Constrained Optimization MT 2.0

for GAUSS™ Mathematical and Statistical System

GAUSS
Expanding Probabilities

Version 2.0
Friday, April 14, 2017

Contents

1 Installation

Constrained Optimization MT 2.0 requires **GAUSS 16** or later. In **GAUSS 16+** there is an Applications Installation Wizard available to install your application.

From within **GAUSS**, go to **Tools -> Install Applications** and follow the prompts to install **GAUSS Applications** from the CD or downloaded .zip file.

Difference Between the Linux/Mac 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 Linux/Mac version.

2 Getting Started

The **Constrained Optimization MT** module contains a set of procedures for the solution of the optimization problem.

GAUSS version 16+ is required to use these routines.

The **Constrained Optimization MT 2.0** version number is stored in a global variable:

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

If you contact technical support, you may be asked for the version of your **Constrained Optimization MT** license.

2.1 README Files

If there is a README.comt file, it contains any last minute information on the **Constrained Optimization MT 2.0** procedures. Please read it before using them.

2.2 Setup

In order to use the procedures in **Constrained Optimization MT** or **COMT** module, the **COMT** library must be active. This is done by including 'comt' in the library statement at the top of your program or command file:

```
library comt;
```

This enables **GAUSS** to find the **COMT** procedures.

3 Special Features in Constrained Optimization MT

The following sections describe the special features found in **Constrained Optimization MT**

3.1 Structures

In **COMT** the same procedure that computes the objective function will also be used to compute analytical derivatives if they are being provided. This procedure will have an additional argument, which tells the function whether to compute the log-likelihood or objective, the first derivatives, the second derivatives, or all three. This means that calculations in common will not have to be redone.

3.1.1 modelResults structure

This objective procedure will return a *modelResults* structure which has three member variables:

modelResults structure	
function	Scalar, value of the objective function.
gradient	Optional, Kx1 vector of first derivatives.
Hessian	Optional, KxK matrix of second derivatives.

```
//Example objective function
proc (1) = myobjective(parms, ind);
   struct modelResults mm;

   //Perform any calculations common to
   //objective function, gradient and Hessian

   //If the first element of 'ind' is
   //non-zero, calculate objective function
   if ind[1];
      mm.function = //Calculate objective function
   endif;

   //If the second element of 'ind' is
   //non-zero, calculate gradient
   if ind[2];
      mm.gradient = //Calculate gradient
   endif;

   //If the third element of 'ind' is
   //non-zero, calculate Hessian
   if ind[3];
      mm.Hessian = //Calculate Hessian
   endif;

   //Return modelResults structure
   retp(mm);
endp;
```

Of course, the derivatives are optional, or even partially optional, i.e., you can compute a subset of the derivatives, if you like, and the remaining will be computed numerically. When computing only a subset of the derivatives, set the uncomputed element of the gradient vector to a missing value. **COMT** will attempt to compute numerical derivatives for any element of the gradient vector that contains a missing value.

3.1.2 Parameter Vector (PV) Structure

COMT allows you to use the *PV* structure from the standard **GAUSS Run-Time Library**, to pass parameters to the objective function. The *PV* structure makes it easy to store your parameters as vectors, matrices or n-dimensional arrays. For cases in which your parameter vector is simply a vector, **COMT** allows you to pass in your parameter vector directly without the use of the *PV* structure.

```
//Add symmetric matrix of starting
//values to 'PV' structure
omega_strt = {  1 0.8 -0.4,
              0.8   1  0.6,
             -0.4 0.6    1 };
p = pvPackS(pvCreate(), omega_strt, "omega");

proc (1) = myobjective(struct PV parms, ind);
   local omega;

   //Retrieve updated symmetric matrix
   //inside of objective function
   omega = pvUnpack(parms, "omega");

   //Perform calcuations and return
```

No more do you have to struggle to get the parameter vector into matrices for calculating the function and its derivatives, trying to remember or figure out which parameter is where in the vector. If your log-likelihood uses matrices or arrays, you can store them directly into the *PV* structure, and remove them as matrices or arrays with the parameters already plugged into them. The *PV* structure can even efficiently handle symmetric matrices, where parameters below the diagonal are repeated above the diagonal.

The functions **pvPackM** and **pvPackMI**, allow you to specify some elements inside your *PV* structure as fixed values and others as free parameters. It remembers the fixed values and only updates the values of the free parameters.

3.1.3 Optional Dynamic Arguments

There will no longer be any need to use global variables. Any inputs that your procedure needs, other than the parameters of the model, can be passed into **COMT** as optional, dynamic arguments. These optional arguments will be passed directly, and untouched, to your objective function.

```
//Inputs to objective function for
//COMT version 1.0
proc (1) = myobjective(struct PV parms, struct DS d, ind);

//Inputs to objective function for
//COMT current version, that requires no
//data other than model parameters.
//And the parameters are simply a vector.
proc (1) = myobjective(x, ind);

//Inputs to objective function for
//COMT current version, that requires no
//data other than model parameters.
//And the parameters are packed in a PV struct.
proc (1) = myobjective(struct PV parms, ind);

//Inputs to objective function for
//COMT current version, that requires
//2 extra matrices, 'theta' and 'gamma'
```

```
//Place extra inputs between the parameter vector and 'ind'
proc (1) = myobjective(x, theta, gamma, ind);

//Inputs to objective function for
//COMT current version, that requires
//2 extra matrices, 'theta' and 'gamma'
//and using the PV structure for parameters
//Place extra inputs between 'PV' struct and 'ind'
proc (1) = myobjective(struct PV parms, theta, gamma, ind);
```

Previous versions of **COMT**, required the use of the *DS* structure for this purpose. The current version is backwards compatible with version 1, so programs written using the *DS* structure will continue to work.

