Constrained Optimization

for GAUSS™

Version 2.0

Aptech Systems, Inc.

Documentation Revision: 1502 March 8, 2010

Part Number: 000013

Contents

Chapter 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.

1.3 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.

Chapter 2

Constrained Optimization

written by

Ronald Schoenberg

This module contains a set of procedures for the solution of the nonlinear programming problem.

2.1 Getting Started

GAUSS 3.2.8+ is required to use these routines.

2.1.1 README Files

The file **README.co** contains any last minute information on this module. Please read it before using the procedures in this module.

2.1.2 Setup

In order to use the procedures in the *CONSTRAINED OPTIMIZATION* Module, the **CO** library must be active. This is done by including `co` in the **library** statement at the top of your program or command file:

```
    library co,pgraph;
```

This enables **GAUSS** to find the *CONSTRAINED OPTIMIZATION* procedures. If you plan to make any right hand references to the global variables (described in the *REFERENCE* section), you also need the statement:

```
    #include co.ext;
```

Finally, to reset global variables in succeeding executions of the command file the following instruction can be used:

```
    coset;
```

This could be included with the above statements without harm and would insure the proper definition of the global variables for all executions of the command file.

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

_co_ver	3×1 matrix, the first element contains the major version number of the *CONSTRAINED OPTIMIZATION* 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.

2.1.3 Converting OPTMUM Command Files

The **CO** module includes a utility for processing command files to change **OPTMUM** global names to **CO** global names and vice versa. This utility is a standalone executable program that is called outside of **GAUSS**. The format for this call is,

chgvar *control_file target_directory file...*

The *control_file* is an ASCII file containing a list of the symbols to change in the first column and the new symbol names in the second column. The **CO** module comes with one control_file:

opt3toco	**OPTMUM** to **CO**

chgvar processes each file and writes a new file with the same name in the target directory.

A common use for **chgvar** is translating a command file that had been used before with **OPTMUM** to one that can be run with **CO**. For example:

```
mkdir new
chgvar opt3oco new opt*.cmd
```

This would convert every file matching `opt*.cmd` in the current directory and create a new file with the same name in the new directory.

The reverse translation is also possible. However, there are some global names in **CO** that don't have a corresponding global in **OPTMUM**, and in these cases no translation occurs.

Further editing of the file may be necessary after processing by **chgvar**.

You may edit the control files or create your own. They are ASCII files with each line containing a pair of names, the first column being the old name, and the second column the new name.

2.2 The Nonlinear Programming Problem

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

$$min\, F(\theta)$$

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 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_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

$$\begin{aligned}\Sigma(\theta) &= \frac{\partial^2 F}{\partial\theta\partial\theta'}\\ \Psi(\theta) &= \frac{\partial F}{\partial\theta}\end{aligned}$$

and the Jacobians

$$\begin{aligned}\dot{G}(\theta) &= \frac{\partial G(\theta)}{\partial\theta}\\ \dot{H}(\theta) &= \frac{\partial H(\theta)}{\partial\theta}\end{aligned}$$

For the purposes of this exposition, and without lost 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

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

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

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

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

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 Lagrangean coefficients of the inequality constraints.

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

2.2.1 Derivatives

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

If you provide a proc for computing Ψ, the first derivative of L, **CO** 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.

2.2.2 The Quasi-Newton 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 **CO** 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 **CO**.

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 quasi-Newton algorithm should do nearly as well as Newton iteration with much less computation.

There are two basic types of quasi-Newton 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**.

Quasi-Newton 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 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 quasi-Newton 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 **CO**, 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.

2.2.3 Scaled Quasi-Newton Methods

In the unscaled versions of the quasi-Newton methods, the estimate of the Hessian is stored as a Cholesky factorization. The scaled quasi-Newton methods (Han, 1986), store a different type of factorization, called a conjugate gradient factorization. In this factorization,

$$\Sigma = ZZ'$$

where Z is a square matrix. Z is highly overdetermined, that is, there are many matrices that satisfy this relationship. Han (1986) describes a technique that chooses a matrix that is orthogonal and the columns of which can be scaled to reduce the effects of ill-conditioning.

The scaled methods are more time-consuming than their unscaled counterparts, and thus they won't be prefered for general use. However, they may produce better results for ill-conditioned models.

2.2.4 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. **CO** 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 **_co_RandRadius** is greater than zero).

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 a 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.

2.2.5 Trust Regions

The idea behind the trust region methods (Dennis and Schnabel, 1994) is to find a new direction, δ, that minimizes

$$m(\theta_t + \delta) = F(\theta_c) + \Psi(\theta_t)\delta + \frac{1}{2}\delta'\Sigma(\theta_t)\delta$$

subject to

$$\| \delta \| \leq \Delta$$

where Δ is the *trust radius.*

In the usual implementation this is solved using approximate methods. In the context of the SQP method, however, this problem can be solved exactly at each iteration by placing constraints on the direction being computed in the quadratic programming step of the SQP algorithm. If we define the norm to be a one-norm, the constraints amount to simple bounds on the direction vector δ. That is, we minimize $m(\theta + \delta)$ such that

$$-\Delta \leq \delta \leq \Delta$$

The trust region method can be invoked by setting

```
_co_Trust = 1;
```

or by adding "trust" to the options list,

```
_co_Options = { trust };
```

You may also toggle the trust region method by pressing "t" on the keyboard during the iterations.

The default region size is .01. This may be modifed by setting the global **_co_TrustRegion** to some other value.

2.2.6 Switching Algorithms

For some problems, descent methods perform differently over different regions of the descent. There might be an advantage, therefore, to be able to switch algorithms at some point of the descent. For this reason a global is provided to give control over algorithm switching.

