Optimization

Aptech Systems, Inc.
Maple Valley, WA

Documentation Version: January 15, 2001

Contents

Chapter 1

Installation

1.1 UNIX

If you are unfamiliar with UNIX, see your system administrator or system documentation for information on the system commands referred to below. The device names given are probably correct for your system.

1. Use `cd` to make the directory containing **GAUSS** the current working directory.
2. Use `tar` to extract the files.

   `tar xvf` *device_name*

   If this software came on diskettes, repeat the `tar` command for each diskette.

The following device names are suggestions. See your system administrator. If you are using Solaris 2.x, see Section 1.1.1.

Operating System	3.5-inch diskette	1/4-inch tape	DAT tape
Solaris 1.x SPARC	`/dev/rfd0`	`/dev/rst8`	
Solaris 2.x SPARC	`/dev/rfd0a` (vol. mgt. off)	`/dev/rst12`	`/dev/rmt/1l`
Solaris 2.x SPARC	`/vol/dev/aliases/floppy0`	`/dev/rst12`	`/dev/rmt/1l`
Solaris 2.x x86	`/dev/rfd0c` (vol. mgt. off)		`/dev/rmt/1l`
Solaris 2.x x86	`/vol/dev/aliases/floppy0`		`/dev/rmt/1l`
HP-UX	`/dev/rfloppy/c20Ad1s0`		`/dev/rmt/0m`
IBM AIX	`/dev/rfd0`	`/dev/rmt.0`	
SGI IRIX	`/dev/rdsk/fds0d2.3.5hi`		

1.1.1 Solaris 2.x Volume Management

If Solaris 2.x volume management is running, insert the floppy disk and type

```
volcheck
```

to signal the system to mount the floppy.

The floppy device names for Solaris 2.x change when the volume manager is turned off and on. To turn off volume management, become the superuser and type

```
/etc/init.d/volmgt off
```

To turn on volume management, become the superuser and type

```
/etc/init.d/volmgt on
```

1.2 DOS

1. Place the diskette in a floppy drive.
2. Log onto the root directory of the diskette drive. For example:

   ```
   A:<enter>
   cd\<enter>
   ```

3. Type: **ginstall** *source_drive target_path*

source_drive	Drive containing files to install with colon included For example: **A:**
target_path	Main drive and subdirectory to install to without a final \ For example: **C:\GAUSS**

   A directory structure will be created if it does not already exist and the files will be copied over.

*target_path***src**	source code files
*target_path***lib**	library files
*target_path***examples**	example files

4. The screen output option used may require that the DOS screen driver ANSI.SYS be installed on your system. If ANSI.SYS is not already installed on your system, you can put the command like this one in your CONFIG.SYS file:

   DEVICE=C:\DOS\ANSI.SYS

   (This particular statement assumes that the file ANSI.SYS is on the subdirectory DOS; modify as necessary to indicate the location of your copy of ANSI.SYS.)

1.3 Differences Between the UNIX and DOS Versions

- In the DOS version, when the global **__output** = 2, information may be written to the screen using commands requiring the ANSI.SYS screen driver. These are not available in the current UNIX version, and therefore setting **__output** = 2 may have the same effect as setting **__output** = 1.

- 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.

- On the Intel math coprocessors used by the DOS machines, intermediate calculations have 80-bit precision, while on the current UNIX machines, all calculations are in 64-bit precision. For this reason, **GAUSS** programs executed under UNIX may produce slightly different results, due to differences in roundoff, from those executed under DOS.

Chapter 2

Optimization

written by

Ronald Schoenberg

This module contains the procedure **OPTMUM**, which solves the problem:

minimize: $f(x)$

where $f : R^n \rightarrow R$. It is assumed that f has first and second derivatives.

2.1 Getting Started

GAUSS 3.1.0+ is required to use these routines.

2.1.1 README Files

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

2.1.2 Setup

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

```
library optmum,quantal,pgraph;
```

This enables **GAUSS** to find the procedure **OPTMUM** and other procedures used by **OPTMUM**. If you plan to make any right hand references to the global variables (which are described in a later section), you will also need the statement:

```
#include optmum.ext;
```

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

```
optset;
```

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

The version number of each module is stored in a global variable. For *OPTIMIZATION* this global is:

_op_ver 3×1 matrix, the first element contains the major version number of the *OPTMIMIZATION* 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 the *OPTIMIZATION* Module.