3.1.4 Control Structures

The functions in this library also use control structures to set optimization options, rather than global control variables. This means, in addition to thread safety, that it will be straightforward to nest calls to **COMT** inside of a call to **COMT**, or other multi-threaded **GAUSS** functions..

3.2 Threading

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

```
//Declare 'c0' to be a comtControl struct
struct comtControl c0;

//Fill 'c0' with default settings
c0 = comtControlCreate();
```

```
//Turn on threading of numerical derivatives in COMT
c0.useThreads = 1;
```

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

Note that the useThreads structure member controls the high-level threading of sections of the **COMT** source code, but does not control the low-level threads that are internal to the **GAUSS** intrinsic functions.

3.3 Augmented Lagrangian Penalty Line Search Method

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

4 The Objective Function

COMT is a set of procedures for estimating the parameters of models via the SQP Nonlinear Programming method with general constraints on the parameters

$$min\ F(\theta)$$

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.

4.1 The Descent Algorithms

COMT uses a Sequential Quadratic Programming Method in which the nonlinear constraints, if any, are linearized and added to the linear constraints and then a descent iteration is accomplished by the use of a quadratic programming solve. This method also requires the Hessian, the matrix of second order derivatives of the the objective function with respect to the parameters, and the gradient, the vector of first derivatives. If analytical derivatives are not provided, they can be numerically estimated.

The available methods for estimating the Hessian, are a direct numerical estimate, called the Newton method, the secant methods, BFGS and DFP, where the estimate is an update from the previous iteration starting with a diagonal matrix.

5 Algorithm

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

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

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

Direction

Define

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

and the Jacobians

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

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

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

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

This solution requires that Σ be positive semi-definite.

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

Line Search

Define the merit function

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

where g_j is the j-th row of *G*, $h_\ell$is the ℓ-th row of *H*, κ is the vector of Lagrangean coefficients of the equality constraints, and λ is the vector of Lagrangean coefficients of the inequality constraints.

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

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

Trust Radius

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

5.1 The Secant Algorithms

The Hessian may be very expensive to compute at every iteration, and poor start values may produce an ill-conditioned Hessian. For these reasons alternative algorithms are provided in **COMT** 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 **COMT**.

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

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

Secant Methods (BFGS and DFP)

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

The secant methods are commonly implemented as updates of the *inverse* of the Hessian. This is not the best method numerically for the BFGS algorithm (Gill and Murray, 1972). This version of **COMT**, 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.

5.2 Line Search Methods

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

Augmented Penalty Line Search Method

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

STEPBT

STEPBT is an implementation of a similarly named algorithm described in Dennis and Schnabel (1983). It first attempts to fit a quadratic function to $m(\theta_t + \rho\delta)$ and computes 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:

$$\left(\sqrt{5}-1\right)/2 = .6180\ldots .$$

This constant is the inverse of the so-called "golden ratio":

$$\left(\sqrt{5}+1\right)/2 = 1.6180\ldots$$

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 $m(x+\delta)$, i.e., sets $\rho = 1$. 1. If $m(x+\delta) < m(x)$ then the step length is set to 1. If not, then it tries $m(x+.5\delta)$. 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.

WOLFE

The Strong Wolfe condition we have

$$m(\theta_t + \delta) \le m(\theta_t) + g_1 \rho \theta_t' [m(\theta_t) - m(\theta_{t-1})]$$
$$\theta_t [m(\theta_t + \rho) - m(\theta_{t-1} + \rho)] \le g_2 \theta_t' [m(\theta_t) - m(\theta_{t-1})]$$

where

$$0 < g_1 < g_2 < 1.$$

The first condition ensures that m decreases sufficiently and the second condition ensures that the slope has been reduced sufficiently.

5.3 Active and Inactive Parameters

The member *active* of the *comtControl* structure may be used to fix parameters to their start values. *active* must be set to a vector of the same length as the vector of start values. Elements of *active* set to zero will fix the corresponding parameter to its starting value, while the parameters corresponding to nonzero elements will be estimated.

For example, to fix the third parameter of a model with five parameters, to its starting value:

```
//Declare 'ctl' to be a comtControl struct
struct comtControl ctl;

//Fill 'ctl' with default settings
ctl = comtControlCreate();

//Fix the third parameter to its start value
//NOTE: This setting will not take effect until
```

```
//'ctl' is passed in to a call to 'comt'
ctl.active = { 1, 1, 0, 1, 1 };
```

This allows estimation of different models without having to modify the function procedure. For example, setting a parameter to be 'inactive' and setting its starting value to zero will eliminate it from the estimation.

6 Constraints

There are two general types of constraints:

- Nonlinear equality constraints
- Nonlinear inequality constraints

However, for computational convenience they are divided into five types:

- Linear equality
- Linear inequality
- Nonlinear equality
- Nonlinear inequality
- Bounds

6.1 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 members, *A* and *B*, of an instance of a *comtControl* structure.

