Gauss for Econometrics: Optimisation

R.G. Pierse

1. Introduction

Almost every econometric estimation method can be characterised as a problem of the maximisation or minimisation of a nonlinear function, possibly subject to set of constraints on the parameters. Sometimes an explicit analytic solution to the problem can be found. With more complicated problems, however, no analytic solution exists and so the maximisation or minimisation problem must be solved numerically using iterative methods. These methods are known as techniques of *non-linear optimisation*.

The *GAUSS* language version 3.x does not contain an intrinsic procedure for non-linear optimisation (although the forthcoming verion of *GAUSS for Windows* does promise such a procedure). However, three optional *GAUSS* applications modules are available from Aptech: **optmum**, for general unconstrained optimisation, **maxlik** which specifically maximises likelihood functions, and **cml**, for maximising likelihood functions subject to constraints on the parameters. The **cml** module was only released in 1995; before this, *GAUSS* programs had to make do with unconstrained optimisation, using reparameterisation techniques to transform constrained problems into unconstrained problems. These reparameterisation techniques can still be very useful, and are discussed in Section 4.

2. Non-linear Optimisation Techniques

In order to make efficient use of the *GAUSS* optimisation modules, some understanding of the different techniques of non-linear optimisation is useful. The fundamental problem of unconstrained optimisation can be defined as:

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \tag{2.1}$$

where f is a scalar-valued function of a $k \times 1$ vector of parameters, $\boldsymbol{\theta}$. Note that, by convention, the problem is defined as a minimisation problem. A maximisation problem can be converted into a minimisation problem by using the function $-f$ in place of f. It is assumed that f is *continuous* and everywhere *twice differentiable.*

The first-order condition for the solution of (2.1) is given by the set of k equations:

$$\mathbf{g}(\boldsymbol{\theta}) = \frac{\partial f}{\partial \boldsymbol{\theta}} = \mathbf{0}$$

and the second-order condition, for the solution to be a minimum, is that the $k \times k$ *Hessian* matrix

$$\mathbf{H}(\boldsymbol{\theta}) = \frac{\partial^2 f}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}'}$$

is *positive-definite.*

The techniques of non-linear optimisation are iterative: starting from a set of initial values for the parameters, $\boldsymbol{\theta}^0$, each iteration generates a new set of parameter values; iteration continues until a minimum is found. At the ith iteration, a new vector of parameter values is generated from the formula:

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \alpha^i \mathbf{p}^i$$

where $\mathbf{p}$ is a vector specifying the search direction, and α is a scalar representing the step to take along the direction $\mathbf{p}$. Thus at every iteration, two sub-problems need to be solved: choosing a search direction and choosing the step length along that direction.

2.1. Steepest Descent

The method of steepest descent is based on the iteration

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \alpha^i \mathbf{g}(\boldsymbol{\theta}^i)$$

so that the search direction $\mathbf{p}^i$ is taken to be minus the parameter gradient vector $\mathbf{g}(\boldsymbol{\theta}^i)$. This is an intuitively appealing choice of search direction since it is guaranteed to be downhill at the current point $\boldsymbol{\theta}^i$. However, in practice, the method can be very inefficient as can be seen from an example. The order of convergence can be shown to be, at least, *1.* This is called *linear convergence.*

2.2. Conjugate Gradient Methods

Conjugate gradient methods determine the search direction $\mathbf{p}^i$ as a linear function of the negative gradient $-\mathbf{g}(\boldsymbol{\theta}^i)$ and the direction vector in the previous iteration, $\mathbf{p}^{i-1}$:

$$\mathbf{p}^i = -\mathbf{g}(\boldsymbol{\theta}^i) + \beta^{i-1}\mathbf{p}^{i-1}$$

where β^{i-1} is a scalar parameter.The method of Polak and Ribiere (1969) uses the formula

$$\beta^{i-1} = \frac{\mathbf{g}(\boldsymbol{\theta}^i)'(\mathbf{g}(\boldsymbol{\theta}^i) - \mathbf{g}(\boldsymbol{\theta}^{i-1}))}{\mathbf{g}(\boldsymbol{\theta}^{i-1})'\mathbf{g}(\boldsymbol{\theta}^{i-1})}$$

where $\beta^0 = 0$. Conjugate gradient methods make use of first derivatives but not the Hessian matrix of second derivatives. It can be proved that these methods will find the minimum of a quadratic function in, at most, k iterations. This property is known as *quadratic convergence*.