2.2 Algorithms

OPTMUM is a procedure for the minimization of a user-provided function with respect to parameters. It is assumed that the derivatives with respect to the parameters exist and are continuous. If the procedures to compute the derivatives analytically are not supplied by the user, **OPTMUM** calls procedures to compute them numerically. The user is required to supply a procedure for computing the function.

Six algorithms are available in **OPTMUM** for minimization. These algorithms, as well as the step length methods, may be modified during execution of **OPTMUM**.

OPTMUM minimizes functions iteratively and requires initial values for the unknown coefficients for the first iteration. At each iteration a *direction*, d, which is a $NP \times 1$ vector where NP is the number of coefficients, and a *step length*, α, are computed.

Direction

The direction, d, is a vector of quantities to be added to the present estimate of the coefficients. Intuitively, the term refers to the fact that these quantities measure where the coefficients are going in this iteration. It is computed as the solution to the equation

$$Hd = -g$$

where g is an NP×1 gradient vector, that is, a vector of the first derivatives of the function with respect to the coefficients, and where H is an NP×NP symmetric matrix H.

Commonly, as well as in previous versions of **OPTMUM**, the direction is computed as

$$d = H^{-1}g$$

Directly inverting H, however, is a numerically risky procedure, and the present version of **OPTMUM** avoids inverting matrices altogether. Instead a *solve* function is called which results in greater numerical stability and accuracy.

H is calculated in different ways depending on the type of algorithm selected by the user, or it can be set to a conformable identity matrix. For best results H should be proportional to the matrix of the second derivatives of the function with respect to pairs of the coefficients, i.e., the *Hessian* matrix, or an estimate of the Hessian.

The Newton Algorithm

In this method H is the Hessian:

$$H = \frac{\partial f}{\partial x \partial x^t}$$

By default H is computed numerically. If a function to compute the Hessian is provided by the user, then that function is called. If the user has provided a function to compute the gradient, then the Hessian is computed as the gradient of the gradient.

After H has been computed

$$Hd = -g$$

is solved for d.

The Secant Algorithms

The calculation of the Hessian is generally a very large computational problem. The secant methods (sometimes called *quasi-Newton* methods) were developed to avoid this. Starting with an initial estimate of the Hessian, or a conformable identity matrix, an update is calculated that requires far less computation. 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 the Newton iteration with much less computation.

Commonly, as well as in the previous versions of **OPTMUM**, an estimate of the *inverse* of the Hessian is updated on each iteration. This is a good strategy for reducing computation but is less favorable numerically. This version of **OPTMUM** instead updates a Cholesky factorization of the estimate of the Hessian (not its inverse). This method has superior numerical properties (Gill and Murray, 1972). The direction is then computed by applying a solve to the factorization and the gradient.

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 factorizations of the estimate of the Hessian is updated using the **GAUSS** functions **CHOLUP** and **CHOLDN**.

For given C, the Cholesky factorizatoin of the estimate of the Hessian,

$$C'Cd = -g$$

is solved for d using **GAUSS**'s **CHOLSOL** function.

2.2.1 The Steepest Descent Method (STEEP)

In the steepest descent method H is set to the identity matrix. This reduces computational and memory requirements, and if the problem is large the reduction is considerable. In regions far from the minimum it can be more efficient than other descent methods which have a tendency to get confused when the starting point is poor. It descends quite inefficiently, however, compared to the Newton and secant methods when closer to the minimum. For that reason, the default switching method begins with steepest descent and switches to the BFGS secant method.

2.2.2 Newton's Method (NEWTON)

Newton's method makes the most demands on the model of all the methods. The method succeeds only when everything is well behaved. The tradeoff is that the

method converge in fewer iterations than other methods. NEWTON uses both the first and second derivatives and thus the Hessian must be computed on each iteration. If the Hessian is being computed numerically, there is likely to be very little gain over DFP or BFGS because while the latter may take many more iterations, the total time to converge is less.

