Maximum Likelihood Estimation and Nonlinear Least Squares in Stata

Christopher F Baum

Faculty Micro Resource Center
Boston College

July 2007

Maximum likelihood estimation

A key resource is the book *Maximum Likelihood Estimation in Stata*, Gould, Pitblado and Sribney, Stata Press: 3d ed., 2006. A good deal of this presentation is adapted from that excellent treatment of the subject, which I recommend that you buy if you are going to work with MLE in Stata.

To perform maximum likelihood estimation (MLE) in Stata, you must write a short Stata program defining the likelihood function for your problem. In most cases, that program can be quite general and may be applied to a number of different model specifications without the need for modifying the program.

Maximum likelihood estimation

A key resource is the book *Maximum Likelihood Estimation in Stata*, Gould, Pitblado and Sribney, Stata Press: 3d ed., 2006. A good deal of this presentation is adapted from that excellent treatment of the subject, which I recommend that you buy if you are going to work with MLE in Stata.

To perform maximum likelihood estimation (MLE) in Stata, you must write a short Stata program defining the likelihood function for your problem. In most cases, that program can be quite general and may be applied to a number of different model specifications without the need for modifying the program.

Let's consider the simplest use of MLE: a model that estimates a binomial probit equation, as implemented in official Stata by the `probit` command. We code our probit ML program as:

```
program myprobit_lf
  version 10.0
  args lnf xb
  quietly replace `lnf' = ln(normal( `xb' )) ///
      if $ML_y1 == 1
  quietly replace `lnf' = ln(normal( -`xb' )) ///
      if $ML_y1 == 0
end
```

This program is suitable for ML estimation in the **linear form** or `lf` context. The local macro `lnf` contains the contribution to log-likelihood of each observation in the defined sample. As is generally the case with Stata's `generate` and `replace`, it is not necessary to loop over the observations. In the linear form context, the program need not sum up the log-likelihood.

Several programming constructs show up in this example. The `args` statement defines the program's *arguments*: `lnf`, the variable that will contain the value of log-likelihood for each observation, and `xb`, the linear form: a single variable that is the product of the "X matrix" and the current vector **b**. The arguments are local macros within the program.

The program replaces the values of `lnf` with the appropriate log-likelihood values, conditional on the value of `$ML_y1`: the first dependent variable or "y"-variable. Thus, the program may be applied to any 0–1 variable as a function of any set of X variables without modification.

Several programming constructs show up in this example. The `args` statement defines the program's *arguments*: `lnf`, the variable that will contain the value of log-likelihood for each observation, and `xb`, the linear form: a single variable that is the product of the "X matrix" and the current vector **b**. The arguments are local macros within the program.

The program replaces the values of `lnf` with the appropriate log-likelihood values, conditional on the value of `$ML_y1`: the first dependent variable or "y"-variable. Thus, the program may be applied to any 0–1 variable as a function of any set of X variables without modification.

Given the program—stored in the file `myprobit_lf.ado` on the `ADOPATH`—how do we execute it?

```
webuse auto, clear
gen gpm = 1/mpg
ml model lf myprobit_lf ///
    (foreign = price gpm displacement)
ml maximize
```

The `ml model` statement defines the context to be the linear form (`lf`), the likelihood evaluator to be `myprobit_lf`, and then specifies the model. The binary variable `foreign` is to be explained by the factors `price, gpm, displacement`, by default including a constant term in the relationship. The `ml model` command only defines the model: it does not estimate it. That is performed with the `ml maximize` command.

Given the program—stored in the file `myprobit_lf.ado` on the `ADOPATH`—how do we execute it?

```
webuse auto, clear
gen gpm = 1/mpg
ml model lf myprobit_lf ///
    (foreign = price gpm displacement)
ml maximize
```

The `ml model` statement defines the context to be the linear form (`lf`), the likelihood evaluator to be `myprobit_lf`, and then specifies the model. The binary variable `foreign` is to be explained by the factors `price, gpm, displacement`, by default including a constant term in the relationship. The `ml model` command only defines the model: it does not estimate it. That is performed with the `ml maximize` command.

You can verify that this routine duplicates the results of applying `probit` to the same model. Note that our ML program produces estimation results in the same format as an official Stata command. It also stores its results as `ereturn`s, so that postestimation commands—such as `test` and `lincom`—are available.

Of course, we need not “roll our own” binomial probit. To understand how we might apply Stata’s ML commands to a likelihood function of our own, we must establish some notation, and explain what the linear form context implies.

You can verify that this routine duplicates the results of applying `probit` to the same model. Note that our ML program produces estimation results in the same format as an official Stata command. It also stores its results as `ereturn`s, so that postestimation commands—such as `test` and `lincom`—are available.

Of course, we need not "roll our own" binomial probit. To understand how we might apply Stata's ML commands to a likelihood function of our own, we must establish some notation, and explain what the linear form context implies.

The log-likelihood function can be written as a function of variables and parameters:

$$\begin{aligned} \ell &= \ln L\{(\theta_{1j}, \theta_{2j}, \ldots, \theta_{Ej}; y_{1j}, y_{2j}, \ldots, y_{Dj}),\ j = 1, N\} \\ \theta_{ij} &= \mathbf{x}_{ij}\beta_i = \beta_{i0} + x_{ij1}\beta_{i1} + \cdots + x_{ijk}\beta_{ik} \end{aligned}$$

or in terms of the whole sample:

$$\begin{aligned} \ell &= \ln L(\theta_1, \theta_2, \ldots, \theta_E; \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D) \\ \theta_i &= \mathbf{X}_i\beta_i \end{aligned}$$

where we have D dependent variables, E equations (indexed by i) and the data matrix X_i for the i^{th} equation, containing N observations indexed by j.

