Constrained Maximum Likelihood MT 2.0

for GAUSS™ Mathematical and Statistical System

Part Number: 007237
Version 2.0
Documentation Revision: 1931 October 11, 2010

Contents

1 Installation

1.1 UNIX/Linux/Mac

If you are unfamiliar with UNIX/Linux/Mac, see your system administrator or system documentation for information on the system commands referred to below.

1.1.1 Download

1. Copy the `.tar.gz` or `.zip` file to `/tmp`.
2. If the file has a `.tar.gz` extension, unzip it using `gunzip`. Otherwise skip to step 3.

   `gunzip app_`*appname*`_`*vernum*`.`*revnum*`_UNIX.tar.gz`
3. `cd` to your **GAUSS** or **GAUSS Engine** installation directory. We are assuming `/usr/local/gauss` in this case.

   ```
   cd /usr/local/gauss
   ```

4. Use `tar` or `unzip`, depending on the file name extension, to extract the file.

   ```
   tar xvf /tmp/app_appname_vernum.revnum_UNIX.tar
   ```

   – or –

   ```
   unzip /tmp/app_appname_vernum.revnum_UNIX.zip
   ```

1.1.2 CD

1. Insert the Apps CD into your machine's CD-ROM drive.
2. Open a terminal window.
3. `cd` to your current **GAUSS** or **GAUSS Engine** installation directory. We are assuming `/usr/local/gauss` in this case.

   ```
   cd /usr/local/gauss
   ```

4. Use `tar` or `unzip`, depending on the file name extensions, to extract the files found on the CD. For example:

   ```
   tar xvf /cdrom/apps/app_appname_vernum.revnum_UNIX.tar
   ```

   – or –

   ```
   unzip /cdrom/apps/app_appname_vernum.revnum_UNIX.zip
   ```

   However, note that the paths may be different on your machine.

1.2 Windows

1.2.1 Download

Unzip the `.zip` file into your **GAUSS** or **GAUSS Engine** installation directory.

1.2.2 CD

1. Insert the Apps CD into your machine's CD-ROM drive.

2. Unzip the `.zip` files found on the CD to your **GAUSS** or **GAUSS Engine** installation directory.

1.2.3 64-Bit Windows

If you have both the 64-bit version of **GAUSS** and the 32-bit Companion Edition installed on your machine, you need to install any **GAUSS** applications you own in both **GAUSS** installation directories.

1.3 Difference Between the UNIX and Windows Versions

- If the functions can be controlled during execution by entering keystrokes from the keyboard, it may be necessary to press ENTER after the keystroke in the UNIX version.

2 Getting Started

GAUSS 10+ and the **GAUSS Run-Time Library** 10+ are required to use these routines for all platforms except Linux, which requires 10.0.4+. See **_rtl_ver** in `src/gauss.dec`.

The **Constrained Maximum Likelihood MT** version number is stored in a global variable:

_comt_ver 3×1 matrix, the first element contains the major version number, the second element the minor version number, and the third element the revision number.

If you call for technical support, you may be asked for the version of your copy of this module.

2.0.1 README Files

If there is a **README.cmlmt** file, it contains any last minute information on the **Constrained Maximum Likelihood MT** procedures. Please read it before using them.

Constrained Maximum Likelihood MT 3

written by

Ronald Schoenberg

This module contains a set of procedures for the solution of the constrained maximum likelihood problem.

3.0.1 Setup

In order to use the procedures in the **Constrained Maximum Likelihood Estimation MT** or **CMLMT** Module, the **CMLMT** library must be active. This is done by including **cmlmt** in the **library** statement at the top of your program or command file:

```
library cmlmt,pgraph;
```

This enables **GAUSS** to find the **CMLMT** procedures. The statement

```
#include cml.sdf
```

is also required. It sets the definitions of the structures used by **CMLMT**.

The version number of each module is stored in a global variable:

_cmlmt_ver 3×1 matrix, the first element contains the major version number of the **CMLMT** Module, the second element the minor version number, and the third element the revision number.

If you call for technical support, you may be asked for the version number of your copy of this module.

3.1 Special Features in Constrained Maximum Likelihood Estimation MT

3.1.1 Structures

In **CMLMT** the same procedure computing the log-likelihood or objective function will be used to compute analytical derivatives as well if they are being provided. Its return argument is a **`cmlmtResults`** structure with three members, a scalar, or Nx1 vector containing the log-likelihood (or objective), a 1XK vector, or NxK matrix of first derivatives, and a KxK matrix or NxKxK array of second derivatives (it needs to be an array if the log-likelihood is weighted). Of course the derivatives are optional, or even partially optional, i.e., you can compute a subset of the derivatives if you like and the remaining will be computed numerically. This procedure will have an additional argument which tells the function which to compute, the log-likelihood or objective, the first

derivatives, or the second derivatives, or all three. This means that calculations in common won't have to be redone.

The new **CMLMT** will use the **DS** and **PV** structures that are now in use in the **GAUSS Run-Time Library**. The **DS** structure is completely flexible, allowing you to pass anything you can think of into your procedure. The **PV** structure revolutionizes how you pass the parameters into the procedure. No more do you have to struggle to get the parameter vector into matrices for calculating the function and its derivatives, trying to remember, or figure out, which parameter is where in the vector. If your log-likelihood uses matrices or arrays, you can store them directly into the **PV** structure, and remove them as matrices or arrays with the parameters already plugged into them. The PV structure can handle matrices and arrays where some of their elements are fixed and some free. It remembers the fixed parameters and knows where to plug in the current values of the free parameters. It can handle symmetric matrices where parameters below the diagonal are repeated above the diagonal.

There will no longer be any need to use global variables. Anything the procedure needs can be passed into it through the **DS** structure. And these new applications will use control structures rather than global variables. This means, in addition to thread safety, that it will be straightforward to nest calls to **CMLMT** inside of a call to **CMLMT**, not to mention Run-Time Library functions like **QNewtonmt**, **QProgmt**, and **EQsolvemt**.

3.1.2 Threading

If you have a multi-core processor in your computer, you may take advantage of this capability by selecting threading. This is done by setting the **useThreads** member of the **cmlmtControl** instance:

```
struct cmlmtControl c0;
c0 = cmlmtControlCreate;
c0.useThreads = 1;
```

An important advantage of threading occurs in computing numerical derivatives. If the derivatives are computed numerically, threading will significantly decrease the time of computation.

Resampling in **CMLMTBoot** and **CMLMTBayes** procedures also takes advantage of threading increasing the speed of calculations up to several times.

3.1.3 Augmented Lagrangian Penalty Line Search Method

An augmented Lagrangian penalty method with second order correction described by Conn, Gould, and Toint (2000), Section 15.3.1, is implemented in **CMLMT**.

3.1.4 Hypothesis Testing for Constrained Models

A special procedure is included in **CMLMT** that computes a test statistic and its probability for the hypotheses $H_0 : \psi = 0 \quad \text{against} \quad H_1 : G(\psi) \geq 0, \psi \neq 0$ where $G(\psi)$ is a general function of the parameters and ψ is a subset of the parameters. See Section 3.8.2 for a discussion of a special case where $G(\psi)$ is a linear constraint function. Also see Silvapulle and Sen, 2005, Section 4.6.2, page 177.

3.2 The Log-likelihood Function

CMLMT is a set of procedures for the estimation of the parameters of models via the maximum likelihood method with general constraints on the parameters, along with an additional set of procedures for statistical inference.

CMLMT solves the general weighted maximum likelihood problem

$$L = \sum_{i=1}^{N} \log P(Y_i; \theta)^{w_i}$$

where N is the number of observations, w_i is a weight. $P(Y_i, \theta)$ is the probability of Y_i given θ, a vector of parameters, subject to the linear constraints,

$$A\theta = B$$
$$C\theta \geq D$$

the nonlinear constraints

$$G(\theta) = 0$$
$$H(\theta) \geq 0$$

and bounds

$$\theta_l \leq \theta \leq \theta_u$$

$G(\theta)$ and $H(\theta)$ are functions provided by the user and must be differentiable at least once with respect to θ.

The procedure **CMLMT** finds values for the parameters in θ such that L is maximized. In fact **CMLMT** minimizes $-L$. It is important to note, however, that the user must specify the log-probability to be *maximized*. **CMLMT** transforms the function into the form to be minimized.

CMLMT has been designed to make the specification of the function and the handling of the data convenient. The user supplies a procedure that computes $\log P(Y_i; \theta)$, i.e., the log-likelihood, given the parameters in θ, for either an individual observation or set of observations (i.e., it must return either the log-likelihood for an individual observation or a vector of log-likelihoods for a matrix of observations). **CMLMT** uses this procedure to construct the function to be minimized.

3.3 Algorithm

CMLMT uses the Sequential Quadratic Programming method. In this method the parameters are updated in a series of iterations beginning with starting values that you provide. Let θ_t be the current parameter values. Then the succeeding values are

$$\theta_{t+1} = \theta_t + \rho\delta$$

where δ is a $K \times 1$ *direction* vector, and ρ a scalar *step length*.

Direction

Define

$$\Sigma(\theta) = \frac{\partial^2 L}{\partial\theta\partial\theta'}$$

$$\Psi(\theta) = \frac{\partial L}{\partial \theta}$$

and the Jacobians

$$\dot{G}(\theta) = \frac{\partial G(\theta)}{\partial \theta}$$
$$\dot{H}(\theta) = \frac{\partial H(\theta)}{\partial \theta}$$

For the purposes of this exposition, and without loss of generality, we may assume that the linear constraints and bounds have been incorporated into G and H.

The direction, δ is the solution to the quadratic program

$$\textit{minimize } \frac{1}{2}\delta'\Sigma(\theta_t)\delta + \Psi(\theta_t)\delta$$

$$\begin{aligned} \textit{subject to} \quad & \dot{G}(\theta_t)\delta + G(\theta_t) = 0 \\ & \dot{H}(\theta_t)\delta + H(\theta_t) \geq 0 \end{aligned}$$

This solution requires that Σ be positive semi-definite.

In practice, linear constraints are specified separately from the G and H because their Jacobians are known and easy to compute. And the bounds are more easily handled separately from the linear inequality constraints.

Line Search

Define the merit function

$$m(\theta) = L - \sum_j \kappa_j g_j(\theta) - \sum_\ell \lambda_\ell h_\ell(\theta)) + \frac{1}{2\mu}(\|g_j(\theta)\|_2^2 + \|h_j(\theta)\|_2^2)$$

where g_j is the j-th row of G, h_ℓ is the ℓ-th row of H, κ is the vector of Lagrangean coefficients of the equality constraints, and λ the vector of Lagrangean coefficients of the inequality constraints.

The line search finds a value of ρ that minimizes or decreases $m(\theta_t + \rho\delta)$.

The penalty coefficient μ increases at each iteration. The amount of increase in this coefficient is set by the **Penalty** member of the instance of the **cmlmtControl** structure.

Trust Radius

By default a "trust radius" is set around all of the parameters being estimated. Constraints are set for each parameter that bounds the new direction ensuring the iterations against extreme movements in the estimates. This provides for safer iterations but can add to the total number of iterations to convergence. To turn this off set the **TrustRadius** member of the instance of the **cmlmtControl** structure.

3.3.1 The Secant Algorithms

The Hessian may be very expensive to compute at every iteration, and poor start values may produce an ill-conditioned Hessian. For these reasons alternative algorithms are provided in **CMLMT** for updating the Hessian rather than computing it directly at each iteration. These algorithms, as well as step length methods, may be modified during the execution of **CMLMT**.

Beginning with an initial estimate of the Hessian, or a conformable identity matrix, an update is calculated. The update at each iteration adds more “information” to the estimate of the Hessian, improving its ability to project the direction of the descent. Thus after several iterations the secant algorithm should do nearly as well as Newton iteration with much less computation.

There are two basic types of secant methods, the BFGS (Broyden, Fletcher, Goldfarb, and Shanno), and the DFP (Davidon, Fletcher, and Powell). They are both rank two updates, that is, they are analogous to adding two rows of new data to a previously computed moment matrix. The Cholesky factorization of the estimate of the Hessian is updated using the functions **CHOLUP** and **CHOLDN**.

Secant Methods (BFGS and DFP)

BFGS is the method of Broyden, Fletcher, Goldfarb, and Shanno, and DFP is the method of Davidon, Fletcher, and Powell. These methods are complementary (Luenberger 1984, page 268). BFGS and DFP are like the NEWTON method in that they use both first and second derivative information. However, in DFP and BFGS the Hessian is approximated, reducing considerably the computational requirements. Because they do not explicitly calculate the second derivatives they are sometimes called *quasi-Newton* methods. While it takes more iterations than the NEWTON method, the use of an approximation produces a gain because it can be expected to converge in less overall time (unless analytical second derivatives are available in which case it might be a toss-up).

The secant methods are commonly implemented as updates of the *inverse* of the Hessian. This is not the best method numerically for the BFGS algorithm (Gill and Murray, 1972). This version of **CMLMT**, following Gill and Murray (1972), updates the Cholesky factorization of the Hessian instead, using the functions **CHOLUP** and **CHOLDN** for BFGS. The new direction is then computed using **CHOLSOL**, a Cholesky solve, as applied to the updated Cholesky factorization of the Hessian and the gradient.

3.3.2 Line Search Methods

Given a direction vector d, the updated estimate of the parameters is computed

$$\theta_{t+1} = \theta_t + \rho\delta$$

where ρ is a constant, usually called the *step length*, that increases the descent of the function given the direction. **CMLMT** includes a variety of methods for computing ρ. The value of the function to be minimized as a function of ρ is

$$m(\theta_t + \rho\delta)$$

Given θ and d, this is a function of a single variable ρ. Line search methods attempt to find a value for ρ that decreases m. STEPBT is a polynomial fitting method, BRENT and HALF are iterative search methods. A fourth method called ONE forces a step length of 1. The default line search method is STEPBT. If this, or any selected method, fails, then BRENT is tried. If BRENT fails, then HALF is tried. If all of the line search methods fail, then a random search is tried, provided the **RandRadius** member of the **cmlmtControl** instance is greater than zero which it is by default.

Augmented Penalty Line Search Method

When the *LineSearch* member of the instance of the **cmlmtControl** structure is set to zero, **CMLMT** uses an "augmented Lagrangian penalty" method for the line search described in Conn, Gould, and Toint (2000). The Hessian and gradient for the Quadratic Programming problem in the SQP method is augmented as described in their Section 15.3.1. This method requires that constraints be imposed on the parameters. This method is not available for solving maximum likelihood problems without constraints on parameters.

STEPBT

STEPBT is an implementation of a similarly named algorithm described in Dennis and Schnabel (1983). It first attempts to fit a quadratic function to $m(\theta_t + \rho\delta)$ and computes an ρ that minimizes the quadratic. If that fails, it attempts to fit a cubic function. The cubic function more accurately portrays the F which is not likely to be very quadratic but is, however, more costly to compute. STEPBT is the default line search method because it generally produces the best results for the least cost in computational resources.

BRENT

This method is a variation on the *golden section* method due to Brent (1972). In this method, the function is evaluated at a sequence of test values for ρ. These test values are determined by extrapolation and interpolation using the constant, $(\sqrt{5} - 1)/2 = .6180....$ This constant is the inverse of the so-called "golden ratio" $((\sqrt{5} + 1)/2 = 1.6180...$ and is why the method is called a golden section method. This method is generally more efficient than STEPBT but requires significantly more function evaluations.