2.3. Newton's Method

Newton's method is based on the iteration

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \alpha^i \mathbf{H}(\boldsymbol{\theta}^i)^{-1}\mathbf{g}(\boldsymbol{\theta}^i)\,.$$

This method uses information from the Hessian matrix to determine the direction $\mathbf{p}^i = -\mathbf{H}(\boldsymbol{\theta}^i)^{-1}\mathbf{g}(\boldsymbol{\theta}^i)$. If the function f is quadratic, then this method will find the minimum in a single iteration. Thus it is *quadratically convergent*. In fact the order of convergence is at least *2*. Since, in the neighbourhood of a minimum, any function will be approximately quadratic, this makes the Newton method a very powerful method. However, it does have some serious drawbacks:

1. The direction $\mathbf{p}^i$ will only be downhill if $\mathbf{H}(\boldsymbol{\theta}^i)$ is positive-definite. Unless the function $f(\boldsymbol{\theta})$ is everywhere convex, there will be parameter regions where $\mathbf{H}(\boldsymbol{\theta}^i)$ is *not* positive-definite so that $\mathbf{p}^i$ can lead to an increase in the function

2. At every iteration, the Hessian matrix needs to be inverted. When k is large, then this is computationally expensive, especially if derivatives are being computed numerically

3. There may be parameter regions where the Hessian is singular and so cannot be inverted. At such points, the algorithm will fail.

2.4. Quasi-Newton methods

Quasi-Newton methods, or *secant* methods, aim to overcome some of the drawbacks of Newton's method, while retaining the powerful property of quadratic convergence. They are based on the iteration:

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \alpha^i \mathbf{G}^i \mathbf{g}(\boldsymbol{\theta}^i)$$

where $\mathbf{G}^i$ is a $k \times k$ matrix that approximates the inverse Hessian $\mathbf{H}(\boldsymbol{\theta}^i)^{-1}$ but which is *guaranteed* to be symmetric positive-definite. In each iteration, $\mathbf{G}^i$ is updated using the formula

$$\mathbf{G}^{i+1} = \mathbf{G}^i + \mathbf{C}^i \,.$$

Different *quasi-Newton* methods use different choices for the correction matrix $\mathbf{C}^i$. The *DFP* method of Davidon (1959) and Fletcher and Powell (1963) uses

$$\mathbf{C}^i = \frac{\mathbf{u}^i \mathbf{u}^{i\prime}}{\mathbf{u}^{i\prime}\mathbf{y}^i} + \frac{\mathbf{v}^i \mathbf{v}^{i\prime}}{\mathbf{v}^{i\prime}\mathbf{y}^i}$$

where $\mathbf{u}^i = \boldsymbol{\theta}^{i+1} - \boldsymbol{\theta}^i$, $\mathbf{y}^i = \mathbf{g}(\boldsymbol{\theta}^{i+1}) - \mathbf{g}(\boldsymbol{\theta}^i)$ and $\mathbf{v}^i = \mathbf{G}^i \mathbf{y}^i$. This is an example of a *rank-two updating formula* since $\mathbf{C}^i$ is a matrix of rank two. Another example of a *quasi-Newton* method using a rank-two updating formula is the Broydon-Fletcher-Goldfarb-Shanno *BFGS* method of Broydon (1970), Fletcher (1970) and Shanno (1970) which uses the formula

$$\mathbf{C}^i = \frac{\mathbf{u}^i \mathbf{u}^{i\prime}}{\mathbf{u}^{i\prime}\mathbf{y}^i} + \frac{\mathbf{v}^i \mathbf{v}^{i\prime}}{\mathbf{v}^{i\prime}\mathbf{y}^i} + \mathbf{v}^{i\prime}\mathbf{y}^i \mathbf{r}^i \mathbf{r}^{i\prime}$$

where

$$\mathbf{r}^i = \frac{\mathbf{u}^i}{\mathbf{u}^{i\prime}\mathbf{y}^i} - \frac{\mathbf{v}^i}{\mathbf{v}^{i\prime}\mathbf{y}^i}.$$

Other *quasi-Newton* methods make use of a *rank-one updating formula*. See Gill, Murray and Wright (1981) or Wolfe (1978) for more details.