The numerically-computed gradient requires NP functions calls where NP is the number of parameters, and the numerically-computed Hessian requires NP^2 function calls. If the number of parameters is large, or the function calculation time-consuming, the Newton method becomes prohibitively computationally expensive. The computational expense can be significantly reduced, however, if you provide a function to compute the gradient. This reduces the calculation of the gradient to one function call. Moreover, for the numerical calculation of the Hessian **OPTMUM** uses the gradient function – in effect computing the Hessian as the gradient of the gradient – and thus reduces the number of function calls to NP for the Hessian.

2.2.3 Secant Methods (BFGS, DFP, scaled BFGS)

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. The use of an approximation produces gains in two ways: first, it is less sensitive to the condition of the model and data, and second, it performs better in all ways than the STEEPEST DESCENT method, and while it takes more iterations than NEWTON 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 Scaled BFGS is another version of the BFGS update method in which the formula for the computation of the update has been modified to make it scale-free.

The secant methods are commonly implemented as updates of the *inverse* of the Hessian. This is not the best method numerically (Gill and Murray, 1972). This version of OPTMUM, following Gill and Murray (1972), updates the Cholesky factorization of the Hessian instead, using the GAUSS functions **CHOLUP** and **CHOLDN**. The new direction is then computed using **CHOLSOL**, a Cholesky solve, as applied to the updated Cholesky factorization of the Hessian and the gradient.

2.2.4 Polak-Ribiere-type Conjugate Gradient (PRCG)

The conjugate gradient method is an improvement on the steepest descent method without the increase in memory and computational requirements of the secant methods. Only the gradient is stored, and the calculation of the new direction is different:

$$d_{\ell+1} = -g_{\ell+1} + \beta_\ell d_\ell$$

where ℓ indicates ℓ-th iteration, d is the direction, g is the gradient. The conjugate gradient method used in OPTMUM is a variation called the Polak-Ribiere method where

$$\beta_\ell = \frac{(g_{\ell+1} - g_\ell)' g_{\ell+1}}{g_\ell' g_\ell}$$

The Newton and secant methods require the storage on the order of Hessian in memory, i.e., $8NP^2$ bytes of memory, where NP is the number of parameters. For a very large problem this can be prohibitive. For example, 200 parameters requires 3.2 megabytes of memory, and this doesn't count the copies of the Hessian that may be generated by the program. For large problems, then, the PRCG and STEEP methods may be the only alternative. As described above, STEEP can be very inefficient in the region of the minimum, and therefore the PRCG is the method of choice in these cases.

2.3 Line Search

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

$$x_+ = x + \alpha d$$

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

$$F(x + \alpha d)$$

Given x and d, this is a function of a single variable α. Line search methods attempt to find a value for α that decreases F. 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.

2.3.1 STEPBT

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

2.3.2 BRENT

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

2.3.3 HALF

This method first computes $F(x+d)$, i.e., set $\alpha = 1$. If $F(x+d) < F(x)$ then the step length is set to 1. If not, then it tries $F(x+.5d)$. 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 very likely fails to find the step length that decreases F the most.

2.3.4 Random Search

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

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

2.4 Calling OPTMUM Recursively

The procedure provided by the user for computing the function to be minimized can itself call **OPTMUM**. In fact the number of nested levels is limited only by the amount

of workspace memory. Each level contains its own set of global variables. Thus nested copies can have their own set of attributes and optimization methods.

It is important to call **OPTSET** for all nested copies, and generally if you wish the outer copy of **OPTMUM** to retain control over the keyboard, you need to set the global variable **_opkey** to zero for all the nested copies.

2.5 Using _OPTMUM Directly

When **OPTMUM** is called, it directly references all the necessary globals and passes its arguments and the values of the globals to a function called **_OPTMUM**. When **_OPTMUM** returns, **OPTMUM** then sets the output globals to the values returned by **_OPTMUM** and returns its arguments directly to the user. **_OPTMUM** makes no global references to matrices or strings, and all procedures it references have names that begin with an underscore "**_**".

_OPTMUM can be used directly in situations where you do not want any of the global matrices and strings in your program. If **OPTMUM**, **OPTPRT** and **OPTSET** are not referenced, the global matrices and strings in `optmum.dec` is not included in your program.

The documentation for **OPTMUM**, the globals it references, and the code itself should be sufficient documentation for using **_OPTMUM**.

2.6 Hints on Optimization

The critical elements in optimization are scaling, starting point, and the condition of the model.