HALF

This method first computes $m(x + d)$, i.e., sets $\rho = 1$. If $m(x + d) < m(x)$ then the step length is set to 1. If not, then it tries $m(x + .5d)$. The attempted step length is divided by one half each time the function fails to decrease and exits with the current value when it does decrease. This method usually requires the fewest function evaluations (it often only requires one), but it is the least efficient in that it is not very likely to find the step length that decreases m the most.

BHHHSTEP

This is a variation on the golden search method. A sequence of step lengths are computed, interpolating or extrapolating using a golden ratio, and the method exits when the function decreases.

3.3.3 Weighted Maximum Likelihood

Weights are specified by setting the **Weights** member of the **cmlmtControl** instance to a weighting vector or by assigning it the name of a column in the **GAUSS** data set being used in the estimation.

CMLMT assumes that the weights sum to the number of observations, i.e, that the weights are frequencies. This will be an issue only with statistical inference. Otherwise, any multiple of the weights will produce the same results.

3.3.4 Active and Inactive Parameters

The member **Active** of the instance of the **cmlmtControl** structure may be used to fix parameters to their start values. This allows estimation of different models without having to modify the function procedure. **Active** must be set to a vector of the same length as the vector of start values. Elements of **Active** set to zero will be fixed to their starting values while nonzero elements will be estimated.

This feature may also be used for model testing. *NumObs* times the difference between the function values from the two estimations is chi-squared distributed with degrees of freedom equal to the number of fixed parameters in **Active**.

3.4 Constraints

There are two general types of constraints: nonlinear equality constraints and nonlinear inequality constraints. However, for computational convenience they are divided into five types: linear equality, linear inequality, nonlinear equality, nonlinear inequality, and bounds.

3.4.1 Linear Equality Constraints

Linear constraints are of the form:

$$A\theta = B$$

where A is an $m_1 \times k$ matrix of known constants, and B an $m_1 \times 1$ vector of known constants, and θ the vector of parameters.

The specification of linear equality constraints is done by assigning the A and B matrices to members, **A** and **B**, of an instance of a **cmlmtControl** structure. For example, to constrain the first of four parameters to be equal to the third,

```
struct cmlmtControl ctl;
ctl = cmlmtControlCreate;
ctl.A = { 1 0 -1 0 };
ctl.B = { 0 };
```

3.4.2 Linear Inequality Constraints

Linear constraints are of the form:

$$C\theta \geq D$$

where C is an $m_2 \times k$ matrix of known constants, and D an $m_2 \times 1$ vector of known constants, and θ the vector of parameters.

The specification of linear equality constraints is done by assigning the C and D matrices to members, **C** and **D**, of an instance of a **cmlmtControl** structure. For example, to constrain the first of four parameters to be greater than the third, and as well the second plus the fourth greater than 10:

```
struct cmlmtControl ctl;
ctl = cmlmtControlCreate;
ctl.C = { 1 0 -1 0,
          0 1  0 1 };
ctl.D = {  0,
          10 };
```

3.4.3 Nonlinear Equality

Nonlinear equality constraints are of the form:

$$G(\theta) = 0$$

where θ is the vector of parameters and $G(\theta)$ is an arbitrary, user-supplied function. Nonlinear equality constraints are specified by assigning the procedure pointer to the **EqProc** member of an instance of the **cmlmtControl** structure. This procedure has two input arguments, a **PV** structure containing the parameters, and a **DS** structure containing data.

For example, suppose you wish to constrain the norm of the parameters to be equal to 1:

```
struct cmlmtControl ctl;
ctl = cmlmtControlCreate;
```

```
proc eqp(struct PV p, struct DS d);
    local b;
    b = pvUnpack(p,"coefficients");
    retp(b'b - 1);
endp;

ctl.eqProc = &eqp;
```

3.4.4 Nonlinear Inequality

Nonlinear inequality constraints are of the form:

$$H(\theta) \geq 0$$

where θ is the vector of parameters and $H(\theta)$ is an arbitrary, user-supplied function. Nonlinear equality constraints are specified by assigning the pointer to the **IneqProc** member of an instance of the **cmlmtControl** structure. This procedure has two input arguments, a **PV** structure containing the parameters, and a **DS** structure containing data.

For example, suppose you wish to constrain a covariance matrix to be positive definite

```
proc ineqp(struct PV p, struct DS d);
    local v;
    v = pvUnpack(p,"covariance");
    retp(minc(eigh(v)) - 1e-5);
endp;
ctl.IneqProc = &ineqp;
```

This constrains the minimum eigenvalue of the covariance matrix to be greater than a small number (1e-5). This guarantees the covariance matrix to be positive definite.

3.4.5 Bounds

Bounds are a type of linear inequality constraint. For computational convenience they may be specified separately from the other inequality constraints. To specify bounds, the lower and upper bounds respectively are entered in the first and second columns of a matrix that has the same number of rows as the parameter vector. This matrix is assigned to the **Bounds** member of an instance of a **cmlmtControl** structure.

If the bounds are the same for all of the parameters, only the first row is necessary.

To bound four parameters:

```
struct cmlmtControl ctl;
ctl = cmlmtControlCreate;
ctl.Bounds = { -10 10,
               -10  0,
                 1 10,
                 0  1 };
```

Suppose all of the parameters are to be bounded between -50 and +50, then,

```
ctl.Bounds = { -50 50 };
```

is all that is necessary.

3.5 The CMLMT Procedure

The call to **CMLMT** has four input arguments and one output argument.

3.5.1 First Input Argument: Pointer to Procedure

The first input argument is the pointer to the procedure computing the log-likelihood function and optionally the gradient and/or Hessian. See Section 3.6 for details.

3.5.2 Second Input Argument: PV Parameter Instance

The **GAUSS Run-Time Library** contains special functions that work with the **PV** structure. They are prefixed by "pv" and defined in `pv.src`. These functions store matrices and arrays with parameters in the structure, and retrieve the original matrices and arrays along with various kinds of information about the parameters and parameter vector from it.

The advantage of the **PV** structure is that it permits you to retrieve the parameters in the form of matrices and/or arrays ready for use in calculating your log-likelihood. The matrices and arrays are defined in your command file when the start values are set up. It isn't necessary that a matrix or array be completely free parameters to be estimated. There are **pvPack** functions that take mask arguments defining what is a parameter versus what is a fixed value. There are also functions for handling symmetric matrices where the parameters below the diagonal are duplicated above the diagonal.

For example, a PV structure is created in your command file:

```
struct PV p;
p = pvCreate;    // creates default structure

garch = { .1, .1, .1 };
p = pvPack(p,garch,"garch");
```

A matrix or array in the model may contain a mixture of fixed values along with parameters to be estimated. This type of matrix or array uses **pvPackm** which has an additional argument, called a "mask", strictly conformable to the input matrix or array

indicating which elements are fixed (the corresponding element in the mask is zero) or being estimated (the corresponding element in the mask is nonzero). For example,

```
struct PV p;
p = pvCreate;

b = { 1.0  0.0  0.0,
      0.5  1.0  0.2,
      0.3  0.0  1.0 };

b_mask = { 0  0  0,
           1  0  1,
           1  0  1 };

p = pvPackm(p,b,"beta",b_mask);
```

In this case there are four free parameters to be estimated, b_{21}, b_{23}, b_{31}, and b_{33}. b_{11} and b_{22} are fixed to `1.0`, and b_{22}, b_{23}, and b_{32} are fixed to 0.0.

pvPacks "packs" a symmetric matrix into the **PV** structure in which only the lower left portion of the matrix contains independent parameters while the upper left is duplicated from the lower left. The following packed matrix contains three nonredundant parameters. When this matrix is unpacked, it will contain the upper nonredundant portion of the matrix equal to the lower portion.

```
vc = { 1.2  0.4,
       0.4  2.1  };
p = pvPacks(p,vc,"phi");  // pack symmetric matrix
```

Suppose that you wish to specify a correlation matrix in which only the correlations are free parameters. You would then use **pvPacksm**.

```
cor = { 1.0  0.2,
        0.2  1.0 };
msk = { 0  1,
        1  0 };
pv = pvPacksm(p,cor,msk,"R");
```

Some computation speedup can be achieved by packing and unpacking by number rather than name. Each packing function has a version with an `i` suffix that packs by number. Then **pvUnpack** can be used with that number:

```
garch = { .1, .1, .1 };
p = pvPacki(p,garch,"garch",1);
```

which is unpacked using its number

```
g0 = pvUnpack(g,1);
```

3.5.3 Third Input Argument: DS Data Instance

The DS structure, or "data" structure, is a very simple structure. It contains a member for each **GAUSS** data type. This is its definition (see `ds.sdf` in the **GAUSS** `src` subdirectory):

```
struct DS {
            scalar type;
            matrix dataMatrix;
            array  dataArray;
            string dname;
            string array vnames;
          };
```

Data in Matrices or Arrays

If you are passing your data in as matrices or arrays, you can set the data structure in any way you want, except that the **dname** member of the first element of the data structure must be a null string. **CMLMT** will pass this instance, or a matrix of instances, to your log-likelihood procedure untouched. For example:

```
struct DS d0;
d0 = reshape(dsCreate,2,1);
d0[1].DataMatrix = y;
d0[2].DataMatrix = x;
```

GAUSS Data Sets

You may choose to have **CMLMT** read a **GAUSS** data set and pass selected columns to your log-likelihood procedure. For this set the **Dname** member of an instance of **DS** structure to the name of the **GAUSS** data set:

```
struct DS d0;
d0.Dname = "mydatafile";
d0.Vnames = "price" $| "X1" $| "X2";
```

CMLMT will first determine how many rows of the data set can be read in at a time. Then it reads in the appropriate number of rows (possibly all), selects the appropriate columns (all of them if **Vnames** set to null string), and passes the resulting matrix to your log-likelihood. If only part of the data can read it at a time, your procedure will be called repeatedly and the log-likelihood and derivatives accumulated.

3.5.4 Fourth Input Argument: cmlmtControl Instance

The members of the **cmlmtControl** instance determine everything about the optimization. For example, suppose you want **CMLMT** to stop after 100 iterations:

```
struct cmlmtControl c0;
c0 = cmlmtControlCreate;

c0.maxIters = 100;
```

The **cmlmtControlCreate** procedure sets all of the defaults. The default values for all the members of a **cmlmtControl** instance can be found in that procedure, located at the top of `cmlmtutil.src` in the **GAUSS** `src` subdirectory.

3.6 The Log-likelihood Procedure

CMLMT requires that you write a procedure computing the log-likelihood. The output from this procedure is a **modelResults** structure containing the log-likelihood and optionally the first and second derivatives of the log-likelihood with respect to the parameters. There are three input arguments to this procedure

1. instance of a **PV** structure containing parameter values
2. instance of a **DS** structure containing data
3. indicator vector

and one return argument

1. instance of a **modelResults** structure containing computational results.

3.6.1 First Input Argument: PV Parameter Instance

This argument contains the parameter matrices and arrays that you need for computing the log-likelihood and (optionally) derivatives. The **pvUnpack** function retrieves them from the **PV** instance.

```
proc lpr(struct PV p, struct DS d, ind);
  local beta, gamma;
  beta = pvUnpack(p,"beta");
  gamma = pvUnpack(p,"gamma");
  .
  .
  .
endp;
```

You may have decided to speed the program up a bit by packing the matrices or arrays using the "i" pack functions, **pvPacki**, **pvPackmi**, **pvPacksi**, etc., You can then unpack the matrices and arrays with the integers used in packing them:

```
proc lpr(struct PV p, struct DS d, ind);
  local beta, gamma;
  beta = pvUnpack(p,1);
  gamma = pvUnpack(p,2);
  .
  .
  .
endp;
```

where it has been assumed that they've been packed accordingly:

```
struct PV p;
```

```
p = pvCreate;

p = pvPacki(p,1.|.1,"beta",1);
p = pvPacksi(p,(1~0)|(0~1),"gamma",2);
```

3.6.2 Second Input Argument: DS Data Instance

There are two cases,

1 the **Dname** member of the first element of the **DS** instance is set to the name of a **GAUSS** data set.

2 the **Dname** member of the first element of the instance is set to a null string (default).

Case 1

In case 1, **CMLMT** will pass the observations in the data set to the log-likelihood procedure in the **DataMatrix** member of the first element of the **DS** instance in this argument.

For example, if the **DS** instance is set up this way in the command file:

```
struct DS d;
d = dsCreate;
d.dname = "mydataset";
```

Then in your log-likelihood procedure you can expect N_0 rows of the data set (where $N_0 \leq N$ and N is the total number of rows in the data set) in the **Datamatrix** member of the **DS** instance passed to your procedure from **CMLMT**.

```
proc lpr(struct PV p, struct DS d, ind);
   local y,x;
   .
   .
   .
   y = d.datamatrix[.,1];
   x = d.dataMatrix[.,2:4];
   .
   .
   .
endp;
```

CMLMT will determine whether or not the entire data set can be stored in memory at once. If it can be, then the entire data set will be passed to the procedure. If not, it will pass the data in chunks and generate the log-likelihood and derivatives by accumulation.

Case 2

In Case 2, **CMLMT** passes the **DS** instance you have constructed completely untouched. You can, therefore, design this instance completely for your convenience in computing the log-likelihood and optionally its derivatives.

For example, you can write a general log-likelihood procedure that computes a variety of log-likelihoods, e.g., a probit and a logit. Then you can set the **Type** member of a **DS** instance to a value in your command file that chooses which to compute for that run.

In your command file

```
struct DS d;
d = dsCreate;
d.Type = 1;
d.dataMatrix = z;
```

and in your log-likelihood procedure

```
proc lpr(struct PV p, struct DS d, ind);
    .
    .
    .
    if d.type == 1;  // compute probit log-likelihood
      .
      .
    elseif d.type == 2;  // compute logit
      .
      .
    endif;
   .
   .
   .
endp;
```


3.6.3 Third Input Argument: Indicator Vector

The third argument is a vector with elements set to zero or one, indicating whether or not function, first derivatives, or second derivatives are to be computed.

1st element if nonzero, the function is to be computed.

2nd element if nonzero, the first derivatives are to be computed.

3rd element if nonzero, the second derivatives are to be computed.

The second and third elements associated with the first and second derivatives are optional.

For example,

```
proc logl( struct PV p0, struct DS d0, ind );
    local b0,b,y,x;
    b0 = pvUnpack(p0,"b0");
    b = pvUnpack(p0,"beta");
    y = d0[1].DataMatrix;
    x = d0[2].DataMatrix;

    struct modelResults mm;
    if ind[1];  // compute log-likelihood
       mm.Function = ....
    endif;
    if ind[2];  // compute optional first derivatives
       mm.Gradient = ....
    endif;
    if ind[3];  // compute optional second derivatives
       mm.Hessian = ....
    endif;
    retp(mm);
endp;
```

If **mm.Gradient** and **mm.Hessian** are not set, they will be computed numerically by **CMLMT**.