To activate algorithm switching **_co_Switch** must be set to a three column vector. The first element is a number indicating which algorithm to which the iterations will be switched, where

1	BFGS - Broyden, Fletcher, Goldfarb, Shanno method
2	DFP - Davidon, Fletcher, Powell method
3	NEWTON - Newton-Raphson method
4	scaled BFGS
5	scaled DFP

The second element is amount of change in the function less than which the algorithm will be switched. The idea here is that the algorithm should be switched when the descent progress is too small, that is, when the function changes less than some value.

The third element is the iteration number at which the algorithm is to be switched.

The SQP method used in **CO** works best with the NEWTON descent algorithm. BFGS doesn't work as well but is much faster. BFGS will work better in the region of the optimum and therefore it is often useful to start out with the NEWTON method and switch to the BFGS method when either the descent progress is small or after a number of iterations. Useful values for lack of progress and number of iterations can be determined by looking over results from a run without switching. Choose some value for lack of progress or number of iterations that allow the initial algorithm to have its chance to work.

2.2.7 Random Search

If the line search fails, i.e., no ρ is found such that $m(\theta_t + \rho\delta) < m(\theta_t)$, 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. **CO** 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 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.

2.2.8 Active and Inactive Parameters

The **CO** 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.

2.3 Managing Optimization

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

For best results, therefore, you want to prepare the problem so that the model is well-specified, and properly scaled, and for which 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.

2.3.1 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, **CO** 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.

2.3.2 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 **CO** has difficulty determining the optimal values since a large region of the function appears virtually flat to **CO**. 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.

2.3.3 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.

2.3.4 Diagnosis

When the optimization is not proceeding well, it is sometimes useful to examine the function, the gradient Ψ , the direction δ, the Hessian Σ, the parameters θ_t, or the step length ρ, during the iterations. The current values of these matrices can be printed out or stored in the global **_co_Diagnostic** by setting **_co_Diagnostic** to a nonzero value. Setting it to 1 causes **CO** to print them to the screen or output file, 2 causes **CO** to store then in **_co_Diagnostic**, and 3 does both.

When you have selected **_co_Diagnostic** = 2 or 3, **CO** inserts the matrices into **_co_Diagnostic** using the **vput** command. the matrices are extracted using the **vread** command. For example,

```
_co_Diagnostic = 2;
call COPrt(CO("tobit",0,&lpr,x0));
h = vread(_co_Diagnostic,"hessian");
d = vread(_co_Diagnostic,"direct");
```

The following table contains the strings to be used to retrieve the various matrices in the **vread** command:

matrix	string
θ	"params"
δ	"direct"
Σ	"hessian"
Ψ	"gradient"
ρ	"step"

2.4 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.

2.4.1 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 **CO** 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 };
```

2.4.2 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 **CO** 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 };
```

2.4.3 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 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;
```

2.4.4 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 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.

2.4.5 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 **CO** 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.

2.5 Gradients

2.5.1 Analytical Gradient

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

2.5.2 User-Supplied Numerical Gradient

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

2.5.3 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, **CO** 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.

2.5.4 User-Supplied Numerical Hessian

You may substitute your own numerical Hessian procedure for the one used by **CO** by default. This done by setting the **CO** 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.

2.5.5 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 **CO** globals, **_co_EqJacobian** and **_co_IneqJacobian** are set to pointers to the nonlinear equality and inequality Jacobians, respectively.

2.6 Minimization Problems and Suggested Solutions

2.6.1 "feasible step length could not be found" error

The iterations halt abnormally with a "feasible step length could not be found" error.

The line search tests each potential direction for feasibility. This is necessary because one or more of the constraints may be preventing undefined calculations in the function. Usually **CO** can recover from any attempt to go outside the boundary but sometimes can't, and the error is generated.

If none of the constraints are protecting the function from undefined calculations, then the test can be removed. The line search may produce a direction outside of the constraint boundaries, but other parts of the algorithm will bring it back. In that case the global, **_co_FeasibleTest**, can be set to zero, turning off the test for feasibility.

2.6.2 Directions become very large and step lengths become very small

On occasion, descent sequences become an oscillation between large and small directions alternating with small and large step lengths. The NEWTON method in particular can do this, especially from a poor start. The algorithm produces a large step because it is far from the optimum. This extrapolation is based on a quadratic approximation to the function which may not be appropriate for the actual surface. The line search then has difficulty finding an actual descent with that direction, and ends up with a very small step length to compensate for the inappropriate extrapolation. This often results in a very small direction on the next iteration and a normal step length.

This oscillation of direction and step length usually result in a very slow descent. The fundamental cause of this problem is the departure of the objective function surface from the quadratic that is assumed by the algorithms. However, the oscillation can be prevented, and a more rapid descent generated, if the trust region is activated. The directions are contained within the trust radius and the step lengths are kept to normal size, and the descent is restored.

To activate the trust region method, set **_co_Trust** to a nonzero value. If progress continues to be slow, set the trust radius to a larger value by setting **_co_TrustRadius** to that value. The default value is .01.

The trust region method can also be interactively toggled on and off during the iterations by pressing "t" on the keyboard.

2.6.3 One of the parameters gets very large and CO bombs

This indicates that the function is underconstrained. If the increase in size is rapid, the Hessian is probably ill-conditioned. In effect, the objective surface is flattening out along one of the coefficients, turning into a valley, and the descent is wandering down the valley.

This can be solved by constraining the errant parameter. Either a boundary could be placed, or a more complicated constraint that brings the Hessian back into condition.

2.6.4 How to interpret the constraint number when CO halts with "constraint no. inconsistent"

This error is written when the quadratic programming solve fails due to an inconsistency in the constraints. A constraint inconsistency means that there is no value for a coefficient that can satisfy all of the constraints. This would occur if, for example, a coefficient was constrained in one instance to be greater than zero and in another to be less than zero.

The constraint number referred to in the error message is to a list of the constraints in the following order

1	inactivated coefficients using **_co_Active**, if any
2	linear equality constraints, if any
3	nonlinear equality constraints, if any
4	linear inequality constraints, if any
5	nonlinear inequality constraints, if any
6	lower bounds
7	upper bounds

There are always lower and upper bounds on coefficients. If there are no other constraints placed on the coefficients, a dummy equality constraint is placed because at least one constraint is required by the quadratic programming solve.

If, for example, there are no inactivated coefficients, no equality constraints, five linear inequality constraints, no nonlinear inequality constraints, and 10 coefficients, then an error message stating that the 7th constraint is inconsistent would indicate that the second lower bound was inconsistent with one or more preceding constraints.

2.6.5 Re-using calculation results between the objective function proc and the gradient proc

When analytical gradients are provided, it is often the case that the function proc and the gradient proc contain identical calculations. If the problem is large and time-consuming, some gain in speed can be achieved by sharing these calculations.

The function and gradient are computed separately because in many instances in the iterations one or the other are needed but not both. Computing them together would then add considerably to the computational burden.

Even though they are computed separately, calculational results can be shared through globals. First, initialize the necessary globals in the command file before the call to **CO**. One of these globals should be used to store the coefficient vector being passed to the proc, and other globals will contain the calculational results to be shared.

In the function proc, test the difference between the coefficient vector stored in the global against the coefficient vector passed to it by **CO**. If they are the same, use the results stored in the other globals, otherwise re-compute the calculations.

In the gradient proc, also test the difference between the coefficient vector stored in the global against that passed to it by **CO**. If they are the same, use the stored results, but if they are different, recompute the results and then update the globals.

After a new direction has been computed and the line search completed, the gradient is the first quantity calculated with the new direction, and therefore that is where the updating should occur.

2.7 EXAMPLES

2.7.1 Nonlinear Programming Problem

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

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

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;

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.

2.7.2 Providing Procedures for Gradients and Jacobians

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

$$\begin{bmatrix} .1 & .7 & .2 \end{bmatrix}$$

is feasible. The published solution is

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

The procedure for solving this problem is

