Optimization MT 2.0

for GAUSS™ Mathematical and Statistical System

Version 2.0
Monday, April 17, 2017

Contents

1 Installation

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 downloaded .zip file.

1.1 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 **OPTMT** module contains a set of procedures for the solution of the optimization problem.

GAUSS version 16+ is required to use these routines.

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

_optmt_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 **Optimization MT** license.

2.1 README Files

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

2.2 Setup

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

```
library optmt;
```

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

3 Special Features in Optimization MT

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

3.1 Structures

3.1.1 modelResults structure

In **OPTMT** the objective function will also be used to compute the analytical derivatives when they are provided. Its return argument is a *modelResults* structure with three members:

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 func
    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;
```

The derivatives are optional, or even partially optional. For example, you can compute a subset of the derivatives if you like and the remaining will be computed numerically. The procedure to compute the objective function has an additional input argument that 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 won't have to be repeated.

3.1.2 Parameter Vector (PV) Structure

OPTMT 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 model parameters as vectors, matrices or n-dimensional arrays. For cases in which your parameters are simply a vector, **OPTMT** 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

Any inputs that your procedure needs, other than the parameters of the model, can be passed into **OPTMT** as optional, dynamic arguments. These optional arguments will be passed directly, and untouched, to your objective function.


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

//Inputs to objective function for
//OPTMT 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
//OPTMT 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
//OPTMT 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
```

```
//OPTMT 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 **OPTMT**, required the use of the *DS* structure for this purpose. The current version is backwards compatible with version 1.0, 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, rather than global variables, to set optimization options. This means, in addition to thread safety, that it will be straightforward to nest calls to **OPTMT** inside of a call to **OPTMT**, or other multi-threaded **GAUSS** functions..

3.2 Threading

You can enable **GAUSS** level threads inside of **OPTMT** by setting the *useThreads* member of the *optmtControl* instance:

```
//Declare c0 to be an instance of an optmtControl structure
struct optmtControl c0;

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

//Turn on internal threading
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 **OPTMT** source code, but does not control the low-level threads that are internal to the **GAUSS** intrinsic functions.

4 The Objective Function

The procedure **OPTMT** finds values for the parameters in θ such that $F(\theta)$ is minimized. **OPTMT** has been designed to make the specification of the function and the handling of the data convenient. The user supplies a procedure that computes F given the parameters in θ.

5 Algorithm

OPTMT includes four descent methods, BFGS, DFP, Newton and steepest descent. In these methods 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 $K \times 1$ vector and ρ a scalar step length.

Direction

Define

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

The direction, δ is the solution to

$$\Sigma(\theta_t)\delta = \Psi(\theta_t)$$

This solution requires that Σ be positive semi-definite.

Line Search

Define the merit function

$$m(\theta) = L - \sum_{\ell} \lambda_{\ell} h_{\ell}(\theta)$$

h_{ℓ} is the ℓ-th constraint and λ_{ℓ} the Lagrangean coefficient of the ℓ-th bounds.

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

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 **OPTMT** 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 **OPTMT**.

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**.

5.1.1 Secant Methods (BFGS and DFP)

The BFGS and DFP methods are complementary methods which, like the Newton Method, use both the 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 these methods take more iterations than the Newton method, they converge in less overall time (unless analytical second derivatives are available in which case it 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 **OPTMT**, 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. **OPTMT** 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

member of the *optmtControl* instance is greater than zero. The default setting for *randRadius* is greater than zero by default.

5.2.1 STEPBT

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

5.2.2 BRENT

This method is a variation on the *golden section* method due to Brent (1972). In this method, the function is evaluated at a sequence of test values for 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.

5.2.3 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.

5.2.4 WOLFE

The Strong Wolfe condition we have

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

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

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.

6 Bounds

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 *optmtControl* structure.