3.6.4 Output Argument: modelResults Instance

The return argument for your log-likelihood procedure is an instance of a **modelResults** structure. The members of this structure are

1 scalar log-likelihood

Function scalar log-likelihood

Gradient $1 \times$ K vector of first derivatives (optional)

Hessian K × K matrix of second derivatives (optional)

NumObs scalar, number of observations

2 vector of log-likelihoods by observation

Function N × 1 vector of log-likelihoods

Gradient N × K matrix of first derivatives (optional)

Hessian K × K matrix of second derivatives (optional)

3 weighted log-likelihood

Function N × 1 vector of log-likelihoods

Gradient N × K matrix of first derivatives (optional)

Hessian N × K × K array of second derivatives computed by observation (optional)

3.6.5 Examples

```
proc logitLL(struct PV p, struct DS d, ind);
   local mu,const,coefs;
   struct modelResults mm;

   const = pvUnpack(p,"constant");
   coefs = pvUnpack(p,"coefficients");

   mu = const + d[2].DataMatrix * coefs;
   emu = exp(mu');

   if ind[1];
       f = mu - ln(sumc(emu));
       mm.Function = subvec(f,1+d[1].DataMatrix);
   endif;

   if ind[2] or ind[3];
```

```
        w = emu./sumc(emu);
        y = d[1].DataMatrix ~ (1 - d[1].DataMatrix);
        g = sumc((y.*(y - w)');
        mm.Gradient = g~(g.*d[2].DataMatrix);

        if ind[3];
            mm.Hessian = mm.Gradient' * mm.Gradient;
        endif;
    endif;
    retp(mm);
endp;


proc FactorAnalysisLL(struct PV p, struct DS d, ind);
    local lambda,phi,psi,sigma;

    struct modelResults mm;

    lambda = pvUnpack(p,"lambda");
    phi = pvUnpack(p,"phi");
    psi = pvUnpack(p,"psi");

    sigma = lambda * phi * lambda' + psi;

    if ind[1];
        mm.Function = lnpdfmvn(d.DataMatrix,sigma);
    endif;

    retp(mm);

endp;


proc garchLL(struct PV p0, struct DS d0, ind);

    local b0,garch,arch,omega,p,q,h,u,vc,w;
    struct modelResults mm;

    b0 = pvUnpack(p0,"b0");
```

```
    garch = pvUnpack(p0,"garch");
    arch = pvUnpack(p0,"arch");
    omega = pvUnpack(p0,"omega");

    p = rows(garch);
    q = rows(arch);

    u = d0.DataMatrix - b0;
    vc = moment(u,0)/rows(u);
    w = omega + (zeros(q,q) | shiftr((u.*ones(
        1,q))',seqa(q-1,-1,q))) * arch;
    h = recserar(w,vc*ones(p,1),garch);

    mm.Function = -0.5 * ((u.*u)./h + ln(2*pi) + ln(h));
    retp(mm);

endp;
```


3.7 Managing Optimization

The critical elements in optimization are scaling, starting point, and the condition of the model. When the data are scaled, the starting point is reasonably close to the solution, and the data and model go together well, the iterations converge quickly and without difficulty.

For best results, therefore, you want to prepare the problem so that model is well-specified, the data scaled, and that a good starting point is available.

The tradeoff among algorithms and step length methods is between speed and demands on the starting point and condition of the model. The less demanding methods are generally time consuming and computationally intensive, whereas the quicker methods (either in terms of time or number of iterations to convergence) are more sensitive to conditioning and quality of starting point.

3.7.1 Scaling

For best performance, the diagonal elements of the Hessian matrix should be roughly equal. If some diagonal elements contain numbers that are very large and/or very small with respect to the others, **CMLMT** has difficulty converging. How to scale the diagonal elements of the Hessian may not be obvious, but it may suffice to ensure that the constants (or "data") used in the model are about the same magnitude.

3.7.2 Condition

The specification of the model can be measured by the condition of the Hessian. The solution of the problem is found by searching for parameter values for which the gradient is zero. If, however, the Jacobian of the gradient (i.e., the Hessian) is very small for a particular parameter, then **CMLMT** has difficulty determining the optimal values since a large region of the function appears virtually flat to **CMLMT**. When the Hessian has very small elements, the inverse of the Hessian has very large elements and the search direction gets buried in the large numbers.

Poor condition can be caused by bad scaling. It can also be caused by a poor specification of the model or by bad data. Bad models and bad data are two sides of the same coin. If the problem is highly nonlinear, it is important that data be available to describe the features of the curve described by each of the parameters. For example, one of the parameters of the Weibull function describes the shape of the curve as it approaches the upper asymptote. If data are not available on that portion of the curve, then that parameter is poorly estimated. The gradient of the function with respect to that parameter is very flat, elements of the Hessian associated with that parameter is very small, and the inverse of the Hessian contains very large numbers. In this case it is necessary to respecify the model in a way that excludes that parameter.

3.7.3 Starting Point

When the model is not particularly well-defined, the starting point can be critical. When the optimization doesn't seem to be working, try different starting points. A closed form solution may exist for a simpler problem with the same parameters. For example, ordinary least squares estimates may be used for nonlinear least squares problems or nonlinear regressions like probit or logit. There are no general methods for computing start values and it may be necessary to attempt the estimation from a variety of starting points.

3.7.4 Example

The following example illustrates the estimation of a tobit model with nonlinearly inequality constraints, and bounds on the parameters. The nonlinear inequality constraints constrain the first coefficient to be greater than the constant, and the product of the coefficients to be less than one. The bounds are provided essentially to constrain the variance parameter to be greater than zero.

```
library cmlmt;
#include cmlmt.sdf

proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d[1].DataMatrix;
    x = d[2].DataMatrix;

    yh = b0 + x * b;
    res = y - yh;
```

```
    u = y[.,1] ./= 0;

    if ind[1];
        mm.Function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.Gradient = u.*g1 + (1-u).*g2;
    endif;

    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
              -10 10,
              -10 10,
              .1 10 };

proc ineqp(struct PV p, struct DS d);
     local c,b0,b;
```

```
    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    c = zeros(2,1);
    c[1] = b[1] - b0;
    c[2] = 1 - b[1] * b[2];
    retp(c);
endp;

c0.IneqProc = &ineqp;

struct DS d0;
d0 = reshape(dsCreate,2,1);
z = loadd("cmlmttobit");
d0[1].DataMatrix = z[.,1];
d0[2].DataMatrix = z[.,2:4];

struct cmlmtResults out1;
out1 = CMLmtprt(CMLmt(&lpr,p0,d0,c0));

print "nonlinear Lagrangeans";
print out1.lagr.nlinineq;
print;
print "bounds Lagrangeans";
print out1.lagr.bounds;
```

and the output looks like this:

```
============================================================
                        tobit example
============================================================
 CMLMT Version 2.0.0                    3/07/2010   3:41 pm
============================================================

return code =    0
normal convergence

Log-likelihood        -99.8204
```

```
Number of cases     100

Covariance of the parameters computed by the following method:
ML covariance matrix
Parameters   Estimates  Std. err. Est./s.e. Prob.  Gradient
-------------------------------------------------------------
b0[1,1]       0.9690     0.0614    15.790  0.0000   61.1939
b[1,1]        0.9690     0.0614    15.790  0.0000  -61.1943
b[2,1]        0.5186     0.1027     5.051  0.0000    0.0007
b[3,1]        0.3914     0.0876     4.470  0.0000   -0.0005
variance[1,1] 0.5716     0.0871     6.562  0.0000    0.0000

Correlation matrix of the parameters
           1            1 -0.39399796 0.0035754728 -0.043471706
           1            1 -0.39399796 0.0035754728 -0.043471706
 -0.39399796  -0.39399796           1  -0.32467011   0.07251157
 0.003575471  0.003575471 -0.32467012            1   0.03239667
-0.043471725 -0.043471725 0.072511576  0.032396662             1



Wald Confidence Limits

                          0.95 confidence limits
Parameters  Estimates   Lower Limit  Upper Limit   Gradient
-----------------------------------------------------------
b0[1,1]       0.9690       0.8472       1.0908      61.1939
b[1,1]        0.9690       0.8472       1.0908     -61.1943
b[2,1]        0.5186       0.3148       0.7225       0.0007
b[3,1]        0.3914       0.2176       0.5653      -0.0005
variance[1,1] 0.5716       0.3986       0.7445       0.0000

Number of iterations    15
Minutes to convergence     0.00333
nonlinear Lagrangeans

 61.1941
 0.0000
```

```
bounds Lagrangeans

 0.0000   0.0000
 0.0000   0.0000
 0.0000   0.0000
 0.0000   0.0000
 0.0000   0.0000
```

If the Lagrangeans are "empty" matrices, the associated constraints are not active. If they are zeros but not "empty" matrices, then they are still inactive at the solution but were active at some point during the iterations.

3.7.5 Algorithmic Derivatives

Algorithmic Derivatives is a program that can be used to generate a **GAUSS** procedure to compute derivatives of the log-likelihood function. If you have **Algorithmic Derivatives**, be sure to read its manual for details on doing this.

First, copy the procedure computing the log-likelihood to a separate file. Second, from the command line enter

```
ad file_name d_file_name
```

where `file_name` is the name of the file containing the input function procedure, and `d_file_name` is the name of the file containing the output derivative procedure.

If the input function procedure is named `lpr`, the output derivative procedure has the name `d_A_lpr` where the addition to the "`_A_`" indicates that the derivative is with respect to the first of the two arguments.

For example, put the following function into a file called `lpr.fct`

```
proc lpr(c,x,y);

   local b,b0,yh,res,yh,u,logl;

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;
    logl = u.*lnpdfmvn(res,s2) +
        (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    retp(logl);
endp;
```

Then enter the following at the **GAUSS** command line

```
library ad;
ad lpr.fct d_lpr.fct;
```

If successful, the following is printed to the screen

```
java -jar d:\gauss10\src\GaussAD.jar lpr.fct d_lpr.fct
```

and the derivative procedure is written to file named `d_lpr.fct`:

```
/* Version:1.1 - May 15, 2004 */
/* Generated from:lpr.src */

/* Taking derivative with respect to argument 1 */
Proc(1)=d_A_lpr(c, x, y);
    Clearg _AD_fnValue;
        Local b, b0, yh, res, yh, u, logl;
        b0 =  c[(1)] ;
```

```
        b =  c[(2):(4)] ;
        yh = b0 + (x * b);
        res = y - yh;
        u =  y[.,(1)]  ./= 0;
        logl = (u .* lnpdfmvn(res, s2)) + ((1 - u)
            .* ln(cdfnc(yh / sqrt(s2))));
        _AD_fnValue = logl;
        /* retp(_AD_fnValue); */
        /* endp; */
        struct _ADS_optimum _AD_d_c ,_AD_d_b ,_AD_d_b0 ,
            _AD_d_yh ,_AD_d_logl ,_AD_d_res ,
            _AD_d__AD_fnValue;
        /* _AD_d_b = 0; _AD_d_b0 = 0; _AD_d_yh = 0;
            _AD_d_res = 0; _AD_d_res = 0;  */
    _AD_d__AD_fnValue = _ADP_d_x_dx(_AD_fnValue);
    _AD_d_logl = _ADP_DtimesD(_AD_d__AD_fnValue,
        _ADP_d_x_dx(logl));
    _AD_d_yh = _ADP_DtimesD(_AD_d_logl,
        _ADP_DtimesD(_ADP_d_yplusx_dx( u .*
        lnpdfmvn(res, s2), (1 - u) .* ln(cdfnc(yh /
        sqrt(s2)))),
_ADP_DtimesD(_ADP_d_ydotx_dx(1 - u, ln(cdfnc(yh /
    sqrt(s2)))),
_ADP_DtimesD(_ADP_d_ln(cdfnc(yh / sqrt(s2))),
_ADP_DtimesD(_ADP_internal(d_cdfnc(yh / sqrt(s2))),
_ADP_DtimesD(_ADP_d_xdivy_dx(yh, sqrt(s2)),
    _ADP_d_x_dx(yh)))))));
    _AD_d_res = _ADP_DtimesD(_AD_d_logl,
    _ADP_DtimesD(_ADP_d_xplusy_dx(
        u .* lnpdfmvn(res, s2), (1 - u) .*
        ln(cdfnc(yh / sqrt(s2)))),
_ADP_DtimesD(_ADP_d_ydotx_dx(u, lnpdfmvn(res, s2)),
_ADP_DtimesD(_ADP_internal(d_A_lnpdfmvn(res, s2)),
    _ADP_d_x_dx(res)))));
    /* u =  y[.,(1)]  ./= 0; */
    _AD_d_yh = _ADP_DplusD(_ADP_DtimesD(_AD_d_res,
_ADP_DtimesD(_ADP_d_yminusx_dx(y, yh), _ADP_d_x_dx(yh))),
    _AD_d_yh); _AD_d_b = _ADP_DtimesD(_AD_d_yh,
    _ADP_DtimesD(_ADP_d_yplusx_dx(b0,
        x * b) , _ADP_DtimesD(_ADP_d_yx_dx(x, b),
        _ADP_d_x_dx(b))));
    _AD_d_b0 = _ADP_DtimesD(_AD_d_yh, _ADP_DtimesD(
```

```
            _ADP_d_xplusy_dx(b0, x * b) , _ADP_d_x_dx(b0)));
        Local _AD_s_c;
        _AD_s_c = _ADP_seqaMatrix(c);
        _AD_d_c = _ADP_DtimesD(_AD_d_b, _ADP_d_xIdx_dx(c,
            _AD_s_c[(2):(4)] ));
        _AD_s_c = _ADP_seqaMatrix(c);
        _AD_d_c = _ADP_DplusD(_ADP_DtimesD(_AD_d_b0,
            _ADP_d_xIdx_dx(c,
     _AD_s_c[(1)] )), _AD_d_c);
        retp(_ADP_external(_AD_d_c));
    endp;
```

If there's a syntax error in the input function procedure, the following is written to the screen

```
java -jar d:\gauss10\src\GaussAD.jar lpr.fct d_lpr.fct
Command 'java -jar d:\gauss10\src\GaussAD.jar lpr.fct
    d_lpr.fct' exit status 1
```

the `exit status 1` indicating that an error has occurred. The output file then contains the reason for the error:

```
/* Version:1.1 - May 15, 2004 */
/* Generated from:lpr.src */

/* Taking derivative with respect to argument 1 */

proc lpr(c,x,y);

   local b,b0,yh,res,yh,u,logl;

    b0 = c[1];
    b = c[2:4];
    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;
```

```
    logl = u.*lnpdfmvn(res,s2) + (1-u).*(ln(cdfnc(
        yh/sqrt(s2)));
Error: lpr.src:12:64: expecting ')', found ';'
```

Finally, call the above procedure from your log-likelihood procedure, for example,

```
proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d[1].DataMatrix;
    x = d[2].DataMatrix;

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;

    if ind[1];
        mm.Function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        mm.Gradient = d_A_lpr(pvGetParvector(p),y,x);
    endif;

    retp(mm);
endp;
```

3.8 Inference

CMLMT includes four broad classes of methods for analyzing the distributions of the estimated parameters:

- tests of hypotheses for models with constrained parameters
- Taylor Series covariance matrix of the parameters. This includes two types: the inverted Hessian and the heteroskedastic- consistent covariance matrix computed from both the Hessian and the cross-product of the first derivatives.
- Confidence limits computed by inversion of the Wald and likelihood ratio statistics that take into account constraints
- Bootstrap
- Likelihood profile and profile t traces

CMLMT computes a Taylor-series covariance matrix of the parameters that includes the sampling distributions of the Lagrangean coefficients. However, when the model includes inequality constraints, confidence limits computed from the usual t-statistics, i.e., by simply dividing the parameter estimates by their standard errors, are incorrect because they do not account for boundaries placed on the distributions of the parameters by the inequality constraints.

Inference for Models with Constraints on Parameters

The likelihood ratio statistic becomes a mixture of chi-squared distributions in the region of constraint boundaries (Gourieroux et al., 1982). If there are no parameters with limits near constraint boundaries, bootstrapping will suffice. Taylor-series methods assume that it is reasonable to truncate the Taylor-series approximation to the distribution of the

parameters at the second order. If this is not reasonable, bootstrapping is an alternative not requiring this assumption. It is important to note that if the limit of the parameter of interest or any other parameters with which it is correlated more than .6 are near constraint boundaries, then bootstrapping will not produce correct inference (Andrews, 1999).