2.5. Berndt, Hall, Hall and Hausman

Berndt, Hall, Hall and Hausman (1974), *BHHH*, is an algorithm that uses the iteration:

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \alpha^i \mathbf{G}(\boldsymbol{\theta}^i)^{-1} \mathbf{g}(\boldsymbol{\theta}^i)$$

where $\mathbf{G}(\boldsymbol{\theta}^i)$ is the $k \times k$ matrix defined by

$$\mathbf{G}(\boldsymbol{\theta}^i) = \sum_{j=1}^{T} \mathbf{g}_j(\boldsymbol{\theta}^i)\mathbf{g}_j(\boldsymbol{\theta}^i)' \,.$$

This algorithm is a form of score algorithm that uses the fact that, if f is a likelihood function, then

$$\operatorname*{plim}_{T\to\infty} \frac{1}{T}\mathbf{H}(\boldsymbol{\theta}) = \operatorname*{plim}_{T\to\infty} \frac{1}{T} \sum_{j=1}^{T} \mathbf{g}_j(\boldsymbol{\theta})\mathbf{g}_j(\boldsymbol{\theta})' \,.$$

In general, of course, this condition will not hold.

2.6. Line Search Methods

Given a search direction $\mathbf{p}^i$ determined by one of the methods considered above, line search methods determine the step length α^i so as to minimise the function along this direction. This can be treated as a problem of minimising a function of a single variable, defined by:

$$f^*(\alpha^i) \equiv f(\boldsymbol{\theta}^i + \alpha^i \mathbf{p}^i) \,.$$

Line search methods are themselves iterative. An initial step length is chosen. Then steps are taken along the search direction until a point is found that satisfies the convergence criterion for a minimum. The steps are determined by the particular method selected. Sometimes it is efficient to set quite a loose convergence criterion for line search at first and to tighten up the criterion as the optimisation progresses.

2.6.1. Bracketing the Minimum

Simple line search algorithms work by starting from an initial step-size, and doubling the step until the function stops decreasing. This provides a bracket for the minimum which must then lie somewhere between the last three points evaluated. An alternative very crude procedure starts from a maximum step length and continues halving it until the function decreases. If the initial step length happens to bracket a minimum, then this method will provide an improved estimate of the step length α^i. If not, then it will finish immediately without having improved the initial estimate of the step length.

2.6.2. Golden Section

The method of golden section makes use of the *golden ratio* defined by

$$q = \frac{1}{1+q} \Rightarrow q = \frac{\sqrt{5}-1}{2} = 0.61803\cdots$$

to determine the sequence of steps taken along the search direction. Brent (1972) provides an implementation of this method.

2.6.3. Quadratic or Cubic interpolation

Given three function values at points along the search direction: $f^*(\alpha_1)$, $f^*(\alpha_2)$ and $f^*(\alpha_3)$, a quadratic function can be fitted to these points and the minimum, α_*, determined by interpolation. Similarly, with four function values, or two function values and two derivative values, a cubic function can be fitted. Given the function value $f^*(\alpha_*)$, one of the original points can be dropped and a new polynomial fitted to the remaining points. Dennis and Schnabel (1983) provides an implementation of this method, which usually involves far fewer function evaluations than the method of golden section.

2.7. Efficiency

The efficiency of an optimisation procedure depends both on the number of iterations and the number of function evaluations per iteration. This varies considerably according to the nature of the problem. If a method depends on computation of first or second derivatives, then the cost will depend on whether or not analytic derivatives are available. Where they are available, then typically the cost will be about the same as a single function evaluation. If not, then derivatives will have to be computed numerically which is equivalent to k function evaluations per gradient computation and $k*(k+1)/2$ evaluations per Hessian computation. When k is large, this cost can become prohibitive, so that methods that do not require derivatives can be more efficient, despite needing more iterations.

2.8. Local and Global Minima

The non-linear optimisation methods will, if successful, converge to a minimum of the function $f(\boldsymbol{\theta})$. However, if the function has more than one minimum, then this may be merely a *local* minimum and not necessarily the *global* minimum.