6.1.1 Examples

Example 1

Set the constraint represented by the equation:

$$1.5\theta_1 + 2.1\theta_2 + 0\theta_3 + 0\theta_4 = 14$$

```
struct comtControl ctl;
ctl = comtControlCreate();

//Set constraint
ctl.A = { 1.5 2.1 0 0 };
ctl.B = 14;
```

Example 2

Constrain the first of four parameters to be equal to the third, represented by the equation:

$$1\theta_1 + 0\theta_2 + (-1\theta_3) + 0\theta_4 = 0$$

```
struct comtControl ctl;
ctl = comtControlCreate();

//Set constraint
ctl.A = { 1 0 -1 0 };
ctl.B = { 0 };
```

6.2 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, D an $m_2 \times 1$ vector of known constants, and θ the vector of parameters.

The specification of linear inequality constraints is done by assigning the *C* and *D* matrices to members, *C* and *D*, of an instance of a *comtControl* structure.

6.2.1 Examples

Example 1

Constrain the first of three parameters to be greater than or equal to 2.5, represented by the equation:

$$1\theta_1 + 0\theta_2 + 0\theta_3 \geq 2.5$$

```
struct comtControl ctl;
ctl = comtControlCreate();

//Add the linear inequality constraint:
ctl.C = { 1 0 0 };
ctl.B = 2.5;
```

Example 2

Constrain the first of four parameters to be greater than the third, and as well the second plus the fourth greater than 10, represented by the equations:

$$1\theta_1 + 0\theta_2 + -1\theta_3 + 0\theta_4 \geq 0$$
$$0\theta_1 + 1\theta_2 + 0\theta_3 + 1\theta_4 \geq 10$$

```
struct comtControl ctl;
ctl = comtControlCreate();

//Add the linear inequality constraint
ctl.C = { 1 0 -1 0,
          0 1  0 1 };
ctl.B = { 0,
         10 };
```

6.3 Nonlinear Equality

Nonlinear equality constraints are of the form:

$$G(\theta) = 0$$

where θ is the vector of parameters and $G(\theta)$ is an arbitrary, user-supplied function. Nonlinear equality constraints are specified by assigning the procedure pointer to the *eqProc* member of an instance of the *comtControl* structure. This procedure has one required input argument, the model parameters--either as a Px1 matrix or a *PV* structure. Any optional dynamic arguments passed to **comt**, will also be passed to this function.

6.3.1 Examples

Apply the nonlinear equality constraint for the equation:

$$\theta_1 + \theta_2^2 = 0$$

```
//User defined procedure to compute
//nonlinear equality constraints
proc (1) = myEqProc(theta);
   local L, K;

   retp(theta[1] + theta[2].^2);
endp;
```

```
//Declare 'ctl' to be a cmtControl struct
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();

//Assign pointer to equality constraint procedure
ctl.eqProc = &myEqProc;
```

Example 2

Suppose you wish to constrain a covariance matrix to be positive definite:

```
struct comtControl ctl;
ctl = comtControlCreate();

proc eqp(b);
    retp(b'b - 1);
endp;

ctl.eqProc = &eqp;
```

6.4 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 inequality constraints are specified by assigning the procedure pointer to the *ineqProc* member of an instance of the *comtControl* structure. This procedure has one required input argument, the model parameters. This can be in the form of a *PV* structure containing the parameters, or a standard **GAUSS** Px1 matrix. Make sure to use the same form that is expected by your objective procedure. Any optional dynamic arguments passed to **comt**, will also be passed to this function.

6.4.1 Examples

Example 1

Minimize the production cost of 1000 widgets, with two variables *Labor* and *Capital*. The production equation is:

$$\begin{aligned}&\mathrm{Q} = \text{Quantity produced}\\&\mathrm{L} = \text{Labor quantity}\\&\mathrm{K} = \text{Capital quantity}\\&Q = 20\sqrt{L}\sqrt{K}\end{aligned}$$

```
proc (1) = ineqProc(theta);
   local L, K;
   L = theta[1];
   K = theta[2];

   //Return the difference between the
   //required quantity, 1000 units, and the
   //quantity produced at the given parameters
   retp(20 .* sqrt(L) .* sqrt(K) - 1000);
endp;

//Declare 'ctl' to be a comtControl structure
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();
```

```
//Assign pointer to inequality procedure
ctl.ineqProc = &ineqProc;
```

or

```
proc (1) = ineqProc(struct PV p);
   local theta, L, K;
   theta = pvUnpack(p, "theta");
   L = theta[1];
   K = theta[2];

   //Return the difference between the
   //required quantity, 1000 units, and the
   //quantity produced at the given parameters
   retp(20 .* sqrt(L) .* sqrt(K) - 1000);
endp;

//Declare 'ctl' to be a comtControl structure
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();

//Assign pointer to inequality procedure
ctl.ineqProc = &ineqProc;
```

Example 2

Suppose you wish to constrain a covariance matrix to be positive definite:

```
proc ineqp(x);
    local v;
    //Exapand 'x' into symmetric matrix
    v = xpnd(x);
    retp(minc(eigh(v)) - 1e-5);
endp;

//Declare 'ctl' to be a comtControl structure
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();

//Assign pointer to inequality procedure
ctl.ineqProc = &ineqp;
```

or

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

//Declare 'ctl' to be a comtControl structure
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();