```
library co;
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 zero nonlinear inequality Lagrangean indicates that the nonlinear inequality boundary was encountered somewhere during the iterations, but is now inactive.

2.7.3 A Portfolio Optimization Problem

Because CO handles general nonlinear functions and constraints, it can solve a more general problem than the Markowitz problem. The efficient frontier is essentially a quadratic programming problem where the Markowitz Mean/Variance portfolio allocation model is solved for a range of expected portfolio returns which are then plotted against the portfolio risk measured as the standard deviation:

$$\begin{aligned} min \quad & w_k' \Sigma w_k \\ s.t. \quad & w_k' \mu = r_k \\ & w_k \iota = 1 \\ & 0 \leq w_k \leq 1 \end{aligned}$$

where ι is a conformable vector of ones, and where Σ is the observed covariance matrix of the returns of a portfolio of securities, and μ are their observed means.

This model is solved for $r_k, k = 1, \ldots, K$, and the efficient frontier is the plot of $r_k, k = 1, \ldots, K$, on the vertical axis against $\sqrt{w_k' \Sigma w_k}, k = 1, \ldots, K$, on the horizontal axis. The portfolio weights in w_k describe the optimum distribution of portfolio resources across the securities given the amount of risk to return one is comfortable with.

Because of CO's ability to handle general constraints, more elaborate models may be considered. For example, this model frequently concentrates the allocation into a minority of the securities. To spread out the allocation one could solve the problem subject to a maximum variance for the weights, i.e., subject to

$$w_k' w_k \leq \phi$$

where ϕ is a constant setting a ceiling on the sums of squares of the weights.

Alternatively, you might have a previous portfolio allocation, and you might want to develop an allocation based on new data that only departed some maximum amount from the old allocation. Let w_0 be the original portfolio allocation. Then we might want to find an optimum set of weights, w, such that

$$-\delta \leq w - w_0 \leq \delta$$

where δ is set to the maximum amount of change in allocation.

In this example, efficient frontiers and associated allocations are computed for both cases where the portfolio allocation is unrestricted and where the change in allocation is restricted to be between -.3 and +.3, i.e., $\delta = 3$, and the previous weights are $.6, .05, .1, 0, .2, .05$.

```
library co, pgraph;
coset;
graphset;

/*
**  The data are taken from from Harry S. Marmer and F.K. Louis Ng,
**  "Mean-Semivariance Analysis of Option-Based Strategies:  A Total
**  Asset Mix Perspective", Financial Analysts Journal, May-June 1993:
*/

corr = {
  1,
 .097,  1,
-.039, .231,  1,
 .159, .237, .672,  1,
-.035, .211, .391, .454,  1,
-.024, .247, .424, .432, .941, 1 };

s = { .94, 11.26, 19.21, 13.67, 17.73, 12.19 };
Sigma = xpnd(corr) .* s .* s';
Mu = { 10.67, 10.54, 12.76, 13.67, 17.73, 13.68 };


proc ObjectiveFunction(w);
     retp(w' * Sigma * w);   /* volatility */
endp;

/*
** Constraints
*/

_co_A = ones(1,6);
_co_B = 1;

_co_A = _co_A | Mu';
_co_B = _co_B | 0;

_co_Bounds = { 0 1 };


start = { 1, 0, 0, 0, 0, 0 };

MN = seqa(10.75,.025,20);

W = {};
SD = {};
```

```
for i(1,20,1);

    _co_B[2,1] = MN[i];
    __output = 0;

    { x,f,g,ret } = co(&ObjectiveFunction,start);
    start = x;
    w = w | x';
    SD = SD | sqrt(f);    /* portfolio volatility */

endfor;

output file = eff.out reset;

format /rd 8,4;
print "Unrestricted Weights and Efficient Frontier";
print;
print "  r_{k}      sd_{k}      w_{k}";
print mn~sd~w;
print;
print;

/*
**  Now an efficient frontier restricting change from previous weights
*/

PreviousWeights = {.6,.05,.1,.0, .2,.05 };

/*
** Inequality Constraints
*/

_co_C = -eye(6) | eye(6);
_co_D = (- PreviousWeights - .3) | (PreviousWeights - .3);


W1 = {};
SD1 = {};
for i(1,20,1);

    _co_B[2,1] = MN[i];

    { x,f,g,ret } = co(&ObjectiveFunction,start);
    start = x;
    w1 = w1 | x';
    SD1 = SD1 | sqrt(f);    /* portfolio volatility */
```

```
endfor;

print "Restricted Weights and Efficient Frontier";
print;
print "  r_{k}      sd_{k}      w_{k}";
print mn~sd1~w1;

output off;

title("Efficient Frontier");
Xlabel("Variance");
Ylabel("Return");
_plegstr = "Unrestricted Solution\000Restricted Solution";
_plegctl = { 1 5 .95 11.1 };

xy(SD~SD1,MN~MN);
```