There is no way to ensure that a global minimum is found, and different starting values may well lead to a different solution. One way to check the robustness of the solution, and to check for the existence of multiple minima is to start the optimisation procedure from a range of starting values $\boldsymbol{\theta}^0$ and see if the same solution is reached each time.

2.9. Reducing the Parameter Space

The computational cost of optimisation depends largely on the number of parameters. Any trick that can reduce the number of parameters will reduce the computational burden. Consider the function

$$f(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$$

where the parameter vector $\boldsymbol{\theta}$ has been partitioned into two subsets and, for the second subset, $\boldsymbol{\theta}_2$, the first order conditions

$$\mathbf{g}(\boldsymbol{\theta}_2) = \frac{\partial f}{\partial \boldsymbol{\theta}_2} = \mathbf{0}$$

can be solved to give an explicit expression for $\boldsymbol{\theta}_2$ as a function of the remaining parameters $\boldsymbol{\theta}_1$:

$$\boldsymbol{\theta}_2 = \mathbf{h}(\boldsymbol{\theta}_1)\,.$$

Then, the function f can be written as

$$f(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = f(\boldsymbol{\theta}_1, \mathbf{h}(\boldsymbol{\theta}_1)) = f^*(\boldsymbol{\theta}_1)$$

which is a function only of $\boldsymbol{\theta}_1$. Substituting the explicit solution for the parameters $\boldsymbol{\theta}_2$ into f reduces the dimensions of the parameter space, and hence the cost of minimising the function.

In the context of likelihood functions, this is known as *concentrating the likelihood* and it is very often possible. For example, the log-likelihood function of the *FIML* estimator in the simultaneous equations model with n variables and T observations is given by

$$L(\boldsymbol{\theta}, \boldsymbol{\Sigma}) = -\frac{nT}{2}\log 2\pi - \frac{T}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}\operatorname{tr}(\boldsymbol{\Sigma}^{-1}\mathbf{U}(\boldsymbol{\theta})'\mathbf{U}(\boldsymbol{\theta}))$$

where $\boldsymbol{\theta}$ is a $k \times 1$ vector of coefficients on the equations in the system, and $\boldsymbol{\Sigma}$ is an $n \times n$ positive-definite covariance matrix with $n \times (n+1)/2$ unrestricted

coefficients. Thus the function L has $k + n \times (n+1)/2$ parameters. However, an explicit expression for $\mathbf{\Sigma}$ can be derived by solving the first order conditions

$$\frac{\partial L}{\partial \mathbf{\Sigma}} = 0 \Rightarrow \mathbf{\Sigma} = \frac{1}{T}\mathbf{U}(\boldsymbol{\theta})'\mathbf{U}(\boldsymbol{\theta})$$

so that the *likelihood function* can be *concentrated* to give

$$L^*(\boldsymbol{\theta}) = -\frac{nT}{2}\log 2\pi - \frac{T}{2}\log\left|\frac{1}{T}\mathbf{U}(\boldsymbol{\theta})'\mathbf{U}(\boldsymbol{\theta})\right| - \frac{nT}{2}$$

which is a function of only k parameters.

2.10. Parameter Scaling

Another factor that can influence the computational cost of optimisation is the way in which the parameters are scaled. Ideally, all the parameters should be scaled to be of approximately the same magnitude. If one of the parameters is very much smaller than the others, then the Hessian matrix will have very small components for this parameter, compared with the others. This will lead to rounding errors in calculating the search direction and the optimisation algorithm can get stuck. The condition number of the Hessian, defined by the ratio of its largest to its smallest eigenvalue:

$$\lambda_{\max}/\lambda_{\min}$$

gives information as to whether or not the parameters are appropriately scaled. A large value indicates that some rescaling may be necessary. The condition number can be computed using the *GAUSS* procedure **cond**.

3. Using Optmum

The *GAUSS* application procedure **optmum** performs general non-linear optimisation without parameter constraints. Five solution algorithms are available: the steepest descent method **steep**, Newton's method **newton**, two quasi-Newton methods **dfp** and **bfgs**, and the conjugate gradient method of Polak and Ribiere **prcg**. Three line search routines options are available: a quadratic/cubic interpolation option **stepbt**, the golden section algorithm of **Brent**, and an option **half** which implements the crude step length halving routine described above.

