with the parameters() option, nl names them b1, b2, ..., b#. If you specify both parameters() and nparameters(), the number of names in the former must match the number specified in the latter; if not, nl issues an error message with return code 198.

sexp_options refer to any options allowed by your *sexp_prog*.

func_options refer to any options allowed by your *func_prog*.

───────── Model 2 ─────────

lnlsq(*#*) fits the model by using log least-squares, which we define as least squares with shifted lognormal errors. In other words, $\ln(depvar - \#)$ is assumed to be normally distributed. Sums of squares and deviance are adjusted to the same scale as *depvar*.

noconstant indicates that the function does not include a constant term. This option is generally not needed, even if there is no constant term in the model, unless the coefficient of variation (over observations) of the partial derivative of the function with respect to a parameter is less than eps() and that parameter is not a constant term.

hasconstant(*name*) indicates that parameter *name* be treated as the constant term in the model and that nl should not use its default algorithm to find a constant term. As with noconstant, this option is seldom used.

───────── SE/Robust ─────────

vce(*vcetype*) specifies the type of standard error reported, which includes types that are derived from asymptotic theory (gnr), that are robust to some kinds of misspecification (robust), that allow for intragroup correlation (cluster *clustvar*), and that use bootstrap or jackknife methods (bootstrap, jackknife); see [R] *vce_option*.

vce(gnr), the default, uses the conventionally derived variance estimator for nonlinear models fit using Gauss–Newton regression.

nl also allows the following:

vce(hac *kernel* [*#*]) specifies that a heteroskedasticity- and autocorrelation-consistent (HAC) variance estimate be used. HAC refers to the general form for combining weighted matrices to form the variance estimate. There are three kernels available for nl:

$$\underline{n}west \mid \underline{g}allant \mid \underline{an}derson$$

specifies the number of lags. If # is not specified, $N - 2$ is assumed.

vce(hac *kernel* $\begin{bmatrix} # \end{bmatrix}$) is not allowed if weights are specified.

vce(hc2) and vce(hc3) specify alternative bias corrections for the robust variance calculation. vce(hc2) and vce(hc3) may not be specified with the svy prefix. By default, vce(robust) uses $\widehat{\sigma}_j^2 = \{n/(n-k)\}u_j^2$ as an estimate of the variance of the jth observation, where u_j is the calculated residual and $n/(n-k)$ is included to improve the overall estimate's small-sample properties.

vce(hc2) instead uses $u_j^2/(1 - h_{jj})$ as the observation's variance estimate, where h_{jj} is the jth diagonal element of the hat (projection) matrix. This produces an unbiased estimate of the covariance matrix if the model is homoskedastic. vce(hc2) tends to produce slightly more conservative confidence intervals than vce(robust).

vce(hc3) uses $u_j^2/(1 - h_{jj})^2$ as suggested by Davidson and MacKinnon (1993 and 2004), who report that this often produces better results when the model is heteroskedastic. vce(hc3) produces confidence intervals that tend to be even more conservative.

See, in particular, Davidson and MacKinnon (2004, 239), who advocate the use of vce(hc2) or vce(hc3) instead of the plain robust estimator for nonlinear least squares.

___

Reporting

___

level(#); see [R] **Estimation options**.

leave leaves behind after estimation a set of new variables with the same names as the estimated parameters containing the derivatives of $E(y)$ with respect to the parameters. If the dataset contains an existing variable with the same name as a parameter, then using leave causes nl to issue an error message with return code 110.

leave may not be specified with vce(cluster *clustvar*) or the svy prefix.

title(*string*) specifies an optional title that will be displayed just above the table of parameter estimates.

title2(*string*) specifies an optional subtitle that will be displayed between the title specified in title() and the table of parameter estimates. If title2() is specified but title() is not, title2() has the same effect as title().

display_options: cformat(%*fmt*), pformat(%*fmt*), sformat(%*fmt*), and nolstretch; see [R] **Estimation options**.

___

Optimization

___

optimization_options: <u>iterate</u>(#), $\begin{bmatrix} no \end{bmatrix}$log, <u>trace</u>. iterate() specifies the maximum number of iterations, log/nolog specifies whether to show the iteration log (see set iterlog in [R] *set iter*), and trace specifies that the iteration log should include the current parameter vector. These options are seldom used.

eps(#) specifies the convergence criterion for successive parameter estimates and for the residual sum of squares. The default is eps(1e-5).

delta(#) specifies the relative change in a parameter to be used in computing the numeric derivatives. The derivative for parameter β_i is computed as $\{f(X, \beta_1, \beta_2, \ldots, \beta_i + d, \beta_{i+1}, \ldots) - f(X, \beta_1, \beta_2, \ldots, \beta_i, \beta_{i+1}, \ldots)\}/d$, where d is $\delta(\beta_i + \delta)$. The default is delta(4e-7).

The following options are available with nl but are not shown in the dialog box:

coeflegend; see [R] **Estimation options**.

Remarks and examples

<div align="right">stata.com</div>

Remarks are presented under the following headings:

> *Substitutable expressions*
> *Substitutable expression programs*
> *Built-in functions*
> *Lognormal errors*
> *Other uses*
> *Weights*
> *Potential errors*
> *General comments on fitting nonlinear models*
> *Function evaluator programs*

nl fits an arbitrary nonlinear function by least squares. The interactive version allows you to enter the function directly on the command line or dialog box using *substitutable expressions*. You can write a *substitutable expression program* for functions that you fit frequently to save yourself time. Finally, *function evaluator programs* give you the most flexibility in defining your nonlinear function, though they are more complicated to use.

The next section explains the substitutable expressions that are used to define the regression function, and the section thereafter explains how to write substitutable expression program files so that you do not need to type in commonly used functions over and over. Later sections highlight other features of nl.