//Assign pointer to inequality procedure
ctl.ineqProc = &ineqp;
```

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

6.5 Bounds

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

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

To bound four parameters to the ranges:

$$\begin{aligned} -10 &\leq \theta_1 \leq 10 \\ -10 &\leq \theta_2 \leq 0 \\ 1 &\leq \theta_3 \leq 10 \\ 0 &\leq \theta_4 \leq 1 \end{aligned}$$

```
//Declare 'ctl' to be a comtControl struct
//and fill with default settings
struct comtControl ctl;
ctl = comtControlCreate();

//Set separate bounds for each of four parameters
ctl.bounds = { -10 10,
               -10  0,
                 1 10,
                 0  1 };
```

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

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

is all that is necessary.

7 The COMT Procedure

The call to **COMT** has two required input arguments and one output argument.

7.1 First Input Argument: Pointer to Procedure

The first input argument is the pointer to the procedure computing the log-likelihood function and optionally the gradient and/or Hessian. You can create a pointer to a procedure by prepending the name of the procedure with an ampersand (&). For example, if your log-likelihood procedure is named **lpr**, then your call to **comt**, would look like this (at least with respect to the first input):

```
//The & makes the first input a pointer to a procedure
out = comt(&lpr, p);
```

See Section 8 for details.

7.2 Second Input Argument: Model Parameters

COMT allows you to pass the model parameters as either an Px1 matrix, where `P` is the number of parameters in the model, or inside a *PV* structure, containing the model parameters.

7.2.1 PV Parameter Instance

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

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

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

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

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

7.2.2 Fixed Parameters

A matrix or array in the model may contain a mixture of fixed values along with parameters to be estimated. In most cases it is simplest to assign some parameters as fixed by using the *active* member of the comtControl structure.

You can also specify some parameters to be fixed to their start value by 'packing' them in the PV struct with **pvPackM** which has an additional argument, called a "mask," strictly conformable to the input matrix or array indicating which elements are fixed (the corresponding element in the mask is zero) or being estimated (the corresponding element in the mask is nonzero). For example,

```
//Declare 'p' to be a 'PV' structure
struct PV p;

//Initialize PV struct
p = pvCreate();

//Create parameter matrix
b = { 1.0 0.0 0.0,
      0.5 1.0 0.2,
      0.3 0.0 1.0 };

//Create mask with 1 for each
//free parameter to be estimated,
//or a 0 for fixed parameters
b_mask = { 0 0 0,
           1 0 1,
           1 0 1 };

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

In this case there are four free parameters to be estimated, b_{21}, b_{23}, b_{31}, and b_{33}. b_{11} and b_{22} are fixed to 1.0, and b_{12}, b_{13}, and b_{32} are fixed to 0.0.

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

```
//Create symmetric matrix
vc = { 1.2 0.4,
       0.4 2.1 };

//Pack symmetric matrix, using 'pvPackS'
p = pvPackS(p,vc,"phi"); // pack symmetric matrix
```

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

```
//Create starting correlation matrix
cor = { 1.0 0.2,
        0.2 1.0 };

//Fix the diagonal elements at their starting value
msk = { 0 1,
        1 0 };



p = pvPackSM(p,cor,"R",msk);
```

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

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

which is unpacked using its number

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

7.2.3 Optional Input Argument: Instance of an comtControl Structure

The *comtControl* structure is an optional input. If used, it must be the final argument passed in to **comt**. The members of the *comtControl* structure instance set the options for the optimization. For example, suppose you want **COMT** to stop after 100 iterations:

```
//Declare 'c0' to be an comtControl structure
struct comtControl c0;

//Fill 'c0' with default values
c0 = comtControlCreate();

//Set the 'maxIters' member to 100
c0.maxIters = 100;
```

The **comtControlCreate** procedure sets all of the defaults. The default values for all the members of an *comtControl* instance can be found in that procedure, located at the top of comtutil.src in the **GAUSS** src subdirectory.

7.3 Optional Extra Input Arguments

Any data that your objective procedure needs, other than the model parameters, can be passed in as optional arguments to **comt**. These optional input arguments can be any **GAUSS** type such as, matrices, strings, arrays, structures, etc. You will pass these arguments to **comt**, between the parameter vector and the control structure. **comt** will pass them, untouched, to your objective procedure.

For a simple example, suppose that you have a least squares problem for which you need to supply the *X* matrix and *y* vector.

```
//Objective procedure with extra data arguments 'y' and 'X'
proc (1) = myObjective(b_hat, y, X, ind);
    local res;
    struct modelResults mm;
    if ind[1];
        res = y - X * b_hat;
        mm.function = res'res;
    endif;
   retp(mm);
endp;

X = //code to load or create 'X'
y = //code to load or create 'y'

//Starting parameter values
b_start = { 1, 1, 1 };

struct comtResults out;
out = comt(&myObjective, b_start, y, X);
```

Since this example does not pass in a control structure, the extra data arguments, *y* and *X* are the final inputs to **comt**.

8 The Objective Procedure

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

1. The model parameters either as a Px1 matrix, or an instance of a *PV* structure containing parameter values
2. Optional arguments; extra data matrices or arrays (other than the model parameters) used by the objective procedure.
3. Indicator vector

and one return argument

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

8.1 First Input Argument: The Model Parameters

This argument contains either a Px1 matrix, or a *PV* structure, containing the parameter matrices and arrays that you need for computing the log-likelihood and (optionally) derivatives. If the parameters are packed in a *PV* struct, the **pvUnpack** function retrieves them from the *PV* structure.

