Constrained Optimization MT 1.0

for GAUSS™ Mathematical and Statistical System

Aptech Systems, Inc.

Part Number: 007236
Version 1.0
Documentation Revision: 1867 May 14, 2010

Contents

1 Installation

1.1 UNIX/Linux/Mac

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

1.1.1 Download

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

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

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

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

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

   – or –

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

1.1.2 CD

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

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

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

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

   – or –

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

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

1.2 Windows

1.2.1 Download

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

1.2.2 CD

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

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

1.2.3 64-Bit Windows

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

1.3 Difference Between the UNIX and Windows Versions

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

Getting Started 2

2.1 Getting Started

GAUSS 10.0.0+ is required to use these routines. See **_rtl_ver** in src/gauss.dec.

2.1.1 Setup

In order to use the procedures in the **Constrained Optimization MT** 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,pgraph;
```

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

```
#include comt.sdf
```

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

The **COMT** version number is stored in a global variable:

_comt_ver	3×1 matrix, the first element contains the major version number of the **COMT** 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.

2.1.2 README Files

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

Constrained Optimization MT 3

written by

Ronald Schoenberg

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

3.1 Special Features in Constrained Optimization MT

3.1.1 Structures

In **COMT** the same procedure that computes the objective function will also be used to compute analytical derivatives if they are being provided. Its return argument is a

comtResults structure with three members: a scalar function value, a K×1 vector of first derivatives, and a K×K matrix of second derivatives. Of course, the derivatives are optional, or even partially optional, i.e., you can compute a subset of the derivatives, if you like, and the remaining will be computed numerically. This procedure will have an additional argument, which tells the function 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.

The new **COMT** uses the **DS** and **PV** structures, which are in the **GAUSS** Run-Time Library. The **DS** structure is completely flexible, allowing you to pass anything you can think of into your procedure. The **PV** structure revolutionizes how you pass the parameters into the procedure. No longer 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 function uses matrices or arrays, you can store them directly into the **PV** structure and remove them as matrices or arrays with the parameters already plugged into them. The **PV** structure can handle matrices and arrays in which some of the elements are fixed and some free. It remembers the fixed parameters and knows where to plug in the current values of the free parameters. It can also handle symmetric matrices, in which parameters below the diagonal are repeated above the diagonal.

There is no longer any need to use global variables. Anything the procedure needs can be passed into it through the **DS** structure, and **COMT** uses control structures rather than global variables to set options. This means, in addition to thread safety, that it is straightforward to nest a call to **COMT** inside another call to **COMT** or inside Run-Time Library functions like **QNewtonmt**, **QProgmt**, and **EQsolvemt**.

3.1.2 Threading

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

```
struct comtControl c0;
c0 = comtControlCreate;
c0.useThreads = 1;
```

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

3.1.3 Augmented Lagrangian Penalty Line Search Method

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

3.2 The Objective Function

COMT is a set of procedures for estimating the parameters of models via the optimization 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.

3.3 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

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

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 loss of generality, we may assume that the linear constraints and bounds have been incorporated into G and H.

The direction, δ is the solution to the quadratic program

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

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

This solution requires that Σ be positive semi-definite.

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

Line Search

Define the merit function

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

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

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

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

3.3.1 The Secant Algorithms

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

3.3.2 Line Search Methods

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

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

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

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 optimization problems without constraints on parameters.

STEPBT

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

BRENT

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

HALF

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

requires one), but it is the least efficient in that it is not very likely to find the step length that decreases m the most.

3.3.3 Active and Inactive Parameters

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

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

3.4 Constraints

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

3.4.1 Linear Equality Constraints

Linear constraints are of the form:

$$A\theta = B$$

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

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

```
struct comtControl ctl;
ctl = comtControlCreate;
ctl.A = { 1 0 -1 0 };
ctl.B = { 0 };
```


3.4.2 Linear Inequality Constraints

Linear constraints are of the form:

$$C\theta \geq D$$

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

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

```
struct comtControl ctl;
ctl = comtControlCreate;
ctl.C = { 1 0 -1 0,
          0 1  0 1 };
ctl.D = {  0,
           10 };
```

3.4.3 Nonlinear Equality

Nonlinear equality constraints are of the form:

$$G(\theta) = 0$$

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

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

```
struct comtControl ctl;
ctl = comtControlCreate;

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

ctl.eqProc = &eqp;
```

3.4.4 Nonlinear Inequality

Nonlinear inequality constraints are of the form:

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

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

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

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

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

3.4.5 Bounds

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

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

To bound four parameters:

