The GAUSS Constrained Optimization Application

Ronald Schoenberg
Aptech Systems, Inc.

April 7, 1995

For further information about the software package described in this article, or about the GAUSS programming language contact: Aptech Systems, Inc., 23804 SE Kent-Kangley Rd, Maple Valley, WA, 98038, USA, +1-206-432-7855 (voice), +1-206-432-7832 (fax), info@aptech.com (e-mail),

A postscript file of the current version of this manuscript is available via anonymous ftp from `ftp.u.washington.edu : /public/rons`

1 Introduction

CONSTRAINED OPTIMIZATION is a new applications module produced by Aptech Systems, Inc. for the estimation of the parameters of models with general constraints on the parameters. Five types of constraints are provided for:

linear equality

nonlinear equality

linear inequality

nonlinear inequality

bounds

CONSTRAINED OPTIMIZATION is a general purpose program for optimization with constraints. That is, you provide a procedure for computing the function and *CONSTRAINED OPTIMIZATION* computes the estimated coefficients using an iterative method. Only the function and a starting point are required. Optionally, you may provide matrices for the linear constraints and functions for the nonlinear constraints. To make the computations more efficient, you may also provide functions for the gradient or Hessian of the log-likelihood, as well as functions for the Jacobians of the nonlinear constraints. If you do not provide these functions, the derivatives are computed numerically.

A powerful, general program for minimizing functions is made possible, even convenient, as a result of the functions and operators available in the **GAUSS** programming language. The intuitive syntax together with the *CONSTRAINED OPTIMIZATION* package reduces even the most complicated estimation problems to a straightforward writing of a **GAUSS** procedure for computing the function. Moreover, **GAUSS**'s legendary computing speed renders tractable many problems that might have seemed impossible.

This article is a description of the various features of the **GAUSS** *CONSTRAINED OPTIMIZATION* application. If you have any further questions about this software, contact the author at `info@aptech.com`.

2 The Nonlinear Programming Problem

CONSTRAINED OPTIMIZATION is a set of procedures for the solution of the general nonlinear programming problem:

$$minF(\theta)$$

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.

$F(\theta)$ must be convex and have first and second derivatives with respect to the parameters, and the matrix of second derivatives must be positive semi-definite.

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

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

where δ is a $k \times 1$ direction vector, and α a scalar step length.

2.0.1 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

$$\begin{array}{ll} min & \frac{1}{2}\delta'\Sigma(\theta_k)\delta + \Psi(\theta_k)\delta \\ s.t. & \dot{G}(\theta_k)\delta + G(\theta_k) = 0 \\ & \dot{H}(\theta_k)\delta + H(\theta_k) \geq 0. \end{array}$$

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.

2.0.2 line search

Define the *merit* function

$$m(\theta) = F + \max \mid \kappa \mid \sum_j \mid g_j(\theta) \mid - \max \mid \lambda \mid \sum_j \min(0, h_j(\theta))$$

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

The line search finds a value of α that minimizes or decreases $m(\theta_k + \alpha\delta)$.

2.1 Derivatives

The SQP method requires the calculation of a Hessian, Σ, and various gradients and Jacobians, Ψ, $\dot{G}(\theta)$, and $\dot{H}(\theta)$. *CONSTRAINED OPTIMIZATION* computes these numerically if procedures to compute them are not supplied.

If you provide a proc for computing Ψ, the first derivative of L, *CONSTRAINED OPTIMIZATION* uses it in computing Σ, the second derivative of L, i.e., Σ is computed as the Jacobian of the gradient. This improves the computational precision of the Hessian by about four places. The accuracy of the gradient is improved and thus the iterations converge in fewer iterations. Moreover, the convergence takes less time because of a decrease in function calls - the numerical gradient requires k function calls while an analytical gradient reduces that to one.

2.2 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 *CONSTRAINED OPTIMIZATION* for updating the Hessian rather than computing it directly at each iteration. These algorithms as well as the step length methods may be modified during the execution of *CONSTRAINED OPTIMIZATION*.

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**.

2.2.1 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 *CONSTRAINED OPTIMIZATION*, 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.