The final section discusses function evaluator programs. If you find substitutable expressions adequate to define your nonlinear function, then you can skip that section entirely. Function evaluator programs are generally needed only for complicated problems, such as multistep estimators. The program receives a vector of parameters at which it is to compute the function and a variable into which the results are to be placed.

Substitutable expressions

You define the nonlinear function to be fit by nl by using a substitutable expression. Substitutable expressions are just like any other mathematical expressions involving scalars and variables, such as those you would use with Stata's generate command, except that the parameters to be estimated are bound in braces. See [U] 13.2 Operators and [U] 13.3 Functions for more information on expressions.

For example, suppose that you wish to fit the function

$$y_i = \beta_0(1 - e^{-\beta_1 x_i}) + \epsilon_i$$

where β_0 and β_1 are the parameters to be estimated and ϵ_i is an error term. You would simply type

```
. nl (y = {b0}*(1 - exp(-1*{b1}*x)))
```

You must enclose the entire equation in parentheses. Because b0 and b1 are enclosed in braces, nl knows that they are parameters in the model. nl will initialize b0 and b1 to zero by default. To request that nl initialize b0 to 1 and b1 to 0.25, you would type

```
. nl (y = {b0=1}*(1 - exp(-1*{b1=0.25}*x)))
```

That is, inside the braces denoting a parameter, you put the parameter name followed by an equal sign and the initial value. If a parameter appears in your function multiple times, you need only specify an initial value only once (or never, if you wish to set the initial value to zero). If you do specify more than one initial value for the same parameter, nl will use the *last* value given. Parameter names must follow the same conventions as variable names. See [U] 11.3 Naming conventions.

Frequently, even nonlinear functions contain linear combinations of variables. As an example, suppose that you wish to fit the function

$$y_i = \beta_0 \left\{ 1 - e^{-(\beta_1 x_{1i} + \beta_2 x_{2i} + \beta_3 x_{3i})} \right\} + \epsilon_i$$

`nl` allows you to declare a linear combination of variables by using the shorthand notation

```
. nl (y = {b0=1}*(1 - exp(-1*{xb: x1 x2 x3})))
```

In the syntax `{xb: x1 x2 x3}`, you are telling `nl` that you are declaring a linear combination named `xb` that is a function of three variables, `x1`, `x2`, and `x3`. `nl` will create three parameters, named `xb_x1`, `xb_x2`, and `xb_x3`, and initialize them to zero. Instead of typing the previous command, you could have typed

```
. nl (y = {b0=1}*(1 - exp(-1*({xb_x1}*x1 + {xb_x2}*x2 + {xb_x3}*x3))))
```

and yielded the same result. You can refer to the parameters created by `nl` in the linear combination later in the function, though you must declare the linear combination first if you intend to do that. When creating linear combinations, `nl` ensures that the parameter names it chooses are unique and have not yet been used in the function.

In general, there are three rules to follow when defining substitutable expressions:

1. Parameters of the model are bound in braces: `{b0}`, `{param}`, etc.
2. Initial values for parameters are given by including an equal sign and the initial value inside the braces: `{b0=1}`, `{param=3.571}`, etc.
3. Linear combinations of variables can be included using the notation `{eqname:varlist}`, for example, `{xb: mpg price weight}`, `{score: w x z}`, etc. Parameters of linear combinations are initialized to zero.

If you specify initial values by using the `initial()` option, they override whatever initial values are given within the substitutable expression. Substitutable expressions are so named because, once values are assigned to the parameters, the resulting expression can be handled by `generate` and `replace`.

▷ Example 1

We wish to fit the CES production function

$$\ln Q_i = \beta_0 - \frac{1}{\rho} \ln \left\{ \delta K_i^{-\rho} + (1 - \delta) L_i^{-\rho} \right\} + \epsilon_i \tag{1}$$

where $\ln Q_i$ is the log of output for firm i; K_i and L_i are firm i's capital and labor usage, respectively; and ϵ_i is a regression error term. Because ρ appears in the denominator of a fraction, zero is not a feasible initial value; for a CES production function, $\rho = 1$ is a reasonable choice. Setting $\delta = 0.5$ implies that labor and capital have equal impacts on output, which is also a reasonable choice for an initial value. We type

```
. use https://www.stata-press.com/data/r18/production
. nl (lnoutput = {b0} - 1/{rho=1}*ln({delta=0.5}*capital^(-1*{rho}) +
> (1 - {delta})*labor^(-1*{rho})))
Iteration 0:   Residual SS = 29.38631
Iteration 1:   Residual SS = 29.36637
Iteration 2:   Residual SS = 29.36583
Iteration 3:   Residual SS = 29.36581
Iteration 4:   Residual SS = 29.36581
Iteration 5:   Residual SS = 29.36581
Iteration 6:   Residual SS = 29.36581
Iteration 7:   Residual SS = 29.36581
```

Source	SS	df	MS		
				Number of obs =	100
Model	91.144992	2	45.5724962	R-squared =	0.7563
Residual	29.365806	97	.302740263	Adj R-squared =	0.7513
				Root MSE =	.5502184
Total	120.5108	99	1.21728079	Res. dev. =	161.2538

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

Note: Parameter **b0** is used as a constant term during estimation.

nl will attempt to find a constant term in the model and, if one is found, mention it at the bottom of the output. nl found b0 to be a constant because the partial derivative $\partial \ln Q_i / \partial$b0 has a coefficient of variation less than eps() in the estimation sample.