```
struct comtControl ctl;
ctl = comtControlCreate;
ctl.Bounds = { -10 10,
```

```
-10  0,
  1 10,
  0  1 };
```

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

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

is all that is necessary.

3.5 The COMT Procedure

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

3.5.1 First Input Argument: Pointer to Procedure

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

3.5.2 Second Input Argument: PV Parameter Instance

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

The advantage of the **PV** structure is that it permits you to retrieve the parameters in the form of matrices and/or arrays ready for use in calculating your log-likelihood. The

matrices and arrays are defined in your command file when the start values are set up. It isn't necessary that a matrix or array be completely free parameters to be estimated. There are **pvPack** functions that take mask arguments defining what is a parameter versus what is a fixed value. There are also functions for handling symmetric matrices where the parameters below the diagonal are duplicated above the diagonal.

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

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

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

A matrix or array in the model may contain a mixture of fixed values along with parameters to be estimated. This type of matrix or array uses **pvPackm** which has an additional argument, called a "mask", strictly conformable to the input matrix or array indicating which elements are fixed (the corresponding element in the mask is zero) or being estimated (the corresponding element in the mask is nonzero). For example,

```
struct PV p;
p = pvCreate;

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

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

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

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

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

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

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

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

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

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

which is unpacked using its number

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

3.5.3 Third Input Argument: DS Data Instance

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

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

Data in Matrices or Arrays

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

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

3.5.4 Fourth Input Argument: comtControl Instance

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

```
struct comtControl c0;
c0 = comtControlCreate;

c0.maxIters = 100;
```

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

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

and one return argument

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

3.6.1 First Input Argument: PV Parameter Instance

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

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

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


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

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

```
struct PV p;
p = pvCreate;

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

3.6.2 Second Input Argument: DS Data Instance

COMT passes the **DS** instance you have constructed completely untouched. You can, therefore, design this instance completely for your convenience in computing the objective function and optionally its derivatives.

3.6.3 Third Input Argument: Indicator Vector

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

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

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

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

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

For example,

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

    struct modelResults mm;
    if ind[1];  // compute log-likelihood
       mm.Function = ....
    endif;
    if ind[2];  // compute optional first derivatives
```

```
            mm.Gradient = ....
        endif;
        if ind[3];  // compute optional second derivatives
            mm.Hessian = ....
        endif;
        retp(mm);
    endp;
```

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

3.6.4 Output Argument: modelResults Instance

The return argument for your objective function procedure is an instance of a **modelResults** structure. The members of this structure are

Function	scalar value of objective function given parameters
Gradient	$1 \times K$ vector of first derivatives (optional)
Hessian	$K \times K$ matrix of second derivatives (optional)

3.6.5 Examples

```
#include comt.sdf


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

d0[1].dataMatrix =
library comt;
    { 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 };

d0[2].dataMatrix = { 0.76, 0.08, 1.12, 0.68 };

struct PV p0;
p0 = pvPack(pvCreate,1|1|1|1,"x");

struct comtControl c0;
c0 = comtControlCreate;
c0.useThreads = 1;

proc  qfct(struct PV p, struct DS d, ind);
    local x,q,b;
    struct modelResults mm;

    x = pvUnpack(p,"x");
    q = d[1].dataMatrix;
    b = d[2].dataMatrix;

    if ind[1];
        mm.function = .5*x'*q*x - x'b;
    endif;
    retp(mm);

endp;

output file = comt1.out reset;


struct comtResults out;
out = comt(&qfct,p0,d0,c0);

call comtPrt(out);

output off;
```

```
library comt;
#include comt.sdf


/* -- data -- */

  nldat = loadd("nlin");

struct DS d0;
d0 = reshape(d0,2,1);
d0[1].dataMatrix = nldat[.,1];
d0[2].dataMatrix = nldat[.,2];

struct PV p0;
p0 = pvPack(pvCreate,.8|1.1|.2,"B");


proc  ssq(struct PV p, struct DS d, ind);

   struct modelResults mm;

   local y,dev;
   y = d[1].dataMatrix;
   dev = y - fct(p,d);

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

endp;


proc fct(struct PV p, struct DS d);

   local x,b,f;
   b = pvUnpack(p,"B");
   x = d[2].dataMatrix;
   f =  b[1] + b[2] * exp(-b[3]*x);
```

```
   retp(f);

endp;

proc ineqp(struct PV p, struct DS d);     // constrains norm
   local b;                               // of coefficients
   b = pvUnpack(p,"B");                   // to be less than
   retp(2-b'b);                           // two
endp;

proc ineqj(struct PV p, struct DS d);     // jacobian of
    local b;                              // inequality
    b = pvUnpack(p,"B");                  // constraints
   retp(-2*b');
endp;


struct comtControl c0;
c0 = comtControlCreate;
c0.lineSearch = 3;                        // half
c0.ineqProc = &ineqp;
c0.ineqJacobian = &ineqj;
c0.gradcheck = 1;

output file = comt2.out reset;

struct comtResults out;
out = comt(&ssq,p0,d0,c0);

call comtPrt(out);

grd = gradmt(&fct,p0,d0);
cov = (out.fct/rows(d0[1].dataMatrix))*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;
```