A call to **optmum** takes the form:

$$\{x,f,g,retcode\} = \mathbf{optmum}(\&func,x0);$$

The procedure takes two arguments: a pointer to a procedure *func* that evaluates the function to be minimised, and a $k \times 1$ vector of initial values for the parameters, *x0*. There are four returns: a $k \times 1$ vector of final values for the parameters x, a scalar f which is the value of the function $f(x)$ at the minimum, the $k \times 1$ gradient vector $\mathbf{g}(x)$ at the minimum, and a scalar *retcode* that is the return code from **optmum**. If a minimum is found successfully then *retcode* returns zero; any other value indicates abnormal termination of the procedure. Consult the manual for the meaning of other values of *retcode*.

In addition to the arguments, **optmum** has a large number of optional global variables, all with names starting with '**_op**', that control details of the procedure. For example **_opalgr** controls the solution algorithm to be used and can take one of the six values: **steep**, **bfgs**, **bfgs-sc**, **dfp**, **newton**, or **prcg**. If this global is not explicitly set by the user, then **optmum** uses a default value, in this case **bfgs**. Another global variable, **_opfhess**, returns the final Hessian matrix for those solution algorithms (Newton and quasi-Newton) which make use of the Hessian. All global variables can be reset to their default values by the **optset** command.

A simple call to **optmum** is illustrated in the following code fragment:

```
library optmum;
optset;
_opstmth = "bfgs stepbt";
x0 = zeros(5);
{x,f,g,retcode} = optmum(&func,x0);
H = _opfhess;
```

The first line declares the **optmum** library. This is necessary in order for *GAUSS* to be able to find the **optmum** procedures. The second line calls **optset** to reset all the global variables to their default values. Line 3 sets the global **_opstmth** which controls the solution algorithm. The fourth line sets the initial parameter values *x0* equal to zero. Then, the **optmum** procedure is called to start optimisation. The final line saves the Hessian matrix in variable *H*.

When **optmum** is called, it will in turn call the user-supplied procedure *func* to evaluate the function at different values of the parameter vector x. The form of this procedure is given by the example:

```
proc (1) = func(b);
local s;
s = (y-X*b)'(y-X*b);
retp(s);
endp;
```

The procedure takes a single argument: the parameter vector b, and has a single return: the value of the function to be minimised. In this case, the function returns the sum of squared residuals s for the OLS model. Note that in order to evaluate s, **func** has to use the data matrices y and X. These variables are global to the procedure **func** and must be defined in the main program before **optmum** is called. Generally, function evaluation procedures will need to make use of global variables.

3.1. Derivatives

All the solution methods available in **optmum** make use of first derivatives. In addition the **newton** method uses second order derivatives. By default, **optmum** calculates these derivatives numerically, which is computationally expensive. It is possible, however, to provide **optmum** with user-supplied analytic derivatives, if these are available, in which case convergence can be speeded up considerably. Analytic derivatives are defined in two user-supplied procedures. For example, for the function above, the procedures would be given by:

```
proc (1) = grad(b);
retp((-2*X*(y-X*b))');
endp;
proc (1) = hess(b);
retp(2*X'X);
endp;
```

Procedure **grad** takes the $k \times 1$ parameter vector b as argument and returns the $1 \times k$ vector of derivatives of f with respect to b. Note that this is the transpose of the usual gradient vector $g(b)$. Procedure **hess** returns the $k \times k$ matrix of second order derivatives.

To use the analytic first derivatives in **optmum**, the global variable **_opgdprc** needs to be defined before **optmum** is called. This should be set to be a pointer to the procedure that calculates the derivatives as in

$$\mathbf{\_opgdprc} = \&grad;$$

Similarly, to use analytic second derivatives, define the global **_ophsprc** as in:

$$\mathbf{\_ophsprc} = \&hess;$$

3.2. Switching between algorithms

The **optmum** procedure allows the user to switch between solution algorithms in the course of optimisation. The procedure starts using the algorithm defined in **_opstmth**, and then switches, after **_opditer** (20) iterations, to the algorithm defined in **_opmdmth**. If neither global is set then it uses the values set in **_opalg** (default **bfgs**) and the line search algorithm set in **_opstep** (default **stepbt**).