When the data are scaled, the starting point reasonably close to the solution, and the data and model well-conditioned, the iterations converges 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.

2.6.1 Scaling

For best performance the Hessian matrix should be "balanced", i.e., the sums of the columns (or rows) should be roughly equal. In most cases the diagonal elements determined these sums. If some diagonal elements are very large and/or very small with respect to others, **OPTMUM** 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. 90% of the technical support calls complaining about **OPTMUM** failing to converge are solved by simply scaling the problem.

2.6.2 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 gradient of the gradient (i.e., the Hessian) is very small for a particular parameter, then **OPTMUM** has difficulty deciding on the optimal value since a large region of the function appears virtually flat to **OPTMUM**. 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 model and bad data are two sides of the same coin. If the problem is highly nonlinear it is important that data be available to describe the features of the curve described by each of the parameters. For example, one of the parameters of the Weibull function describes the shape of the curve as it approaches the upper asymptote. If data are not available on that portion of the curve then that parameter is poorly estimated. The gradient of the function with respect to that parameter is very flat, elements of the Hessian associated with that parameter is very small, and the inverse of the Hessian contains very large numbers. In this case it is necessary to respecify the model in a way that excludes that parameter.

2.6.3 Starting Point

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

2.6.4 Managing the Algorithms and Step Length Methods

A lack of a good starting point may be overcome to some extent by managing the algorithms, step length methods, and computation of the Hessian. This is done with the use of the global variables **_opstmth** (start method) and **_opmdmth** (middle method). The first of these globals determines the starting algorithm and step length method. When the number of iterations exceeds **_opditer** or when the function *fails* to change by **_opdfct** percent or if **Alt-T** is pressed during the iterations, the algorithm and/or step length method are switched by **OPTMUM** according to the specification in **_opmdmth**.

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.

The least demanding algorithm is steepest descent (STEEP). The secant methods, BFGS, BFGS-SC, and DFP are more demanding than STEEP, and NEWTON is the most demanding. The least demanding step length method is step length set to 1. More demanding is STEPBT, and the most demanding is BRENT. For bad starting points and ill-conditioned models, the following setup might be useful:

```
_opstmth = "steep one";
_opmdmth = "bfgs brent";
```

or

```
_opstmth = "bfgs brent";
_opmdmth = "newton stepbt";
```

Either of these would start out the iterations without strong demands on the condition and starting point, and then switch to more efficient methods that make greater demands after the function has been moved closer to the minimum.

The complete set of available strings for **_opstmth** and **_opmdmth** are described in the **OPTMUM** reference section on global variables.

2.6.5 Managing the Computation of the Hessian

Convergence using the secant methods (BFGS, BFGS-SC, and DFP) can be considerably accelerated by starting the iterations with a computed Hessian. However, if the starting point is bad, the iterations can be sent into nether regions from which **OPTMUM** may never emerge. To prevent this character strings can be added to **_opstmth** and **_opmdmth** to control the computation of the Hessian. For example, the following

```
_opstmth = "bfgs stepbt nohess";
_opmdmth = "hess";
```

forces **OPTMUM** to start the iterations with the identity matrix in place of the Hessian, and then compute the Hessian after **_opditer** iterations or the function fails to change by **_opdfct** percent. The setting for **_opstmth** is the default setting and thus if the default settings haven't been changed, only the string **_opmdmth** is necessary. The alternative

```
_opmdmth = "inthess";
```

causes **OPTMUM** to compute the Hessian *every* **_opditer** iterations.

2.6.6 Diagnosis

When the optimization is not proceeding well, it is sometimes useful to examine the function, gradient, Hessian and/or coefficients during the iterations. Previous versions of **OPTMUM** saved the current coefficients in a global, but this interfered with the recursive operation of **OPTMUM**, so the global was removed. For meaningful diagnosis you would want more than the coefficients anyway. Thus, we now recommend the method described in `optutil.src` (search for "DIAGNOSTIC").

This commented section contains code that will save the function, gradient, Hessian and/or coefficients as globals, and other code that will print them to the screen. Saving these as globals is useful when your run is crashing during the iterations because the globals will contain the most recent values before the crash. On the other hand, it is sometimes more useful to observe one or more of them during the iterations, in which case the **PRINT** statements will be more helpful. To use this code, simply uncomment the desired lines.