2.3 Line Search Methods

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

$$\theta_{k+1} = \theta_k + \alpha\delta$$

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

$$m(\theta_k + \alpha\delta).$$

Given θ and d, this is a function of a single variable α. Line search methods attempt to find a value for α 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 **_co_RandRadius** is greater than zero).

2.3.1 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 $m(\theta_k + \alpha\delta)$ and computes an α 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.

2.3.2 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 α. 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.

2.3.3 HALF

This method first computes $m(x+d)$, i.e., sets $\alpha = 1$. 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 effective in that it is not very likely to find the step length that decreases m the most.

2.4 Random Search

If the line search fails, i.e., no α is found such that $m(\theta_k + \alpha\delta) < m(\theta_k)$, then a search is attempted for a random direction that decreases the function. The radius of the random search is fixed by the global variable, **_co_RandRadius** (default = .01), times a measure of the magnitude of the gradient. *CONSTRAINED OPTIMIZATION* makes **_co_MaxTry** attempts to find a direction that decreases the function, and if all of them fail, the direction with the smallest value for m is selected.

The function should never increase, but this assumes a well-defined problem. In practice, many functions are not so well-defined, and it often is the case that convergence is more likely achieved by a direction that puts the function somewhere else on the hyper-surface even if it is at a higher point on the surface. Another reason for permitting an increase in the function here is that halting the minimization altogether is the only alternative if it is not at the minimum, and so one might as well retreat to another starting point. If the function repeatedly increases, then you would do well to consider improving either the specification of the problem or the starting point.

2.5 Active and Inactive Parameters

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

3 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.

3.1 Linear Equality Constraints

Linear equality 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 the *CONSTRAINED OPTIMIZATION* globals, **_co_A** and **_co_B**, respectively. For example, to constrain the first of four parameters to be equal to the third,

```
_co_A = { 1 0 -1 0 };
_co_B = { 0 };
```

3.2 Linear Inequality Constraints

Linear inequality 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 the *CONSTRAINED OPTIMIZATION* globals, **_co_C** and **_co_D**, respectively. 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:

```
_co_C = { 1 0 -1 0,
           0 1  0 1 };
_co_D = { 0,
          10 };
```

3.3 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 pointer to the user-supplied function to the **GAUSS** global, **_co_EqProc**.

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

```
proc eqp(b);
   retp(b'b - 1);
endp;
_co_EqProc = &eqp;
```

3.4 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 user-supplied function to the **GAUSS** global, **_co_IneqProc**.

For example, suppose you wish to constrain a covariance matrix to be positive definite, the lower left nonredundant portion of which is stored in elements `r:r+s` of the parameter vector:

```
proc ineqp(b);
    local v;
    v = xpnd(b[r:r+s]); /* r and s defined elsewhere */
    retp(minc(eigh(v)) - 1e-5);
endp;
_co_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.

3.5 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 *CONSTRAINED OPTIMIZATION* global, **_co_Bounds**.

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

To bound four parameters:

```
_co_Bounds = { -10 10,
               -10  0,
                 1 10,
                 0  1 };
```

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

```
_co_Bounds = { -50 50 };
```

is all that is necessary.

3.6 Example

The following example is taken from Hock and Schittkowski (1981, page 76): minimize

$$F(\theta) = (\theta_1 - \theta_2)^2 + (\theta_2 + \theta_3 - 2)^2 + (\theta_4 - 1)^2 + (\theta_5 - 1)^2$$

subject to equality constraints

$$\begin{aligned} \theta_1 + 3\theta_2 &= 0 \\ \theta_3 + \theta_4 - 2\theta_5 &= 0 \\ \theta_2 - \theta_5 &= 0 \end{aligned}$$

and bounds

$$-10 \leq \theta_i \leq 10, \quad i = 1, \cdots, 5$$

The starting point

$$\theta_{st} = \begin{pmatrix} 2 & 2 & 2 & 2 & 2 \end{pmatrix}$$

is infeasible, i.e, doesn't satisfy the equality constraints. This usually won't be a problem. Failing to satisfy inequality constraints can sometimes be a problem however, in particular when the inequality constraints are enforcing well-defined behavior. Therefore, feasibility of starting points should be considered when there are inequality constraints.

The published solution is

$$\begin{aligned}\theta_{sol} &= \left[\begin{array}{ccccc} \frac{-33}{43} & \frac{11}{43} & \frac{27}{43} & \frac{-5}{43} & \frac{11}{43} \end{array}\right] \\ &= \left[\begin{array}{ccccc} -.7674 & 0.2558 & 0.6279 & -.1163 & 0.2558 \end{array}\right]\end{aligned}$$

The procedure for solving this problem is