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

To bound four parameters:

```
//Declare 'ctl' to be an optmtControl struct
struct optmtControl ctl;

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

//Set lower and upper bounds for 4 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 OPTMT Procedure

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. See Section 8 for details.

7.2 Second Input Argument: Model Parameters

OPTMT allows you to pass the model parameters as either a 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:

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

//Set PV defaults (required)
p = pvCreate();

//Create 3x1 vector
garch = { .1, .1, .1 };

//Add 3x1 vector, 'garch' to 'p' with name "garch"
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 *optmtControl* structure.

You can also specify some parameters as fixed by 'packing' them into the *PV* structure using the function **pvPackM**. **pvPackM** has an additional argument, called a "mask," strictly element-by-element 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' structure
```

```
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}. While b_{11} and b_{22} are fixed to 1.0, and b_{12}, b_{13}, and b_{32} are fixed to 0.0.

7.2.3 PV Structures with Symmetric Matrices

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");
```

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);
```

7.2.4 Unpack PV Structures by Index

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);
```

7.2.5 Optional Input Argument: Instance of an optmtControl Structure

The members of the *optmtControl* structure instance set the options for the optimization. For example, suppose you want **OPTMT** to stop after 100 iterations:

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

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

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

The **optmtControlCreate** procedure sets all of the defaults. The default values for all the members of an *optmtControl* instance can be found in that procedure, located at the top of optmtutil.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 **optmt**. These optional input arguments can be any **GAUSS** type such as, matrices, strings, arrays, structures, etc. You will pass these arguments to **optmt**, between the parameter vector and the control structure. **optmt** 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 optmtResults out;
out = optmt(&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 **optmt**.

8 The Objective Procedure

OPTMT requires that you write a procedure computing the value of the objective function. The output from this procedure is a *modelResults* structure containing the value 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* struct, 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

You may pass in data that your objective function procedure needs, other than the parameter vector, as extra optional arguments. These arguments will be passed untouched to your objective function, and will be passed in between the *PV* structure and the indicator variable.

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

```
//Optional arguments 'y' and 'x' are passed in
//between the 'PV' struct and the indicator variable
proc  fct(struct PV p, y, x, ind);
   struct modelResults mm;
   local dev, b;

   b = pvUnpack(p,"B");
   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. Note that this argument is created and passed to your objective function when it is called by **optmt**. Your program does not need to declare or initialize 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 myobjective(b, y, x, ind);
    local y,x;

    struct modelResults mm;

    // compute objective function
    if ind[1];
       mm.function = ....
    endif;

    // compute optional first derivatives
    if ind[2];
       mm.gradient = ....
    endif;

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

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

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

modelResults structure	
function	scalar log-likelihood
gradient	Kx1 vector of first derivatives (optional)
Hessian	K×K matrix of second derivatives (optional)

8.5 Examples

Example 1

A procedure that takes one extra data argument, a matrix *omega*, and calculates the gradient and Hessian as well as the scalar objective function.

```
proc fct(x, omega, ind);

    //Declare 'mm' to be a modelResults struct
    //that is local to this procedure
    struct modelResults mm;

    //Calculate objective function
    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;

    //Calculate gradient
    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;

    //Calculate Hessian
    if ind[3];
         mm.Hessian = omega;
         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;

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


Example 2

A procedure that takes two extra data arguments, *q* and *b*, and calculates only the scalar objective function.

```
proc qfct(x, q, b, ind);

    //Declare 'mm' to be a modelResults struct
    //that is local to this procedure
    struct modelResults mm;

    if ind[1];
        //Calculate objective function
        mm.Function = .5*x'*q*x - x'b;
    endif;

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

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, **OPTMT** 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 **OPTMT** has difficulty determining the optimal values since a large region of the function appears virtually flat to **OPTMT**. 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 as starting values 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.

9.4 Example

```
//Activate OPTMT library
library optmt;
```

```
//Create extra data needed by objective function
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 };

//Create a 4x1 matrix of 1's to be the starting values
x0 = ones(4, 1);

//Declare 'c0' to be an 'optmtControl' struct
struct optmtControl c0;

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

//Set lower and upper bounds
//for each of the 4 parameters
c0.bounds = { 0 1,
              0 1,
              1 2,
              1 2 };

//Procedure to be minimized
proc qfct(x, Q, b, ind);

    //Declare 'mm' to be a 'modelResults'
    //local to this procedure, 'qfct'
    struct modelResults mm;
```

```
    if ind[1];

        //Calculate the scalar objective
        mm.Function = .5*x'*Q*x - x'b;
    endif;

    //Return the 'modelResults' struct
    retp(mm);
endp;

//Send all printed output to new file 'optmt1.out'
output file = optmt1.out reset;

//Declare 'out' to be an 'optmtResults' struct
//to hold the results of the optimization.
struct optmtResults out;

//Minimize 'qfct'
out = optmt(&qfct,x0,Q,b,c0);

//Print report of optmization results
call optmtPrt(out);