Px1 Matrix case

Below is part of a simple example in which the parameter vector contains two values. The first element will be *beta_* and the second will be *gamma_*.

```
proc lpr(p, ind);
  local beta_, gamma_;
  beta_ = p[1];
  gamma_ = p[2];
  .
  .
  .
endp;
```

PV struct case

Next is the same as the example above, but using a *PV* structure.

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

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

```
proc lpr(struct PV p, ind);
  local beta_, gamma_;
  beta_ = pvUnpacki(p, 1);
  gamma_ = pvUnpacki(p, 2);
  .
  .
  .
endp;
```

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

```
struct PV p;
p = pvCreate();

//Pack vector by index
b = { 1, 0.1 };
p = pvPacki(p,b,"beta",1);

//Pack symmetric matrix by index
g = { 1 0,
      0 1 };

p = pvPacksi(p,g,"gamma",2);
```

8.2 Optional Arguments

The optional arguments are available for use in your objective function procedure.

For example, for an objective function with a dependent variable vector and a matrix of independent variables, we have:

```
//Loadd all data from 'nlin.dat'
nldat = loadd("nlin.dat");

//Set 'y' equal to the first column of 'nldat'
//and set 'x' equal to the second column of 'nldat'
y = nldat[.,1];
x = nldat[.,2];

proc  fct(b, y, x, ind);
   struct modelResults mm;
   local dev;

   dev = y - b[1] - b[2] * exp(-b[3] * x);

   if ind[1];
      mm.function = dev'dev;
   endif;
   retp(mm);
endp;
```

8.3 Final Input Argument: Indicator Vector

The final argument is a vector with three elements set to zero or one, indicating whether or not the function, first derivatives, or second derivatives are to be computed. This vector is created inside of **comt** and passed to your objective procedure when it is called by **comt**. You do not need to create or declare the indicator vector.

1st element	if nonzero, the function is to be computed.
2nd element	if nonzero, the first derivatives are to be computed.
3rd element	if nonzero, the second derivatives are to be computed.

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

For example,

```
proc logl(b, y, x, ind);

    struct modelResults mm;
    if ind[1]; // compute objective function
       mm.function = ....
    endif;
    if ind[2]; // compute optional first
               // derivatives
       mm.gradient = ....
    endif;
    if ind[3]; // compute optional
               // second derivatives
       mm.Hessian = ....
    endif;
    retp(mm);
endp;
```

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

8.4 Output Argument: modelResults Structure

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

function	scalar log-likelihood
gradient	K×1 vector of first derivatives (optional)
Hessian	K×K matrix of second derivatives (optional)

8.5 Examples

Example 1

```
library comt;

//Create extra data matrix and vector
Q = { 0.78 -0.02 -0.12 -0.14,
     -0.02  0.86 -0.04  0.06,
     -0.12 -0.04  0.72 -0.08,
     -0.14  0.06 -0.08  0.74 };

b = { 0.76, 0.08, 1.12, 0.68 };

//Starting parameter values
x0 = { 1, 1, 1, 1 };

//Declare 'c0' to be a comtControl struct
//and fill with default values
struct comtControl c0;
c0 = comtControlCreate();

//Turn on threading of the numerical derivatives
c0.useThreads = 1;

//User created objective function
proc  qfct(x, q, b, ind);

    //Declare 'mm' to be a modelResults struct
    //local to this procedure
    struct modelResults mm;
    //Compute objective function
    //and assign to 'function' member of mm
    if ind[1];
        mm.function = .5*x'*q*x - x'b;
    endif;

    //Return modelResults struct
    retp(mm);

endp;

//Send all output to a file named 'comt1.out'
//in the current working directory
output file = comt1.out reset;

//Declare 'out' to be a comtResults struct,
//to contain the results of the optimization
struct comtResults out;

//Minimize the objective function
out = comt(&qfct,x0,q,b,c0);

//Print final results
call comtPrt(out);

//Stop sending output to file 'comt4.out'
output off;
```

Example 2

```
library comt;

//Load data and assign variables
nldat = loadd("nlin");
y = nldat[.,1];
x = nldat[.,2];

//Starting parameter values
b0 = {  0.8, 1.1, 0.2 };

//User defined objective function to
//compute sum of squared residuals
proc ssq(b, y, x, ind);
   struct modelResults mm;
   local dev;
   dev = y - fct(b,x);
   if ind[1];
       mm.function = dev'dev;
   endif;
   retp(mm);

endp;

//User defined procedure to compute nonlinear model.
//Used by objective procedure, 'ssq'
proc fct(b, x);
   local f;
   f =  b[1] + b[2] * exp(-b[3]*x);
   retp(f);
```

```
endp;

//Constrains norm of coefficients
//to be less than two. Optional data
//arguments are required in the call
//even though they aren't being used
proc ineqp(b,y,x);
   retp(2-b'b);
endp;

//Jacobian of inequality constraints.
//Optional arguments are required in the call
//even though they aren't being used
proc ineqj(b,y,x);
   retp(-2*b');
endp;

struct comtControl c0;
c0 = comtControlCreate();
c0.lineSearch = 3; // half

//Assign pointers to procedures for computing
//nonlinear inequality constraints and Jacobian
//of nonlinear inequality constraints
c0.ineqProc = &ineqp;
c0.ineqJacobian = &ineqj;
c0.gradcheck = 1;

output file = comt2.out reset;

struct comtResults out;
```

```
out = comt(&ssq,b0,y,x,c0);

call comtPrt(out);

//Compute gradient, given final parameter estimates
b_hat = pvGetParVector(out.par);
grd = gradp(&fct,b_hat,x);

//Compute and print covariance matrix
//of the parameters and other output stats
cov = (out.fct/rows(y))*invpd(grd'*grd);

print "covariance matrix";
print cov;
print;
print "standard errors of parameters";
sd = sqrt(diag(cov));
print sd';
print ;
print "t-statistics";
print (pvGetParVector(out.par)./sd)';

output off;
```

9 Managing Optimization

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

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

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

9.1 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, **COMT** 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.