3.7 Managing Optimization

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

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

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

3.7.1 Scaling

For best performance, the diagonal elements of the Hessian matrix should be roughly equal. If some diagonal elements contain numbers that are very large and/or very small with respect to the others, **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.

3.7.2 Condition

The specification of the model can be measured by the condition of the Hessian. The solution of the problem is found by searching for parameter values for which the gradient is zero. If, however, the Jacobian of the gradient (i.e., the Hessian) is very small for a particular parameter, then **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.

3.7.3 Starting Point

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

3.7.4 Example

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

```
library comt;
#include comt.sdf

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

d0[1].dataMatrix = {
   109,
   149,
   149,
   191,
   213,
   224 };

d0[2].dataMatrix = {
    1,
    2,
    3,
    5,
    7,
   10 };

struct PV p0;
p0 = pvPack(pvCreate, 100|0.75 ,"B");


proc  ssq(struct PV p, struct DS d, ind);

   struct modelResults mm;
```

```
    local y,f,g;
    y = d[1].dataMatrix;
    f = fct(p,d);

    if ind[1];
        mm.function = (y - f)'(y - f);
    endif;
    if ind[2];
        g = grd(p,d);
        mm.gradient = -2*g'(y-f);
    endif;

    retp(mm);

endp;


proc fct(struct PV p, struct DS d);
    local b,x;
    b = pvUnpack(p,"B");
    x = d[2].dataMatrix;
    retp( b[1]*(1 - exp(-b[2]*x)) );
endp;

proc grd(struct PV p, struct DS d);
    local b,x,f,g;
    b = pvUnpack(p,"B");
    x = d[2].dataMatrix;
    g = zeros(rows(x),rows(b));
    f = exp(-b[2]*x);
    g[.,1] = 1 - f;
    g[.,2] = b[1]*x.*f;
    retp(g);
endp;


struct comtControl c0;
c0 = comtControlCreate;
```

```
c0.gradCheck = 1;

struct comtResults out;
out = comt(&ssq,p0,d0,c0);

call comtPrt(out);

g = grd(out.par,d0);
cov = (out.fct/rows(d0[1].dataMatrix))*invpd(g'*g);

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)';
```


and the output looks like this:

```
analytical gradient        numerical gradient
-932.8196153813061100      -932.8196142625807900
-18609.6357937713840000    -18609.6358822984800000

============================================================
 COMT Version 1.0.0                    1/29/2010   2:59 pm
============================================================

return code =    0
normal convergence

Objective Function         1168.01

  Parameters    Estimates  Gradient
------------------------------------------------------------
```

```
  B[1,1]        213.8094      0.0000
  B[2,1]          0.5472     -0.0006

Number of iterations    3706
Minutes to convergence     0.01694
covariance matrix

       101.75603      -0.62853681
     -0.62853681     0.0072885202

standard errors of parameters
       10.087419      0.085372831

t-statistics
       21.195650        6.4099725
```

3.8 Run-Time Switches

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

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

V	Set **DirTol**

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

3.9 COMT Structures

3.9.1 comtControl

matrix	A
matrix	B
matrix	C
matrix	D
scalar	EqProc
scalar	IneqProc
scalar	IneqJacobian
scalar	EqJacobian
matrix	Bounds
matrix	Algorithm
matrix	Switch
matrix	LineSearch
matrix	Active
matrix	MaxIters

matrix	DirTol
matrix	FeasibleTest
matrix	MaxTries
matrix	RandRadius
matrix	GradMethod
matrix	HessMethod
matrix	GradStep
matrix	HessStep
matrix	GradCheck
matrix	State
string	Title
scalar	PrintIters
matrix	TrustRadius
matrix	Penalty
matrix	DisableKey

3.9.2 comtResults

struct	PV Par
scalar	Fct
struct	ComtLagrange lagr
scalar	Retcode

string	ReturnDescription
matrix	Hessian
matrix	Gradient
matrix	NumIterations
matrix	ElapsedTime
string	Title

3.9.3 comtLagrange

matrix	Lineq
matrix	Nlineq
matrix	Linineq
matrix	Nlinineq
matrix	Bounds

3.9.4 modelResults

matrix	Function
matrix	Gradient
matrix	Hessian

3.10 Error Handling

3.10.1 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
9	error with constraints
10	secant update failed
11	maximum time exceeded
13	quadratic program failed
14	equality Jacobian failed
15	inequality Jacobian failed
16	function evaluated as complex
20	Hessian failed to invert

3.10.2 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.