2.7 Function

You must supply a procedure for computing the objective function to be minimized. **OPTMUM** always minimizes. If you wish to maximize a function, minimize the negative of the function to be maximized.

This procedure has one input argument, an NP $\times$ 1 vector of coefficients. It returns a scalar, the function evaluated at the NP coefficients.

Occasionally your function may fail due to illegal calculations - such as an overflow. Or your function may not fail when it fact it should - such as taking the logarithm of a negative number, which is a legal calculation in **GAUSS** but which is not usually desired. In either of these cases, you may want to recover the value of the coefficients at that point but you may not want **OPTMUM** to continue the iterations. You can control this be having your procedure return a missing value when there is any condition which you wish to define as failure. This allows **OPTMUM** to return the state of the iterations at the point of failure. If your function attempts an illegal calculation and you have *not* tested for it, **OPTMUM** ends with an error message and the state of the iterations at that point is lost.

2.8 Gradient

2.8.1 User-Supplied Analytical Gradient

To increase accuracy and reduce time, the user may supply a procedure for computing the gradient. In practice, unfortunately, most time spent on writing the gradient

procedure is spent in debugging. The user is urged to first check the procedure against numerical derivatives. Put the function and gradient procedures into their own command file along with a call to **GRADFD**:

The user-supplied procedure has one input argument, an NP $\times$ 1 vector of coefficients. It returns a 1 $\times$ NP row vector of gradients of the function with respect to the NP coefficients.

```
library optmum;
optset;

c0 = { 2, 4 };
x = rndu(100,1);
y = model(c0);
print "analytical gradient ";
print grd(c0);
print;
print "numerical gradient ";
print gradfd(&fct,c0);

proc model(c);
    retp(c[1]*exp(-c[2]*x));
endp;

proc fct(c);
    local dev;
    dev = y - model(c);
    retp(dev'*dev);
endp;

proc grd(c);
    local g;
    g = exp(-c[2]*x);
    g = g~(-c[1]*x.*g);
    retp(-2*(y - model(c))'*g);
endp;
```

2.8.2 User-Supplied Numerical Gradient

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

Included in the **OPTMUM** library of procedures are functions for computing numerical derivatives: **GRADCD**, numerical derivatives using central differences, **GRADFD**,

numerical derivatives using forward differences, and **GRADRE**, which applies the Richardson Extrapolation method to the forward difference method.

GRADRE can come very close to analytical derivatives. It is considerably more time-consuming, however, than using analytical derivatives. The results of **GRADRE** are controlled by three global variables, **_grnum**, **_grstp**, and **_grsca**. The default settings of these variables, for a reasonably well-defined problem, produces convergence with moderate speed. If the problem is difficult and doesn't converge then try setting **_grnum** to 20, **_grsca** to 0.4, and **_grstp** to 0.5. This slows down the computation of the derivatives by a factor of 3 but increases the accuracy to near that of analytical derivatives.

To use any of these procedures put

```
#include gradient.ext;
```

at the top of your command file, and

```
_opusrgd = &gradre;
```

somewhere in the command file after the called to **OPTSET** and before the call to **OPTMUM**.

You may use one of your own procedures for computing numerical derivatives. This procedure has two arguments, the pointer to the function being optimized and an NP $\times$ 1 vector of coefficients. It returns a 1 $\times$ NP row vector of the derivatives of the function with respect to the NP coefficients. Then simply add

```
_opusrgd = &yourgrd;
```

where `yourgrd` is the name of your procedure.

2.9 Hessian

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

2.9.1 User-Supplied Analytical Hessian

The Hessian is computed on each iteration in the Newton-Raphson algorithm, at the start of the BFGS, Scaled BFGS, and DFP algorithms if **_opshess** = 0, when **Alt-I** is

pressed during iterations, **_opstmth** = "hess", after **_opditer** iterations or when the function has failed to change by **_opdfct** percent or when **Alt-T** is pressed if **_opmdmth** = "hess", All of these computations may be speeded up by the user by providing a procedure for computing analytical second derivatives. This procedure has one argument, the NP×1 vector of parameters, and returns a NP×NP symmetric matrix of second derivatives of the objection function with respect to the parameters. The pointer to this procedure is stored in the global variable **_ophsprc**.