The elasticity of substitution for the CES production function is $\sigma = 1/(1 + \rho)$; and, having fit the model, we can use nlcom to estimate it:

```
. nlcom (1/(1 + _b[/rho]))
       _nl_1: 1/(1 + _b[/rho])
```

lnoutput	Coefficient	Std. err.	z	P>|z|	[95% conf. interval]	
_nl_1	.4189372	.0829424	5.05	0.000	.256373	.5815014

See [R] **nlcom** and [U] **13.5 Accessing coefficients and standard errors** for more information.

◁

nl's output closely mimics that of regress; see [R] **regress** for more information. The R^2, sums of squares, and similar statistics are calculated in the same way that regress calculates them. If no "constant" term is specified, the usual caveats apply to the interpretation of the R^2 statistic; see the comments and references in Goldstein (1992). Unlike regress, nl does not report a model F statistic, because a test of the joint significance of all the parameters except the constant term may not be relevant in a nonlinear model.

Substitutable expression programs

If you fit the same model often or if you want to write an estimator that will operate on whatever variables you specify, then you will want to write a substitutable expression program. That program will return a macro containing a substitutable expression that `nl` can then evaluate, and it may optionally calculate initial values as well. The name of the program must begin with the letters `nl`.

To illustrate, suppose that you use the CES production function often in your work. Instead of typing in the formula each time, you can write a program like this:

```
program nlces, rclass
        version 18.0
        syntax varlist(min=3 max=3) [if]
        local logout : word 1 of 'varlist'
        local capital : word 2 of 'varlist'
        local labor : word 3 of 'varlist'
        // Initial value for b0 given delta=0.5 and rho=1
        tempvar y
        generate double 'y' = 'logout' + ln(0.5*'capital'^-1 + 0.5*'labor'^-1)
        summarize 'y' 'if', meanonly
        local b0val = r(mean)
        // Terms for substitutable expression
        local capterm "{delta=0.5}*'capital'^(-1*{rho})"
        local labterm "(1-{delta})*'labor'^(-1*{rho})"
        local term2   "1/{rho=1}*ln('capterm' + 'labterm')"
        // Return substitutable expression and title
        return local eq "'logout' = {b0='b0val'} - 'term2'"
        return local title "CES ftn., ln Q='logout', K='capital', L='labor'"
end
```

The program accepts three variables for log output, capital, and labor, and it accepts an `if` *exp* qualifier to restrict the estimation sample. All programs that you write to use with `nl` must accept an `if` *exp* qualifier because, when `nl` calls the program, it passes a binary variable that marks the estimation sample (the variable equals one if the observation is in the sample and zero otherwise). When calculating initial values, you will want to restrict your computations to the estimation sample, and you can do so by using `if` with any commands that accept `if` *exp* qualifiers. Even if your program does not calculate initial values or otherwise use the `if` qualifier, the `syntax` statement must still allow it. See [P] **syntax** for more information on the `syntax` command and the use of `if`.

As in the previous example, reasonable initial values for δ and ρ are 0.5 and 1, respectively. Conditional on those values, (1) can be rewritten as

$$\beta_0 = \ln Q_i + \ln(0.5 K_i^{-1} + 0.5 L_i^{-1}) - \epsilon_i \tag{2}$$

so a good initial value for β_0 is the mean of the right-hand side of (2) ignoring ϵ_i. Lines 7–10 of the function evaluator program calculate that mean and store it in a local macro. Notice the use of `if` in the `summarize` statement so that the mean is calculated only for the estimation sample.

The final part of the program returns two macros. The macro `title` is optional and defines a short description of the model that will be displayed in the output immediately above the table of parameter estimates. The macro `eq` is required and defines the substitutable expression that `nl` will use. If the expression is short, you can define it all at once. However, because the expression used here is somewhat lengthy, defining local macros and then building up the final expression from them is easier.

To verify that there are no errors in your program, you can call it directly and then use `return list`:

```
. use https://www.stata-press.com/data/r18/production
. nlces lnoutput capital labor
  (output omitted )
. return list
macros:
             r(title) : "CES ftn., ln Q=lnoutput, K=capital, L=labor"
                r(eq) : "lnoutput = {b0=3.7116062646663641} - 1/{rho=1}*ln({delt
> a=0.5}*capital^(-1*{rho}) + (1-{delta})*labor^(-1*{rho}))"
```

The macro `r(eq)` contains the same substitutable expression that we specified at the command line in the preceding example, except for the initial value for b0. In short, an `nl` substitutable expression program should return in `r(eq)` the same substitutable expression you would type at the command line. The only difference is that when writing a substitutable expression program, you do not bind the entire expression inside parentheses.

Having written the program, you can use it by typing

```
. nl ces: lnoutput capital labor
```

(There is a space between `nl` and `ces`.) The output is identical to that shown in example 1, save for the title defined in the function evaluator program that appears immediately above the table of parameter estimates.

❑ Technical note

You will want to store `nlces` as an ado-file called `nlces.ado`. The alternative is to type the code into Stata interactively or to place the code in a do-file. While those alternatives are adequate for occasional use, if you save the program as an ado-file, you can use the function anytime you use Stata without having to redefine the program. When `nl` attempts to execute `nlces`, if the program is not in Stata's memory, Stata will search the disk(s) for an ado-file of the same name and, if found, automatically load it. All you have to do is name the file with the `.ado` suffix and then place it in a directory where Stata will find it. You should put the file in the directory Stata reserves for user-written ado-files, which, depending on your operating system, is `c:\ado\personal` (Windows), `~/ado/personal` (Unix), or `~:ado:personal` (Mac). See [U] **17 Ado-files**. ❑

Sometimes you may want to pass additional options to the substitutable expression program. You can modify the `syntax` statement of your program to accept whatever options you wish. Then when you call `nl` with the syntax

```
. nl func_prog: varlist, options
```

any *options* that are not recognized by `nl` (see the table of options at the beginning of this entry) are passed on to your function evaluator program. The only other restriction is that your program cannot accept an option named `at` because `nl` uses that option with function evaluator programs.