The hypotheses $H(\theta) = 0$ versus $H(\theta) \geq 0$ can be tested using the **CMLMTChibarSq** procedure. See Section 3.8.2 for details.

The procedure **CMLMTBoot** generates the mean vector and covariance matrix of the bootstrapped parameters. The likelihood profile and profile t traces explicated by Bates and Watts (1988) provide diagnostic material for evaluating parameter distributions. **CMLMTProfile** generates trace plots which are used for this evaluation.

3.8.1 Covariance Matrix of the Parameters

An argument based on a Taylor-series approximation to the likelihood function (e.g., Amemiya, 1985, page 111) shows that

$$\hat{\theta} \rightarrow N(\theta, A^{-1}BA^{-1})$$

where

$$\begin{aligned} A &= E\left[\frac{\partial^2 L}{\partial\theta\partial\theta'}\right] \\ B &= E\left[\left(\frac{\partial L}{\partial\theta}\right)'\left(\frac{\partial L}{\partial\theta}\right)\right] \end{aligned}$$

Estimates of A and B are

$$\begin{aligned}\hat{A} &= \frac{1}{N}\sum_{i}^{N}\frac{\partial^2 L_i}{\partial\theta\partial\theta'} \\ \hat{B} &= \frac{1}{N}\sum_{i}^{N}\left(\frac{\partial L_i}{\partial\theta}\right)'\left(\frac{\partial L_i}{\partial\theta}\right)\end{aligned}$$

Assuming the correct specification of the model plim(A) = plim(B) and thus

$$\hat{\theta} \rightarrow N(\theta, \hat{A}^{-1})$$

Without loss of generality we may consider two types of constraints, the nonlinear equality and the nonlinear inequality constraints (the linear constraints are included in nonlinear, and the bounds are regarded as a type of linear inequality). Furthermore, the inequality constraints may be treated as equality constraints with the introduction of "slack" parameters into the model:

$$H(\theta) \geq 0$$

is changed to

$$H(\theta) = \zeta^2$$

where ζ is a conformable vector of slack parameters.

Further distinguish *active* from *inactive* inequality constraints. Active inequality constraints have nonzero Lagrangeans, γ_j, and zero slack parameters, ζ_j, while the reverse is true for inactive inequality constraints. Keeping this in mind, define the diagonal matrix, Z, containing the slack parameters, ζ_j, for the inactive constraints, and another

diagonal matrix, Γ, containing the Lagrangean coefficients. Also, define $H_{\oplus}(\theta)$ representing the active constraints, and $H_{\ominus}(\theta)$ the inactive.

The likelihood function augmented by constraints is then

$$\begin{aligned} L_A \quad = \quad & L + \lambda_1 g(\theta)_1 + \cdots + \lambda_I g(\theta)^I + \gamma_1 h_{\oplus 1}(\theta) + \cdots + \gamma_J h_{\oplus J}(\theta) + \\ & h_{\ominus 1}(\theta)_i - \zeta_1^2 + \cdots + h_{\ominus K}(\theta) - \zeta_K^2 \end{aligned}$$

and the Hessian of the augmented likelihood is

$$E(\frac{\partial^2 L_A}{\partial\theta\partial\theta'}) = \begin{bmatrix} \Sigma & 0 & 0 & \dot{G}' & \dot{H}'_{\oplus} & \dot{H}'_{\ominus} \\ 0 & 2\Gamma & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2Z \\ \dot{G} & 0 & 0 & 0 & 0 & 0 \\ \dot{H}_{\oplus} & 0 & 0 & 0 & 0 & 0 \\ \dot{H}_{\ominus} & 0 & 2Z & 0 & 0 & 0 \end{bmatrix}$$

where the dot represents the Jacobian with respect to θ, $L = \sum_{i=1}^{N} \log P(Y_i; \theta)$, and $\Sigma = \partial^2 L / \partial\theta\partial\theta'$. The covariance matrix of the parameters, Lagrangeans, and slack parameters is the Moore-Penrose inverse of this matrix. Usually, however, we are interested only in the covariance matrix of the parameters, as well as the covariance matrices of the Lagrange coefficients associated with the active inequality constraints and the equality constraints.

These matrices may be computed without requiring the storage and manipulation of the entire Hessian. Construct the partitioned array

$$\tilde{B} == \begin{bmatrix} \dot{G} \\ \dot{H}_{\oplus} \\ \dot{H}_{\ominus} \end{bmatrix}$$

and denote the i-th row of $\tilde{B}$ as $\tilde{b}_i$. Then the $k \times k$ upper left portion of the inverse, that is, that part associated with the estimated parameters, is calculated recursively. First, compute

$$\Omega_1 = \Sigma^{-1} - \frac{1}{\tilde{b}_1 \Sigma^{-1} \tilde{b}_1'} \Sigma^{-1} \tilde{b}_1' \tilde{b}_1 \Sigma^{-1}$$

then continue to compute for all rows of $\tilde{B}$:

$$\Omega_i = \Omega_{i-1} - \frac{1}{\tilde{b}_i \Omega_{i-1} \tilde{b}_i'} \Omega_{i-1} \tilde{b}_i' \tilde{b}_i \Omega_{i-1}$$

Rows associated with the inactive inequality constraints in $\tilde{B}$, i.e., with $\dot{H}_{\ominus}$, drop out and therefore they need not be considered.

Standard errors for some parameters associated with active inequality constraints may not be available, i.e., the rows and columns of Ω associated with those parameters may be all zeros.

3.8.2 Testing against inequality constraints

Constraints of the form

$$H\theta \geq 0, \tag{1}$$

where H is a matrix of constants, arise in various empirical studies. There is a large literature on statistical inference under such linear inequality constraints, and more generally under nonlinear inequality constraints as well. An up-to-date account of these developments may be found in Silvapulle and Sen (2005). In what follows, we shall provide an introduction to tests against inequality constraints and indicate how **GAUSS** may be used for implementing a simple score test against inequality constraints.

Let ψ denote a $q \times 1$ subvector of θ and λ denote the remaining components of θ. For simplicity, let us write $\theta = \begin{pmatrix} \lambda \\ \psi \end{pmatrix}$ where $\psi = (\psi_1, \ldots, \psi_q)'$ and $\lambda = (\lambda_1, \ldots, \lambda_{p-q})'$. Suppose that we wish to test

$$H_0 : \psi = 0 \quad \text{against} \quad H_1 : R\psi \geq 0, \psi \neq 0 \tag{2}$$

where R is a given matrix of constants; thus, R does not depend on θ and it is nonstochastic.

If our objective were to test $\psi = 0$ against $\psi \neq 0$, then a simple and easy to apply test is the Rao's Score test, or equivalently the Lagrange Multiplier test. This test is also a valid for the inequality constrained testing problem in (2), but it may not be the best because it ignores the inequality constraint $R\psi \geq 0$ in the alternative hypothesis. Various tests of (2), including likelihood ratio and score tests, have been developed. Now, we provide the essential details for testing (2) using a *one-sided score test*.

First, it is convenient to introduce the so called chi-bar square distribution that plays an important role in constrained statistical inference. The asymptotic null distribution of the likelihood ratio/Wald/Score test of $\psi = 0$ against $\psi \neq 0$ is a chi-square. When there are inequality constraints, such as $R\psi \geq 0$, in the null or the alternative hypothesis, the role of the chi-square distribution is replaced by a chi-bar square distribution; this is defined in the next paragraph.

Let $Z \sim N(0, V)$, where Z is a $q \times 1$ random vector and V is a $q \times q$ positive definite matrix. Let

$$\bar{\chi}^2(V, R) = Z'V^{-1}Z - \min_{Ra \geq 0}(Z - a)'V^{-1}(Z - a); \tag{3}$$

in the second term, it is implicit that a is a vector of the same length as Z. We shall use the notation $\bar{\chi}^2(V, R)$ is used for the random variable on the RHS of (3) and also for its distribution. The random variable, $\bar{\chi}^2(V, R)$, is said to have a *chi-bar square distribution*

and it can be expressed as follows:

$$\Pr\{\bar{\chi}^2(V,R) \leq c\} = \sum_{i=0}^{q} w_i \Pr(\chi_i^2 \leq c)$$

for some non-negative numbers, $w_i, i = 0, \ldots q$, that are functions of (q, V, R); these quantities are known as *chi-bar square weights* and also as *level probabilities*. Except in some very spacial cases, $\Pr\{\bar{\chi}^2(V,R) \leq c\}$ is difficult to compute exactly. However, it can be estimated by simulation to a desired degree of precision as follows:

1. Generate Z from $N(0, V)$.
2. Compute $\bar{\chi}^2(V,R)$.
3. Repeat the first two steps M times, say $M = 10000$.
4. Estimate $\Pr\{\bar{\chi}^2(V,R) \leq c\}$ by the proportion of times $\bar{\chi}^2(V,R)$ turned out to be less than or equal to c.

This is the method employed by **GAUSS**; for a similar method for estimating $\{w_i\}$ see Wolak (1987). When the number of repeated samples M is 10000, the standard error of the estimate of the probability obtained by this simulation method does not exceed 0.005; if c is large so that $\Pr\{\bar{\chi}^2(V,R) \leq c\}$ is less than 0.1, then the standard error is less than 0.003. Thus, the precision in the estimation can be controlled by adjusting the number of repeated samples, M.

The asymptotic null distributions of several statistics for testing (2) turns out to be a chi-bar square distribution. Therefore, the **chibarsq()** procedure plays an important role in the implementation of tests against inequality constraints.

3.8.3 One-sided score test

As in (2) let $\psi = (\psi_1, \ldots, \psi_q)'$ denote a $q \times 1$ subvector of θ, λ denote the remaining components of θ, and $\theta = \begin{pmatrix} \lambda \\ \psi \end{pmatrix}$. Suppose that we wish to test

$$H_0 : \psi = 0 \quad \text{against} \quad H_1 : R\psi \geq 0, \psi \neq 0 \tag{4}$$

where R is a given matrix of constants. A generalized version of Rao's Score test can be applied for testing H_0 vs H_1. Let us first introduce the following: Let $L(\theta)$ denote the log-likelihood and

$$s(\theta) = \frac{\partial L(\theta)}{\partial \theta} \quad : \text{score function.} \tag{5}$$

Let $s(\theta)$ be partitioned as follows to conform with (λ, ψ) :

$$\begin{pmatrix} s_\lambda \\ s_\psi \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial \lambda} \\ \frac{\partial L}{\partial \psi} \end{pmatrix}. \tag{6}$$

Similarly, let us introduce the following notation for partitioning any given matrix P of the same order as θ, to conform with the partition, (ψ, λ):

$$P = \begin{pmatrix} P_{\lambda\lambda} & P_{\lambda\psi} \\ P_{\psi\lambda} & P_{\psi\psi} \end{pmatrix} \tag{7}$$

Let $\tilde{\lambda}$ denote the *mle* of λ under $H_0 : \psi = 0$, and let

$$\tilde{\theta} = \begin{pmatrix} \tilde{\lambda} \\ 0 \end{pmatrix}, \tag{8}$$

denote the *mle* of θ under $H_0 : \psi = 0$. Let

$$A(\theta) \quad = \quad -E[n^{-1}\frac{\partial}{\partial\theta'}s(\theta)] = -n^{-1}E[\frac{\partial^2}{\partial\theta'\partial\theta}L(\theta)], \tag{9}$$

$$\tag{10}$$

Let $\tilde{s}, \tilde{A}$, and $\tilde{B}$ denote the corresponding quantities evaluated at $\tilde{\theta}$. These three quantities can be obtained by calling the constrained maximum likelihood procedure under the constraint $H\theta = 0$ where

$$H = \begin{pmatrix} 0 & I \end{pmatrix}$$

and I is the identity matrix of the same order as the dimension of ψ; note that $H\theta = \psi$ and hence $H\theta = 0$ is equivalent to $\psi = 0$.

Now, the *one-sided score statistic* of Silvapulle and Silvapulle (1995) [SS, hereafter], which is a generalized version of Rao's Score statistic, for testing $H_0 : \psi = 0$ against the one-sided alternative $H_1 : R\psi \geq 0, \psi \neq 0$ is

$$T_S = \tilde{u}'\tilde{D}^{-1}\tilde{u} - \min_{Ra\geq 0}(\tilde{u} - a)'\tilde{D}^{-1}(\tilde{u} - a) \tag{11}$$

where

$$\tilde{D} = [(\tilde{A}\tilde{B}^{-1}\tilde{A}')^{-1}]_{\psi\psi}, \tag{12}$$

and

$$\tilde{u} = n^{-1/2}[\tilde{A}_{\psi\psi} - \tilde{A}_{\psi\lambda}\tilde{A}_{\lambda\lambda}^{-1}\tilde{A}_{\lambda\psi}]^{-1}[\tilde{s}_{\psi} - \tilde{A}_{\psi\lambda}(\tilde{A}_{\psi\psi})^{-1}\tilde{s}_{\lambda}]. \tag{13}$$

An attractive feature of this one-sided score test of SS is that it does not require estimation of the model under the inequality constraints in the alternative hypothesis, and further, the test is applicable for methods based on estimating equations such as Generalized Estimating Equations (GEE) of Liang and Zeger (1986).

The asymptotic distribution of T_S under the null hypothesis is $\bar{\chi}^2(D, R)$ where $D = [(AB^{-1}A')^{-1}]_{\psi\psi}$. Therefore, if t_s denotes the sample value of T_S and D does not depend on λ then an approximate large sample p-value is $\Pr\{\bar{\chi}^2(D, R) \geq t_s\}$. Further, if the exact form of D is unknown, then an estimate of the p-value is obtained by substituting an estimate for D.

Usually D depends on λ. In this case, it is customary to define the asymptotic p-value as

$$\sup_{\lambda} \Pr\{\bar{\chi}^2(D_\lambda, R) \geq t_s\}$$

where the suffix λ is used to indicate the D matrix depends on λ. This can be computed approximately by evaluating $\Pr\{\bar{\chi}^2(D_\lambda, R) \geq t_s\}$ over a grid of λ values and finding the maximum over that grid; if the dimension q of λ is large, this may be computing intensive. Alternatively, some authors have suggested to estimate the large sample p-value by

$$\tilde{p} = \Pr\{\bar{\chi}^2(\tilde{D}, R) \geq t_s\} \tag{14}$$

where $\tilde{D}$ is treated as nonstochastic; its suitability would depend on the particular case, and hence should be used with caution.

An upper bound for the large sample p-value is

$$p_u = 0.5[\Pr(\chi^2_{q-1} \geq t_s) + \Pr(\chi^2_q \geq t_s)]$$

where q is the number of components in ψ.

3.8.4 Likelihood ratio test

The likelihood ratio statistic is defined as

$$LRT = 2[\max_{H_1} L(\theta) - \max_{H_0} L(\theta)]. \tag{15}$$

The asymptotic null distribution of LRT is $\bar{\chi}^2(HA^{-1}H', I)$ where I is the identity matrix (see Theorem 4.3.1 in Silvapulle and Sen, 2005). Therefore, an estimate of the p-value, corresponding to (14), for the likelihood ratio test is

$$\Pr\{\bar{\chi}^2(H\tilde{A}^{-1}H', I) \geq LRT).$$

An upper bound for the p-value of LRT is

$$0.5[\Pr(\chi^2_2 \geq LRT) + \Pr(\chi^2_3 \geq LRT)].$$

Example

This example replicates a test of an AR-ARCH model described in Silvapulle and Sen (2005), Section 4.6.6, page 181. The data are observations on the *All Ordinaries Index* of Australian companies. The model is an AR-ARCH with three lagged error terms in the conditional variance equation, and four lagged AR terms in the mean equation. The test we have in mind is whether ARCH effects exist. This test is complicated by the fact that they are constrained to be positive to ensure stationarity of the process as well as positive conditional variances. The null and alternative hypotheses are therefore $H_0 : \Psi = 0$ and $H_1 : \Psi \geq 0$ where Ψ includes the three ARCH parameters.