```
library co,qp;
#include co.ext;

coset;

proc fct(x);
   retp((x[1]-x[2])^2+(x[2]+x[3]-2)^2+(x[4]-1)^2+(x[5]-1)^2);
endp;

proc eqp(x);
    local result;
    result = zeros(3,1);
    result[1] = x[1] + 3*x[2];
    result[2] = x[3] + x[4] - 2 * x[5];
    result[3] = x[2] - x[5];
    retp(result);
endp;

_co_Bounds = { -10 10 };
_co_EqProc = &eqp;

start = { 2,2,2,2,2 };

output file = co6.out reset;

{ x,f,g,ret } = co( &fct,start );

call coprt(x,f,g,ret);

print;
print "Equality Lagrangeans";
print vread(_co_Lagrange,"nlineq");
```

```
print;
print "Boundary Lagrangeans";
print vread(_co_Lagrange,"bounds");
```

The output from this run is

```
=================================================================
 CO Version 1.0.0                             2/20/95  10:28 am
=================================================================

return code =    0
normal convergence

Value of objective function        4.093023

Parameters    Estimates      Gradient
-----------------------------------------
P01            -0.7674       -2.0465
P02             0.2558       -0.1861
P03             0.6279       -2.2326
P04            -0.1163       -2.2326
P05             0.2558       -1.4884

Number of iterations        4
Minutes to convergence     0.00650

Equality Lagrangeans

-2.0465
-2.2326
 5.9535

Boundary Lagrangeans
.
```

The missing value for the Lagrangeans associated with the bounds indicates that none of the bounds constraints are active.

4 Gradients

4.1 Analytical Gradient

To increase accuracy and reduce time, you may supply a procedure for computing the gradient, $\Psi(\theta) = \partial L / \partial \theta$, analytically.

This procedure has two input arguments, a $K \times 1$ vector of parameters and an $N_i \times L$ submatrix of the input data set. The *CONSTRAINED OPTIMIZATION* global, **_co_GradProc** is then set to the pointer to that procedure.

In practice, unfortunately, much of the time spent on writing the gradient procedure is devoted to debugging. To help in this debugging process, *CONSTRAINED OPTIMIZATION* can be instructed to compute the numerical gradient along with your prospective analytical gradient for comparison purposes. In the example above this is accomplished by setting **_co_GradCheckTol** to a small nonzero value.

4.2 User-Supplied Numerical Gradient

You may substitute your own numerical gradient procedure for the one used by *CONSTRAINED OPTIMIZATION* by default. This is done by setting the *CONSTRAINED OPTIMIZATION* global **_co_UserGrad** to a pointer to the procedure.

4.3 Analytical Hessian

You may provide a procedure for computing the Hessian, $\Sigma(\theta) = \partial^2 F / \partial\theta\partial\theta'$. This procedure has one argument, the $K \times 1$ vector of parameters, and returns a $K \times K$ symmetric matrix of second derivatives of the objection function with respect to the parameters.

The pointer to this procedure is stored in the global variable **_co_HessProc**.

In practice, unfortunately, much of the time spent on writing the Hessian procedure is devoted to debugging. To help in this debugging process, *CONSTRAINED OPTIMIZATION* can be instructed to compute the numerical Hessian along with your prospective analytical Hessian for comparison purposes. To accomplish this **_co_GradCheckTol** is set to a small nonzero value.

4.4 User-Supplied Numerical Hessian

You may substitute your own numerical Hessian procedure for the one used by *CONSTRAINED OPTIMIZATION* by default. This done by setting the *CONSTRAINED OPTIMIZATION* global, **_co_UserHess** to a pointer to the procedure. This procedure has two input arguments, a pointer to the function, and a $K \times 1$ vector of parameters. It must return a K $\times$ K matrix which is the estimated Hessian evaluated at the parameter vector.

4.5 Analytical Nonlinear Constraint Jacobians

When nonlinear equality or inequality constraints have been placed on the parameters, the convergence can be improved by providing a procedure for computing their Jacobians, i.e., $\dot{G}(\theta) = \partial G(\theta)/\partial\theta$ and $\dot{H}(\theta) = \partial H(\theta)/\partial\theta$.

These procedures have one argument, the $K \times 1$ vector of parameters, and return an $M_j \times K$ matrix, where M_j is the number of constraints computed in the corresponding constraint function. Then the *CONSTRAINED OPTIMIZATION* globals, **_co_EqJacobian** and **_co_IneqJacobian** are set to pointers to the nonlinear equality and inequality Jacobians, respectively.

4.6 Example

The following example illustrates furnishing procedures for computing gradients and Jacobians. It is taken from Hock and Schittkowski (1981, page 55). The function to be minimized is

$$F(\theta) = (\theta_1 + 3\theta 2 + \theta_3)^2 + 4(\theta_1 - \theta 2)^2$$

subject to the inequality constraint,

$$6\theta_2 + 4\theta_3 - \theta_1^3 - 3 \geq 0$$

and the equality constraint

$$1 - \sum_{i=1}^{3} \theta_i = 0$$

and bounds

$$\theta_i \geq 0, \quad i = 1, 2, 3.$$

The starting point

$$[\ .1 \quad .7 \quad .2\ ]$$

is feasible. The published solution is

$$\begin{bmatrix} 0 & 0 & 1. \end{bmatrix}$$

The procedure for solving this problem is