The Results are

```
 Unrestricted Weights and Efficient Frontier

 r_{k}     sd_{k}    w_{k}

10.7500    0.9431    0.9872  0.0000  0.0021  0.0000  0.0107  0.0000
10.7750    0.9534    0.9839  0.0000  0.0017  0.0000  0.0144  0.0000
10.8000    0.9677    0.9807  0.0000  0.0012  0.0000  0.0181  0.0000
10.8250    0.9857    0.9775  0.0000  0.0007  0.0000  0.0217  0.0000
10.8500    1.0072    0.9743  0.0000  0.0003  0.0000  0.0254  0.0000
10.8750    1.0321    0.9710  0.0000  0.0000  0.0000  0.0290  0.0000
10.9000    1.0601    0.9674  0.0000  0.0000  0.0000  0.0326  0.0000
10.9250    1.0911    0.9639  0.0000  0.0000  0.0000  0.0361  0.0000
10.9500    1.1247    0.9603  0.0000  0.0000  0.0000  0.0397  0.0000
10.9750    1.1608    0.9568  0.0000  0.0000  0.0000  0.0432  0.0000
11.0000    1.1991    0.9533  0.0000  0.0000  0.0000  0.0467  0.0000
11.0250    1.2394    0.9497  0.0000  0.0000  0.0000  0.0503  0.0000
11.0500    1.2815    0.9462  0.0000  0.0000  0.0000  0.0538  0.0000
11.0750    1.3253    0.9426  0.0000  0.0000  0.0000  0.0574  0.0000
11.1000    1.3706    0.9391  0.0000  0.0000  0.0000  0.0609  0.0000
11.1250    1.4173    0.9356  0.0000  0.0000  0.0000  0.0644  0.0000
11.1500    1.4652    0.9320  0.0000  0.0000  0.0000  0.0680  0.0000
11.1750    1.5141    0.9285  0.0000  0.0000  0.0000  0.0715  0.0000
11.2000    1.5641    0.9246  0.0000  0.0000  0.0006  0.0748  0.0000
11.2250    1.6149    0.9207  0.0000  0.0000  0.0012  0.0781  0.0000
```

```
  Restricted Weights and Efficient Frontier

  r_{k}      sd_{k}     w_{k}

 10.7500     1.3106     0.9000  0.0684  0.0066  0.0000  0.0000  0.0249
 10.7750     1.2891     0.9000  0.0604  0.0068  0.0000  0.0000  0.0328
 10.8000     1.2771     0.9000  0.0528  0.0057  0.0027  0.0000  0.0388
 10.8250     1.2734     0.9000  0.0454  0.0038  0.0073  0.0000  0.0435
 10.8500     1.2778     0.9000  0.0379  0.0019  0.0119  0.0000  0.0483
 10.8750     1.2903     0.9000  0.0305  0.0001  0.0164  0.0000  0.0530
 10.9000     1.3077     0.9000  0.0300  0.0000  0.0172  0.0058  0.0470
 10.9250     1.3265     0.9000  0.0299  0.0000  0.0177  0.0119  0.0405
 10.9500     1.3466     0.9000  0.0298  0.0000  0.0183  0.0180  0.0340
 10.9750     1.3679     0.9000  0.0297  0.0000  0.0189  0.0240  0.0274
 11.0000     1.3904     0.9000  0.0296  0.0000  0.0195  0.0301  0.0209
 11.0250     1.4140     0.9000  0.0294  0.0000  0.0200  0.0362  0.0143
 11.0500     1.4387     0.9000  0.0293  0.0000  0.0206  0.0423  0.0078
 11.0750     1.4643     0.9000  0.0292  0.0000  0.0212  0.0484  0.0012
 11.1000     1.4914     0.9000  0.0264  0.0000  0.0212  0.0524  0.0000
 11.1250     1.5209     0.9000  0.0229  0.0000  0.0212  0.0559  0.0000
 11.1500     1.5526     0.9000  0.0195  0.0000  0.0211  0.0594  0.0000
 11.1750     1.5863     0.9000  0.0161  0.0000  0.0211  0.0629  0.0000
 11.2000     1.6220     0.9000  0.0126  0.0000  0.0210  0.0664  0.0000
 11.2250     1.6596     0.9000  0.0092  0.0000  0.0210  0.0699  0.0000
```

Figure 2.1 contains the efficient frontiers for both the restricted and unrestricted analyses

2.8 Run-Time Switches

If the user presses **H** (or **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.