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

9.3 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.

10 Run-Time Switches

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

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

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

11 COMT Structures

11.1 comtControl

matrix	`A`
matrix	`B`
matrix	`C`
matrix	`D`
scalar	`eqProc`
scalar	`ineqProc`
scalar	`ineqJacobian`
scalar	`eqJacobian`
matrix	`bounds`
matrix	`useThreads`
matrix	`algorithm`
matrix	`switch`

matrix	`lineSearch`
matrix	`active`
matrix	`maxIters`
matrix	`maxTime`
matrix	`dirTol`
matrix	`feasibleTest`
matrix	`maxTries`
matrix	`randRadius`
matrix	`gradMethod`
matrix	`hessMethod`
matrix	`gradStep`
matrix	`gradCheck`
matrix	`state`
string	`title`
scalar	`printIters`
matrix	`trustRadius`
matrix	`penalty`
matrix	`disableKey`

11.2 comtResults

struct	`PV par`
scalar	`fct`
struct	`comtLagrange lagr`
scalar	`retcode`
string	`returnDescription`
matrix	`Hessian`
matrix	`gradient`
matrix	`numIterations`
matrix	`elapsedTime`
string	`title`

11.3 comtLagrange

matrix	`lineq`
matrix	`nlineq`
matrix	`linIneq`
matrix	`nlinIneq`
matrix	`bounds`

11.4 modelResults

matrix	`function`
matrix	`gradient`
matrix	`Hessian`

12 Error Handling

12.1 Return Codes

The *retcode* member of an instance of a *comtResults* structure, which is returned by **COMT**, contains a scalar number that contains information about the status of the iterations upon exiting **COMT**. 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

10	secant update failed
11	maximum time exceeded
12	error with weights
13	quadratic program failed
14	equality Jacobian failed
15	inequality Jacobian failed
16	function evaluated as complex
20	Hessian failed to invert

12.2 Error Trapping

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

```
trap 4;
```

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

13 References

1. Brent, R.P., 1972. **Algorithms for Minimization Without Derivatives**. Englewood Cliffs, NJ: Prentice-Hall.
2. Conn, Andrew R., Gould, Nicholas I.M., Toint, Philippe L., 2000. **Trust-Region Methods**. Philadelphia: SIAM.
3. Dennis, Jr., J.E., and Schnabel, R.B., 1983. **Numerical Methods for Unconstrained Optimization and Nonlinear Equations**. Englewood Cliffs, NJ: Prentice-Hall.
4. Fletcher, R., 1987. **Practical Methods of Optimization**. New York: Wiley.
5. Gill, P. E. and Murray, W. 1972. Quasi-Newton methods for unconstrained optimization." *J. Inst. Math. Appl*., 9, 91-108.
6. Han, S.P., 1977. "A globally convergent method for nonlinear programming." *Journal of Optimization Theory and Applications*, 22:297-309.
7. Hock, Willi and Schittkowski, Klaus, 1981. **Lecture Notes in Economics and Mathematical Systems**. New York: Springer-Verlag.
8. 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.
9. Schoenberg, Ronald, 1997. "Constrained Maximum Likelihood". *Computational Economics*, 1997:251-266.

14 COMT Reference

The COMT Reference chapter describes each of the commands, procedures and functions available in **COMT**.

comt

Purpose

Computes estimates of parameters of a constrained objective function.

Library

comt

Format

out = **comt**(&*fct*, *p*);

out = **comt**(&*fct*, *p*, ...);

out = **comt**(&*fct*, *p*, ..., *ctl*);

out = **comt**(&*fct*, *p*, *ctl*);

Input

&fct	a pointer to a procedure that returns the value of the objective function.	
p	Either a Px1 matrix, or an instance of a *PV* structure containing start values for the parameters constructed using the **pvPack** functions.	
. . .	Optional extra arguments. These arguments are passed untouched to the user-provided objective function, by **comt**. You must use the same inputs that were passed into the call to **comt** that produced the *out* structure.	
ctl	Optional input, an instance of a *comtControl* structure. Normally an instance is initialized by calling **comtCreate** and members of this instance can be set to other values by the user. For an instance named *ctl*, the members are:	
	ctl.A	M×K matrix, linear equality constraint coefficients: *ctl.A* * *p* = *ctl.B* where *p* is a vector of the parameters.
	ctl.B	M×1 vector, linear equality constraint constants: *ctl.A* * *p* = *ctl.B* where *p* is a vector of the parameters.
	ctl.C	M×K matrix, linear inequality constraint coefficients: *ctl.C* * *p* ≥ *ctl.D* where *p* is a vector of the parameters.
	ctl.D	M×1 vector, linear inequality constraint constants: *ctl.C* * *p* ≥ *ctl.D* where *p* is a vector of the parameters.
	ctl.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it must accept the same input arguments as the objective procedure, except for the indicator vector. It must have one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = 0, i.e., no equality procedure.
	ctl.ineqProc	scalar, pointer to a procedure that computes the nonlinear