```
library co;
#include co.ext;
coset;

proc fct(x);
   retp( (x[1] + 3*x[2] + x[3])^2 + 4*(x[1] - x[2])^2 );
endp;

proc ineqp(x);
    retp(6*x[2] + 4*x[3] - x[1]^3 - 3);
endp;

proc ineqj(x);
   retp(-3*x[1]^2~6~4);
endp;

proc eqp(x);
    retp(1-sumc(x));
endp;

proc eqj(x);
   retp(-ones(1,3));
endp;


proc gp(x);
   local g, t;
   g = zeros(3,1);
   g[1] = 10*x[1] - 2*x[2] + 2*x[3];
   g[2] = -2*x[1] + 26*x[2] + 6*x[3];
   g[3] = 2*x[1] + 6*x[2] + 2*x[3];
   retp(g);
endp;

proc hsp(x);
   local h;
   h = zeros(3,3);
   h[1,1] = 10;
   h[1,2] = -2;
```

```
   h[1,3] = 2;

   h[2,1] = -2;
   h[2,2] = 26;
   h[2,3] = 6;

   h[3,1] = 2;
   h[3,2] = 6;
   h[3,3] = 2;
   retp(h);
endp;

_co_Bounds = { 0 1e256 };

start = { .1, .7, .2 };

_co_GradProc = &gp;
_co_HessProc = &hsp;
_co_IneqProc = &ineqp;
_co_IneqJacobian = &ineqj;
_co_EqProc = &eqp;
_co_EqJacobian = &eqj;

{ x,f,g,ret } = co( &fct,start );

call coprt(x,f,g,ret);

print;
print;
print "published solution";
print "     0       0      1";

print;
print "nonlinear equality Lagrangeans";
print vread(_co_Lagrange,"nlineq");
print;
print "nonlinear inequality Lagrangeans";
print vread(_co_Lagrange,"nlinineq");
print;
print "boundary Lagrangeans";
print vread(_co_Lagrange,"bounds");
```

The output from this run is

```
===============================================================
 CO Version 1.0.0                          2/20/95   1:05 pm
===============================================================

return code =    0
normal convergence

Value of objective function        1.000000

Parameters    Estimates    Gradient
-----------------------------------------
P01              0.0000       2.0000
P02              0.0000       6.0000
P03              1.0000       2.0000

Number of iterations        3
Minutes to convergence     0.00267


published solution
     0       0      1

nonlinear equality Lagrangeans
-2.0000

nonlinear inequality Lagrangeans
 0.0000

boundary Lagrangeans
.
```

The missing value for the boundary Lagrangeans indicates that they are all inactive. The nonlinear inequality Lagrangean is zero but is not set to scalar missing. This indicates that the constraint was active at some point in the iterations but was not active at convergence.

References

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

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

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

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

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

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

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.