G (or **g**)	Toggle **_co_GradMethod**
V (or **v**)	Revise **_co_DirTol**
O (or **o**)	Toggle **__output**
M (or **m**)	Maximum Tries
I (or **i**)	Compute Hessian
E (or **e**)	Edit Parameter Vector
C (or **c**)	Force Exit
A (or **a**)	Change Algorithm
J (or **j**)	Change Line Search Method
T (or **t**)	Trust Region
H (or **h**)	Help Table

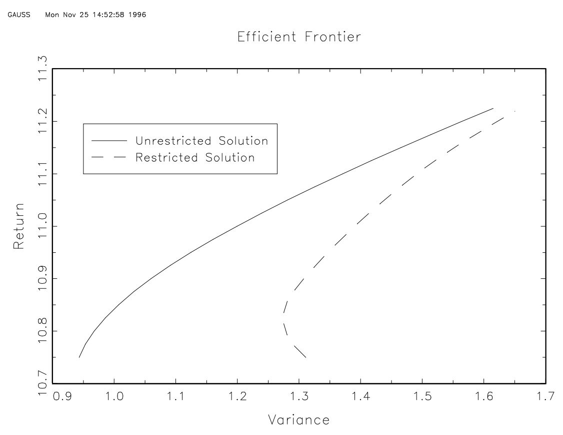

Figure 2.1: Efficient Frontier for Restricted and Unrestricted Portfolio Weights

The algorithm may be switched during the iterations either by pressing **A** (or **a**), or by pressing one of the following:

1	Broyden-Fletcher-Goldfarb-Shanno (BFGS)
2	Davidon-Fletcher-Powell (DFP)
3	Newton-Raphson (NEWTON) or (NR)
4	Scaled Broyden-Fletcher-Goldfarb-Shanno (BFGS-SC)
5	Scaled Davidon-Fletcher-Powell (DFP-SC)

The line search method may be switched during the iterations either by pressing **S**, or by pressing one of the following:

Shift-1	no search (1.0 or 1 or ONE)
Shift-2	cubic or quadratic method (STEPBT)
Shift-3	step halving method (HALF)
Shift-4	Brent's method (BRENT)

2.9 Error Handling

2.9.1 Return Codes

The fourth argument in the return from **CO** contains a scalar number that contains information about the status of the iterations upon exiting **CO**. 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	quasi-Newton update failed
11	maximum time exceeded
13	quadratic program failed
14	equality Jacobian failed
15	inequality Jacobian failed
20	Hessian failed to invert
99	termination condition unknown

2.9.2 Error Trapping

Setting the global **__output** = 0 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 **CO** to *not* send the messages to the screen:

```
trap 4;
```

Whatever the setting of the trap flag, **CO** 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 **CO** function is embedded in a larger program, and you want the larger program to handle the errors.

2.10 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.

Chapter 3

Constrained Optimization Reference

Purpose

Computes estimates of parameters of a constrained optimization function.

Library

co

Format

{ *x*,*f*,*g*,*retcode* } = **CO(&*fct*,*start*)**

Input

&*fct*	a pointer to a procedure that returns the function evaluated at the parameters
start	$K \times 1$ vector, start values.

Output

x	$K \times 1$ vector, estimated parameters
f	scalar, function at minimum
g	$K \times 1$ vector, gradient evaluated at x
retcode	scalar, return code. If normal convergence is achieved, then $retcode = 0$, otherwise a positive integer is returned indicating the reason for the abnormal termination:

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

13	quadratic program failed
20	Hessian failed to invert
99	termination condition unknown.

Globals

The globals variables used by **CO** can be organized in the following categories according to which aspect of the optimization they affect:

Options **_co_Options**

Constraints **_co_A**, **_co_B**, **_co_C**, **_co_D**, **_co_EqProc**, **_co_IneqProc**, **_co_EqJacobian**, **_co_IneqJacobian**, **_co_Bounds**, **_co_Lagrange**

Descent and Line Search **_co_Algorithm**, **_co_Delta**, **_co_LineSearch**, **_co_MaxTry**, **_co_Extrap**, **_co_FeasibleTest**, **_co_Interp**, **_co_RandRadius**, **_co_UserSearch** **_co_Switch**, **_co_Trust**, **_co_TrustRadius**,

Gradient **_co_GradMethod**, **_co_GradProc**, **_co_UserNumGrad**, **_co_HessProc**, **_co_UserNumHess**, **_co_GradStep**, **_co_GradCheckTol**

Terminations Conditions **_co_DirTol**, **_co_MaxIters**, **_co_MaxTime**

Parameters **_co_Active**, **_co_ParNames**

Miscellaneous **__title**, **_co_IterData**, **_co_Diagnostic**

The list below contains an alphabetical listing of each global with a complete description.

_co_A $M_1 \times K$ matrix, linear equality constraint coefficient matrix **_co_A** is used with **_co_B** to specify linear equality constraints:

```
_co_A * X = _co_B
```

where X is the $K \times 1$ unknown parameter vector.

_co_Active vector, defines fixed/active coefficients. This global allows you to fix a parameter to its starting value. This is useful, for example, when you wish to try different models with different sets of parameters without having to re-edit the function. When it is to be used, it must be a vector of the same length as the starting vector. Set elements of **_co_Active** to 1 for an active parameter, and to zero for a fixed one.

_co_Algorithm scalar, selects optimization method:

1	BFGS - Broyden, Fletcher, Goldfarb, Shanno method
2	DFP - Davidon, Fletcher, Powell method
3	NEWTON - Newton-Raphson method
4	scaled BFGS
5	scaled DFP

Default = 3

_co_Delta scalar, floor for eigenvalues of Hessian in the NEWTON algorithm. When nonzero, the eigenvalues of the Hessian are augmented to this value.

_co_B $M_1 \times 1$ vector, linear equality constraint constant vector **_co_B** is used with **_co_A** to specify linear equality constraints:

```
_co_A * X = _co_B
```

where X is the $K \times 1$ unknown parameter vector.