Built-in functions

Some functions are used so often that `nl` has them built in so that you do not need to write them yourself. `nl` automatically chooses initial values for the parameters, though you can use the `initial(...)` option to override them.

Three alternatives are provided for exponential regression with one asymptote:

exp3 $y_i = \beta_0 + \beta_1 \beta_2^{x_i} + \epsilon_i$

exp2 $y_i = \beta_1 \beta_2^{x_i} + \epsilon_i$

exp2a $y_i = \beta_1 \left(1 - \beta_2^{x_i}\right) + \epsilon_i$

For instance, typing nl exp3: ras dvl fits the three-parameter exponential model (parameters β_0, β_1, and β_2) using y_i = ras and x_i = dvl.

Two alternatives are provided for the logistic function (symmetric sigmoid shape; not to be confused with logistic regression):

log4 $y_i = \beta_0 + \beta_1 \Big/ \left[1 + \exp\{-\beta_2(x_i - \beta_3)\}\right] + \epsilon_i$

log3 $y_i = \beta_1 \Big/ \left[1 + \exp\{-\beta_2(x_i - \beta_3)\}\right] + \epsilon_i$

Finally, two alternatives are provided for the Gompertz function (asymmetric sigmoid shape):

gom4 $y_i = \beta_0 + \beta_1 \exp\left[-\exp\{-\beta_2(x_i - \beta_3)\}\right] + \epsilon_i$

gom3 $y_i = \beta_1 \exp\left[-\exp\{-\beta_2(x_i - \beta_3)\}\right] + \epsilon_i$

Lognormal errors

A nonlinear model with errors that are independent and identically distributed normal may be written as

$$y_i = f(\mathbf{x}_i, \boldsymbol{\beta}) + u_i, \qquad u_i \sim N(0, \sigma^2) \tag{3}$$

for $i = 1, \ldots, n$. If the y_i are thought to have a k-shifted lognormal instead of a normal distribution— that is, $\ln(y_i - k) \sim N(\zeta_i, \tau^2)$, and the systematic part $f(\mathbf{x}_i, \boldsymbol{\beta})$ of the original model is still thought appropriate for y_i—the model becomes

$$\ln(y_i - k) = \zeta_i + v_i = \ln\{f(\mathbf{x}_i, \boldsymbol{\beta}) - k\} + v_i, \quad v_i \sim N(0, \tau^2) \tag{4}$$

This model is fit if lnlsq(k) is specified.

If model (4) is correct, the variance of $(y_i - k)$ is proportional to $\{f(\mathbf{x}_i, \boldsymbol{\beta}) - k\}^2$. Probably the most common case is $k = 0$, sometimes called "proportional errors" because the standard error of y_i is proportional to its expectation, $f(\mathbf{x}_i, \boldsymbol{\beta})$. Assuming that the value of k is known, (4) is just another nonlinear model in $\boldsymbol{\beta}$, and it may be fit as usual. However, we may wish to compare the fit of (3) with that of (4) using the residual sum of squares (RSS) or the deviance D, $D = -2 \times$ log-likelihood, from each model. To do so, we must allow for the change in scale introduced by the log transformation.

Assuming, then, the y_i to be normally distributed, Atkinson (1985, 85–87, 184), by considering the Jacobian $\prod |\partial \ln(y_i - k)/\partial y_i|$, showed that multiplying both sides of (4) by the geometric mean of $y_i - k$, $\dot{y}$, gives residuals on the same scale as those of y_i. The geometric mean is given by

$$\dot{y} = e^{n^{-1} \sum \ln(y_i - k)}$$

which is a constant for a given dataset. The residual deviance for (3) and for (4) may be expressed as

$$D(\widehat{\boldsymbol{\beta}}) = \left\{1 + \ln(2\pi\widehat{\sigma}^2)\right\}n \tag{5}$$

where $\widehat{\boldsymbol{\beta}}$ is the maximum likelihood estimate (MLE) of $\boldsymbol{\beta}$ for each model and $n\widehat{\sigma}^2$ is the RSS from (3), or that from (4) multiplied by $\dot{y}^2$.

Because (3) and (4) are models with different error structures but the same functional form, the arithmetic difference in their RSS or deviances is not easily tested for statistical significance. However, if the deviance difference is large (>4, say), we would naturally prefer the model with the smaller deviance. Of course, the residuals for each model should be examined for departures from assumptions (nonconstant variance, nonnormality, serial correlations, etc.) in the usual way.

Alternatively, consider modeling

$$E(y_i) = 1/(C + Ae^{Bx_i}) \tag{6}$$

$$E(1/y_i) = E(y_i') = C + Ae^{Bx_i} \tag{7}$$

where C, A, and B are parameters to be estimated. Using the data $(y, x) = (0.04, 5)$, $(0.06, 12)$, $(0.08, 25)$, $(0.1, 35)$, $(0.15, 42)$, $(0.2, 48)$, $(0.25, 60)$, $(0.3, 75)$, and $(0.5, 120)$ (Danuso 1991), fitting the models yields

Model	C	A	B	RSS	Deviance
(6)	1.781	25.74	-0.03926	-0.001640	-51.95
(6) with `lnlsq(0)`	1.799	25.45	-0.04051	-0.001431	-53.18
(7)	1.781	25.74	-0.03926	8.197	24.70
(7) with `lnlsq(0)`	1.799	27.45	-0.04051	3.651	17.42

There is little to choose between the two versions of the logistic model (6), whereas for the exponential model (7), the fit using `lnlsq(0)` is much better (a deviance difference of 7.28). The reciprocal transformation has introduced heteroskedasticity into y_i', which is countered by the proportional errors property of the lognormal distribution implicit in `lnlsq(0)`. The deviances are not comparable between the logistic and exponential models because the change of scale has not been allowed for, although in principle it could be.