First, a **CMLMT** estimation is generated where the ARCH parameters are fixed to zero. A **cmlmtControl** instance is created for this estimation where its **Active** member is used to fix the ARCH parameters to their initial values, zero in this case. Additional **CovParType** is set to 2 instructing **CMLMT** to generate the cross-product of the matrix of first derivatives which is required by **chiBarSq**.

Second, another **cmlmtControl** instance is created containing the specification of the constraints on the parameters in the hypothesis. In this case they are bounds constraining the ARCH parameters to be positive.

Third, the **cmlmtResults** instance returned from the call to **CMLMT** along with the **DS** data structure, and the second **cmlmtControl** instance with the specification of the constraints on the parameters, are passed to **chiBarSq** for the calculation of the test statistic and its probability.

Finally, additional constraints that might be placed on ancillary parameters are ignored here. The method for testing hypotheses described here and employed by **chiBarSq** does not allow for constraints on ancillary parameters which is a considerably more complicated test. The additional constraints that could be placed on the AR parameters to ensure stationarity are not active and so may be ignored here. It is important to remember therefore that the test described here only holds for hypotheses where constraints are only placed on the parameters of interest and not the ancillary parameters.

```
library cmlmt;
#include cmlmt.sdf

struct DS d1;
d1 = reshape(dsCreate,2,1);

load z0[] = aoi.asc;
z = packr(lagn(251*ln(trimr(z0,1,0)./
    trimr(z0,0,1)),0|1|2|3|4));
d1[1].dataMatrix = z[.,1];
d1[2].dataMatrix = z[.,2:5];


proc lpr(struct PV p, struct DS d, ind);
    local series,b,omega,arch,const,phi,u2,q,n,h,v;

    struct modelResults mm;

    omega = pvUnpack(p,"omega");
    arch = pvUnpack(p,"arch");
    const = pvUnpack(p,"constant");
    phi = pvUnpack(p,"phi");

    u2 = (d[1].dataMatrix - d[2].dataMatrix * phi -
        const)^2;
```

```
    q = rows(arch);
    n = rows(u2);
    h = ones(n,1);
    v = seqa(1,1,q)' + seqa(0,1,n-q);
    h[q+1:rows(h)] = omega + reshape(u2[v],n-q,q) * arch;
    h[1:q] = ones(q,1)*meanc(h[q+1:rows(h)]);

    mm.function = -0.5*( (u2 ./ h) + ln(2 * pi) + ln(h) );
    retp(mm);

endp;


/*
**  hypothesis test that the arch parameters
**  are zero versus greater than zero
*/

struct cmlmtControl c1;
c1 = cmlmtControlCreate;

c1.A = zeros(3,6) ~ eye(3);
c1.B = zeros(3,1);
c1.covParType = 2;  // causes Jacobian to be computed
                    // which is needed for chibarsq

struct PV p1;
p1 = pvPack(pvCreate,.08999,"constant");
p1 = pvPack(p1,.25167|-.12599|.09164|.07517,"phi");
p1 = pvPack(p1,3.22713,"omega");
p1 = pvPack(p1,0|0|0,"arch");

/*
**  ML estimation of parameters where
```

```
**  parameters under hypothesis are fixed
**  to zero.
*/

struct cmlmtResults out1;
out1 = cmlmt(&lpr,p1,d1,c1);


/*
** The cmlmtControl instance, c2, contains the
** the constraints on the arch parameters
*/

struct cmlmtControl c2;
c2 = cmlmtcontrolcreate;

c2.bounds = {
    -10 10,
    -10 10,
    -10 10,
    -10 10,
    -10 10,
      0 10,
      0 10,
      0 10,
      0 10  };



psi = { 7, 8, 9 };
{ chibar,chibarprob } = chibarsq(out1,d1,c2,psi);

print;
```

```
print;
print "-----------------------------------------";
print "  test of H(arch) = 0 vs. H(arch) >= 0";
print;
print "       chibar " chibar;
print "   chibarprob " chibarprob;
```

The results are

```
-----------------------------------------
  test of H(arch) = 0 vs. H(arch) >= 0

       chibar  3.9152
   chibarprob  0.0913
```

3.8.5 Testing Lagrangeans

Equality Constraints

When equality constraints are present in the model, their associated Lagrange coefficients may be tested to determine their reasonableness. An estimate of the covariance matrix of the joint distribution of the Lagrange coefficients associated with the equality constraints is $\dot{G}\Sigma^{-1}\dot{G}'$ and therefore

$$\hat{\lambda}'\dot{G}\Sigma^{-1}\dot{G}'\hat{\lambda}$$

is asymptotically $\chi^2(p)$ where p is the length of $\hat{\lambda}$. Individual constraints may be tested using their associated t-statistics.

When appropriate this covariance matrix is assigned to the **Lagr.Eqcov** member of an instance of the **cmlmtResults** structure.

Active Inequality Constraints

When inequality constraints are active, their associated Lagrange coefficients are nonzero. The expected value of their Lagrange coefficients is zero (assuming correct specification of the model), and they are active only in occasional samples. How many samples this occurs in depends on their covariance matrix, which is estimated by $\dot{H}_{\oplus}\Sigma^{-1}\dot{H}'_{\oplus}$.

When appropriate this covariance matrix is assigned to **Lagr.Ineqcov** member of an instance of the **cmlmtResults** structure.

3.8.6 Heteroskedastic-consistent Covariance Matrix

When the **CovParType** member of an instance of **cmlmtControl** is set to 2, **CMLMT** returns heteroskedastic-consistent covariance matrices of the parameters in the **CovPar** member of an instance of the **cmlmtResults** structure returned from the call to **CMLMT**.

Define

$$B = \left(\frac{\partial F}{\partial \theta}\right)' \left(\frac{\partial L}{\partial \theta}\right)$$

evaluated at the estimates. Then the covariance matrix of the parameters is

$$\Omega B \Omega$$

3.8.7 Confidence Limits by Inversion

When the model includes inequality constraints, confidence limits computed as the ratio of the parameter estimate to its standard error are not correct because they do not take into account that the distribution of the parameter is restricted by its boundaries.

Inversion of the Likelihood Ratio Statistic. Partition a k-vector of parameters, $\theta = (\theta_1\ \theta_2)$, and let $\tilde{\theta}$ be a maximum likelihood estimate of θ, where θ_1 is fixed to some value. A $100(1-\alpha)\%$ confidence region for the parameters in θ_1 is defined by

$$-2 * log(L(\tilde{\theta})/L(\hat{\theta})) \leq \chi^2_{(1-\alpha,k)}.$$

Let

$$F_{lr}(\phi) = \min(-2 * log(L(\tilde{\theta})/L(\hat{\theta})) \mid \eta_i'\theta = \phi, H(\theta) \geq 0)$$

where η is a vector with a one in the i-th position and zeros elsewhere, and $H(\theta)$ is a function describing the constraints. The lower limit of the $(1-\alpha)$ interval for θ_i is the value of ϕ such that

$$F_{lr}(\phi) = \chi^2_{(1-\alpha,k)}. \tag{16}$$

A modified secant method is used to find the value of ϕ that satisfies (16). The upper limit is found by defining F_{lr} as a maximum.

The **CMLMT** procedure **`CMLProfileLimits`** solves this problem. Corrections are made by **`CMLMTProfileLimits`** when the limits are near constraint boundaries.

Inversion of the Wald Statistic. A $1-\alpha$ joint confidence region for θ is the hyper-ellipsoid

$$JF(J, N-K; \alpha) = (\theta - \hat{\theta})' V^{-1} (\theta - \hat{\theta}) \tag{17}$$

where V is the covariance matrix of the parameters, J is the number of parameters involved in the hypothesis, and $F(J, N-K; \alpha)$ is the upper α area of the F-distribution with J, N-K degrees of freedom.

If there are no constraints in the model, the $1 - \alpha$ confidence interval for any selected parameter is

$$\hat{\theta} \pm \sqrt{\eta_k' V^{-1} \eta_k}\; t(N - K; \alpha/2)$$

where η_k is a vector of zeros with the k-th element corresponding to the parameter being tested set to one.

When there are constraints no such simple description of the interval is possible. Instead it is necessary to state the confidence limit problem as a parametric nonlinear programming problem.

The lower limit of the confidence limit is the solution to

$$\min \left\{ \eta_k' \theta \mid (\theta - \hat{\theta})' V^{-1} (\theta - \hat{\theta}) \geq JF(J, N - K; \alpha), G(\theta) = 0, H(\theta) \geq 0) \right\}$$

where now η can be an arbitrary vector of constants and $J = \sum \eta_k \neq 0$, and where again we have assumed that the linear constraints and bounds have been folded in among nonlinear constraints. The upper limit is the maximum of this same function.

In this form, the minimization is not convex and can't be solved by the usual methods. However, the problem can be re-stated as a parametric nonlinear programming problem (Rust and Burrus, 1972). Define the function

$$F(\phi) = \min((\theta - \hat{\theta})' V^{-1} (\theta - \hat{\theta}) \mid \eta_k' \theta = \phi, G(\theta) = 0, H(\theta) \geq 0)$$

The upper and lower limits of the $1 - \alpha$ interval are the values of ϕ such that

$$F(\phi) = JF(J, N - K; \alpha)$$

To find this value it is necessary to iteratively refine ϕ by interpolation until 3.8.7 is satisfied. The **CMLMT** procedure **`CMLMTInverseWaldLimits`** solves this problem.

3.8.8 Bootstrap

The bootstrap method is used to generate empirical distributions of the parameters, thus avoiding the difficulties with the usual methods of statistical inference described above.

CMLMTBoot

Rather than randomly sample with replacement from the data set, **CMLMTBoot** performs **NumSample** weighted maximum likelihood estimations where the weights are Poisson pseudo-random numbers with expected value equal to the the number of observations, where **NumSample** is a member of an instance of the **cmlmtControl** structure. This is asymptotically equivalent to simple random sampling with replacement. The number of resamplings is determined by setting the **NumSample** member of an instance of a **cmlmtControl** structure. The default is 100 re-samplings. Efron and Tibshirani (1993:52) suggest that 100 is satisfactory, 50 is often enough to give a good estimate, and rarely are more than 200 needed.

The mean of the bootstrapped parameters is returned by **CMLMTBoot** in an instance of a **cmlmtResults** structure as the member **Par**, an instance of a **PV** structure. The covariance matrix is returned as the member **CovPar**. Confidence limits are returned as the member **BootLimits**. In addition **CMLMTBoot** writes the bootstrapped parameter estimates to a **GAUSS** data set with the name set in the member **BootFileName**. If the name is not specified, **CMLMTBoot** selects the name `BOOTxxxx`, where `xxxx` starts at 0000 and increments by 1 until a name is found that is not already in use.

Example

To bootstrap the example in Section 3.7.4, the only necessary alteration is the change the call to **CMLMT** to a call to **CMLMTBoot**:

```
library cmlmt,pgraph;
```

```
#include cmlmt.sdf
#include kern.sdf

proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.DataMatrix[.,1];
    x = d.DataMatrix[.,2:4];

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;
    if ind[1];
        mm.Function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;
    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/(2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.Gradient = u.*g1 + (1-u).*g2;
    endif;
    retp(mm);
endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
```

```
c0 = cmlmtcontrolcreate;
c0.Title = "CML6 - bootstrap example";
c0.Bounds = { -10 10,
              -10 10,
              -10 10,
              -10 10,
              .1 10 };

c0.BootFilename = "example6";
c0.State = 324235;

struct DS d0;
d0.Dname = "cmlmttobit";

struct cmlmtResults out1;
out1 = cmlmtBoot(&lpr,p0,d0,c0);

call cmlmtPrt(out1);

call kern(loadd("example6"),kernControlCreate);
```

This example calls **kern** to generate kernel density plots of the parameters from the **GAUSS** data set generated by **CMLMTBoot**.

3.8.9 Profiling

The **CMLMT** proc, **CMLMTProfile** generates profile t plots as well as plots of the likelihood profile traces for all of the parameters in the model in pairs. The profile t plots are used to assess the nonlinearity of the distributions of the individual parameters, and the likelihood profile traces are used to assess the bivariate distributions. The input and output arguments to **CMLMTProfile** are identical to those of **CMLMT**. But in addition to providing the maximum likelihood estimates and covariance matrix of the parameters, a series of plots are printed to the screen using **GAUSS**' Publication Quality Graphics. A screen is printed for each possible pair of parameters. There are three plots, a profile t plot

for each parameter, and a third plot containing the likelihood profile traces for the two parameters.

The discussion in this section is based on Bates and Watts (1988), pages 205-216, which is recommended reading for the interpretation and use of profile t plots and likelihood profile traces.

The Profile t Plot

Define

$$\tilde{\theta}_k = (\tilde{\theta}_1, \tilde{\theta}_2, ..., \tilde{\theta}_{k-1}, \theta_k, \tilde{\theta}_{k+1}, ..., \tilde{\theta}_K)$$

This is the vector of maximum likelihood estimates *conditional* on θ_k, i.e., where θ_k is fixed to some value. Further define the profile t function

$$\tau(\theta_k) = sign(\theta_k - \hat{\theta}_k)(N - K)\sqrt{2N\left[L(\tilde{\theta}_k) - L(\hat{\theta}_k)\right]}$$

For each parameter in the model, τ is computed over a range of values for θ_k. These plots provide exact likelihood intervals for the parameters, and reveal how nonlinear the estimation is. For a linear model, τ is a straight line through the origin with unit slope. For nonlinear models, the amount of curvature is diagnostic of the nonlinearity of the estimation. High curvature suggests that the usual statistical inference using the t-statistic is hazardous.

The Likelihood Profile Trace

The likelihood profile traces provide information about the bivariate likelihood surfaces. For nonlinear models the profile traces are curved, showing how the parameter estimates

affect each other and how the projection of the likelihood contours onto the (θ_k, θ_ℓ) plane might look. For the (θ_k, θ_ℓ) plot, two lines are plotted, $L(\tilde{\theta}_k)$ against θ_k and $L(\tilde{\theta}_\ell)$ against θ_ℓ.

If the likelihood surface contours are long and thin, indicating the parameters to be collinear, the profile traces are close together. If the contours are fat, indicating the parameters to be more uncorrelated, the profile traces tend to be perpendicular. And if the contours are nearly elliptical, the profile traces are straight. The surface contours for a linear model would be elliptical and thus the profile traces would be straight and perpendicular to each other. Significant departures of the profile traces from straight, perpendicular lines, therefore, indicate difficulties with the usual statistical inference.

To generate profile t plots and likelihood profile traces from the example in Section 3.7.4, it is necessary only to change the call to **CMLMT** to a call to **CMLMTProfile**:

```
call CMLMTPrt(CMLMTProfile("cmlmttobit",0,&lpr,x0));
```

CMLMTProfile produces the same output as **CMLMT** which can be printed out using a call to **CMLMTPRT**.

For each pair of parameters a plot is generated containing an xy plot of the likelihood profile traces of the two parameters, and two profile t plots, one for each parameter.

The likelihood profile traces indicate that the distributions of parameters 1 and 2 are highly correlated. Ideally, the traces would be perpendicular and the trace in this example is far from ideal.

The profile t plots indicate that the parameter distributions are somewhat nonlinear. Ideally the profile t plots would be straight lines and this example exhibits significant nonlinearity. It is clear that any interpretations of the parameters of this model must be made quite carefully.