_co_Bounds $K \times 2$ matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds. If the bounds for all the coefficients are the same, a 1x2 matrix may be used. Default = { -1e256 1e256 }.

_co_C $M_3 \times K$ matrix, linear inequality constraint coefficient matrix **_co_C** is used with **_co_D** to specify linear inequality constraints:

```
_co_C * X = _co_D
```

where X is the $K \times 1$ unknown parameter vector.

_co_D $M_3 \times 1$ vector, linear inequality constraint constant vector **_co_D** is used with **_co_C** to specify linear inequality constraints:

```
_co_C * X = _co_D
```

where X is the $K \times 1$ unknown parameter vector.

_co_Diagnostic scalar.

0	nothing is stored or printed
1	current estimates, gradient, direction, function value, Hessian, and step length are printed to the screen
2	the current quantities are stored in **_co_Diagnostic** using the **VPUT** command. Use the following strings to extract from **_co_Diagnostic** using **VREAD**:

function	"function"
estimates	"params"
direction	"direct"
Hessian	"hessian"
gradient	"gradient"
step	"step"

_co_DirTol scalar, convergence tolerance for gradient of estimated coefficients. When this criterion has been satisifed **CO** exits the iterations. Default = 1e-5.

_co_EqJacobian scalar, pointer to a procedure that computes the Jacobian of the nonlinear equality constraints with respect to the parameters. The procedure has one input argument, the $K \times 1$ vector of parameters, and one output argument, the $M_2 \times K$ matrix of derivatives of the constraints with respect to the parameters. For example, if the nonlinear equality constraint procedure was,

```
proc eqproc(p);
    retp(p[1]*p[2]-p[3]);
endp;
```

the Jacobian procedure and assignment to the global would be,

```
proc eqj(p);
    retp(p[2]~p[1]~-1);
endp;

_co_EqJacobian = &eqj;
```

_co_EqProc scalar, pointer to a procedure that computes the nonlinear equality constraints. For example, the statement:

```
_co_EqProc = &eqproc;
```

tells **CO** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the $K \times 1$ vector of parameters, and one output argument, the $M_2 \times 1$ vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
P[1] * P[2] = P[3]
```

The proc for this is:

```
proc eqproc(p);
    retp(p[1]*[2]-p[3]);
endp;
```

_co_Extrap scalar, extrapolation constant in BRENT. Default = 2.

_co_FeasibleTest scalar, if zero, testing for feasibility in the line search is turned off. Default = 1, i.e., the test for feasibility is turned on.

_co_FinalHess $K \times K$ matrix, the Hessian used to compute the covariance matrix of the parameters is stored in **_co_FinalHess**. This is most useful if the inversion of the hessian fails, which is indicated when **CO** returns a missing value for the covariance matrix of the parameters. An analysis of the Hessian stored in **_co_FinalHess** can then reveal the source of the linear dependency responsible for the singularity.

_co_GradCheckTol scalar. Tolerance for the deviation of numerical and analytical gradients when proc's exist for the computation of analytical gradients, Hessians, and/or Jacobians. If set to zero, the analytical gradients will not be compared to their numerical versions. When adding procedures for computing analytical gradients it is highly recommended that you perform the check. Set **_co_GradCheckTol** to some small value, 1e-3, say when checking. It may have to be set larger if the numerical gradients are poorly computed to make sure that **CO** doesn't fail when the analytical gradients are being properly computed.

_co_GradMethod scalar, method for computing numerical gradient.

0	central difference
1	forward difference (default)

_co_GradProc scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. For example, the statement:

```
_co_GradProc=&gradproc;
```

tells **CO** that a gradient procedure exists as well where to find it. The user-provided procedure has one input argument, a $K \times 1$ vector of parameter values The procedure returns a single output argument, a $K \times 1$ vector of gradients of the function with respect to the parameters evaluated at the vector of parameter values.

For example, suppose the function is $b_1 \exp -b_2$, then the following would be added to the command file:

```
proc lgd(b);
    retp(exp(-b[2])|-b[1]*b[2]*exp(-b[2]));
endp;

_co_GradProc = &lgd;
```

Default = 0, i.e., no gradient procedure has been provided.

_co_GradStep scalar, or 1×2 or $K \times 1$, or $K \times 2$ matrix, increment size for computing gradient and/or Hessian. If scalar, stepsize will be value times parameter estimates for the numerical gradient. If 1x2, the first element is multiplied times parameter value for gradient and second element the same for the Hessian. If Kx1, the step size for the gradient will be the elements of the vector, i.e., it will not be multiplied times the parameters, and if Kx2, the second column sets the step sizes for the Hessian.

When the numerical gradient is not performing well, set to a larger value (1e-3, say). Default is the cube root of machine precision.

_co_HessProc scalar, pointer to a procedure that computes the hessian, i.e., the matrix of second order partial derivatives of the function with respect to the parameters. For example, the instruction:

```
_co_HessProc = &hessproc;
```

tells **CO** that a procedure has been provided for the computation of the hessian and where to find it. The procedure that is provided by the user must have one input argument, a $K \times 1$ vector of parameter values. The procedure returns a single output argument, the $K \times K$ symmetric matrix of second order derivatives of the function evaluated at the parameter values.

_co_IneqJacobian scalar, pointer to a procedure that computes the Jacobian of the nonlinear equality constraints with respect to the parameters. The procedure has one input argument, the $K \times 1$ vector of parameters, and one output argument, the $M_4 \times K$ matrix of derivatives of the constraints with respect to the parameters. For example, if the nonlinear equality constraint procedure was,

```
proc ineqproc(p);
    retp(p[1]*p[2]-p[3]);
endp;
```

the Jacobian procedure and assignment to the global would be,

```
proc ineqj(p);
    retp(p[2]~p[1]~-1);
endp;

_co_IneqJacobian = &ineqj;
```

_co_IneqProc scalar, pointer to a procedure that computes the nonlinear inequality constraints. For example the statement:

```
_co_IneqProc = &ineqproc;
```

tells **CO** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the $K \times 1$ vector of parameters, and one output argument, the $M_4 \times 1$ vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
P[1] * P[2] >= P[3]
```

The proc for this is:

```
proc ineqproc(p);
    retp(p[1]*[2]-p[3]);
endp;
```

_co_Interp scalar, interpolation constant in BRENT. Default = .25.