//Stop sending printed output to 'optmt1.out'
output off;
```

and the output looks like this:

```
=====================================
 Optmt Version 2.0.0
=====================================
```

```
return code =    0
function                  -2.07055
normal convergence
  Parameters    Estimates   Gradient
------------------------------------
   x[1,1]        1.0000     -0.3892
   x[2,1]        0.1126      0.0000
   x[3,1]        1.8731      0.0000
   x[4,1]        1.3015      0.0000


Number of iterations    9
Minutes to convergence     0.00118

Lagraneans

0.0000000 0.389223
0.0000000 0.000000
0.0000000 0.000000
0.0000000 0.000000
```

If the Lagrangeans are "empty" matrices, the associated constraints are not active. If they are zeros but not "empty" matrices, then they are still inactive at the solution but were active at some point during the iterations.

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
V	Set *Tol*

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

11 OPTMT Structures

11.1 optmtControl

matrix	bounds
matrix	algorithm
matrix	switch
matrix	lineSearch
matrix	active
matrix	maxTries
matrix	maxIters
matrix	maxTime
matrix	tol
matrix	feasibleTest
matrix	randRadius
matrix	gradMethod
matrix	hessMethod

matrix	gradStep
matrix	gradCheck
matrix	state
string	title
scalar	printIters
matrix	disableKey

11.2 optmtResults

struct	PV par
scalar	fct
struct	lagrangeans
scalar	retcode
string	returnDescription
matrix	Hessian
matrix	gradient
matrix	numIterations
matrix	elapsedTime
string	title

11.3 modelResults

modelResults structure	
matrix	function
matrix	gradient
matrix	Hessian

12 Error Handling

12.1 Return Codes

The *Retcode* member of an instance of a *optmtResults* structure, which is returned by **OPTMT**, contains a scalar number that contains information about the status of the iterations upon exiting **OPTMT**. 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
16	function evaluated as complex
20	Hessian failed to invert

34	data set could not be opened

12.2 Error Trapping

Setting the *printIters* member of an instance of a *optmtControl* 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 **OPTMT** not to send the messages to the screen:

```
trap 4;
```

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

13 References

1. Amemiya, Takeshi, 1985. **Advanced Econometrics**. Cambridge, MA: Harvard University Press.
2. Brent, R.P., 1972. **Algorithms for Minimization Without Derivatives**. Englewood Cliffs, NJ: Prentice-Hall.
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. Judge, G.G., R.C. Hill, W.E. Griffiths, H. Lütkepohl and T.C. Lee. 1988. **Introduction to the Theory and Practice of Econometrics**. 2nd Edition. New York:Wiley.
7. Judge, G.G., W.E. Griffiths, R.C. Hill, H. Lütkepohl and T.C. Lee. 1985. **The Theory and Practice of Econometrics**. 2nd Edition. New York:Wiley.

14 Command Reference

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

optmt

Purpose

Computes estimates of parameters of an objective function.

Library

optmt

Format

out = **optmt**(&*fct*, *p*);
out = **optmt**(&*fct*, *p*, ...);
out = **optmt**(&*fct*, *p*, ..., *ctl*);
out = **optmt**(&*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 **optmt**. You must use the same inputs that were passed into the call to **optmt** that produced the *out* structure.		
ctl	Optional input, an instance of an *optmtControl* structure. Normally an instance is initialized by calling **optmtCreate** and members of this instance can be set to other values by the user. For an instance named *ctl*, the members are:		
	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.	
		1	BFGS (default).
		2	DFP.
		3	Newton.
		4	Steepest Descent.
	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]	**optmt** switches if function changes less than this amount.
		ctl.switch [3,1:2]	**optmt** switches if this number of iterations is exceeded.
		ctl.switch [4,1:2]	**optmt** switches if line search step changes less than this amount.
			`Default = { 1   3,` `1e-4 1e-4,`

		10 10, 1e-4 1e-4 };	
	ctl.lineSearch	scalar, sets line search method.	
		1	STEPBT (quadratic and cubic curve fit) (default).
		2	Brent's method.
		3	half.
		4	Strong Wolfe's condition.
	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 = 10000.	
	ctl.tol	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 derivatives. If scalar, stepsize will be value times parameter estimates for the numerical gradient or Hessian. If K×1, *gradStep* 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.maxTime	scalar, maximum number of minutes to convergence.	
	ctl.disableKey	scalar, if nonzero, keyboard input disabled.	

Output

out	instance of an *optmtResults* 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.lagrangeans	K×2 matrix, Lagrangeans for the bounds constraints. 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.
		16	function evaluated as complex.
		20	Hessian failed to invert.
		34	data set could not be opened.

Remarks

Writing the Objective Function

optmt 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.

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. Full examples are below.

Examples

Example 1: Basic Usage

The following is a complete example. The code can be found in the GAUSSHOME/examples/optmt directory, named `optmt_basic.e`.