Other uses

Even if you are fitting linear regression models, you may find that `nl` can save you some typing. Because you specify the parameters of your model explicitly, you can impose constraints on them directly.

▷ Example 2

In example 2 of [R] **cnsreg**, we showed how to fit the model

mpg $= \beta_0 + \beta_1$price $+ \beta_2$weight $+ \beta_3$displ $+ \beta_4$gear_ratio $+ \beta_5$foreign $+$
    $\beta_6$length $+ u$

subject to the constraints

$$\beta_1 = \beta_2 = \beta_3 = \beta_6$$
$$\beta_4 = -\beta_5 = \beta_0/20$$

An alternative way is to use `nl`:

```
. use https://www.stata-press.com/data/r18/auto, clear
(1978 automobile data)
. nl (mpg = {b0} + {b1}*price + {b1}*weight + {b1}*displ +
> {b0}/20*gear_ratio - {b0}/20*foreign + {b1}*length)
Iteration 0:  Residual SS = 1578.522
Iteration 1:  Residual SS = 1578.522
```

Source	SS	df	MS		
				Number of obs =	74
Model	34429.478	2	17214.7389	R-squared =	0.9562
Residual	1578.5223	72	21.9239203	Adj R-squared =	0.9549
				Root MSE =	4.682299
Total	36008	74	486.594595	Res. dev. =	436.4562

mpg	Coefficient	Std. err.	t	P>|t|	[95% conf. interval]	
/b0	26.52229	1.375178	19.29	0.000	23.78092	29.26365
/b1	-.000923	.0001534	-6.02	0.000	-.0012288	-.0006172

The point estimates and standard errors for β_0 and β_1 are identical to those reported in example 2 of [R] **cnsreg**. To get the estimate for β_4, we can use `nlcom`:

```
. nlcom _b[/b0]/20
      _nl_1: _b[/b0]/20
```

mpg	Coefficient	Std. err.	z	P>|z|	[95% conf. interval]	
_nl_1	1.326114	.0687589	19.29	0.000	1.191349	1.460879

The advantage to using `nl` is that we do not need to use the `constraint` command six times.

◁

`nl` is also a useful tool when doing exploratory data analysis. For example, you may want to run a regression of y on a function of x, though you have not decided whether to use $\mathrm{sqrt}(x)$ or $\ln(x)$. You can use `nl` to run both regressions without having first to generate two new variables:

```
. nl (y = {b0} + {b1}*ln(x))
. nl (y = {b0} + {b1}*sqrt(x))
```

Poi (2008) shows the advantages of using `nl` when marginal effects of transformed variables are desired as well.

Weights

Weights are specified in the usual way—analytic and frequency weights as well as `iweight`s are supported; see [U] **20.24 Weighted estimation**. Use of analytic weights implies that the y_i have different variances. Therefore, model (3) may be rewritten as

$$y_i = f(\mathbf{x}_i, \boldsymbol{\beta}) + u_i, \qquad u_i \sim N(0, \sigma^2/w_i) \tag{3a}$$

where w_i are (positive) weights, assumed to be known and normalized such that their sum equals the number of observations. The residual deviance for (3a) is

$$D(\widehat{\boldsymbol{\beta}}) = \left\{1 + \ln(2\pi\widehat{\sigma}^2)\right\}n - \sum \ln(w_i) \tag{5a}$$

[compare with (5)], where

$$n\widehat{\sigma}^2 = \text{RSS} = \sum w_i \big\{ y_i - f(\mathbf{x}_i, \widehat{\boldsymbol{\beta}}) \big\}^2$$

Defining and fitting a model equivalent to (4) when weights have been specified as in (3a) is not straightforward and has not been attempted. Thus, deviances using and not using the `lnlsq()` option may not be strictly comparable when analytic weights (other than 0 and 1) are used.

You do not need to modify your substitutable expression in any way to use weights. If, however, you write a substitutable expression program, then you should account for weights when obtaining initial values. When `nl` calls your program, it passes whatever weight expression (if any) was specified by the user. Here is an outline of a substitutable expression program that accepts weights:

```
program nl name, rclass
        version 18.0
        syntax varlist [aw fw iw] if
        ...
        // Obtain initial values allowing weights
        // Use the syntax ['weight''exp'].  For example,
        summarize varname ['weight''exp'] 'if'
        regress depvar varlist ['weight''exp'] 'if'
        ...
        // Return substitutable expression
        return local eq "substitutable expression"
        return local title "description of estimator"
end
```

For details on how the `syntax` command processes weight expressions, see [P] **syntax**.

Potential errors

`nl` is reasonably robust to the inability of your nonlinear function to be evaluated at some parameter values. `nl` does assume that your function can be evaluated at the initial values of the parameters. If your function cannot be evaluated at the initial values, an error message is issued with return code 480. Recall that if you do not specify an initial value for a parameter, then `nl` initializes it to zero. Many nonlinear functions cannot be evaluated when some parameters are zero, so in those cases specifying alternative initial values is crucial.

Thereafter, as `nl` changes the parameter values, it monitors your function for unexpected missing values. If these are detected, `nl` backs up. That is, `nl` finds a point between the previous, known-to-be-good parameter vector and the new, known-to-be-bad vector at which the function can be evaluated and continues its iterations from that point.

`nl` requires that once a parameter vector is found where the predictions can be calculated, small changes to the parameter vector be made to calculate numeric derivatives. If a boundary is encountered at this point, an error message is issued with return code 481.