3.11 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 Optimization". *Computational Economics*, 1997:251-266.

COMT Reference 4

comt

PURPOSE Computes estimates of parameters of a constrained objective function.

LIBRARY **comt**

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

INPUT *&modelProc* a pointer to a procedure that returns the value of the objective function given the parameters.

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

data instance or matrix of instances of a **DS** structure containing data. It is passed to the user-provided procedure pointed at by `&fct` to be used in the objective function. A scalar or vector of **DS** instances passed to **comt** are passed unchanged to the user-provided procedure that computes the objective function. All members of all the instances can be set at the discretion of the programmer.

ctl 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 has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a vector of computed equality constraints. For more details

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

`ctl.`**IneqProc** scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, and an instance of a **DS** data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure.

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

`ctl.`**IneqJacobian** scalar, pointer to a procedure that computes the Jacobian of the inequality constraints. When such a procedure has been provided, it has two input arguments, an instance of a **PV** parameter structure, an instance of a **DS** data structure, and one output argument, a matrix of derivatives of the inequality constraints with respect to the parameters. Default = {.}, i.e.,

no inequality Jacobian procedure.

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

ctl.Algorithm scalar, descent algorithm.

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

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

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

- **ctl.Switch[1]** algorithm number to switch to.
- **ctl.Switch[2]** **comt** switches if function changes less than this amount.
- **ctl.Switch[3]** **comt** switches if this number of iterations is exceeded.
- **ctl.Switch[4]** **comt** switches if line search step changes less than this amount.

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

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

- **0** augmented trust region method (requires constraints)
- **1** STEPBT (quadratic and cubic curve fit) (default)
- **2** Brent's method
- **3** 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 = .001.

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

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

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

`ctl.`**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 is attempted. If nonzero, it is the radius of the random search. Default = .001.

ctl.GradMethod scalar, method for computing numerical gradient.

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

ctl.HessMethod scalar, method for computing numerical Hessian.

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

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

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

times the parameters.

ctl.**GradCheck**	scalar, if nonzero and if analytical gradients and/or Hessian have been provided, numerical gradients and/or Hessian are computed and compared against the analytical versions.
ctl.**State**	scalar, seed for random number generator.
ctl.**Title**	string, title of run.
ctl.**printIters**	scalar, if nonzero, prints iteration information. Default = 0.
ctl.**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.structmemRetcode` return code:

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

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

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

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

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

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

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

```
proc fct(struct PV p, struct DS d, ind);
    local x,h0;
    struct modelResults mm;

    x = pvUnpack(p,"x");

    if ind[1];
        mm.function = 100*(x[2]-x[1]^2)^2 + (1-x[1])^2 +
          90*(x[4]-x[3]^2)^2 + (1-x[3])^2 + 10.1*((x[2]-1)^2 +
          (x[4]-1)^2) + 19.8*(x[2]-1)*(x[4]-1);
    endif;

    if ind[2];
        local a,b;
```

```
        a = x[2] - x[1]^2;
        b = x[4] - x[3]^2;
        mm.gradient = zeros(4,1);
        mm.gradient[1] = -2*(200*x[1]*a + 1 - x[1]);
        mm.gradient[2] = 2*(100*a + 10.1*(x[2]-1) +
          9.9*(x[4]-1));
        mm.gradient[3] = -2*(180*x[3]*b + 1 - x[3]);
        mm.gradient[4] = 2*(90*b + 10.1*(x[4]-1) +
          9.9*(x[2]-1));
    endif;


    if ind[3];
        mm.hessian = d.dataMatrix;
        mm.hessian[1,1] = -2*(200*(x[2]-x[1]^2) -
          400*x[1]^2 - 1);
        mm.hessian[1,2] = -400*x[1];
        mm.hessian[2,1] = mm.hessian[1,2];
        mm.hessian[3,3] = -2*(180*(x[4]-x[3]^2) -
          360*x[3]^2 - 1);
        mm.hessian[4,3] = -360*x[3];
        mm.hessian[3,4] = mm.hessian[4,3];
     endif;

     retp(mm);
endp;
```