2.9.2 User-Supplied Numerical Hessian

You may substitute your own numerical Hessian procedure for the one used by OPTMUM by default. This is done by setting the OPTMUM global, **_opusrhs** to a pointer to the procedure. This procedure has two input arguments, a pointer to the function being minimized and an NP × 1 vector of coefficients. It returns an NP × NP matrix containing the second derivatives of the function evaluated at the input coefficient vector.

2.10 Run-Time Switches

If the user presses **Alt-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.

Alt-G	Toggle **_opgdmd**
Alt-V	Revise **_opgtol**
Alt-F	Revise **_opdfct**
Alt-P	Revise **_opditer**
Alt-O	Toggle **__output**
Alt-M	Maximum Backstep
Alt-I	Compute Hessian
Alt-E	Edit Parameter Vector
Alt-C	Force Exit
Alt-A	Change Algorithm
Alt-J	Change Step Length Method
Alt-T	Force change to mid-method
Alt-H	Help Table

The algorithm may be switched during the iterations either by pressing **Alt-A**, or by pressing one of the following:

Alt-1	Steepest Descent (STEEP)
Alt-2	Broyden-Fletcher-Goldfarb-Shanno (BFGS)
Alt-3	Scaled BFGS (BFGS-SC)
Alt-4	Davidon-Fletcher-Powell (DFP)
Alt-5	Newton-Raphson (NEWTON) or (NR)
Alt-6	Polak-Ribiere Conjugate Gradient (PRCG)

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

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

2.11 Error Handling

2.11.1 Return Codes

The fourth argument in the return from **OPTMUM** contains a scalar number that contains information about the status of the iterations upon exiting **OPTMUM**. 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	step length calculation failed
7	function cannot be evaluated at initial parameter values
8	number of elements in the gradient function is inconsistent with the number of starting values
9	the gradient function returned a column vector rather than the required row vector
20	Hessian failed to invert

2.11.2 Error Trapping

Setting the global **__output** = 0 turns off all printing to the screen. Error codes, however, still is printed to the screen unless error trapping is also turned on. Setting the trap flag to 4 causes **OPTMUM** to *not* send the messages to the screen:

```
trap 4;
```

Whatever the setting of the trap flag, **OPTMUM** discontinues computations and return 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 **OPTMUM** program is embedded in a larger program, and you want the larger program to handle the errors.

2.12 Example

The example OPT1.E is taken from D.G. Luenberger (1984) **Linear and Nonlinear Programming**, Addison-Wesley, page 219. The function to be optimized is a quadratic function:

$$f(x) = 0.5(b - x)'Q(b - x)$$

where a is a vector of known coefficients and Q is a symmetric matrix of known coefficients. For computational purposes this equation is restated:

$$f(x) = 0.5x'Qx - x'b$$

From Luenberger we set

$$Q = \begin{matrix} .78 & -.02 & -.12 & -.14 \\ -.02 & .86 & -.04 & .06 \\ -.12 & -.04 & .72 & -.08 \\ -.14 & .06 & -.08 & .74 \end{matrix}$$

and

$$b' = \begin{matrix} .76 & .08 & 1.12 & .68 \end{matrix}$$

First, we do the setup:

```
library optmum;
optset;
```

The procedure to compute the function is

```
proc qfct(x);
    retp(.5*x'*Q*x-x'b);
endp;
```

Next a vector of starting values is defined,

```
x0 = { 1, 1, 1, 1 };
```

and finally a call to **OPTMUM**:

```
{ x,f,g,retcode } = optmum(&qfct,x0);
```

The estimated coefficients are returned in x, the value of the function at the minimum is returned in f and the gradient is returned in g (you may check this to ensure that minimum has actually been reached).

2.13 References

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

Broyden, G., 1965. "A class of methods for solving nonlinear simultaneous equations." *Mathematics of Computation* 19:577-593.

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

Fletcher, R. and Powell, M. 1963. "A rapidly convergent descent method for minimization." *The Computer Journal* 6:163-168.

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

Johnson, Lee W., and Riess, R. Dean, 1982. *Numerical Analysis, 2nd Ed.* Reading, MA: Addison-Wesley.