3.9 Run-Time Switches

If the user presses H during the iterations, a help table is printed to the screen which describes the run-time switches. By this method, important global variables may be modified during the iterations. The case may also be ignored, that is, either upper or lower case letters suffice.

A	Change Algorithm
C	Force Exit
G	Toggle **GradMethod**
H	Help Table
O	Set **PrintIters**
S	Set line search method
T	set **TrustRadius**
V	Set **DirTol**

Keyboard polling can be turned off completely by setting the **disableKey** member of the **cmlmtControl** instance to a nonzero value.

3.10 CMLMT Structures

3.10.1 cmlmtControl

matrix	A
matrix	B

matrix	C
matrix	D
scalar	EqProc
scalar	IneqProc
scalar	IneqJacobian
scalar	EqJacobian
matrix	Bounds
matrix	Algorithm
matrix	Switch
matrix	LineSearch
matrix	Active
matrix	NumObs
matrix	MaxIters
matrix	DirTol
matrix	Weights
matrix	CovParType
matrix	Alpha
matrix	FeasibleTest
matrix	MaxTries
matrix	RandRadius
matrix	GradMethod

matrix	HessMethod
matrix	GradStep
matrix	HessStep
matrix	GradCheck
matrix	State
string	Title
scalar	PrintIters
matrix	TrustRadius
matrix	Aug
matrix	DisableKey
matrix	Select
matrix	Center
matrix	Increment
matrix	Width
matrix	NumCat
string	BootFileName
string	BayesFileName
matrix	BayesAlpha
scalar	PriorProc
matrix	NumSamples
matrix	MaxTime
matrix	MaxBootTime

3.10.2 cmlmtResults

struct	PV Par
scalar	Fct
struct	CcmlmtLagrange lagr
scalar	Retcode
string	ReturnDescription
matrix	CovPar
string	CovParDescription
matrix	NumObs
matrix	Hessian
matrix	Xproduct
matrix	Waldlimits
matrix	Inversewaldlimits
matrix	Bayeslimits
matrix	Profilelimits
matrix	Bootlimits
matrix	Gradient
matrix	NumIterations
matrix	ElapsedTime
matrix	Alpha
string	Title

3.10.3 cmlmtLagrange

matrix	Lineq
matrix	Nlineq
matrix	Linineq
matrix	Nlinineq
matrix	Bounds

3.10.4 modelResults

matrix	Function
matrix	Gradient
matrix	Hessian
array	Hessianw
matrix	NumObs

3.11 Error Handling

3.11.1 Return Codes

The **Retcode** member of an instance of a **cmlmtResults** structure, which is returned by **CMLMT**, contains a scalar number that contains information about the status of the iterations upon exiting **CMLMT**. The following table describes their meanings:

0	normal convergence

1	forced exit
2	maximum iterations exceeded
3	function calculation failed
4	gradient calculation failed
5	Hessian calculation failed
6	line search failed
7	function cannot be evaluated at initial parameter values
8	error with gradient
9	error with constraints
10	secant update failed
11	maximum time exceeded
12	error with weights
13	quadratic program failed
14	equality Jacobian failed
15	inequality Jacobian failed
16	function evaluated as complex
20	Hessian failed to invert
34	data set could not be opened

3.11.2 Error Trapping

Setting the **PrintIters** member of an instance of a **cmlmtControl** structure to zero turns off all printing to the screen. Error codes, however, still are printed to the screen unless error trapping is also turned on. Setting the trap flag to 4 causes **CMLMT** *not* to send the messages to the screen:

```
trap 4;
```

Whatever the setting of the trap flag, **CMLMT** discontinues computations and returns with an error code. The trap flag in this case only affects whether messages are printed to the screen or not. This is an issue when the **CMLMT** function is embedded in a larger program, and you want the larger program to handle the errors.

3.12 References

1. Andrews, D.W.K, 1999. “Inconsistency of the bootstrap when a parameter is on the boundary of the parameter space”, *Econometrica*, 99.

2. Amemiya, Takeshi, 1985. *Advanced Econometrics*. Cambridge, MA: Harvard University Press.

3. Bates, Douglas M. and Watts, Donald G., 1988. *Nonlinear Regression Analysis and Its Applications*. New York: John Wiley & Sons.

4. Berndt, E., Hall, B., Hall, R., and Hausman, J., 1974. “Estimation and inference in nonlinear structural models”. *Annals of Economic and Social Measurement* 3:653-665.

5. Brent, R.P., 1972. *Algorithms for Minimization Without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall.

6. Conn, Andrew R., Gould, Nicholas I.M., Toint, Philippe L., 2000. *Trust-Region Methods.* Philadelphia: SIAM.

7. Dennis, Jr., J.E., and Schnabel, R.B., 1983. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.

8. Efron, Gradley, Robert J. Tibshirani, 1993. *An Introduction to the Bootstrap*. New York: Chapman & Hall.

9. Fletcher, R., 1987. *Practical Methods of Optimization*. New York: Wiley.

10. Gill, P. E. and Murray, W. 1972. "Quasi-Newton methods for unconstrained optimization." *J. Inst. Math. Appl.*, 9, 91-108.

11. Gourieroux, Christian, Holly, Alberto, and Monfort, Alain (1982). "Likelihood Ratio test, Wald Test, and Kuhn-Tucker test in linear models with inequality constraints on the regression parameters", *Econometrica* 50: 63-80.

12. Han, S.P., 1977. "A globally convergent method for nonlinear programming." *Journal of Optimization Theory and Applications*, 22:297-309.

13. Hock, Willi and Schittkowski, Klaus, 1981. *Lecture Notes in Economics and Mathematical Systems*. New York: Springer-Verlag.

14. Jamshidian, Mortaza and Bentler, P.M., 1993. "A modified Newton method for constrained estimation in covariance structure analysis." *Computational Statistics & Data Analysis*, 15:133-146.

15. Judge, G.G., R.C. Hill, W.E. Griffiths, H. Lütkepohl and T.C. Lee. 1988. *Introduction to the Theory and Practice of Econometrics.* 2nd Edition. New York:Wiley.

16. Judge, G.G., W.E. Griffiths, R.C. Hill, H. Lüutkepohl and T.C. Lee. 1985. *The Theory and Practice of Econometrics.* 2nd Edition. New York:Wiley.

17. Liang, Kung-Yee and Zeger, Scott L. (1986). Longitudinal Data Analysis Using Generalized Linear Models, *Biometrika*, 73, pp 13–22,

18. Schoenberg, Ronald, 1997. “Constrained Maximum Likelihood”. *Computational Economics*, 1997:251-266.

19. Silvapulle, Mervyn J. and Sen, Pranab K., 2005. *Constrained Statistical Inference.* New York: Wiley.

20. Silvapulle, Mervyn J. and Silvapulle, Paramsothy (1995). A score test against one-sided alternatives. *Journal of the American Statistical Association*, volume 90, pages 342–349.

21. Wolak, Frank A. (1987). An exact test for multiple inequality and equality constraints in the linear regression model. *Journal of the American Statistical Association*, volume 82, pages 782–793.

22. White, H. 1981. “Consequences and detection of misspecified nonlinear regression models.” *Journal of the American Statistical Association* 76:419-433.

23. White, H. 1982. “Maximum likelihood estimation of misspecified models.” *Econometrica* 50:1-25.

CMLMT Reference 4

CMLMT

PURPOSE Computes estimates of parameters of a constrained maximum likelihood function.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMT(&***modelProc*,*par*,*data*,*ctl*);

INPUT *&modelProc* a pointer to a procedure that returns either the log-likelihood for one observation or a vector of log-likelihoods for a matrix of observations.

par instance of a **PV** structure containing start values for the parameters constructed using the **pvPack** functions.

data instance or matrix of instances of a **DS** structure containing data. it is passed to the user-provided procedure pointed at by `&fct` to be used in the objective function. There are two cases,

1 a scalar or vector of **DS** instances passed to **cmlmt** are passed unchanged to the user-provided log-likelihood procedure. In this case the structure can be a scalar, vector, or matrix of **DS** instances, and all members of all the instances can be set at the discretion of the programmer, **except** that the **dname** member of the [1,1] element of the structure must be a null string.

2 if the **dname** member of the **DS** instance contains the name of a **GAUSS** data set, **cmlmt** passes the contents of that data set either in whole or in part to the user-provided log-likelihood procedure in the **DataMatrix** member of the first **DS** instance. If the member, **Vnames**, contains a string array of column names in the data set **cmlmt** will select those columns for passing to that procedure. All other members, as well as all members of succeeding elements of a vector of instances may be used at the programmer's discretion.

To clarify, if you do **not** want **cmlmt** to read the data from a **GAUSS** data set and pass it to your procedure, you can pass a **DS** structure containing whatever you wish to **cmlmt** and it will be passed untouched to your procedure.

If you do wish to have **cmlmt** to read the data from a **GAUSS** data set, set the **dname** member of the first instance in the **DS** structure to the name of the **GAUSS** data set, and **cmlmt** will pass the contents in the **DataMatrix** member of the first instance.

ctl an instance of a **cmlmtControl** structure. Normally an instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user. For an instance named **ctl**, the members are:

ctl.**A**	M × K matrix, linear equality constraint coefficients: *ctl*.**A** * *p* = *ctl*.**B** where **p** is a vector of the parameters.
ctl.**B**	M × 1 vector, linear equality constraint constants: *ctl*.**A** * *p* = *ctl*.**B** where **p** is a vector of the parameters.
ctl.**C**	M × K matrix, linear inequality constraint coefficients: *ctl*.**C** * *p* ≥ *ctl*.**D** where **p** is a vector of the parameters.
ctl.**D**	M × 1 vector, linear inequality constraint constants: *ctl*.**C** * *p* ≥ *ctl*.**D** where **p** is a vector of the parameters.
ctl.**EqProc**	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure.
ctl.**IneqProc**	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a vector of computed inequality constraints. For more

details see Remarks below. Default = {.}, i.e., no inequality procedure.

ctl.**EqJacobian** scalar, pointer to a procedure that computes the Jacobian of the equality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a matrix of derivatives of the equality constraints with respect to the parameters. Default = {.}, i.e., no equality Jacobian procedure.

ctl.**IneqJacobian** scalar, pointer to a procedure that computes the Jacobian of the inequality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a matrix of derivatives of the inequality constraints with respect to the parameters. Default = {.}, i.e., no inequality Jacobian procedure.

ctl.**Bounds** 1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }.

ctl.**Algorithm** scalar, descent algorithm.

- **0** Modified BFGS
- **1** BFGS (default)
- **2** DFP
- **3** Newton

4 BHHH

ctl.**useThreads** scalar, if nonzero, the calculation of numerical derivatives will be threaded. Default = 0.

ctl.**Switch** 4 × 1 or 4 × 2 vector, controls algorithm switching:
if 4 × 1:

ctl.**Switch[1]** algorithm number to switch to.

ctl.**Switch[2]** **cmlmt** switches if function changes less than this amount.

ctl.**Switch[3]** **cmlmt** switches if this number of iterations is exceeded.

ctl.**Switch[4]** **cmlmt** switches if line search step changes less than this amount.

else if 4 × 2 **cmlmt** switches between the algorithm in column 1 and column 2. Default = { 1 3, .0001 .0001, 10 10, .0001 .0001 }.

ctl.**LineSearch** scalar, sets line search method.

0 augmented trust region method (requires constraints)

1 STEPBT (quadratic and cubic curve fit) (default)

2 Brent's method

3 BHHHStep

4 half

5 Wolfe's condition

ctl.**TrustRadius** scalar, radius of the trust region. If scalar missing, trust region not

applied. The trust sets a maximum amount of the direction at each iteration. Default = .001.

ctl.**Aug** scalar, augmentation constant for trust region line search method.

ctl.**Active** K × 1 vector, set K-th element to zero to fix it to start value. Use the **GAUSS** function **pvGetIndex** to determine where parameters in the **PV** structure are in the vector of parameters. Default = {.}, all parameters are active.

ctl.**NumObs** scalar, number of observations, required if the log-likelihood procedure returns a scalar.

ctl.**MaxIters** scalar, maximum number of iterations. Default = 10000.

ctl.**DirTol** scalar, convergence tolerance. Iterations cease when all elements of the direction vector are less than this value. Default = $1e-5$.

ctl.**Weights** vector, weights for objective function returning a vector. Default = 1.

ctl.**CovParType** scalar. If 2, QML covariance matrix, else if 0, no covariance matrix is computed, else ML covariance matrix is computed. Default = 1.

ctl.**Alpha** scalar, probability level for statistical tests. Default = .05.

ctl.**FeasibleTest** scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality

boundaries, then this test can be turned off. Default = 1.

ctl.**MaxTries** scalar, maximum number of attemps in random search. Default = 100.

ctl.**RandRadius** scalar, If zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = .001.

ctl.**GradMethod** scalar, method for computing numerical gradient.

- **0** central difference
- **1** forward difference (default)
- **2** backward difference

ctl.**HessMethod** scalar, method for computing numerical Hessian.

- **0** central difference
- **1** forward difference (default)
- **2** backward difference

ctl.**GradStep** scalar or K × 1, increment size for computing numerical gradient. If scalar, stepsize will be value times parameter estimates for the numerical gradient. If K × 1, the step size for the gradient will be the elements of the vector, i.e., it will not be multiplied times the parameters.

ctl.**HessStep** scalar or K × 1, increment size for computing numerical Hessian. If scalar, stepsize will be value times parameter estimates for the numerical Hessian. If K × 1, the step size for the gradient will be the elements of the vector, i.e., it will not be multiplied times the parameters.

	ctl.**GradCheck**	scalar, if nonzero and if analytical gradients and/or Hessian have been provided, numerical gradients and/or Hessian are computed and compared against the analytical versions.
	ctl.**State**	scalar, seed for random number generator.
	ctl.**Title**	string, title of run.
	ctl.**printIters**	scalar, if nonzero, prints iteration information. Default = 0.
	ctl.**MaxBootTime**	scalar, maximum number of minutes to convergence.
	ctl.**DisableKey**	scalar, if nonzero, keyboard input disabled.

OUTPUT *out* instance of a **cmlmtResults** structure. For an instance named **out**, the members are:

out.**Par**	instance of a **PV** structure containing the parameter estimates. Use **pvUnpack** to retrieve matrices and arrays or **pvGetParvector** to get the parameter vector.
out.**Fct**	scalar, function evaluated at parameters in *out*.**Par**
out.**ReturnDescription**	string, description of return values.
out.**CovPar**	K × K matrix, covariance matrix of parameters.
out.**CovParDescription**	string, description of *out*.**CovPar**.
out.**NumObs**	scalar, number of observations.
out.**Hessian**	K × K matrix, Hessian evaluated at parameters in *out*.**Par**.

out.**Xproduct** K × K matrix, cross-product of N × K matrix of first derivatives evaluated at parameters in *out*.**Par**. Not available if loglikelihood function returns a scalar.

out.**WaldLimits** K × 2 matrix, Wald confidence limits.

out.**inverseWaldLimits** K × 2 matrix, confidence limits by inversion of Wald statistics. Available only

out.**ProfileLimits** K × 2 matrix, profile likelihood confidence limits, i.e., by inversion of likelihood ratio statistics. Only available if **cmlmtProfileLimits** has been called.

out.**BayesLimits** K × 2 matrix, weighted likelihood Bayesian confidence limits. Only available if **cmlmtBayes** has been called.

out.**BootLimits** K × 2 Matrix, bootstrap confidence limits. Available only if **cmlmtBoot** has been called.

out.**Gradient** K × 1 vector, gradient evaluated at the parameters in *out*.**Par**.

out.**NumIterations** scalar, number of iterations.

out.**ElapsedTime** scalar, elapsed time of iterations.

out.**Alpha** scalar, probability level of confidence limits. Default = .05.

out.**Title** string, title of run.

out.**Lagr** an instance of a **cmlmtLagrange** structure containing the Lagrangeans for the constraints. For an instance named *out*.**Lagr**, the members are:

out.**Lagr.Lineq** M × 1 vector,

Lagrangeans of linear equality constraints.

out.**Lagr.Nlineq** N × 1 vector, Lagrangeans of nonlinear equality constraints.

out.**Lagr.Linineq** P × 1 vector, Lagrangeans of linear inequality constraints.

out.**Lagr.Nlinineq** Q × 1 vector, Lagrangeans of nonlinear inequality constraints.

out.**Lagr.Bounds** K × 2 matrix, Lagrangeans of bounds.

out.**Lagr.EqCov** (M+N) × (M+N) matrix, covariance matrix of equality constraints.

out.**Lagr.IneqCov** (P+Q) × (P+Q) matrix, covariance matrix of inequality constraints.

Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.

out.**Retcode** return code:

0 normal convergence

1 forced exit

2 maximum number of iterations exceeded

3 function calculation failed

4 gradient calculation failed

5 Hessian calculation failed
6 line search failed
7 functional evaluation failed
8 error with initial gradient
9 error with constraints
10 second update failed
11 maximum time exceeded
12 error with weights
13 quadratic program failed
14 equality constraint Jacobian failed
15 inequality constraint Jacobian failed
16 function evaluated as complex
20 Hessian failed to invert
34 data set could not be opened

REMARKS **Writing the Log-likelihood Function** There is one required user-provided procedure, the one computing the log-likelihood function and optionally the first and/or second derivatives, and four other optional procedures, one each for computing the equality constraints, the inequality constraints, the Jacobian of the equality constraints, and the Jacobian of the inequality constraints.

The main procedure, computing the log-likelihood and optionally the first and/or second derivatives, has three arguments, an instance of type struct **PV** containing the parameters, a second argument that is an instance of type struct **DS** containing the data, and a third argument that is a vector of zeros and ones indicating which of the results, the function, first derivatives, or second derivatives, are to be computed.

The remaining optional procedures take just two arguments, the instance of the **PV** structure containing the parameters and the the instance of the **DS** structure containing the data.

The instance of the **PV** structure is set up using the **PV** pack procedures, **pvPack**, **pvPackm**, **pvPacks**, and **pvPacksm**. These procedures allow for setting up a parameter vector in a variety of ways.

The instance of the **DS** structure containing the data is set up in two distinct ways depending on whether **cmlmt** is to read the data in from a **GAUSS** data set, or whether the data is in a matrix.

For example, the following procedure computes the log-likelihood and the first derivatives for a tobit model:

```
proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d[1].dataMatrix;
    x = d[2].dataMatrix;

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;

    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
```

```
            g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
                (2*s2);
            g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
                (yh/(2*s2));
            g2 = (pdfn(yh)./cdfnc(yh)).*g2;
            mm.gradient = u.*g1 + (1-u).*g2;
        endif;
        retp(mm);

    endp;
```

Nonlinear Equality or Inequality Constraints Procedures

The procedures for nonlinear equality and inequality constraints take two input arguments, an instance of a **PV** parameter structure and an instance of a **DS** data structure. For example, to constrain the sum of squares of coefficients to be greater than one, provide the following procedure:

```
proc ineqConst(struct PV par1, struct DS data1);
    local b;
    b = pvUnpack(p,"b");
    retp( sumc(b^2) - 1 );
endp;
```

EXAMPLE The following is a complete example. It applys the Biochemical Oxygen Demand model to data taken from Douglas M. Bates and Donald G. Watts, *Nonlinear Regression Analysis and Its Applications*, page 270.

```
library cmlmt;
#include cmlmt.sdf
```

```
proc lnlk(struct PV p, struct DS d, ind);
    local dev,s2,m,r,b0,b;

    struct modelResults mm;

    b0 = pvUnpack(p,1);
    b = pvUnpack(p,2);
    r = exp(-b*d[2].dataMatrix);
    m = 1 - r;

    dev = d[1].dataMatrix -  b0*m;
    s2 = dev'dev/rows(dev);

    if ind[1];
        mm.function = lnpdfmvn(dev,s2);
    endif;
    if ind[2];
        mm.gradient =  (dev/s2) .*
            (m ~ b0*d[2].dataMatrix.*r);
    endif;
    retp(mm);

endp;

struct DS d0;
d0 = reshape(dsCreate,2,1);
d0[1].dataMatrix =
  {
     8.3,
    10.3,
    19.0,
    16.0,
    15.6,
```

```
    19.8
  };

d0[2].dataMatrix =
  {
    1,
    2,
    3,
    4,
    5,
    7
  };


struct PV p0;
p0 = pvPacki(pvCreate,19.143,"b0",1);
p0 = pvPacki(p0,.5311,"b",2);


struct cmlmtControl c0;
c0 = cmlmtControlCreate;

c0.Bounds = { 10 35,
              0  2 };

struct cmlmtResults out;
out = cmlmt(&lnlk,p0,d0,c0);
```

SOURCE cmlmt.src

CMLMTBayes

PURPOSE Bayesian Inference using weighted maximum likelihood bootstrap.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMTBayes(&***modelProc*,*par*,*data*,*ctl*);

INPUT *&modelProc* a pointer to a procedure that returns either the log-likelihood for one observation or a vector of log-likelihoods for a matrix of observations.

par instance of a **PV** structure containing start values for the parameters constructed using the **pvPack** functions.

data instance or matrix of instances of a **DS** structure containing data. it is passed to the user-provided procedure pointed at by `&fct` to be used in the objective function. There are two cases,

1 a scalar or vector of **DS** instances passed to **cmlmt** are passed unchanged to the user-provided log-likelihood procedure. In this case the structure can be a scalar, vector, or matrix of **DS** instances, and all members of all the instances can be set at the discretion of the programmer, **except** that the **dname** member of the [1,1] element of the structure must be a null string.

2 if the **dname** member of the **DS** instance contains the name of a **GAUSS** data set, **cmlmt** passes the contents of that data set either in whole or in part to the user-provided log-likelihood procedure in the **DataMatrix** member of the first **DS** instance. If the member, **Vnames**, contains a string array of column names in the data set **cmlmt** will select those columns for passing to

that procedure. All other members, as well as all members of succeeding elements of a vector of instances may be used at the programmers discretion.

To clarify, if you do **not** want **cmlmt** to read the data from a **GAUSS** data set and pass it to your procedure, you can pass a **DS** structure containing whatever you wish to **cmlmt** and it will be passed untouched to your procedure.

If you do wish to have **cmlmt** read the data from a **GAUSS** data set, set the **dname** member of the first instance in the **DS** structure to the name of the **GAUSS** data set, and **cmlmt** will pass the contents in the **DataMatrix** member of the first instance.

ctl an instance of a **cmlmtControl** structure. Normally an instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user. For an instance named **ctl**, the members are:

ctl.**useThreads**	scalar, if nonzero, resampling will be threaded. Default = 0.
ctl.**BayesAlpha**	scalar, exponent of the Dirichlet random variates used in the weights for the weighted bootstrap. See Newton and Raftery, "Approximate Bayesian Inference with the Weighted Likelihood Bootstrap", *J.R.Statist. Soc. B* (1994), 56:3-48. Default = 1.4.
ctl.**PriorProc**	scalar, pointer to proc for computing prior. This proc takes the parameter vector as its only argument and returns a scalar probability. If a proc is not provided, a uniform prior is assumed.
ctl.**NumSample**	scalar, number of re-samples in the weighted likelihood bootstrap.

ctl.**BayesFname** string, file name of **GAUSS** data set (do not include the .DAT extension) containing simulated posterior of the parameters. If not specified, **CMLMTBayes** will select the file name, BAYESxxxx where xxxx is 0000 incremented by 1 until a name is found that doesn't exist on the current directory.

ctl.**MaxBootTime** scalar, maximum number of minutes for resampling.

For description of additional members of the **cmlmtControl** structure see reference for **cmlmt**.

OUTPUT *out* instance of a **cmlmtResults** structure.

out.**Par** instance of a **PV** structure containing the mean of the resampled estimates. Use **pvUnpack** to retrieve matrices and arrays or **pvGetParvector** to get the parameter vector.

out.**Fct** scalar, mean log-likelihood across resamples.

out.**ReturnDescription** string, description of return values.

out.**CovPar** K × K matrix, covariance matrix of resampled parameter estimates.

out.**CovParDescription** string, description of *out*.**CovPar**.

out.**NumObs** scalar, number of observations.

out.**BayesLimits** K × 2 matrix, weighted likelihood Bayesian confidence limits. Only available if **cmlmtBayes** has been called.

out.**Gradient** K × 1 vector, mean gradient.

out.**NumIterations** scalar, average number of iterations.

out.**ElapsedTime** scalar, average elapsed time of iterations.

out.**Alpha** scalar, probability level of confidence limits. Default = .05.

out.**Title** string, title of run.

out.**Retcode** return code:

- **0** normal convergence
- **1** forced exit
- **2** maximum number of iterations exceeded
- **3** function calculation failed
- **4** gradient calculation failed
- **5** Hessian calculation failed
- **6** line search failed
- **7** functional evaluation failed
- **8** error with initial gradient
- **9** error with constraints
- **10** second update failed
- **11** maximum time exceeded
- **12** error with weights
- **13** quadratic program failed
- **14** equality constraint Jacobian failed
- **15** inequality constraint Jacobian failed
- **16** function evaluated as complex
- **20** Hessian failed to invert
- **34** data set could not be opened

EXAMPLE

```
library cmlmt,pgraph;
#include cmlmt.sdf
#include kern.sdf

graphset;


proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.dataMatrix[1];
    x = d[2].dataMatrix[2:4];

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;

    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
```

```
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.gradient = u.*g1 + (1-u).*g2;
    endif;


    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
              -10 10,
              -10 10,
              .1 10 };

c0.numSamples = 200;
c0.bayesFileName = "bayes";

proc prior(b);   /*  unit normal prior  */
   retp(prodc(pdfn(b)));
endp;
c0.PriorProc = &prior;
```

```
struct DS d0;
d0 = dsCreate;
d0.dname = "cmlmttobit";

out1 = cmlmtBayes(&lpr,p0,d0,c0);

call cmlmtPrt(out1);
```

SOURCE cmlmtbayes.src

CMLMTBoot

PURPOSE Computes bootstrap estimates.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMTBoot(&***modelProc*,*par*,*data*,*ctl*);

INPUT *&modelProc* a pointer to a procedure that returns either the log-likelihood for one observation or a vector of log-likelihoods for a matrix of observations.

par instance of a **PV** structure containing start values for the parameters constructed using the **pvPack** functions.

data instance or matrix of instances of a **DS** structure containing data. It is passed to the user-provided procedure pointed at by &fct to be used in the objective function. There are two cases,

1 a scalar or vector of **DS** instances passed to **cmlmt** are passed unchanged to the user-provided log-likelihood procedure. In this case the structure can be a scalar,

vector, or matrix of **DS** instances, and all members of all the instances can be set at the discretion of the programmer, **except** that the **dname** member of the [1,1] element of the structure must be a null string.

2 if the **dname** member of the **DS** instance contains the name of a **GAUSS** data set, **cmlmt** passes the contents of that data set either in whole or in part to the user-provided log-likelihood procedure in the **DataMatrix** member of the first **DS** instance. If the member, **Vnames**, contains a string array of column names in the data set **cmlmt** will select those columns for passing to that procedure. All other members, as well as all members of succeeding elements of a vector of instances may be used at the programmers discretion.

To clarify, if you do **not** want **cmlmt** to read the data from a **GAUSS** data set and pass it to your procedure, you can pass a **DS** structure containing whatever you wish to **cmlmt** and it will be passed untouched to your procedure.

If you do wish to have **cmlmt** to read the data from a **GAUSS** data set, set the **dname** member of the first instance in the **DS** structure to the name of the **GAUSS** data set, and **cmlmt** will pass the contents in the **DataMatrix** member of the first instance.

ctl an instance of a **cmlmtControl** structure. Normally an instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user. For an instance named **ctl**, the members are:

ctl.**useThreads**	scalar, if nonzero, resampling will be threaded. Default = 0.
ctl.**NumSample**	scalar, number of re-samples in the weighted likelihood bootstrap.
ctl.**BootFname**	string, file name of **GAUSS** data set (do not include the .DAT extension)

containing simulated posterior of the parameters. If not specified, **CMLMTBoot** will select the file name, BAYESxxxx where xxxx is 0000 incremented by 1 until a name is found that doesn't exist on the current directory.

ctl.**MaxBootTime** scalar, maximum number of minutes for resampling.

For description of additional members of the **cmlmtControl** structure see reference for **cmlmt**.

OUTPUT *out* instance of a **cmlmtResults** structure.

out.**Par** instance of a **PV** structure containing the mean of the resampled estimates. Use **pvUnpack** to retrieve matrices and arrays or **pvGetParvector** to get the parameter vector.

out.**Fct** scalar, mean log-likelihood across resamples.

out.**ReturnDescription** string, description of return values.

out.**CovPar** K × K matrix, covariance matrix of resampled parameter estimates.

out.**CovParDescription** string, description of *out*.**CovPar**.

out.**NumObs** scalar, number of observations.

out.**BootLimits** K × 2 Matrix, bootstrap confidence limits. Available only if **cmlmtBoot** has been called.

out.**Gradient** K × 1 vector, mean gradient.

out.**NumIterations** scalar, average number of iterations.

out.**ElapsedTime** scalar, average elapsed time of iterations.

out.**Alpha** scalar, probability level of confidence limits. Default = .05.

out.**Title** string, title of run.

out.**Retcode** return code:

- **0** normal convergence
- **1** forced exit
- **2** maximum number of iterations exceeded
- **3** function calculation failed
- **4** gradient calculation failed
- **5** Hessian calculation failed
- **6** line search failed
- **7** functional evaluation failed
- **8** error with initial gradient
- **9** error with constraints
- **10** second update failed
- **11** maximum time exceeded
- **12** error with weights
- **13** quadratic program failed
- **14** equality constraint Jacobian failed
- **15** inequality constraint Jacobian failed
- **16** function evaluated as complex
- **20** Hessian failed to invert
- **34** data set could not be opened

EXAMPLE

```
library cmlmt,pgraph;
```

```
#include cmlmt.sdf
#include kern.sdf

graphset;


proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.dataMatrix[1];
    x = d[2].dataMatrix[2:4];

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;

    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) +
           (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
```

```
        mm.gradient = u.*g1 + (1-u).*g2;
    endif;


    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
              -10 10,
              -10 10,
              .1 10 };

c0.numSamples = 200;
c0.bootFileName = "boot";

proc prior(b);   /*  unit normal prior  */
   retp(prodc(pdfn(b)));
endp;
c0.PriorProc = &prior;

struct DS d0;
d0 = dsCreate;
```

```
d0.dname = "cmlmttobit";

out1 = cmlmtBoot(&lpr,p0,d0,c0);

call cmlmtPrt(out1);
```

SOURCE cmlmtboot.src

CMLMTProfile

PURPOSE Computes profile t plots and likelihood profile traces for constrained maximum likelihood models.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMTProfile(&***modelProc*,*par*,*data*,*ctl*);

INPUT *&modelProc* a pointer to a procedure that returns either the log-likelihood for one observation or a vector of log-likelihoods for a matrix of observations.

par instance of a **PV** structure containing start values for the parameters constructed using the **pvPack** functions.

data instance or matrix of instances of a **DS** structure containing data. it is passed to the user-provided procedure pointed at by &fct to be used in the objective function. There are two cases,

1 a scalar or vector of **DS** instances passed to **cmlmt** are passed unchanged to the user-provided log-likelihood procedure. In this case the structure can be a scalar, vector, or matrix of **DS** instances, and all members of all

the instances can be set at the discretion of the programmer, **except** that the **dname** member of the [1,1] element of the structure must be a null string.