By setting the three globals **_opstmth**, **_opditer** and **_opmdmth**, the user can control how **optmum** switches between algorithms. In addition, solution and line search algorithms can be switched interactively as the optimisation progresses using the 'ALT-A' and 'ALT-S' keys respectively. At any point, solution can be terminated with the 'ALT-C' key.

4. Dealing With Constraints

Often some of the parameters in the optimisation problem are subject to some form of constraint. For example, variance terms must be greater than zero and autoregressive coefficients must lie in the open interval $(-1, 1)$. There are two ways of dealing with constraints. The first is to make use of a constrained optimisation procedure that solves the problem

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \text{ subject to } \mathbf{r}(\boldsymbol{\theta}) = \mathbf{0} \tag{4.1}$$

where $\mathbf{r}(\boldsymbol{\theta})$ is an $l \times 1$ vector valued function of parameter constraints. *GAUSS* now has such a procedure and this will be considered in the next section.

Alternatively, a constrained optimisation problem generally can be transformed into an unconstrained problem by a suitable reparameterisation. For example,

consider a function of two parameters $f(\rho, \sigma^2)$ where the parameters must satisfy the restrictions:

$$-1 < \rho < 1 \quad \text{and} \quad \sigma^2 > 0\,. \tag{4.2}$$

Define a new pair of parameters ρ^* and σ^* by the transformations

$$\rho = h_1(\rho^*) = \frac{\rho^*}{1 + |\rho^*|} \quad , \quad \rho^* = h_1^{-1}(\rho) = \frac{\rho}{1 - |\rho|}$$

and

$$\sigma^2 = h_2(\sigma^*) = \exp(\sigma^*) \quad , \quad \sigma^* = h_2^{-1}(\sigma^2) = \log(\sigma^2)\,.$$

Note that, for all finite values of ρ^* and σ^*, the functions h_1 and h_2 ensure that ρ and σ^2 will satisfy the restrictions (4.2). The function $f(\rho, \sigma^2)$ can now be rewritten in terms of the transformed parameters as

$$f(\rho, \sigma^2) = f(h_1(\rho^*), h_2(\sigma^*)) = f^*(\rho^*, \sigma^*)$$

where the new parameters ρ^* and σ^* are not subject to any restrictions.

Note that the transformations h_1 and h_2 are by no means unique. In choosing a transformation, one important criterion is *monotonicity* or *invertibility*. Monotonicity is the property that the gradient of the transformation function h has a constant sign. This ensures that the inverse function $h^{-1}$is *unique*. If the inverse function is not unique, then the transformation will cause the optimisation problem to have multiple minima. For example, an alternative transformation to h_2 would be the function $h_2^*(\sigma^*) = (\sigma^*)^2$. However, the inverse of this function, $h_2^{*-1}(\sigma^*) = \pm\sqrt{\sigma^*}$, is not unique so that, if $f^*(\rho^*, \sigma^*)$ is an optimum then so also is $f^*(\rho^*, -\sigma^*)$. Multiple minima cause identification problems and are best avoided if possible.

4.1. Computing standard errors

If a constrained optimisation problem is solved using parameter transformation, then the solution will be in terms of the transformed parameters. The solution values for the untransformed parameters are given by the inverse transformation $\mathbf{h}^{-1}(\boldsymbol{\theta})$. Similarly, the Hessian matrix of second derivatives returned by **optmum** will be with respect to the transformed parameters $\boldsymbol{\theta}$ rather than the original parameters, so that, for the example problem, the *GAUSS* command

$$se = \mathbf{sqrt(diag(invpd(\_opfhess)));}$$

will give standard errors of ρ^* and σ^* rather than ρ and σ^2.

In order to compute standard errors of the original parameters, there are two possibilities. Either, standard errors can be derived analytically, making use of the formula

$$\mathrm{Var}(\mathbf{h}^{-1}(\boldsymbol{\theta})) \simeq \frac{\partial \mathbf{h}^{-1}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \mathrm{Var}(\boldsymbol{\theta}) \frac{\partial \mathbf{h}^{-1}(\boldsymbol{\theta})'}{\partial \boldsymbol{\theta}}$$

or they can be computed numerically using the *GAUSS* command

$$se = \mathbf{sqrt}(\mathbf{diag}(\mathbf{invpd}(\mathbf{hessp}(\&func1,x))));$$

where x is the vector of final values of the original untransformed parameters $\mathbf{h}^{-1}(\boldsymbol{\theta})$ and *func1* is a procedure to evaluate the function *with respect to these original parameters.*