When specifying `lnlsq()`, an attempt to take logarithms of $y_i - k$ when $y_i \leq k$ results in an error message with return code 482.

If `iterate()` iterations are performed and estimates still have not converged, results are presented with a warning, and the return code is set to 430.

If you use the programmed substitutable expression version of `nl` with a function evaluator program, or vice versa, Stata issues an error message. Verify that you are using the syntax appropriate for the program you have.

General comments on fitting nonlinear models

Achieving convergence is often problematic. For example, a unique minimum of the sum-of-squares function may not exist. Much literature exists on different algorithms that have been used, on strategies for obtaining good initial parameter values, and on tricks for parameterizing the model to make its behavior as linear-like as possible. Selected references are Kennedy and Gentle (1980, chap. 10) for computational matters and Ross (1990) and Ratkowsky (1983) for all three aspects. Ratkowsky's book is particularly clear and approachable, with useful discussion on the meaning and practical implications of intrinsic and parameter-effects nonlinearity. An excellent text on nonlinear estimation is Gallant (1987). Also see Davidson and MacKinnon (1993 and 2004).

To enhance the success of `nl`, pay attention to the form of the model fit, along the lines of Ratkowsky and Ross. For example, Ratkowsky (1983, 49–59) analyzes three possible three-parameter yield-density models for plant growth:

$$E(y_i) = \begin{cases} (\alpha + \beta x_i)^{-1/\theta} \\ (\alpha + \beta x_i + \gamma x_i^2)^{-1} \\ (\alpha + \beta x_i^\phi)^{-1} \end{cases}$$

All three models give similar fits. However, he shows that the second formulation is dramatically more linear-like than the other two and therefore has better convergence properties. In addition, the parameter estimates are virtually unbiased and normally distributed, and the asymptotic approximation to the standard errors, correlations, and confidence intervals is much more accurate than for the other models. Even within a given model, the way the parameters are expressed (for example, ϕ^{x_i} or $e^{\theta x_i}$) affects the degree of linearity and convergence behavior.

Function evaluator programs

Occasionally, a nonlinear function may be so complex that writing a substitutable expression for it is impractical. For example, there could be many parameters in the model. Alternatively, if you are implementing a two-step estimator, writing a substitutable expression may be altogether impossible. Function evaluator programs can be used in these situations.

`nl` will pass to your function evaluator program a list of variables, a weight expression, a variable marking the estimation sample, and a vector of parameters. Your program is to replace the dependent variable, which is the first variable in the variables list, with the values of the nonlinear function evaluated at those parameters. As with substitutable expression programs, the first two letters of the name must be `nl`.

To focus on the mechanics of the function evaluator program, again let's compare the CES production function to the previous examples. The function evaluator program is

```
program nlces2
        version 18.0
        syntax varlist(min=3 max=3) if, at(name)
        local logout : word 1 of `varlist'
        local capital : word 2 of `varlist'
        local labor : word 3 of `varlist'
        // Retrieve parameters out of at matrix
        tempname b0 rho delta
        scalar `b0' = `at'[1, 1]
        scalar `rho' = `at'[1, 2]
        scalar `delta' = `at'[1, 3]
        tempvar kterm lterm
        generate double `kterm' = `delta'*`capital'^(-1*`rho') `if'
        generate double `lterm' = (1-`delta')*`labor'^(-1*`rho') `if'
        // Fill in dependent variable
        replace `logout' = `b0' - 1/`rho'*ln(`kterm' + `lterm') `if'
end
```

Unlike the previous `nlces` program, this one is not declared to be r-class. The `syntax` statement again accepts three variables: one for log output, one for capital, and one for labor. An `if` *exp* is again required because `nl` will pass a binary variable marking the estimation sample. All function evaluator programs must accept an option named `at()` that takes a `name` as an argument—that is how `nl` passes the parameter vector to your program.

The next part of the program retrieves the output, labor, and capital variables from the variables list. It then breaks up the temporary matrix `at` and retrieves the parameters `b0`, `rho`, and `delta`. Pay careful attention to the order in which the parameters refer to the columns of the `at` matrix because that will affect the syntax you use with `nl`. The temporary names you use inside this program are immaterial, however.

The rest of the program computes the nonlinear function, using some temporary variables to hold intermediate results. The final line of the program then replaces the dependent variable with the values of the function. Notice the use of `if` to restrict attention to the estimation sample. `nl` makes a copy of your dependent variable so that when the command is finished your data are left unchanged.

To use the program and fit your model, you type

```
. use https://www.stata-press.com/data/r18/production, clear
. nl ces2 @ lnoutput capital labor, parameters(b0 rho delta)
> initial(b0 0 rho 1 delta 0.5)
```

The output is again identical to that shown in example 1. The order in which the parameters were specified in the `parameters()` option is the same in which they are retrieved from the `at` matrix in the program. To initialize them, you simply list the parameter name, a space, the initial value, and so on.

If you use the `nparameters()` option instead of the `parameters()` option, the parameters are named b1, b2, ..., bk, where k is the number of parameters. Thus, you could have typed

```
. nl ces2 @ lnoutput capital labor, nparameters(3) initial(b1 0 b2 1 b3 0.5)
```

With that syntax, the parameters called `b0`, `rho`, and `delta` in the program will be labeled `b1`, `b2`, and `b3`, respectively. In programming situations or if there are many parameters, instead of listing the parameter names and initial values in the `initial()` option, you may find it more convenient to pass a column vector. In those cases, you could type

```
. matrix myvals = (0, 1, 0.5)
. nl ces2 @ lnoutput capital labor, nparameters(3) initial(myvals)
```

In summary, a function evaluator program receives a list of variables, the first of which is the dependent variable that you are to replace with the values of your nonlinear function. Additionally, it must accept an if *exp*, as well as an option named `at` that will contain the vector of parameters at which `nl` wants the function evaluated. You are then free to do whatever is necessary to evaluate your function and replace the dependent variable.