In the special case where the log-likelihood contribution can be calculated separately for each observation and the sum of those contributions is the overall log-likelihood, the model is said to meet the *linear form restrictions*:

$$\begin{aligned} \ln \ell_j &= \ln \ell(\theta_{1j}, \theta_{2j}, \ldots, \theta_{Ej}; y_{1j}, y_{2j}, \ldots, y_{Dj}) \\ \ell &= \sum_{j=1}^{N} \ln \ell_j \end{aligned}$$

which greatly simplify the task of specifying the model. Nevertheless, when the linear form restrictions are not met, Stata provides three other contexts in which the full likelihood function (and possibly its derivatives) can be specified.

One of the more difficult concepts in Stata's MLE approach is the notion of ML *equations*. In the example above, we only specified a single equation:

```
(foreign = price gpm displacement)
```

which served to identify the dependent variable `$ML_y1` to Stata) and the **X** variables in our binomial probit model.

Let's consider how we can implement estimation of a linear regression model via ML. In regression we seek to estimate not only the coefficient vector **b** but the error variance σ^2. The log-likelihood function for the linear regression model with normally distributed errors is:

$$\ln L = \sum_{j=1}^{N} [\ln \phi\{(y_j - x_j\beta)/\sigma\} - \ln \sigma]$$

with parameters β, σ to be estimated.

One of the more difficult concepts in Stata's MLE approach is the notion of ML *equations*. In the example above, we only specified a single equation:

```
(foreign = price gpm displacement)
```

which served to identify the dependent variable `$ML_y1` to Stata) and the **X** variables in our binomial probit model.

Let's consider how we can implement estimation of a linear regression model via ML. In regression we seek to estimate not only the coefficient vector **b** but the error variance σ^2. The log-likelihood function for the linear regression model with normally distributed errors is:

$$\ln L = \sum_{j=1}^{N} [\ln \phi\{(y_j - x_j\beta)/\sigma\} - \ln \sigma]$$

with parameters β, σ to be estimated.

Writing the conditional mean of y for the j^{th} observation as μ_j,

$$\mu_j = E(y_j) = x_j\beta$$

we can rewrite the log-likelihood function as

$$\begin{aligned}
\theta_{1j} &= \mu_j = x_{1j}\beta_1 \\
\theta_{2j} &= \sigma_j = x_{2j}\beta_2 \\
\ln L &= \sum_{j=1}^{N} [\ln \phi\{(y_j - \theta_{1j})/\theta_{2j}\} - \ln \theta_{2j}]
\end{aligned}$$

This may seem like a lot of unneeded notation, but it makes clear the flexibility of the approach. By defining the linear regression problem as a two-equation ML problem, we may readily specify equations for both β and σ. In OLS regression with homoskedastic errors, we do not need to specify an equation for σ, a constant parameter; but the approach allows us to readily relax that assumption and consider an equation in which σ itself is modeled as varying over the data.

Given a program `mynormal_lf` to evaluate the likelihood of each observation—the individual terms within the summation—we can specify the model to be estimated with

```
ml model lf mynormal_lf  ///
    (y = equation for y) (equation for sigma)
```

This may seem like a lot of unneeded notation, but it makes clear the flexibility of the approach. By defining the linear regression problem as a two-equation ML problem, we may readily specify equations for both β and σ. In OLS regression with homoskedastic errors, we do not need to specify an equation for σ, a constant parameter; but the approach allows us to readily relax that assumption and consider an equation in which σ itself is modeled as varying over the data.

Given a program `mynormal_lf` to evaluate the likelihood of each observation—the individual terms within the summation—we can specify the model to be estimated with

```
ml model lf mynormal_lf  ///
    (y = equation for y) (equation for sigma)
```

In the homoskedastic linear regression case, this might look like

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) ()
```

where the trailing set of `()` merely indicate that nothing but a constant appears in the "equation" for σ. This `ml model` specification indicates that a regression of `mpg` on tt weight and `displacement` is to be fit, by default with a constant term.

We could also use the notation

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) /sigma
```

where there is a constant parameter to be estimated.

In the homoskedastic linear regression case, this might look like

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) ()
```

where the trailing set of `()` merely indicate that nothing but a constant appears in the “equation” for σ. This `ml model` specification indicates that a regression of `mpg` on tt weight and `displacement` is to be fit, by default with a constant term.

We could also use the notation

```
ml model lf mynormal_lf ///
     (mpg = weight displacement) /sigma
```

where there is a constant parameter to be estimated.

But what does the program `mynormal_lf` contain?

```
program mynormal_lf
  version 10.0
  args lnf mu sigma
  quietly replace `lnf' = ///
   ln(normalden( $ML_y1,  `mu', `sigma' ))
end
```

We can use Stata's `normalden(x,m,s)` function in this context. The three-parameter form of this Stata function returns the Normal[`m`, `s`] density associated with `x` divided by `s`. `m`, μ_j in the earlier notation, is the conditional mean, computed as $\mathbf{X}\beta$, while `s`, or σ, is the standard deviation. By specifying an "equation" for σ of `()`, we indicate that a single, constant parameter is to be estimated in that equation.

But what does the program `mynormal_lf` contain?