5. Using CML

The *GAUSS* application procedure **cml** performs maximisation of a log-likelihood function allowing for general constraints on the parameters. The constraints can be either nonlinear equality constraints of the form

$$\mathbf{r}(\boldsymbol{\theta}) = \mathbf{0}$$

or inequality constraints

$$\mathbf{r}(\boldsymbol{\theta}) \geq \mathbf{0}\,.$$

In addition, the special cases of linear equality or inequality constraints or bounds constraints of the form

$$\boldsymbol{\theta}_{\min} \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_{\max}$$

are treated separately. Constraints are imposed by setting one (or more) of the global variables: **_cml_EqProc** (for general nonlinear equality constraints), **_cml_IneqProc** (for general nonlinear inequality constraints), **_cml_A** and **_cml_B** (for linear equality constraints), **_cml_C** and **_cml_D** (for linear inequality constraints) or **_cml_Bounds** (for bounds constraints). The first two of these require user-written subroutines specifying the form of the nonlinear constraints (the function $\mathbf{r}(\boldsymbol{\theta})$).

Four solution algorithms are available: Newton's method **newton**, two quasi-Newton methods **dfp** and **bfgs**, and the Berndt, Hall, Hall and Hausman method **bhhh**. The default is **newton**.

A call to **cml** takes the form:

$$\{x,f,g,V,retcode\} = \mathbf{cml}(Data,ind,\&func,x0);$$

The procedure takes four arguments: an $n \times m$ matrix *Data* of data observations to be used for evaluating the log-likelihood which is passed to the user-supplied function *func*, an $m \times 1$ vector of indices *ind* for the variables in *Data* (or the scalar 0), a pointer to a procedure *func* that evaluates the log-likelihood function to be maximised, and a $k \times 1$ vector of initial values for the parameters, *x0*. There are five returns: a $k \times 1$ vector of final values for the parameters x, a scalar f which is the value of the log-likelihood function $f(x)$ at the maximum, the $k \times 1$ final gradient vector $\mathbf{g}(x)$, the $k \times k$ parameter covariance matrix V computed from the final Hessian matrix $\mathbf{H}(x)$, and a scalar *retcode* that is the return code from **cml**. If a maximum is found successfully then *retcode* returns zero; any other value indicates abnormal termination of the procedure. Consult the manual for the meaning of other values of *retcode*.

In addition to its arguments, **cml** has a large number of optional global variables, all with names starting with '**_cml_**', that control details of the procedure. For example **_cml_Algorithm** controls the solution algorithm to be used and can take one of the four values: **bfgs**, **dfp**, **newton**, or **bhhh**. If this global is not explicitly set by the user, then **cml** uses a default value, in this case **newton**. Another global variable, **_cml_CovPar**, determines the type of parameter covariance matrix to return. A value of zero means that the covariance matrix is not computed, whereas **_cml_CovPar** = 2 results in the White heteroscedastic-consistent estimator being computed. All global variables can be reset to their default values by the **cmlset** command.

A simple call to **cml** is illustrated in the following code fragment:

```
library cml;
cmlset;
_cml_Bounds = {-.999 .999,1e-6 500};
theta0 = 0|1;
{theta,f,g,V,retcode} = cml(Data,0,&mlfunc,theta0);
```

The first line declares the **cml** library. The second line calls **cmlset** to reset all the global variables to their default values. Line 3 sets the global **_cml_Bounds** which defines lower and upper bounds for the parameters. The fourth line sets

the initial parameter values *theta0* equal to zero and one respectively. Finally, the **cml** procedure is called to start optimisation.

cml in turn calls the user-supplied procedure *mlfunc* to evaluate the likelihood function at different values of the parameter vector *theta*. The form of this procedure is given by the example:

```
proc (1) = mlfunc(theta,Data);
local k,y,X,b,s2,llf ;
k= rows(theta);
y = Data[.,1]; X = Data[.,2:k];
b = theta[1:k-1]; s2 = theta[k];
llf = -0.5*(ln(2*pi*s2)+((y-X*b)^2)/s2);
retp(llf);
endp;
```