If you wish to use weights, your function evaluator program's `syntax` statement must accept them. If your program consists only of, for example, `generate` statements, you need not do anything with the weights passed to your program. However, if in calculating the nonlinear function you use commands such as `summarize` or `regress`, then you will want to use the weights with those commands.

As with substitutable expression programs, `nl` will pass to it any options specified that `nl` does not accept, providing you with a way to pass more information to your function.

❑ Technical note

Before version 9 of Stata, the `nl` command used a different syntax, which required you to write an *nlfcn* program, and it did not have a syntax for interactive use other than the seven functions that were built-in. The old syntax of `nl` still works, and you can still use those *nlfcn* programs. If `nl` does not see a colon, an at sign, or a set of parentheses surrounding the equation in your command, it assumes that the old syntax is being used.

The current version of `nl` uses scalars and matrices to store intermediate calculations instead of local and global macros as the old version did, so the current version produces more accurate results. In practice, however, any discrepancies are likely to be small.

❑

Stored results

nl stores the following in e():

Scalars
e(N)	number of observations
e(k)	number of parameters
e(k_eq_model)	number of equations in overall model test; always 0
e(df_m)	model degrees of freedom
e(df_r)	residual degrees of freedom
e(df_t)	total degrees of freedom
e(mss)	model sum of squares
e(rss)	residual sum of squares
e(tss)	total sum of squares
e(mms)	model mean square
e(msr)	residual mean square
e(ll)	log likelihood assuming i.i.d. normal errors
e(r2)	R^2
e(r2_a)	adjusted R^2
e(rmse)	root mean squared error
e(dev)	residual deviance
e(N_clust)	number of clusters
e(lnlsq)	value of lnlsq if specified
e(log_t)	1 if lnlsq specified, 0 otherwise
e(gm_2)	square of geometric mean of $(y-k)$ if lnlsq, 1 otherwise
e(cj)	position of constant in e(b) or 0 if no constant
e(delta)	relative change used to compute derivatives
e(rank)	rank of e(V)
e(ic)	number of iterations
e(converged)	1 if converged, 0 otherwise

Macros
e(cmd)	nl
e(cmdline)	command as typed
e(depvar)	name of dependent variable
e(wtype)	weight type
e(wexp)	weight expression
e(title)	title in estimation output
e(title_2)	secondary title in estimation output
e(clustvar)	name of cluster variable
e(hac_kernel)	HAC kernel
e(hac_lag)	HAC lag
e(vce)	*vcetype* specified in vce()
e(vcetype)	title used to label Std. err.
e(type)	1 = interactively entered expression
	2 = substitutable expression program
	3 = function evaluator program
e(sexp)	substitutable expression
e(params)	names of parameters
e(funcprog)	function evaluator program
e(rhs)	contents of variables()
e(properties)	b V
e(predict)	program used to implement predict
e(marginsok)	predictions allowed by margins
e(marginsnotok)	predictions disallowed by margins

Matrices
e(b)	coefficient vector
e(init)	initial values vector
e(V)	variance–covariance matrix of the estimators
e(V_modelbased)	model-based variance

Functions
e(sample)	marks estimation sample

In addition to the above, the following is stored in `r()`:

Matrices
    `r(table)`   matrix containing the coefficients with their standard errors, test statistics, p-values, and confidence intervals

Note that results stored in `r()` are updated when the command is replayed and will be replaced when any r-class command is run after the estimation command.

Methods and formulas

The derivation here is based on Davidson and MacKinnon (2004, chap. 6). Let $\boldsymbol{\beta}$ denote the $k \times 1$ vector of parameters, and write the regression function using matrix notation as $\mathbf{y} = \mathbf{f}(\mathbf{x}, \boldsymbol{\beta}) + \mathbf{u}$ so that the objective function can be written as

$$\text{SSR}(\boldsymbol{\beta}) = \{\mathbf{y} - \mathbf{f}(\mathbf{x}, \boldsymbol{\beta})\}' \, \mathbf{D} \, \{\mathbf{y} - \mathbf{f}(\mathbf{x}, \boldsymbol{\beta})\}$$

The $\mathbf{D}$ matrix contains the weights and is defined in [R] **regress**; if no weights are specified, then $\mathbf{D}$ is the $N \times N$ identity matrix. Taking a second-order Taylor series expansion centered at $\boldsymbol{\beta}_0$ yields

$$\text{SSR}(\boldsymbol{\beta}) \approx \text{SSR}(\boldsymbol{\beta}_0) + \mathbf{g}'(\boldsymbol{\beta}_0)(\boldsymbol{\beta} - \boldsymbol{\beta}_0) + \frac{1}{2}(\boldsymbol{\beta} - \boldsymbol{\beta}_0)'\mathbf{H}(\boldsymbol{\beta}_0)(\boldsymbol{\beta} - \boldsymbol{\beta}_0) \tag{8}$$

where $\mathbf{g}(\boldsymbol{\beta}_0)$ denotes the $k \times 1$ gradient of $\text{SSR}(\boldsymbol{\beta})$ evaluated at $\boldsymbol{\beta}_0$ and $\mathbf{H}(\boldsymbol{\beta}_0)$ denotes the $k \times k$ Hessian of $\text{SSR}(\boldsymbol{\beta})$ evaluated at $\boldsymbol{\beta}_0$. Letting $\mathbf{X}$ denote the $N \times k$ matrix of derivatives of $\mathbf{f}(\mathbf{x}, \boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$, the gradient $\mathbf{g}(\boldsymbol{\beta})$ is