2 if the **dname** member of the **DS** instance contains the name of a **GAUSS** data set, **cmlmt** passes the contents of that data set either in whole or in part to the user-provided log-likelihood procedure in the **DataMatrix** member of the first **DS** instance. If the member, **Vnames**, contains a string array of column names in the data set **cmlmt** will select those columns for passing to that procedure. All other members, as well as all members of succeeding elements of a vector of instances may be used at the programmers discretion.

To clarify, if you do **not** want **cmlmt** to read the data from a **GAUSS** data set and pass it to your procedure, you can pass a **DS** structure containing whatever you wish to **cmlmt** and it will be passed untouched to your procedure.

If you do wish to have **cmlmt** to read the data from a **GAUSS** data set, set the **dname** member of the first instance in the **DS** structure to the name of the **GAUSS** data set, and **cmlmt** will pass the contents in the **DataMatrix** member of the first instance.

ctl an instance of a **cmlmtControl** structure. Normally an instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user. For an instance named **ctl**, the members are:

ctl.**NumCat**	scalar, number of categories in profile table. Default = 16.
ctl.**Increment**	K × 1 vector, increments for cutting points, default is 2 * *ctl*.**Width** * std dev / *ctl*.**NumCat**. If scalar zero, increments are computed by **CMLMTProfile**.

ctl.**Center** K × 1 vector, value of center category in profile table. Default values are coefficient estimates.

ctl.**Select** selection vector for selecting coefficients to be included in profiling, for example

```
ctl.Select = { 1, 3, 4 };
```

selects the 1st, 3rd, and 4th parameters for profiling.

ctl.**Width** scalar, width of profile table in units of the standard deviations of the parameters. Default = 2.

For description of additional members of the **cmlmtControl** structure see reference for **cmlmt**.

OUTPUT *out* instance of a **cmlmtResults** structure.

out.**Par** instance of a **PV** structure containing the mean of the resampled estimates. Use **pvUnpack** to retrieve matrices and arrays or **pvGetParvector** to get the parameter vector.

out.**Fct** scalar, mean log-likelihood across resamples.

out.**ReturnDescription** string, description of return values.

out.**CovPar** K × K matrix, covariance matrix of resampled parameter estimates.

out.**CovParDescription** string, description of *out*.**CovPar**

out.**NumObs** scalar, number of observations.

out.**ProfileLimits** K × 2 matrix, profilestrap confidence limits. Available only if

cmlmtProfile has been called.

out.**Gradient** K × 1 vector, mean gradient.

out.**NumIterations** scalar, average number of iterations.

out.**ElapsedTime** scalar, average elapsed time of iterations.

out.**Alpha** scalar, probability level of confidence limits. Default = .05.

out.**Title** string, title of run.

out.**Retcode** return code:

- **0** normal convergence
- **1** forced exit
- **2** maximum number of iterations exceeded
- **3** function calculation failed
- **4** gradient calculation failed
- **5** Hessian calculation failed
- **6** line search failed
- **7** functional evaluation failed
- **8** error with initial gradient
- **9** error with constraints
- **10** second update failed
- **11** maximum time exceeded
- **12** error with weights
- **13** quadratic program failed
- **14** equality constraint Jacobian failed
- **15** inequality constraint Jacobian failed
- **16** function evaluated as complex
- **20** Hessian failed to invert

34 data set could not be opened

REMARKS For each pair of the selected parameters, three plots are printed to the screen. Two of these are the profile t trace plots that describe the univariate profiles of the parameters, and one of them is the profile likelihood trace describing the joint distribution of the two parameters. Ideally distributed parameters would have univariate profile t traces that are straight lines, and bivariate likelihood profile traces that are two straight lines intersecting at right angles. This ideal is generally not met by nonlinear models, however, large deviations from the ideal indicate serious problems with the usual statistical inference.

EXAMPLE

```
library cmlmt,pgraph;
#include cmlmt.sdf
#include kern.sdf

graphset;


proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.dataMatrix[1];
    x = d[2].dataMatrix[2:4];

    yh = b0 + x * b;
    res = y - yh;
```

```
    u = y[.,1] ./= 0;

    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.gradient = u.*g1 + (1-u).*g2;
    endif;


    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
```

```
                                -10 10,
                                -10 10,
                                .1 10 };


                    struct DS d0;
                    d0 = dsCreate;
                    d0.dname = "cmlmttobit";

                    out1 = cmlmtProfile(&lpr,p0,d0,c0);

                    call cmlmtPrt(out1);
```

SOURCE cmlmtprofile.src

CMLMTProfileLimits

PURPOSE Computes confidence limits by inversion of the likelihood ratio statistic.

LIBRARY **cmlmt**

FORMAT *out* **= CMLMTProfileLimits(&***modelProc***,***out***,***data***,***ctl***);**

INPUT

&modelProc a pointer to the log-likelihood procedure used to generate results of an estimation by a call to **cmlmt**.

out instance of **cmlmtResults** structure containing results of an estimation generated by a call to **cmlmt**.

data instance of the **DS** data structure used in the call to **cmlmt** that produced the results in out.

ctl an instance of a **cmlmtControl** structure. Normally an

instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user.

For description of the **cmlmtControl** structure see reference for **cmlmt**.

OUTPUT *out* instance of a **cmlmtResults** structure. The member *out*.**ProfileLimits** is filled with the confidence limits by inversion of the likelihood ratio statistic. The remaining members are untouched. For description of additional **cmlmtResutls** members see reference for **cmlmt**.

EXAMPLE

```
library cmlmt,pgraph;
#include cmlmt.sdf
#include kern.sdf

graphset;


proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.dataMatrix[1];
    x = d[2].dataMatrix[2:4];

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;
```

```
    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) + (1-u).*
            (ln(cdfnc(yh/sqrt(s2))));
    endif;

    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.gradient = u.*g1 + (1-u).*g2;
    endif;


    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
              -10 10,
```

```
                        -10 10,
                        .1 10 };

    struct DS d0;
    d0 = dsCreate;
    d0.dname = "cmlmttobit";

    struct cmlmtResults out1;
    out1 = cmlmt(&lpr,p0,d0,c0);

    out1 = cmlmtProfileLimits(&lpr,out1,d0,c0);

    call cmlmtPrt(out1);
```

SOURCE cmlmtpflim.src

CMLMTInverseWaldLimits

PURPOSE Computes confidence limits by inversion of the Wald statistic.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMTInverseWaldLimits**(*out*,*ctl*);

INPUT

out instance of **cmlmtResults** structure containing results of an estimation generated by a call to **cmlmt**.

ctl an instance of a **cmlmtControl** structure. Normally an instance is initialized by calling **cmlmtCreate** and members of this instance can be set to other values by the user.

For description of the **cmlmtControl** structure see reference for **cmlmt**.

OUTPUT *out* instance of a **cmlmtResults** structure. The member *out*.**ProfileLimits** is filled with the confidence limits by inversion of the likelihood ratio statistic. The remaining members are untouched. For description of additional **cmlmtResults** members see reference for **cmlmt**.

EXAMPLE

```
library cmlmt,pgraph;
#include cmlmt.sdf
#include kern.sdf

graphset;


proc lpr(struct PV p, struct DS d, ind);
    local s2,b0,b,y,x,yh,u,res,g1,g2;

    struct modelResults mm;

    b0 = pvUnpack(p,"b0");
    b = pvUnpack(p,"b");
    s2 = pvUnpack(p,"variance");

    y = d.dataMatrix[1];
    x = d[2].dataMatrix[2:4];

    yh = b0 + x * b;
    res = y - yh;
    u = y[.,1] ./= 0;

    if ind[1];
        mm.function = u.*lnpdfmvn(res,s2) +
            (1-u).*(ln(cdfnc(yh/sqrt(s2))));
    endif;
```

```
    if ind[2];
        yh = yh/sqrt(s2);
        g1 = ((res~x.*res)/s2)~((res.*res/s2)-1)/
            (2*s2);
        g2 = ( -( ones(rows(x),1)~x )/sqrt(s2) )~
            (yh/(2*s2));
        g2 = (pdfn(yh)./cdfnc(yh)).*g2;
        mm.gradient = u.*g1 + (1-u).*g2;
    endif;


    retp(mm);

endp;

struct PV p0;
p0 = pvPack(pvCreate,1,"b0");
p0 = pvPack(p0,1|1|1,"b");
p0 = pvPack(p0,1,"variance");

struct cmlmtControl c0;
c0 = cmlmtcontrolcreate;

c0.title = "tobit example";

c0.Bounds = { -10 10,
              -10 10,
              -10 10,
              -10 10,
              .1 10 };

struct DS d0;
d0 = dsCreate;
```

```
d0.dname = "cmlmttobit";

struct cmlmtResults out1;
out1 = cmlmt(&lpr,p0,d0,c0);

out1 = cmlmtInverseWaldLimits(&lpr,out1,d0,c0);

call cmlmtPrt(out1);
```

SOURCE cmlmtpflim.src

ChiBarSq

PURPOSE Computes the chi-bar-statistic and its probability for an hypothesis regarding parameters under constraints.

LIBRARY **ChiBarSq**

FORMAT { **chibar, chiparprob** } = **ChiBarSq**(*out*,*data*,*ctl*,*psi*);

INPUT

out instance of a **cmlmtResults** structure. This structure must contain the results from a **cmlmt** estimation in which the a subset of parameters is set equal to zero using *ctl*.**Active** – start values for those parameters are set to zero, and *ctl*.**Active** is set equal to a vector of zeros and ones in which zeros correspond to the parameters in the hypothesis and ones to the remaining parameters.

data instance of a **DS** structure, the same instances used in the **cmlmt** estimation generating **out**.

ctl instance of a **cmlmtControl** structure. It must contain the constraint specifications under the alternate hypothesis.

psi indices of the set of parameters in the hypothesis. The indices can be determined from the list of the parameters generated by calling **pvGetParnames** on the instance of the **PV** structure in **out**.

OUTPUT *chibar* scalar, chi-bar-square statistic of hypothesis.

chibarprob scalar, probability of **chibar**.

REMARKS **Chibarsq** computes the chi-bar-square statistic for the hypothesis $H(theta) = 0$ vs. $H(theta) >= 0$, where theta is the vector of estimated parameters, and H() is a constraint function of the parameters.

First, the model with $H(theta) = 0$ is estimated by calling **cmlmt**. The simplest way to to this is to use *ctl*.**Active**. *ctl*.**covParType** must also be set to 2 so that the Jacobian or score, i.e., the matrix of first derivatives by observation, is generated.

Next **CHIBARSQ** is called with first argument being the instance of the **cmlmtResults** structure output by the call to **cmlmt**, and second argument the **DS** data structure, and third argument the **cmlmtControl** structure containing the specification of the alternate hypothesis, $H(theta) \geq 0$.

EXAMPLE

```
library cmlmt;
#include cmlmt.sdf

struct DS d0;
d0 = reshape(dsCreate,2,1);

load z0[] = aoi.asc;
z = packr(lagn(251*ln(trimr(z0,1,0)./trimr(
    z0,0,1)),0|1|2|3|4));
d0[1].dataMatrix = z[.,1];
d0[2].dataMatrix = z[.,2:5];
```

```
proc lpr(struct PV p, struct DS d, ind);
    local series,b,omega,arch,const,phi,u2,q,n,h,v;

    struct modelResults mm;

    omega = pvUnpack(p,"omega");
    arch = pvUnpack(p,"arch");
    const = pvUnpack(p,"constant");
    phi = pvUnpack(p,"phi");

    u2 = (d[1].dataMatrix - d[2].dataMatrix *
        phi - const)^2;

    q = rows(arch);
    n = rows(u2);
    h = ones(n,1);
    v = seqa(1,1,q)' + seqa(0,1,n-q);
    h[q+1:rows(h)] = omega + reshape(u2[v],n-q,q) *
        arch;
    h[1:q] = ones(q,1)*meanc(h[q+1:rows(h)]);

    mm.function = -0.5*( (u2 ./ h) + ln(2 * pi) +
        ln(h) );
    retp(mm);

endp;


// First, estimate model where H(theta) = 0

struct cmlmtControl c2;
c2 = cmlmtControlCreate;
c2.Active = ones(6,1) | zeros(3,1);
```

```
    c2.covParType = 2;  // causes Jacobian to be computed
                        // which is needed for chibarsq

    struct PV p1;
    p1 = pvPack(pvCreate,.08999,"constant");
    p1 = pvPack(p1,.25167|-.12599|.09164|.07517,"phi");
    p1 = pvPack(p1,3.22713,"omega");
    p1 = pvPack(p1,0|0|0,"arch");

    struct cmlmtResults out2;
    out2 = cmlmt(&lpr,p1,d0,c2);


// Set up cmlmtControl structure for H(theta) >= 0

    struct cmlmtControl c0;
    c0 = cmlmtcontrolcreate;

    c0.bounds = {
        -10 10,
        -10 10,
        -10 10,
        -10 10,
        -10 10,
        -10 10,
          0 10,
          0 10,
          0 10  };

    psi = { 7, 8, 9 };
    { chibar,chibarprob } = chibarsq(out2,d0,c0,psi);

    print;
    print;
```

```
print "-----------------------------------------";
print "  test of H(arch) = 0 vs. H(arch) >= 0";
print;
print "       chibar " chibar;
print "   chibarprob " chibarprob;
```

SOURCE cmlmtchibar.src

CMLMTControlCreate

PURPOSE Creates a default instance of type **CMLMTControl**.

LIBRARY **cmlmt**

FORMAT *s* = **CMLMTControlCreate;**

OUTPUT *s* instance of type **CMLMTControl**.

SOURCE cmlmtutil.src

CMLMTLagrangeCreate

PURPOSE Creates a default instance of type **CMLMTLagrange**.

LIBRARY **cmlmt**

FORMAT *s* = **CMLMTLagrangeCreate;**

OUTPUT *s* instance of type **CMLMTLagrange**.

SOURCE cmlmtutil.src

CMLMTResultsCreate

PURPOSE Creates a default instance of type **CMLMTResults**.

LIBRARY **cmlmt**

FORMAT *s* = **CMLMTResultsCreate;**

OUTPUT *s* instance of type **CMLMTResults**.

SOURCE cmlmtutil.src

ModelResultsCreate

PURPOSE Creates a default instance of type **ModelResults**.

LIBRARY **cmlmt**

FORMAT *s* = **ModelResultsCreate;**

OUTPUT *s* instance of type **ModelResults**.

SOURCE cmlmtutil.src

CMLMTPrt

PURPOSE Formats and prints the output from a call to **cmlmt**.

LIBRARY **cmlmt**

FORMAT *out* = **CMLMTPrt(out);**

INPUT *out* instance of **cmlmtResults** structure containing results of an estimation generated by a call to **cmlmt**.

OUTPUT *out* the input instance of the **cmlmtResults** structure unchanged.

REMARKS The call to **cmlmt** can be nested in the call to **CMLMTPrt**:

```
call CMLMTPrt(CMLMT(&modelProc,par,data,ctl));
```

SOURCE cmlmtutil.src

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X