```
program mynormal_lf
  version 10.0
  args lnf mu sigma
  quietly replace `lnf' = ///
   ln(normalden( $ML_y1,  `mu', `sigma' ))
end
```

We can use Stata's `normalden(x,m,s)` function in this context. The three-parameter form of this Stata function returns the Normal[`m`, `s`] density associated with `x` divided by `s`. `m`, μ_j in the earlier notation, is the conditional mean, computed as $\mathbf{X}\beta$, while `s`, or σ, is the standard deviation. By specifying an "equation" for σ of `()`, we indicate that a single, constant parameter is to be estimated in that equation.

What if we wanted to estimate a heteroskedastic regression model, in which σ_j is considered a linear function of some variable(s)? We can use the same likelihood evaluator, but specify a non-trivial equation for σ:

```
ml model lf mynormal_lf ///
    (mpg = weight displacement) (price)
```

This would model $\sigma_j = \beta_4 \, \textit{price} + \beta_5$. If we wanted σ to be proportional to `price`, we could use

```
ml model lf mynormal_lf ///
    (mu: mpg = weight displacement) ///
    (sigma: price, nocons)
```

which also labels the equations as `mu, sigma` rather than the default `eq1, eq2`.

A better approach to this likelihood evaluator program involves modeling σ in log space, allowing it to take on all values on the real line. The likelihood evaluator program becomes

```
program mynormal_lf2
  version 10.0
  args lnf mu lnsigma
  quietly replace `lnf' = ///
   ln(normalden( $ML_y1,  `mu', exp(`lnsigma' )))
end
```

It may be invoked by

```
ml model lf mynormal_lf2 ///
     (mpg = weight displacement) /lnsigma, ///
     diparm(lnsigma, exp label("sigma"))
```

Where the `diparm( )` option presents the estimate of σ.

A better approach to this likelihood evaluator program involves modeling σ in log space, allowing it to take on all values on the real line. The likelihood evaluator program becomes

```
program mynormal_lf2
  version 10.0
  args lnf mu lnsigma
  quietly replace `lnf' = ///
   ln(normalden( $ML_y1,  `mu', exp(`lnsigma' )))
end
```

It may be invoked by

```
ml model lf mynormal_lf2 ///
     (mpg = weight displacement) /lnsigma, ///
     diparm(lnsigma, exp label("sigma"))
```

Where the `diparm( )` option presents the estimate of σ.

We have illustrated the simplest likelihood evaluator method: the linear form (`lf`) context. It should be used whenever possible, as it is not only easier to code (and less likely to code incorrectly) but more accurate. When it cannot be used—when the linear form restrictions are not met—you may use methods `d0, d1`, or `d2`.

Method `d0`, like `lf`, requires only that you code the log-likelihood function, but in its entirety rather than for a single observation. It is the least accurate and slowest ML method, but the easiest to use when method `lf` is not available.

We have illustrated the simplest likelihood evaluator method: the linear form (`lf`) context. It should be used whenever possible, as it is not only easier to code (and less likely to code incorrectly) but more accurate. When it cannot be used—when the linear form restrictions are not met—you may use methods `d0, d1`, or `d2`.

Method `d0`, like `lf`, requires only that you code the log-likelihood function, but in its entirety rather than for a single observation. It is the least accurate and slowest ML method, but the easiest to use when method `lf` is not available.

Method `d1` requires that you code both the log-likelihood function and the vector of first derivatives, or gradients. It is more difficult than `d0`, as those derivatives must be derived analytically and coded, but is more accurate and faster than `d0` (but less accurate and slower than `lf`.

Method `d2` requires that you code the log-likelihood function, the vector of first derivatives and the matrix of second partial derivatives. It is the most difficult method to use, as those derivatives must be derived analytically and coded, but it is the most accurate and fastest method available. Unless you plan to use a ML program very extensively, you probably do not want to go to the trouble of writing a method `d2` likelihood evaluator.

Method `d1` requires that you code both the log-likelihood function and the vector of first derivatives, or gradients. It is more difficult than `d0`, as those derivatives must be derived analytically and coded, but is more accurate and faster than `d0` (but less accurate and slower than `lf`.

Method `d2` requires that you code the log-likelihood function, the vector of first derivatives and the matrix of second partial derivatives. It is the most difficult method to use, as those derivatives must be derived analytically and coded, but it is the most accurate and fastest method available. Unless you plan to use a ML program very extensively, you probably do not want to go to the trouble of writing a method `d2` likelihood evaluator.

Stata's ML routines provide a variety of optimization methods. The default method is the Newton–Raphson algorithm (`technique(nr)`), which relies on a computed Hessian matrix of second partials to guide its search for the optimum. The method used by Stata is a modified version of Newton–Raphson incorporating improvements by Marquardt.

A related set of methods are the quasi-Newton methods, which avoid explicit computation of the Hessian. A popular choice is the Berndt, Hall, Hall and Hausman algorithm (`technique(bhhh)`). Other quasi-Newton methods are those of Davidon, Fletcher and Powell (`technique(dfp)`) and Broyden, Fletcher, Goldfarb and Shanno (`technique(bfgs)`). The BHHH method is not available with method `d0`, as it relies upon gradients (first derivatives).

Stata's ML routines provide a variety of optimization methods. The default method is the Newton–Raphson algorithm (`technique(nr)`), which relies on a computed Hessian matrix of second partials to guide its search for the optimum. The method used by Stata is a modified version of Newton–Raphson incorporating improvements by Marquardt.

A related set of methods are the quasi-Newton methods, which avoid explicit computation of the Hessian. A popular choice is the Berndt, Hall, Hall and Hausman algorithm (`technique(bhhh)`). Other quasi-Newton methods are those of Davidon, Fletcher and Powell (`technique(dfp)`) and Broyden, Fletcher, Goldfarb and Shanno (`technique(bfgs)`). The BHHH method is not available with method `d0`, as it relies upon gradients (first derivatives).

Many of Stata's standard estimation features are readily available when writing ML programs.

You may estimate over a subsample with the standard `if` *exp* or `in` *range* qualifiers on the `ml model` statement.

The default variance-covariance matrix (`vce(oim)`) estimator is based on the inverse of the estimated Hessian, or information matrix, at convergence. That matrix is available when using the default Newton–Raphson optimization method, which relies upon estimated second derivatives of the log-likelihood function.

Many of Stata's standard estimation features are readily available when writing ML programs.

You may estimate over a subsample with the standard `if` *exp* or `in` *range* qualifiers on the `ml model` statement.

The default variance-covariance matrix (`vce(oim)`) estimator is based on the inverse of the estimated Hessian, or information matrix, at convergence. That matrix is available when using the default Newton–Raphson optimization method, which relies upon estimated second derivatives of the log-likelihood function.

Many of Stata's standard estimation features are readily available when writing ML programs.

You may estimate over a subsample with the standard `if` *exp* or `in` *range* qualifiers on the `ml model` statement.

The default variance-covariance matrix (`vce(oim)`) estimator is based on the inverse of the estimated Hessian, or information matrix, at convergence. That matrix is available when using the default Newton–Raphson optimization method, which relies upon estimated second derivatives of the log-likelihood function.

If any of the quasi-Newton methods are used, you may select the Outer Product of Gradients (`vce(opg)`) estimator of the variance-covariance matrix, which does not rely on a calculated Hessian. This may be especially helpful if you have a lengthy parameter vector. You can specify that the covariance matrix is based on the information matrix (`vce(oim)`) even with the quasi-Newton methods.

The standard heteroskedasticity-robust `vce` estimate is available by selecting the `vce(robust)` option (unless using method `d0`). Likewise, the cluster-robust covariance matrix may be selected, as in standard estimation, with `cluster(`*varname*`)`.

If any of the quasi-Newton methods are used, you may select the Outer Product of Gradients (`vce(opg)`) estimator of the variance-covariance matrix, which does not rely on a calculated Hessian. This may be especially helpful if you have a lengthy parameter vector. You can specify that the covariance matrix is based on the information matrix (`vce(oim)`) even with the quasi-Newton methods.

The standard heteroskedasticity-robust `vce` estimate is available by selecting the `vce(robust)` option (unless using method `d0`). Likewise, the cluster-robust covariance matrix may be selected, as in standard estimation, with `cluster(`*varname*`)`.

You may estimate a model subject to linear constraints using the standard `constraint` command and the `constraints( )` option on the `ml model` command.

You may specify weights on the `ml model` command, using the weights syntax applicable to any estimation command. If you specify `pweights` (probability weights) robust estimates of the variance-covariance matrix are implied.

You may use the `svy` option to indicate that the data have been `svyset`: that is, derived from a complex survey design.

You may estimate a model subject to linear constraints using the standard `constraint` command and the `constraints( )` option on the `ml model` command.

You may specify weights on the `ml model` command, using the weights syntax applicable to any estimation command. If you specify `pweights` (probability weights) robust estimates of the variance-covariance matrix are implied.

You may use the `svy` option to indicate that the data have been `svyset`: that is, derived from a complex survey design.

You may estimate a model subject to linear constraints using the standard `constraint` command and the `constraints( )` option on the `ml model` command.

You may specify weights on the `ml model` command, using the weights syntax applicable to any estimation command. If you specify `pweights` (probability weights) robust estimates of the variance-covariance matrix are implied.

You may use the `svy` option to indicate that the data have been `svyset`: that is, derived from a complex survey design.

A method `lf` likelihood evaluator program will look like:

```
program myprog
   version 10.0
   args lnf theta1 theta2 ...
   tempvar tmp1 tmp2 ...
   qui gen double `tmp1' = ...
   qui replace `lnf' = ...
end
```

`ml` places the name of each dependent variable specified in `ml model` in a global macro: `$ML_y1, $ML_y2`, and so on.

`ml` supplies a variable for each equation specified in `ml model` as `theta1`, `theta2`, etc. Those variables contain linear combinations of the explanatory variables and current coefficients of their respective equations. These variables must not be modified within the program.

`ml` places the name of each dependent variable specified in `ml model` in a global macro: `$ML_y1, $ML_y2`, and so on.

`ml` supplies a variable for each equation specified in `ml model` as `theta1`, `theta2`, etc. Those variables contain linear combinations of the explanatory variables and current coefficients of their respective equations. These variables must not be modified within the program.

If you need to compute any intermediate results within the program, use `tempvar`s, and declare them as `double`. If scalars are needed, define them with a `tempname` statement. Computation of components of the LLF is often convenient when it is a complicated expression.

Final results are saved in `` `lnf' ``: a double-precision variable that will contain the contributions to likelihood of each observation.

The linear form restrictions require that the individual observations in the dataset correspond to independent pieces of the log-likelihood function. They will be met for many ML problems, but are violated for problems involving panel data, fixed-effect logit models, and Cox regression.

If you need to compute any intermediate results within the program, use `tempvar`s, and declare them as `double`. If scalars are needed, define them with a `tempname` statement. Computation of components of the LLF is often convenient when it is a complicated expression.

Final results are saved in `` `lnf' ``: a double-precision variable that will contain the contributions to likelihood of each observation.

The linear form restrictions require that the individual observations in the dataset correspond to independent pieces of the log-likelihood function. They will be met for many ML problems, but are violated for problems involving panel data, fixed-effect logit models, and Cox regression.

If you need to compute any intermediate results within the program, use `tempvar`s, and declare them as `double`. If scalars are needed, define them with a `tempname` statement. Computation of components of the LLF is often convenient when it is a complicated expression.

Final results are saved in `` `lnf' ``: a double-precision variable that will contain the contributions to likelihood of each observation.

The linear form restrictions require that the individual observations in the dataset correspond to independent pieces of the log-likelihood function. They will be met for many ML problems, but are violated for problems involving panel data, fixed-effect logit models, and Cox regression.

Just as linear regression may be applied to many nonlinear models (e.g., the Cobb–Douglas production function), Stata's *linear form restrictions* do not hinder our estimation of a nonlinear model. We merely add equations to define components of the model. If we want to estimate

$$y_j = \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_3 x_{3j}^{\beta_4} + \beta_5 + \epsilon_j$$

with $\epsilon \sim N(0, \sigma^2)$, we can express the log-likelihood as

$$\begin{aligned}
\ln \ell_j &= \ln \phi\{(y_j - \theta_{1j} - \theta_{2j} x_{3j}^{\theta_{3j}})/\theta_{4j}\} - \ln \theta_{4j} \\
\theta_{1j} &= \beta_1 x_{1j} + \beta_2 x_{2j} + \beta_5 \\
\theta_{2j} &= \beta_3 \\
\theta_{3j} &= \beta_4 \\
\theta_{4j} &= \sigma
\end{aligned}$$

The likelihood evaluator for this problem then becomes