```
library optmt;

//Objective function
proc fct(x, ind);

   //Declare 'mm' to be a modelResults
   //struct, local to this procedure
   struct modelResults mm;

   //If the first element of the indicator vector
   //is non-zero, calculate the objective function
   if ind[1];
      //Assign the value of the objective function to the
      //'function' member of the 'modelResults' struct
      mm.function = 4 * x[1].^2 - 4 * x[1] * x[2]
                  + 3 * x[2]^2 + x[1];
   endif;
```

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

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

//Declare 'out' to be an optmtResults struct
//to hold the optimization results
struct optmtResults out;

//Minimize objective function
out = optmt(&fct,x0);

//Print optimtization results
call optmtPrt(out);
```

The above code will print output similar to the following:

```
=======================================
 Optmt
=======================================

Return code    =    0
Function value =   -0.09375
Convergence    :    normal convergence

Parameters  Estimates   Gradient
---------------------------------------
x[1,1]     -0.1875      0.0000
```

```
x[2,1]     -0.1250      0.0000


Number of iterations    6
Minutes to convergence     0.00008

Lagraneans
              {}
```

Example 2: Use 'optmtControl' structure and optional inputs.

The following is a complete example. The code can be found in the GAUSSHOME/examples/optmt directory, named `optmt_spring.e`.

```
library optmt;

//Objective function
proc fct(x, k, F, ind);

    //Declare 'mm' to be a modelResults
    //struct, local to this procedure
    struct modelResults mm;

    //If the first element of the indicator vector
    //is non-zero, calculate the objective function
    if ind[1];
        //Assign the value of the objective function to the
        //'function' member of the 'modelResults' struct
        mm.function = k[1] * (sqrt(x[1]^2 + (x[2] + 1)^2) - 1)^2
                    + k[2] * (sqrt(x[1]^2 + (x[2] - 1)^2) - 1)^2
                    - (F'x);
     endif;
```

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

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

//Extra data needed by function
k = { 100, 90 };
F = { 20, 40 };

//Declare 'out' to be an optmtResults struct
//to hold the optimization results
struct optmtResults out;

//Declare 'ctl' to be an optmtControl structure
struct optmtControl ctl;

//Fill 'ctl' with default options
ctl = optmtControlCreate();

//Print report on every 5th iteration
ctl.printIters = 5;

//Use Newton Algorithm
ctl.algorithm = 3;

//Use Strong Wolfe Conditions
ctl.lineSearch = 4;
```

```
//Minimize objective function
//Pass in extra data needed by objective function
//between PV struct and control structure
out = optmt(&fct, x0, k, F, ctl);

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

The above code will print output similar to the following report:

```
=======================================
 Optmt
=======================================

Return code    =    0
Function value =   -9.65623
Convergence    :    normal convergence

Parameters  Estimates   Gradient
---------------------------------------
x[1,1]      0.5044      0.0000
x[2,1]      0.1219      0.0000


Number of iterations    6
Minutes to convergence     0.00181

Lagraneans
              {}
```

Source

optmt.src

optmtControlCreate

Purpose

Fills a structure of type *optmtControlCreate* with default settings.

Library

optmt

Format

ctl = **optmtControlCreate**();

Output

ctl	instance of a *optmtControl* structure filled in with default values.

Examples

```
//Declare 'ctl' to be an optmtControl struct
struct optmtControl ctl;

//Fill 'ctl' with default settings
ctl = optmtControlCreate();
```

Source

optmtutil.src

optmtPrt

Purpose

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

Library

optmt

Format

out = **optmtPrt**(*out*);

Input

out	instance of a *optmtResults* structure generated by a call to **optmt**.

Output

out	the input instance of the *optmtResults* structure unchanged.

Remarks

The call to **optmt** can be nested in the call to **optmtPrt**.

Source

optmtutil.src

Index