Luenberger, D.G., 1984. *Linear and Nonlinear Programming.* Reading, MA: Addison-Wesley.

Chapter 3

Optimization Reference

Library

`optmum`

Purpose

Minimizes a user-provided function with respect to a set of parameters.

Format

{ *x*,*f*,*g*,*retcode* } = **OPTMUM(&***fct*,*x0*);

Input

x0	NP×1 vector, start values or the name of a proc that takes no input arguments and returns an NP×1 vector of start values.
&*fct*	pointer to a procedure that computes the function to be minimized. This procedure must have one input argument, an NP×1 vector of parameter values, and one output argument, a scalar value of the function evaluated at the input vector of parameter values.

Output

x	NP×1 vector, parameter estimates.
f	scalar, value of function at minimum.
g	NP×1 vector, gradient evaluated at x.
retcode	scalar, return code. If normal convergence is achieved then *retcode* = 0, otherwise a positive integer is returned indicating the reason for the abnormal termination:

1 forced exit.

2 maximum iterations exceeded.

3 function calculation failed.

4 gradient calculation failed.

5 Hessian calculation failed.

6 step length calculation failed.

7 function cannot be evaluated at initial parameter values.

8 number of elements in the gradient function is inconsistent with the number of starting values.

9 the gradient function returned a column vector rather than the required row vector.

20 Hessian failed to invert.

Globals

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

Optimization and Steplengths **_opshess**, **_opalgr**, **_opdelta**, **_opstmth**, **_opdfct**, **_opditer**, **_opmxtry**, **_opmdmth**, **_opmbkst**, **_opstep**, **_opextrp**, **_oprteps**, **_opintrp**, **_opusrch**.

Gradient **_opgdprc**, **_opgrdmd**, **_ophsprc**, **_opusrgd**, **_opusrhs**.

Terminations Conditions **_opmiter**, **_opgtol**.

Output Globals **_opitdta**, **_opfhess**, **_opkey**.

Printout **__output**, **_opparnm**, **__title**.

Please refer to this list if you need to know the name of the globals in affecting particular aspects of your problem. Then see the list below for a complete description of each global. Below, these globals are listed alphabetically.

_opalgr scalar, selects optimization method:

1 STEEP - steepest descent

2 BFGS - Broyden, Fletcher, Goldfarb, Shanno method

3 BFGS-SC

4 DFP - scale free Davidon, Fletcher, Powell method

5 NEWTON - Newton-Raphson method

6 PRCG - Polak-Ribiere Conjugate Gradient method

Default = 2.

_opdelta scalar. At each iteration during the NEWTON method the eigenvalues of the Hessian are inspected, and if any of them are less than or equal to **_opdelta**, the Hessian is sufficiently augmented to generate a Hessian with eigenvalues greater than or equal to **_opdelta**. Default = 0.1.

_opdfct scalar. If the function *fails* to improved by the percentage **_opdfct**, **OPTMUM** switches to the algorithm and/or steplength specified in **_opmdmth**. Default = 0.01.

_opditer scalar integer. After **_opditer** iterations, **OPTMUM** switches to the algorithm and/or steplength specified in **_opmdmth**. Default = 20.

_opextrp scalar, extrapolation constant for the BRENT line search method. Default = 2.

_opfhess NP×NP matrix. Contains last Hessian calculated by **OPTMUM**. If a Hessian is never calculated, then this global is set to a scalar missing value.

_opgdprc pointer to a procedure that computes the gradient of the function with respect to the parameters.

For example, the instruction:

```
_opgdprc = &gradproc;
```

stores the pointer to a procedure with the name **GRADPROC** in **_opgdprc**. The user-provided procedure has a single input argument, an NP × 1 vector of parameter values, and a single output argument, a 1 × NP row vector of gradients of the function with respect to the parameters evaluated at the vector of parameter values. For example, suppose the procedure is named **GRADPROC** and the function is a quadratic function with one parameter:

$$y = x^2 + 2x + 1$$

then

```
proc gradproc(x);
    retp(2*x+2);
endp;
```

By default, **OPTMUM** uses a numerical gradient.

_opgdmd scalar, selects numerical gradient method.

0 Central difference method.

1 Forward difference method.

2 Forward difference method with Richardson Extrapolation.

Default = 0.