```
program mynonlin_lf
  version 10.0
  args lnf theta1 theta2 theta3 sigma
  quietly replace `lnf' = ln(normalden( $ML_y1, ///
       `theta1'+`theta2'*$X3^`theta3', `sigma' ))
end
```

This program evaluates the LLF using a *global macro*, `X3`, which must be defined to identify the Stata variable that is to play the role of x_3.

By making this reference a global macro, we avoid hard-coding the variable name, so that the same model can be fit on different data without altering the program.

We could invoke the program with

```
global X3 bp0
ml model lf mynonlin_lf (bp = age sex) ///
   /beta3 /beta4 /sigma
ml maximize
```

Thus, we may readily handle a nonlinear regression model in this context of the linear form restrictions, redefining the ML problem as one of four equations.

If we need to set starting values, we can do so with the `ml init` command. The `ml check` command is very useful in testing the likelihood evaluator program and ensuring that it is working properly. The `ml search` command is recommended to find appropriate starting values. The `ml query` command can be used to evaluate the progress of ML if there are difficulties achieving convergence.

We could invoke the program with

```
global X3 bp0
ml model lf mynonlin_lf (bp = age sex) ///
   /beta3 /beta4 /sigma
ml maximize
```

Thus, we may readily handle a nonlinear regression model in this context of the linear form restrictions, redefining the ML problem as one of four equations.

If we need to set starting values, we can do so with the `ml init` command. The `ml check` command is very useful in testing the likelihood evaluator program and ensuring that it is working properly. The `ml search` command is recommended to find appropriate starting values. The `ml query` command can be used to evaluate the progress of ML if there are difficulties achieving convergence.

If the `lf` method cannot be used, Stata's ML routines require the use of methods tt d0, d1 or `d2`. In any of those methods, several additional responsibilities are borne by the programmer:

In contrast to `lf`, where the likelihood evaluator's arguments include θ_{ij} expressions incorporating both x_{ij} and β_i, these methods pass the coefficient vector as a single argument. You must generate θ_{ij} expressions with the help of the utility command `mleval`.

If the `lf` method cannot be used, Stata's ML routines require the use of methods tt d0, d1 or `d2`. In any of those methods, several additional responsibilities are borne by the programmer:

In contrast to `lf`, where the likelihood evaluator's arguments include θ_{ij} expressions incorporating both x_{ij} and β_i, these methods pass the coefficient vector as a single argument. You must generate θ_{ij} expressions with the help of the utility command `mleval`.

In the `lf` approach, the likelihood evaluator computes the value of log-likelihood for each observation, storing it in variable `lnf`. In these methods, you must sum the log-likelihood values over observations and store it in a *scalar* `lnf`, an argument to the likelihood evaluator. The utility command `mlsum` may be used.

In the `lf` approach, the program respects the estimation sample and deals with impossible values where the log-likelihood function cannot be calculated. In these methods, you must handle those matters in your likelihood evaluator.

In the `lf` approach, the likelihood evaluator computes the value of log-likelihood for each observation, storing it in variable `lnf`. In these methods, you must sum the log-likelihood values over observations and store it in a *scalar* `lnf`, an argument to the likelihood evaluator. The utility command `mlsum` may be used.

In the `lf` approach, the program respects the estimation sample and deals with impossible values where the log-likelihood function cannot be calculated. In these methods, you must handle those matters in your likelihood evaluator.

A prototype `d0, d1` or `d2` likelihood evaluator:

```
program myprog
   version 10.0
   args todo b lnf g negH
//                 calculate scalar `lnf' = ln L
   if (`todo' == 0 | `lnf' >= . ) exit
//                 for a d1 program, calculate g,
//                 the gradient vector: d ln L / d beta
   if (`todo' == 0 | `lnf' >= . ) exit
//                 for a d2 program, calculate negH,
//                 the negative Hessian:
//                 -d2 ln L / d beta d beta'
end
```

The `todo` argument is 0 if `lnf` must be calculated, 1 if the row vector `g` must also be calculated, and 2 if the matrix `negH` must also be calculated. Thus, for methods `d1` and `d2`, the program may be called with `todo=1`, requesting only calculation of the log-likelihood.

The argument `b` is the current value of the vector of coefficients as a row vector. If you specify

```
ml model d0 myprog (foreign = mpg weight)
```

`b` will contain three values:

$$\theta_j = \beta_1 mpg_j + \beta_2 weight_j + \beta_3$$

The constant term (if included) is always the last coefficient.

If you specify