		inequality constraints. When such a procedure has been provided, it must accept the same input arguments as the objective procedure, except for the indicator vector. It must have one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = 0, i.e., no inequality procedure.	
	ctl.eqJacobian	scalar, pointer to a procedure that computes the Jacobian of the equality constraints. When such a procedure has been provided, it must accept the same input arguments as the objective procedure, except for the indicator vector. It must have one output argument, a matrix of derivatives of the equality constraints with respect to the parameters. Default = 0, i.e., no equality Jacobian procedure.	
	ctl.ineqJacobian	scalar, pointer to a procedure that computes the Jacobian of the inequality constraints. When such a procedure has been provided, it must accept the same input arguments as the objective procedure, except for the indicator vector. It must have one output argument, a matrix of derivatives of the inequality constraints with respect to the parameters. Default = 0, i.e., no inequality Jacobian procedure.	
	ctl.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }.	
	ctl.algorithm	scalar, descent algorithm.	
		0	Modified BFGS.
		1	BFGS (default).
		2	DFP.
		3	Newton.

	ctl.useThreads	scalar, if nonzero, the calculation of numerical derivatives will be threaded. Default = 0.	
	ctl.switch	4×2 vector, controls algorithm switching between algorithm in column 1 and column 2.	
		ctl.switch[1,1:2]	algorithm numbers to switch between.
		ctl.switch[2,1:2]	**comt** switches if function changes less than this amount.
		ctl.switch[3,1:2]	**comt** switches if this number of iterations is exceeded.
		ctl.switch[4,1:2]	**comt** switches if line search step changes less than this amount.
		`Default = { 3   1, 1e-4 1e-4, 10   10, 1e-4 1e-4 };`	
		ctl.switch[4]	**comt** switches if line search step changes less than this amount.
		To disable Switching, set to missing value, error(0).	
	ctl.lineSearch	scalar, sets line search method.	
		0	augmented trust region method (requires constraints).
		1	STEPBT (quadratic and cubic curve fit) (default).
		2	Brent’s method.

		3	half.
		4	Wolfe's condition.
	ctl.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.1.	
	ctl.active	K×1 vector, set K-th element to zero to fix it to start value. If using a *PV* struct, use the **GAUSS** function **pvGetIndex** to determine where parameters in the *PV* structure are in the vector of parameters. Default = {.}, all parameters are active.	
	ctl.maxIters	scalar, maximum number of iterations. Default = 1e5.	
	ctl.maxTime	scalar, maximum number of minutes to convergence.	
	ctl.dirTol	scalar, convergence tolerance. Iterations cease when all elements of the direction vector are less than this value. Default = 1*e*-5.	
	ctl.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.	
	ctl.maxTries	scalar, maximum number of attempts in random search. Default = 100.	
	ctl.randRadius	scalar, If zero, no random search attempted. If nonzero, it is the radius of the random search. Default = 0.001.	
	ctl.gradMethod	scalar, method for computing numerical gradient.	
		0	central difference.
		1	forward difference (default).

		2	backward difference.
	ctl.hessMethod	scalar, method for computing numerical Hessian.	
		0	central difference.
		1	forward difference (default).
		2	backward difference.
	ctl.gradStep	scalar or K×1, increment size for computing numerical gradient. If scalar, stepsize will be value times parameter estimates for the numerical gradient. If K×1, the step size for the gradient will be the elements of the vector, i.e., it will not be multiplied times the parameters.	
	ctl.gradCheck	scalar, if nonzero and if analytical gradients and/or Hessian have been provided, numerical gradients and/or Hessian are computed and compared against the analytical versions.	
	ctl.state	scalar, seed for random number generator.	
	ctl.title	string, title of run.	
	ctl.printIters	scalar, if nonzero, prints iteration information. Default = 0.	
	ctl.disableKey	scalar, if nonzero, keyboard input disabled.	

Output

out	instance of a *comtResults* structure. For an instance named *out*, the members are:	
	out.Par	instance of a *PV* structure containing the parameter estimates. Use **pvUnpack** to retrieve matrices and arrays or **pvGetParvector** to get the parameter vector.

	out.fct	scalar, function evaluated at parameters in *out.Par*.	
	out.returnDescription	string, description of return values.	
	out.Hessian	K×K matrix, Hessian evaluated at parameters in *out.Par*.	
	out.gradient	K×1 vector, gradient evaluated at the parameters in *out.Par*.	
	out.numIterations	scalar, number of iterations.	
	out.elapsedTime	scalar, elapsed time of iterations.	
	out.title	string, title of run.	
	out.lagr	an instance of a *comtLagrange* structure containing the Lagrangeans for the constraints. For an instance named *out.lagr*, the members are:	
		out.lagr.lineq	M×1 vector, Lagrangeans of linear equality constraints.
		out.lagr.nlineq	N×1 vector, Lagrangeans of nonlinear equality constraints.
		out.lagr.linIneq	P×1 vector, Lagrangeans of linear inequality constraints.
		out.lagr.nlinIneq	Q×1 vector, Lagrangeans of nonlinear inequality constraints.
		out.lagr.bounds	K×2 matrix, Lagrangeans of bounds.
		Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.	