_co_IterData 2x1 vector, contains information about the iterations. The first element contains the # of iterations, the second element contains the elapsed time in minutes of the iterations.

_co_Lagrange vector, created using VPUT. Contains the Lagrangean coefficients for the constraints. They may be extracted with the VREAD command using the following strings:

"lineq"	linear equality constraints
"nlineq"	nonlinear equality constraints
"linineq"	linear inequality constraints
"nlinineq"	nonlinear inequality constraints
"bounds"	bounds

When an inequality or bounds constraint is active, its associated Lagrangean is nonzero. The linear Lagrangeans preceed the nonlinear Lagrangeans in the covariance matrices.

_co_LineSearch scalar, selects method for conducting line search. The result of the line search is a *step length*, i.e., a number which reduces the function value when multiplied times the direction..

1	step length = 1.
2	cubic or quadratic step length method (STEPBT)
3	step halving (HALF)
4	Brent's step length method (BRENT)

Default = 2.

Usually **_co_LineSearch** = 2 is best. If the optimization bogs down, try setting **_co_LineSearch** = 1, 4 or 5. **_co_LineSearch** = 3 generates slower iterations but faster convergence and **_co_LineSearch** = 1 generates faster iterations but slower convergence.

When any of these line search methods fails, **CO** attempts a random search of radius **_co_RandRadius** times the truncated log to the base 10 of the gradient when **_co_RandRadius** is set to a nonzero value. If **_co_UserSearch** is set to 1, **CO** enters an interactive line search mode.

_co_MaxIters scalar, maximum number of iterations.

_co_MaxTime scalar, maximum time in iterations in minutes. Default = 1e+5, about 10 weeks.

_co_MaxTry scalar, maximum number of tries to find step length that produces a descent.

_co_Options character vector, specification of options. This global permits setting various **CO** options in a single global using identifiers. For example

```
_co_Options = { newton brent trust central file };
```

sets the line search method to BRENT, the descent method to NEWTON, trust region on, the numerical gradient method to central differences, and **__output** = 1.

The following is a list of the identifiers:

Algorithms BFGS, DFP, NEWTON, BFGS-SC, DFP-SC

Line Search ONE, STEPBT, HALF, BRENT

Trust Method TRUST

Gradient method CENTRAL, FORWARD

Output method NONE, FILE, SCREEN

_co_ParNames $K \times 1$ character vector, parameter labels.

_co_RandRadius scalar, if set to a nonzero value (1e-2, say) and all other line search methods fail then **CO** attempts **_co_MaxTry** tries to find a random direction within radius determined by **_co_RandRadius** that is a descent. Default = 1e-2.

_co_Switch 3×1 vector, algorithm switching. The first element is the descent algorith to which the iterations will be switched. The second element is amount of change in the function less than which the algorithm will be switched. The idea here is that the algorithm should be switched when the descent progress is too small, that is, when the function changes less than some value. The third element is the iteration number at which the algorithm is to be switched. By default switching is turned off.

_co_Trust scalar, if nonzero, the trust region method is turned on. Default = 0.

_co_TrustRadius scalar. The trust region if the trust region method is turned on. Default = .01.

_co_UserNumGrad scalar, pointer to user provided numerical gradient procedure. The instruction

```
_co_UserNumGrad = &userproc;
```

tells **CO** that a procedure for computing the numerical gradients exists. The user-provided procedure has three input arguments, a pointer to a function that computes the function, and a $K \times 1$ vector of parameter values. The procedure returns a single output argument, a $K \times 1$ matrix of gradients of the function with respect to each parameter.

__title string title of run

_co_UserNumHess scalar, pointer to user provided numerical Hessian procedure. The instruction

```
_co_UserHess = &hessproc;
```

tells **CO** that a procedure for computing the numerical Hessian exists. The user-provided procedure three input arguments, a pointer to a function that computes the function, and a $K \times 1$ vector of parameter values. The procedure returns a single output argument, a $K \times K$ Hessian matrix of the function with respect to the parameters.

_co_UserSearch scalar, if nonzero and if all other line search methods fail **CO** enters an interactive mode in which the user can select a line search parameter

■ Remarks

Specifying Constraints.

There are five types of constraints: linear equality, linear inequality, nonlinear equality, nonlinear inequality, bounds Linear constraints are specified by initializing the appropriate **CO** globals to known matrices of constants. The linear equality constraint matrices are **_co_A** and **_co_B**, and they assume the following relationship with the parameter vector:

```
_co_A * x = _co_B
```

where `x` is the parameter vector.

Similarly, the linear *in*equality constraint matrices are **_co_C** and **_co_D**, and assume the following relationship with the parameter vector:

```
_co_C * x >= _co_D
```

The nonlinear constraints are specified by providing procedures and assigning their pointers to **CO** globals. These procedures take a single argument, the vector of parameters, and return a column vector of evaluations of the constraints at the parameters. Each element of the column vector is a separate constraint.

For example, suppose you wish to constrain the product of the first and third coefficients to be equal to 10, and the squared second and fourth coefficients to be equal to the squared fifth coefficient:

```
proc eqp(x);
    local c;
    c = zeros(2,1);
    c[1] = x[1] * x[3] - 10;
    c[2] = x[2] * x[2] + x[4] * x[4] - x[5] * x[5];
    retp(c);
endp;

 _co_EqProc = &eqp;
```

The nonlinear equality constraint procedure causes **CO** to find estimates for which its evaluation is equal to a conformable vector of zeros.