```
ml model d0 myprog (foreign = mpg weight) ///
      (displ = weightsq) /sigma
```

`b` will contain six values:

$$\begin{aligned}
\theta_{1j} &= \beta_1 mpg_j + \beta_2 weight_j + \beta_3 \\
\theta_{2j} &= \beta_4 weightsq_j + \beta_5 \\
\theta_{3j} &= \beta_6
\end{aligned}$$

To form the combinations of coefficients and data, we use `mleval` in our likelihood evaluator:

```
tempvar theta1 theta2 ...
tempname theta3 ...
mleval `theta1' = `b', eq(1)
mleval `theta2' = `b', eq(2)
mleval `theta3' = `b', eq(3) scalar
```

We specify each `theta` as a `tempvar` (if a variable) or a `tempname` (if a scalar, such as σ). These `theta` macros may then be used in writing the likelihood function, just as in method `lf`.

To generate the total log-likelihood of the sample, we should use `mlsum` in our likelihood evaluator:

```
mlsum `lnf' = ...
```

Use of `mlsum` automatically includes only observations specified in the estimation subsample (those for which `$ML_samp==1`), applies weights if specified, and ensures that there are no missing values of observations' likelihood. If there are missing values, it sets the scalar `lnf` to missing to signal that the current coefficient values in tt b are not feasible.

If the likelihood function meets the linear form restrictions (but we wish to use method `d0, d1` or `d2`), we may use the `mlvecsum` and `mlmatsum` commands in our likelihood evaluator to produce the appropriate formulas for the gradient vector `g` and negative Hessian `negH`.

Stata recommends that the routine is tested as a method `d0` evaluator before making use of the first or first and second derivatives. Methods `d1debug` and `d2debug` assist you in determining whether the derivatives have been coded properly.

If the likelihood function meets the linear form restrictions (but we wish to use method `d0, d1` or `d2`), we may use the `mlvecsum` and `mlmatsum` commands in our likelihood evaluator to produce the appropriate formulas for the gradient vector `g` and negative Hessian `negH`.

Stata recommends that the routine is tested as a method `d0` evaluator before making use of the first or first and second derivatives. Methods `d1debug` and `d2debug` assist you in determining whether the derivatives have been coded properly.

We have presented interactive use of Stata's ML facilities, in which commands `ml model`, `ml search` and `ml maximize` are used to perform estimation. If you are going to use a particular maximum likelihood evaluator extensively, it may be useful to write an ado-file version of the routine, using ML's noninteractive mode.

An ado-file version of a ML routine can implement all the features of any Stata estimation command, and itself becomes a Stata command with a *varlist* and options, as well as the ability to replay results.

We have presented interactive use of Stata's ML facilities, in which commands `ml model`, `ml search` and `ml maximize` are used to perform estimation. If you are going to use a particular maximum likelihood evaluator extensively, it may be useful to write an ado-file version of the routine, using ML's noninteractive mode.

An ado-file version of a ML routine can implement all the features of any Stata estimation command, and itself becomes a Stata command with a *varlist* and options, as well as the ability to replay results.

To produce a new estimation command, you must write two ado-files: `newcmd.ado` and `newcmd_ll.ado`, the likelihood function evaluator. The latter ado-file is unchanged from our previous discussion. The `newcmd.ado` file will characteristically contain references to two programs, internal to the routine: `Replay` and `Estimate`. Thus, the skeleton of the command ado-file will look like:

```
program newcmd
  version 10.0
  if replay()  {
    if ("`e(cmd)'" != "newcmd") error 301
    Replay `0'
 }
 else Estimate `0'
end
```

If `newcmd` is invoked by only its name, it will execute the `Replay` subprogram if the last estimation command was `newcmd`. Otherwise, it wlll execute the `Estimate` subprogram, passing the remainder of the command line in local macro `0` to `Estimate`.

The `Replay` subprogram is quite simple:

```
program Replay
   syntax [, Level(cilevel), other-display-options ]
   ml display, level(`level') other-display-options
end
```

If `newcmd` is invoked by only its name, it will execute the `Replay` subprogram if the last estimation command was `newcmd`. Otherwise, it wlll execute the `Estimate` subprogram, passing the remainder of the command line in local macro `0` to `Estimate`.

The `Replay` subprogram is quite simple:

```
program Replay
   syntax [, Level(cilevel), other-display-options ]
   ml display, level(`level') other-display-options
end
```

The `Estimate` subprogram will contain:

```
program Estimate, eclass sortpreserve
   syntax varlist [if] [in]  [,  vce(passthru) ///
   Level(cilevel) other-estimation-options ///
   other-display-options ]

   marksample touse

   ml model method newcmd_ll  if `touse', ///
   `vce' other-estimation-options  maximize