$$\mathbf{g}(\boldsymbol{\beta}) = -2\mathbf{X}'\mathbf{D}\mathbf{u} \tag{9}$$

$\mathbf{X}$ and $\mathbf{u}$ are obviously functions of $\boldsymbol{\beta}$, though for notational simplicity that dependence is not shown explicitly. The (m, n) element of the Hessian can be written as

$$H_{mn}(\boldsymbol{\beta}) = -2 \sum_{i=1}^{i=N} d_{ii} \left[\frac{\partial^2 f_i}{\partial \beta_m \partial \beta_n} u_i - X_{im} X_{in} \right] \tag{10}$$

where d_{ii} is the ith diagonal element of $\mathbf{D}$. As discussed in Davidson and MacKinnon (2004, chap. 6), the first term inside the brackets of (10) has expectation zero, so the Hessian can be approximated as

$$\mathbf{H}(\boldsymbol{\beta}) = 2\mathbf{X}'\mathbf{D}\mathbf{X} \tag{11}$$

Differentiating the Taylor series expansion of $\text{SSR}(\boldsymbol{\beta})$ shown in (8) yields the first-order condition for a minimum

$$\mathbf{g}(\boldsymbol{\beta}_0) + \mathbf{H}(\boldsymbol{\beta}_0)(\boldsymbol{\beta} - \boldsymbol{\beta}_0) = \mathbf{0}$$

which suggests the iterative procedure

$$\boldsymbol{\beta}_{j+1} = \boldsymbol{\beta}_j - \alpha \mathbf{H}^{-1}(\boldsymbol{\beta}_j)\mathbf{g}(\boldsymbol{\beta}_j) \tag{12}$$

where α is a "step size" parameter chosen at each iteration to improve convergence. Using (9) and (11), we can write (12) as

$$\boldsymbol{\beta}_{j+1} = \boldsymbol{\beta}_j + \alpha(\mathbf{X}'\mathbf{D}\mathbf{X})^{-1}\mathbf{X}'\mathbf{D}\mathbf{u} \tag{13}$$

where $\mathbf{X}$ and $\mathbf{u}$ are evaluated at β_j. Apart from the scalar α, the second term on the right-hand side of (13) can be computed via a (weighted) regression of the columns of $\mathbf{X}$ on the errors. nl computes the derivatives numerically and then calls regress. At each iteration, α is set to one, and a candidate value β_{j+1}^* is computed by (13). If $\text{SSR}(\beta_{j+1}^*) < \text{SSR}(\beta_j)$, then $\beta_{j+1} = \beta_{j+1}^*$ and the iteration is complete. Otherwise, α is halved, a new β_{j+1}^* is calculated, and the process is repeated. Convergence is declared when $\alpha|\beta_{j+1,m}| \leq \epsilon(|\beta_{jm}| + \tau)$ for all $m = 1, \ldots, k$. nl uses $\tau = 10^{-3}$ and, by default, $\epsilon = 10^{-5}$, though you can specify an alternative value of ϵ with the eps() option.

As derived, for example, in Davidson and MacKinnon (2004, chap. 6), an expedient way to obtain the covariance matrix is to compute $\mathbf{u}$ and the columns of $\mathbf{X}$ at the final estimate $\widehat{\beta}$ and then regress that $\mathbf{u}$ on $\mathbf{X}$. The covariance matrix of the estimated parameters of that regression serves as an estimate of $\text{Var}(\widehat{\beta})$. If that regression employs a robust covariance matrix estimator, then the covariance matrix for the parameters of the nonlinear regression will also be robust.

All other statistics are calculated analogously to those in linear regression, except that the nonlinear function $f(\mathbf{x}_i, \beta)$ plays the role of the linear function $\mathbf{x}_i'\beta$. See [R] regress.

This command supports estimation with survey data. For details on VCEs with survey data, see [SVY] Variance estimation.

Acknowledgments

References

Atkinson, A. C. 1985. *Plots, Transformations, and Regression: An Introduction to Graphical Methods of Diagnostic Regression Analysis*. Oxford: Oxford University Press.

Canette, I. 2011. A tip to debug your nl/nlsur function evaluator program. *The Stata Blog: Not Elsewhere Classified*. http://blog.stata.com/2011/12/05/a-tip-to-debug-your-nlnlsur-function-evaluator-program/.

Danuso, F. 1991. sg1: Nonlinear regression command. *Stata Technical Bulletin* 1: 17–19. Reprinted in *Stata Technical Bulletin Reprints*, vol. 1, pp. 96–98. College Station, TX: Stata Press.

Davidson, R., and J. G. MacKinnon. 1993. *Estimation and Inference in Econometrics*. New York: Oxford University Press.

———. 2004. *Econometric Theory and Methods*. New York: Oxford University Press.

Gallant, A. R. 1987. *Nonlinear Statistical Models*. New York: Wiley.

Goldstein, R. 1992. srd7: Adjusted summary statistics for logarithmic regressions. *Stata Technical Bulletin* 5: 17–21. Reprinted in *Stata Technical Bulletin Reprints*, vol. 1, pp. 178–183. College Station, TX: Stata Press.

Kennedy, W. J., Jr., and J. E. Gentle. 1980. *Statistical Computing*. New York: Dekker.

Poi, B. P. 2008. Stata tip 58: nl is not just for nonlinear models. *Stata Journal* 8: 139–141.

Ratkowsky, D. A. 1983. *Nonlinear Regression Modeling: A Unified Practical Approach*. New York: Dekker.

Ross, G. J. S. 1987. *MLP User Manual, Release 3.08*. Oxford: Numerical Algorithms Group.

———. 1990. *Nonlinear Estimation*. New York: Springer.

Also see