The procedure takes two arguments: the parameter vector *theta*, and an $n \times m$ data matrix *Data*. It returns an $n \times 1$ vector of the log-likelihood for each observation. In this example, the function returns the *unconcentrated* log-likelihood of the *OLS* model where the parameter vector $\boldsymbol{\theta}$ is ordered as $\boldsymbol{\theta}' = (\boldsymbol{\beta}' : \sigma^2)'$ and the data matrix *Data* is ordered as $Data = (\mathbf{y} : \mathbf{X})$. The **_cml_Bounds** line in the main program calls **cml** to maximise this log-likelihood with two parameters β and σ^2 with the constraints that $-0.999 \leq \beta \leq 0.999$ and that $0.000001 \leq \sigma^2 \leq 500$.

5.1. Derivatives

It is possible to supply **cml** with analytic derivatives to speed up convergence. Analytic derivatives are defined in two user-supplied procedures. For example, for the function **mlfunc** above, the procedures would be given by:

```
proc (1) = mlgrad(theta,Data);
local k,y,X,u,s2,gb,gs2;
k= rows(theta);
y = Data[.,1]; X = Data[.,2:k];
u = y-X*theta[1:k-1]; s2 = theta[k];
```

```
gb = (X.*(u*ones(1,k-1)))/s2;
gs2 = 0.5*((u/s2)^2-1/s2);
retp(gb~gs2);
endp;
proc (1) = mlhess(theta,Data);
local T,k,y,X,u,s2,hb,hbs,hs2;
T = rows(Data); k= rows(theta);
y = Data[.,1]; X = Data[.,2:k];
u = y-X*theta[1:k-1]; s2 = theta[k];
hb = -(X*X)/s2; hbs = -X'u/(s2^2);
hs2 = -0.5*(T+u'u/(s2^3));
retp((hb~hbs)|(hbs'~hs2));
endp;
```

Procedure **mlgrad** takes two arguments: the $k \times 1$ parameter vector *theta* and the $n \times m$ data matrix *Data*, and returns the $n \times k$ matrix of derivatives of *each observation* of the likelihood function f with respect to *theta*. Procedure **mlhess** returns the $k \times k$ matrix of second order derivatives.

To use analytic first derivatives in **cml**, the global variable **_cml_GradProc** needs to be defined before **cml** is called. This should be set to be a pointer to the procedure that calculates the derivatives as in

```
_cml_GradProc = &mlgrad;
```

Similarly, to use analytic second derivatives, define the global **_cml_HessProc** as in:

```
_cml_HessProc = &mlhess;
```

References

[1] Berndt, E., B.H. Hall, R.E. Hall and J.A. Hausman (1974), 'Estimation and inference in nonlinear structural models', *Annals of Economic and Social Measurement*, 3, 653–665.

[2] Brent, R.P. (1972), *Algorithms for Minimisation Without Derivatives*, Prentice Hall, Englewood Cliffs, NJ.

[3] Broydon, C.G. (1970), 'The convergence of a class of double-rank minimisation algorithms: 1. General considerations', *Journal of the Institute of Mathematics and its Applications*, 6, 76–90.

[4] Dennis, J.E. and R.B. Schnabel (1983), *Numerical Methods for Unconstrained Optimisation and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, NJ.

[5] Davidon, W.C. (1959), 'Variable metric method for minimisation', *Research and Development Report ANL-5990*, US Atomic Energy Commission,Argonne National Laboratories.

[6] Fletcher, R. (1970), 'A new approach to variable metric algorithms', *The Computer Journal*, 13, 317–322.

[7] Fletcher, R. and M.J.D. Powell (1963), 'A rapidly convergent descent method for minimisation', *The Computer Journal*, 6, 163–168.

[8] Gill, P.E., W. Murray and M.H. Wright (1981), *Practical Optimization*, Academic Press, New York.

[9] Polak E. and G. Ribiere (1969), 'Note sur la convergence de methodes de directions conjugées', *Revue Française de Information et Récherches Operation*, 16-R1, 35–43.

[10] Shanno, D.F. (1970), 'Conditioning of quasi-Newton methods for function minimisation', *Mathematics for Computation*, 24, 647–656.

[11] Wolfe, M.A. (1978), *Numerical Methods for Unconstrained Optimization*, Van Nostrand Reinhold, New York.