The nonlinear *in*equality constraint procedure is similar to the equality procedure. **CO** finds estimates for which the evaluation of the procedure is greater than or equal to zero. The nonlinear inequality constraint procedure is assigned to the global **_co_IneqProc**. For example, suppose you wish to constrain the norm of the coefficients to be greater than one:

```
proc ineqp(x);
    retp(x'x-3);
endp;

 _co_IneqProc = &ineqp;
```

Bounds are a type of linear inequality constraint. They are specified separately for computational and notational convenience. To declare bounds on the parameters assign a two column vector with rows equal to the number of parameters to the **CO** global, **_co_Bounds**. The first column is the lower bounds and the second column the upper bounds. For example,

```
_co_Bounds = {  0 10,
              -10  0
              -10 20 };
```

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

Writing the Function to be Minimized

The user must provide a procedure for computing the function The procedure has one input argument, a vector of parameters. The output is the scalar value of the function evaluated at the current parameters values. Suppose that the function procedure has been named *pfct*, the following considerations apply:

The format of the procedure is:

f = $fct(x)$;

where x is a column vector of parameters.

Supplying an Analytical GRADIENT Procedure

To decrease the time of computation, the user may provide a procedure for the calculation of the gradient of the function. The global variable **_co_GradProc** must contain the pointer to this procedure. Suppose the name of this procedure is *gradproc*. Then,

g = $gradproc(x)$;

where the input argument is the vector of parameters and the output argument is g is column vector of gradients of the function with respect to coefficients

Providing a procedure for the calculation of the first derivatives also has a significant effect on the calculation time of the Hessian. The calculation time for the numerical computation of the Hessian is a quadratic function of the size of the matrix. For large matrices, the calculation time can be very significant. This time can be reduced to a linear function of size if a procedure for the calculation of analytical first derivatives is available. When such a procedure is available, **CO** automatically uses it to compute the numerical Hessian.

The major problem one encounters when writing procedures to compute gradients and Hessians is in making sure that the gradient is being properly computed. **CO** checks the gradients and Hessian when **_co_GradCheckTol** is nonzero. **CO** generates both numerical and analytical gradients, and viewing the discrepancies between them can help in debugging the analytical gradient procedure.

Supplying an Analytical HESSIAN Procedure.

Selection of the NEWTON algorithm becomes feasible if the user supplies a procedure to compute the Hessian. If such a procedure is provided, the global variable **_co_HessProc** must contain a pointer to this procedure. Suppose this procedure is called *hessproc*, the format is

h = $hessproc(x)$;

The input argument is the $K \times 1$ vector of parameter values. The output argument is the $K \times K$ matrix of second order partial derivatives evaluated at the coefficients in x.

Supplying Analytical Jacobians of the Nonlinear Constraints.

At each iteration the Jacobians of the nonlinear constraints, if they exist, are computed numerically. This is time-consuming and generates a loss of precision. For models with a large number of inequality constraints a significant speed-up can be achieved by providing analytical Jacobian procedures. The improved accuracy can also have a significant effect on convergence.

The Jacobian procedures take a single argument, the vector of parameters, and return a matrix of derivatives of each constraint with respect to each parameter. The rows are associated with the constraints and the columns with the parameters. The pointer to the nonlinear equality Jacobian procedure is assigned to **_co_EqJacobian**. The pointer to the nonlinear *in*equality Jacobian procedure is assigned to **_co_IneqJacobian**.

For example, suppose the following procedure computes the equality constraints:

```
 proc eqp(x);
     local c;
     c = zeros(2,1);
     c[1] = x[1] * x[3] - 10;
     c[2] = x[2] * x[2] + x[4] * x[4] - x[5] * x[5];
     retp(c);
 endp;
_co_EqProc = &eqp;
```

Then the Jacobian procedure would look like this:

```
 proc eqJacob(x);
    local c;
    c = zeros(2,5);
    c[1,1] = x[3];
    c[1,3] = x[1];
    c[2,2] = 2*x[2];
    c[2,4] = 2*x[4];
    c[3,5] = -2*x[5];
    retp(c);
endp;
_co_EqJacobian = &eqJacob;
```

The Jacobian procedure for the nonlinear inequality constraints is specified similarly, except that the associated global containing the pointer to the procedure is **_co_IneqJacobian**.

Source

co.src

Purpose

Resets *CONSTRAINED OPTIMIZATION* global variables to default values.

Library

`co`

Format

COSet;

Input

None

Output

None

Remarks

Putting this instruction at the top of all command files that invoke **CO** is generally good practice. This prevents globals from being inappropriately defined when a command file is run several times or when a command file is run after another command file has executed that calls **CO**.

Source

`co.src`

Purpose

Formats and prints the output from a call to **CO**.

Library

`co`

Format

{ *x*,*f*,*g*,*retcode* } = **COPrt(***x*,*f*,*g*,*retcode***);**

Input

x	$K \times 1$ vector, parameter estimates
f	scalar, value of function at minimum
g	$K \times 1$ vector, gradient evaluated at x
retcode	scalar, return code.

Output

The input arguments are returned unchanged.

Globals

__header string. This is used by the printing procedure to display information about the date, time, version of module, etc. The string can contain one or more of the following characters:

"t"	print title (see **__title**)
"l"	bracket title with lines
"d"	print date and time
"v"	print version number of program
"f"	print file name being analyzed

Example:

```
__header = "tld";
```

Default = "tldvf".

__title string, message printed at the top of the screen and printed out by **COPrt**. Default = "".

Remarks

The call to **CO** can be nested in the call to **COPrt**:

```
{ x,f,g,retcode } = COPrt(CO(&fct,start));
```

Source

`co.src`

Index