Nonlinear Equality or Inequality Constraints Procedures

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

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

EXAMPLE The following is a complete example. It is Problem 32 in Hock and Schittkowski, _Test_Examples_for_ Nonlinear_Programming_Codes, 1981, Springer-Verlag

```
library comt;
#include comt.sdf

struct DS d0;
d0 = dsCreate;

proc fct(struct PV p, struct DS d, ind);
    local x,h0;
    struct modelResults mm;

    x = pvUnpack(p,"x");

    if ind[1];
        mm.function = (x[1] + 3*x[2] + x[3])^2 + 4*(x[1] -
          x[2])^2;
    endif;

    if ind[2];
        mm.gradient = zeros(3,1);
        mm.gradient[1] = 10*x[1] - 2*x[2] + 2*x[3];
        mm.gradient[2] = -2*x[1] + 26*x[2] + 6*x[3];
        mm.gradient[3] = 2*x[1] + 6*x[2] + 2*x[3];
    endif;


    if ind[3];
```

```
            mm.hessian = zeros(3,3);
            mm.hessian[1,1] = 10;
            mm.hessian[1,2] = -2;
            mm.hessian[1,3] = 2;

            mm.hessian[2,1] = -2;
            mm.hessian[2,2] = 26;
            mm.hessian[2,3] = 6;

            mm.hessian[3,1] = 2;
            mm.hessian[3,2] = 6;
            mm.hessian[3,3] = 2;
        endif;

        retp(mm);
    endp;

    proc ineqp(struct PV p, struct DS d);
        local x;
        x = pvUnpack(p,"x");
        retp(6*x[2] + 4*x[3] - x[1]^3 - 3);
    endp;

    proc ineqj(struct PV p, struct DS d);
        local x;
        x = pvUnpack(p,"x");
       retp(-3*x[1]^2~6~4);
    endp;

    proc eqp(struct PV p, struct DS d);
        local x;
        x = pvUnpack(p,"x");
        retp(1-sumc(x));
    endp;

    proc eqj(struct PV p, struct DS d);
        local x;
        x = pvUnpack(p,"x");
       retp(-ones(1,3));
```

```
endp;

struct comtControl c0;
c0 = comtControlCreate;

c0.ineqProc = &ineqp;
c0.ineqJacobian = &ineqj;
c0.eqProc = &eqp;
c0.eqJacobian = &eqj;

c0.bounds = { 0 1e256 };


struct PV p0;
p0 = pvPack(pvCreate,.1|.7|.2,"x");


struct comtResults out;
out = comt(&fct,p0,d0,c0);

call comtPrt(out);

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

print;
print "nonlinear equality Lagrangeans";
print out.lagr.nlineq;
print;
print "nonlinear inequality Lagrangeans";
print out.lagr.nlinineq;
print;
print "boundary Lagrangeans";
print out.lagr.bounds;
```

The output is

```
============================================================
 COMT Version 1.0.0                  2/02/2010   3:04 pm
============================================================

return code =    0
normal convergence

Objective Function         1.00000

  Parameters    Estimates  Gradient
------------------------------------------------------------
x[1,1]    0.0000      2.0000
x[2,1]    0.0000      6.0000
x[3,1]    1.0000      2.0000

Number of iterations    4520
Minutes to convergence     2.04775


published solution
     0       0      1

nonlinear equality Lagrangeans
      -1.9999998

nonlinear inequality Lagrangeans
      0.00000000

boundary Lagrangeans

      0.00000000       0.00000000
       3.9999998       0.00000000
      0.00000000       0.00000000
```

SOURCE comt.src

COMTControlCreate

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

LIBRARY **comt**

FORMAT **s** = COMTControlCreate;

OUTPUT *s* instance of type **COMTControl**.

SOURCE comtutil.src

COMTLagrangeCreate

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

LIBRARY **comt**

FORMAT **s** = COMTLagrangeCreate;

OUTPUT *s* instance of type **COMTLagrange**.

SOURCE comtutil.src

COMTResultsCreate

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

LIBRARY **comt**

FORMAT **s** = COMTResultsCreate;

OUTPUT *s* instance of type **COMTResults**.

SOURCE comtutil.src

ModelResultsCreate

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

LIBRARY **comt**

FORMAT **s** = ModelResultsCreate;

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

SOURCE comtutil.src

COMTPrt

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

LIBRARY **comt**

FORMAT **out** = COMTPrt(*out*);

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

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

REMARKS The call to **COMT** can be nested in the call to **COMTPrt**:

```
call COMTPrt(COMT(&modelProc,par,data,ctl));
```

SOURCE comtutil.src

Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

W