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

Remarks

Writing the Objective Function

comt requires a user-provided procedure to compute the objective function. In the objective function, you have the option to also compute first and/or second derivatives. You also have the option to provide separate procedures to compute the equality constraints, the inequality constraints, the Jacobian of the equality constraints, and the Jacobian of the inequality constraints.

The main objective procedure has two required arguments, the model parameters, and a final argument that is a vector of zeros and ones indicating which of the results, the function, first derivatives, or second derivatives, are to be computed. Between the model parameters and the indicator vector is where you pass in any extra data arguments needed by your objective function.

comt will pass the same input arguments to all of the optional procedures that you provide, except for the indicator vector (which is only passed to the objective procedure). Therefore all of these procedures must accept the same input arguments, or you can add '...' as the final input argument if the procedure does not need certain arguments. See the nonlinear inequality procedure in the example below.

For example, the following procedure computes the objective function and the first derivatives for the function:

$$f(x) = (x_1 - y)^2 + (x_2 - z)^2$$

where *y* and *z* are predetermined constant values. This procedure requires two additional data arguments, the scalar values *y* and *z*. As you can see below, these variables are passed in to the objective procedure between the model parameters and the indicator vector.

```
//Nonlinear objective function with extra
//data input arguments: 'y' and 'z'
proc fct(x, y, z, ind);
```

```
    struct modelResults mm;

    if ind[1];
        //Calculate objective function value
        mm.function = (x[1] - y).^2 + (x[2] - z).^2;
    endif;

    if ind[2];
        //Calculate analytical gradient
        mm.gradient = zeros(2,1);
        mm.gradient[1] = 2 .* (x[1] - y);
        mm.gradient[2] = 2 .* (x[2] - z);
    endif;

    retp(mm);
endp;
```

Nonlinear Equality or Inequality Constraints Procedures **comt** will pass the same input arguments to the procedures for nonlinear equality and inequality constraints as it passes to the objective procedure. For example, below is a procedure to compute the following nonlinear inequality constraints:

$$h_1(x) = x_1^2 - x_2 \geq 4$$
$$h_2(x) = (x_1 - 2)^2 - x_2 \geq 3$$

```
//Procedure to compute inequality constraints
proc ineqp(x, ...);
    local constraint_1, constraint_2, constraint_3;

    //Compute constraints
    constraint_1 = x[1].^2 - x[2] + 4;
    constraint_2 = (x[1] - 2).^2 - x[2] + 3;

    //Return constraints as a 'num_constraints by 1' vector
    //A negative value means the current parameters
    //violate the constraint
    retp(constraint_1 | constraint_2);
endp;
```

Example

The following is a complete example. It is adapted from Arora, **Optimization: Algorithms and Applications**, 2015, CRC Press.

```
library comt;

//Nonlinear objective function
proc fct(x, y, z, ind);
    struct modelResults mm;

    if ind[1];
       //Calculate objective function value
        mm.function = (x[1] - y).^2 + (x[2] - z).^2;
    endif;

    if ind[2];
       //Calculate analytical gradient
        mm.gradient = zeros(2,1);
        mm.gradient[1] = 2 .* (x[1] - y);
        mm.gradient[2] = 2 .* (x[2] - z);
    endif;

    retp(mm);
endp;

//Procedure to compute inequality constraints
proc ineqp(x, ...);
    local constraint_1, constraint_2, constraint_3;

    constraint_1 = x[1].^2 - x[2] + 4;
    constraint_2 = (x[1] - 2).^2 - x[2] + 3;

    retp(constraint_1 | constraint_2);
endp;


//Declare 'c0' to be a comtControl struct
//and fill with default settings
struct comtControl c0;
c0 = comtControlCreate();

//Assign pointer for nonlinear inequality procedure
c0.ineqProc = &ineqp;

//Starting parameter values
x0 = { 1, 1 };

//Declare 'out' to be a comtResults
//struct to hold optimization results
struct comtResults out;

//Expand trustRadius for quicker descent
```

```
c0.trustRadius = 1;
y = 1;
z = 5;
//Minimize objective function
out = comt(&fct,x0,y,z,c0);

//Print optimization results
call comtPrt(out);
```

The output is

```
=====================================
 COMT Version 2.0.0
=====================================


Return code    = 0
Function value = 0.25391
Convergence    : normal convergence

Parameters  Estimates    Gradient
------------------------------------
x[1,1]         0.7500     -0.5000
x[2,1]         4.5625     -0.8750



Number of iterations    6
Minutes to convergence  0.00023
```

Source

comt.src

comtControlCreate

Purpose

Creates a default instance of a structure of type *comtControl*.

Library

comt

Format

s = **comtControlCreate**;

Output

s	instance of a *comtControl* structure filled in with default values.

Source

comtutil.src

comtLagrangeCreate

Purpose

Creates a default instance of a structure of type *comtLagrange*.

Library

comt

Format

```
s = comtLagrangeCreate;
```

Output

s	instance of a *comtLagrange* structure filled in with default values.

Source

comtutil.src

comtResultsCreate

Purpose

Creates a default instance of type *comtResults*.

Library

comt

Format

```
s = comtResultsCreate();
```

Output

s	instance of a *comtResults* structure filled in with default values.

Source

comtutil.src

ModelResultsCreate

Purpose

Creates a default instance of type *ModelResults*.

Library

comt

Format

s = **ModelResultsCreate**();

Output

s	instance of a *ModelResults*.

Source

comtutil.src

Index