   ereturn local cmd "newcmd"
   Replay, `level' other-display-options
end
```

The `Estimate` subprogram is declared as `eclass` so that it can return estimation results. The `syntax` statement will handle the use of `if` or `in` clauses, and pass through any other options provided on the command line. The `marksample touse` ensures that only observations satisfying the `if` or `in` clauses are used.

On the `ml model` statement, `method` would be hard-coded as `lf, d0, d1, d2`. The `if 'touse'` clause specifies the proper estimation sample. The `maximize` option is crucial: it is this option that signals to Stata that ML is being used in its noninteractive mode. With `maximize`, the `ml model` statement not only defines the model but triggers its estimation.

The non-interactive estimation does not display its results, so `ereturn` and a call to the `Replay` subprogram are used to produce output.

The `Estimate` subprogram is declared as `eclass` so that it can return estimation results. The `syntax` statement will handle the use of `if` or `in` clauses, and pass through any other options provided on the command line. The `marksample touse` ensures that only observations satisfying the `if` or `in` clauses are used.

On the `ml model` statement, `method` would be hard-coded as `lf, d0, d1, d2`. The `if 'touse'` clause specifies the proper estimation sample. The `maximize` option is crucial: it is this option that signals to Stata that ML is being used in its noninteractive mode. With `maximize`, the `ml model` statement not only defines the model but triggers its estimation.

The non-interactive estimation does not display its results, so `ereturn` and a call to the `Replay` subprogram are used to produce output.

The `Estimate` subprogram is declared as `eclass` so that it can return estimation results. The `syntax` statement will handle the use of `if` or `in` clauses, and pass through any other options provided on the command line. The `marksample touse` ensures that only observations satisfying the `if` or `in` clauses are used.

On the `ml model` statement, `method` would be hard-coded as `lf, d0, d1, d2`. The `` if `touse' `` clause specifies the proper estimation sample. The `maximize` option is crucial: it is this option that signals to Stata that ML is being used in its noninteractive mode. With `maximize`, the `ml model` statement not only defines the model but triggers its estimation.

The non-interactive estimation does not display its results, so `ereturn` and a call to the `Replay` subprogram are used to produce output.

Additional estimation options can readily be added to this skeleton. For instance, cluster-robust estimates can be included by adding a `CLuster(varname)` option to the syntax statement, with appropriate modifications to `ml model`. In the linear regression example, we could provide a set of variables to model heteroskedasticity in a `HEtero(varlist)` option, which would then define the second equation to be estimated by `ml model`.

The `mlopts` utility command can be used to parse options provided on the command line and pass any related to the ML estimation process to `ml model`. For instance, a `HEtero(varlist)` option is to be handled by the program, while an `iterate(#)` option should be passed to the optimizer.

Additional estimation options can readily be added to this skeleton. For instance, cluster-robust estimates can be included by adding a `CLuster(varname)` option to the syntax statement, with appropriate modifications to `ml model`. In the linear regression example, we could provide a set of variables to model heteroskedasticity in a `HEtero(varlist)` option, which would then define the second equation to be estimated by `ml model`.

The `mlopts` utility command can be used to parse options provided on the command line and pass any related to the ML estimation process to `ml model`. For instance, a `HEtero(varlist)` option is to be handled by the program, while an `iterate(#)` option should be passed to the optimizer.

Nonlinear least squares estimation

Besides the capabilities for maximum likelihood estimation of one or several equations via the `ml` suite of commands, Stata provides facilities for single-equation nonlinear least squares estimation with `nl` and the estimation of nonlinear systems of equations with `nlsur`.

The `nl` and `nlsur` commands may be invoked in three ways: interactively, using a "programmed substitutable expression", and using a "function evaluator program". We discuss the first and third methods here. The function evaluator program is quite similar to the likelihood function evaluators we have discussed.

Nonlinear least squares estimation

Besides the capabilities for maximum likelihood estimation of one or several equations via the `ml` suite of commands, Stata provides facilities for single-equation nonlinear least squares estimation with `nl` and the estimation of nonlinear systems of equations with `nlsur`.

The `nl` and `nlsur` commands may be invoked in three ways: interactively, using a "programmed substitutable expression", and using a "function evaluator program". We discuss the first and third methods here. The function evaluator program is quite similar to the likelihood function evaluators we have discussed.

In the interactive mode, you specify the nonlinear least squares expression, including starting values if necessary, on the command line. For example, consider the two-factor CES production function:

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

with the parameters β_0, ρ, δ.

This could be estimated with:

```
nl (lnQ={b0}-1/{rho=1}*ln({delta=0.5}*K^(-1*{rho}) +
    (1-{delta}*L^(-1*{rho}))))
```

Note that the parameters are enclosed in { }, with initial values given if needed. The entire equation must be enclosed by ( ). You may use options such as `robust` and `cluster(`*varname*`)` with `nl`.

In the interactive mode, you specify the nonlinear least squares expression, including starting values if necessary, on the command line. For example, consider the two-factor CES production function:

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

with the parameters β_0, ρ, δ.

This could be estimated with:

```
nl (lnQ={b0}-1/{rho=1}*ln({delta=0.5}*K^(-1*{rho}) +
    (1-{delta}*L^(-1*{rho}))))
```

Note that the parameters are enclosed in { }, with initial values given if needed. The entire equation must be enclosed by ( ). You may use options such as `robust` and `cluster(`*varname*`)` with `nl`.

The standard apparatus for any estimation command is available after invoking tt nl. For instance, we might want to calculate the elasticity of substitution for the CES function, defined as $\sigma = 1/(1+\rho)$. The `nlcom` command can provide point and interval estimates of this expression via the delta method:

```
nlcom (sigma: 1 / ( 1 + [rho]_b[_cons] ))
```

where we refer to the "constant" in the `rho` equation, and label the resulting expression `sigma` in the output.

After `nl`, all of the standard results from any estimation command are available for further use; `ereturn list` for details.

The standard apparatus for any estimation command is available after invoking tt nl. For instance, we might want to calculate the elasticity of substitution for the CES function, defined as $\sigma = 1/(1+\rho)$. The `nlcom` command can provide point and interval estimates of this expression via the delta method:

```
nlcom (sigma: 1 / ( 1 + [rho]_b[_cons] ))
```

where we refer to the "constant" in the `rho` equation, and label the resulting expression `sigma` in the output.

After `nl`, all of the standard results from any estimation command are available for further use; `ereturn list` for details.

If you want to use `nl` extensively for a particular problem, it makes sense to develop a *function evaluator program*. That program is quite similar to any Stata `ado`-file or `ml` program. It must be named `nl`*func*`.ado`, where *func* is a name of your choice: e.g., `nlces.ado` for a CES function evaluator.

The stylized function evaluator program contains:

```
program nlfunc
   version 10.0
   syntax varlist(min=n max=n) if, at(name)
 // extract vars from varlist
 // extract params as scalars from at matrix
 // fill in dependent variable with replace
end
```

If you want to use `nl` extensively for a particular problem, it makes sense to develop a *function evaluator program*. That program is quite similar to any Stata `ado`-file or `ml` program. It must be named `nl`*func*`.ado`, where *func* is a name of your choice: e.g., `nlces.ado` for a CES function evaluator.

The stylized function evaluator program contains:

```
program nlfunc
   version 10.0
   syntax varlist(min=n max=n) if, at(name)
 // extract vars from varlist
 // extract params as scalars from at matrix
 // fill in dependent variable with replace
end
```

The *varlist* contains all Stata variables referenced in the program, both dependent and independent. Their names are passed to the program, so that different variables may be used without modifying the program.

The *if* argument should be used, as a local macro, on all `generate` or `replace` statements within the program to restrict analysis to the estimation sample.

The *at(name)* contains the name of a Stata matrix—a row vector—containing the current values of the parameters. The parameters must be extracted from this matrix when referenced in the program.

The *varlist* contains all Stata variables referenced in the program, both dependent and independent. Their names are passed to the program, so that different variables may be used without modifying the program.

The *if* argument should be used, as a local macro, on all `generate` or `replace` statements within the program to restrict analysis to the estimation sample.

The *at(name)* contains the name of a Stata matrix—a row vector—containing the current values of the parameters. The parameters must be extracted from this matrix when referenced in the program.

The *varlist* contains all Stata variables referenced in the program, both dependent and independent. Their names are passed to the program, so that different variables may be used without modifying the program.

The *if* argument should be used, as a local macro, on all `generate` or `replace` statements within the program to restrict analysis to the estimation sample.

The *at(name)* contains the name of a Stata matrix—a row vector—containing the current values of the parameters. The parameters must be extracted from this matrix when referenced in the program.

To retrieve the Stata variables from the *varlist*, use statements such as:

```
local logoutput : word 1 of `varlist'
```

and reference `logoutput` as a local macro within the program.

To retrieve the parameters from the matrix specified in *at(name)*, use:

```
tempname b0 rho delta
scalar `b0' = `at'[1,1]
scalar `rho' = `at'[1,2]
scalar `delta' = `at'[1,3]
```

and reference the parameters as local macros within the program.

To retrieve the Stata variables from the *varlist*, use statements such as:

```
local logoutput : word 1 of `varlist'
```

and reference `logoutput` as a local macro within the program.

To retrieve the parameters from the matrix specified in *at(name)*, use:

```
tempname b0 rho delta
scalar `b0' = `at'[1,1]
scalar `rho' = `at'[1,2]
scalar `delta' = `at'[1,3]
```

and reference the parameters as local macros within the program.

If any additional variables are needed, define them as `tempvar`s of type `double` and respect the *if* condition:

```
tempvar kterm lterm
gen double `kterm' = `delta'*`K'^(-1*`rho)  `if'
gen double `lterm' = (1-`delta')*`L'^(-1*`rho)  `if'
replace `logoutput' = `b0'-1/`rho'* ///
    ln(`kterm' + `lterm')  `if'
```

To use the program `nlces`, call it with the `nl` command, but only include the unique part of its name, followed by `@`:

```
nl ces @ lnQ cap lab, parameters(b0 rho delta) ///
 initial(b0 0 rho 1 delta 0.5)
```

You could restrict analysis to a subsample with the *if exp* qualifier:

```
nl ces @ lnQ cap lab if industry==33, ...
```

To use the program `nlces`, call it with the `nl` command, but only include the unique part of its name, followed by `@`:

```
nl ces @ lnQ cap lab, parameters(b0 rho delta) ///
 initial(b0 0 rho 1 delta 0.5)
```

You could restrict analysis to a subsample with the *if exp* qualifier:

```
nl ces @ lnQ cap lab if industry==33, ...
```

Two alternative syntaxes for invoking the program exist:

```
nl ces @ lnQ cap lab, nparameters(3) ///
 initial(b0 0 rho 1 delta 0.5)
```

In this case, you need not spell out the parameter names in the `parameters` option, but merely indicate how many there are in `nparameters`.

You can also pass the parameter values, in the order that they appear in the function evaluator program, in a row vector:

```
matrix startv = (0, 1, 0.5)
nl ces @ lnQ cap lab, nparameters(3) initial(startv)
```

Two alternative syntaxes for invoking the program exist:

```
nl ces @ lnQ cap lab, nparameters(3) ///
 initial(b0 0 rho 1 delta 0.5)
```

In this case, you need not spell out the parameter names in the `parameters` option, but merely indicate how many there are in `nparameters`.

You can also pass the parameter values, in the order that they appear in the function evaluator program, in a row vector:

```
matrix startv = (0, 1, 0.5)
nl ces @ lnQ cap lab, nparameters(3) initial(startv)
```

The techniques for estimating a system of nonlinear equations (for instance, cost share equations for a transcendental logarithmic cost function) are similar to those described above for single-equation nonlinear least squares. You must write a function evaluator program that generates values for each of the LHS variables, working with the entire parameter vector. For more details see `[R] nlsur`. Note, however, that this command does not implement estimation of a nonlinear system of simultaneous equations.