_opgtol scalar, or NP×1 vector, tolerance for gradient of estimated coefficients. When this criterion has been satisfied **OPTMUM** exits the iterations. Default = 10^{-4}.

_ophsprc scalar, pointer to a procedure that computes the Hessian, i.e. the matrix of second order partial derivatives of the function with respect to the parameters.

For example, the instruction:

```
_ophsprc = &hessproc;
```

stores the pointer of a procedure with the name **HESSPROC** in the global variable **_ophsprc**. The procedure that is provided by the user must have single input argument, the NP × 1 vector of parameter values, and a single output argument, the NP × NP symmetric matrix of second order derivatives of the function evaluated at the parameter values.

By default, **OPTMUM** calculates the Hessian numerically.

_opitdta 3×1 matrix, the first element contains the elapsed time in minutes of the iterations and the second element contains the total number of iterations. The third element contains a character variable indicating the type of inverse of Hessian that was computed:

NOCOV	not computed
NOTPD	not positive definite
HESS	computed from Hessian
SECANT	estimated from secant update

_opintrp scalar, interpolation constant for the BRENT line search method. Default = 0.25.

_opkey scalar, controls keyboard trapping, that is, the run-time switches and help table. If zero, keyboard trapping is turned off. This option is useful when nested copies of **OPTMUM** are being called. Turning off the keyboard trapping of the nested copies reserves control for the outer copy of **OPTMUM**. Default = 1.

_opmxtry scalar, maximum number of tries to compute a satisfactory step length. Default = 100.

_opmdmth string, used to specify an algorithm and/or step length and/or computation of Hessian to which **OPTMUM** switches to when either **_opditer** or **_opdfct** are satisfied, or **Alt-T** is pressed. The following character strings may be included in **_opmdmth**:

Algorithm choice:

"STEEP"

"BFGS"

"BFGS_SC"

"DFP"

"NEWTON" (or "NR")
"PRCG"

Step length choice:

"ONE" (or "1" or "'1.0")
"STEPBT"
"BRENT"
"HALF"

Hessian return choice:

"HESS"
"NOHESS".

For example:

```
_opmdmth = "steep brent nohess";
```

The character strings for the algorithms and step lengths causes **OPTMUM** to switch to the specified algorithm and or step length method. The string "HESS" causes the Hessian to be computed, which is the default, and "NOHESS" prevents the Hessian from being computed.

Default = "HESS";

_opmiter scalar, maximum number of iterations. Default = 10000.

_opmbkst scalar, maximum number of backsteps taken to find step length. Default = 10.

_opparnm NP×1 character vector of parameter labels. By default, no labels is used for the output.

_oprteps scalar. If set to nonzero positive real number and if the selected line search method fails, then a random search is tried with radius equal to the value of this global times the truncated log to the base 10 of the gradient. Default = .01.

_opshess scalar, or NP×NP matrix, the starting Hessian for BFGS, Scaled BFGS, DFP, and NEWTON. Possible scalar values are:

0 begin with identity matrix.

1 compute starting Hessian

Default = 0. If set to a matrix, this matrix is used for the starting Hessian. Default = 0.

_opstep scalar, selects method for computing step length.

1 step length = 1.

2 cubic or quadratic step length method (STEPBT)

3 Brent's step length method (BRENT)

4 step halving (HALF)

Default = 2.

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

When any of these line search methods fails, a random search is tried with radius **_oprteps** times the truncated log to the base 10 of the gradient. If **_opusrch** is set to 1 **OPTMUM** enters an interactive line search mode.

_opstmth string, used to specify an algorithm and/or step length and/or computation of Hessian with which **OPTMUM** begins the iterations. The following character strings may be included in **_opstmth**:

Algorithm choice:

"STEEP"

"BFGS"

"BFGS_SC"

"DFP"

"NEWTON" (or "NR")

"PRCG"

Step length choice:

"ONE" (or "1" or "1.0")

"STEPBT"

"BRENT"

"HALF"

Hessian choice:

"HESS"

"NOHESS".

The character strings for the algorithms and step lengths causes **OPTMUM** to begin with the specified algorithm and or step length method. The string "HESS" causes the Hessian to be computed and inverted at the start of the iterations,

The default setting is **_opstmth** = "". The default settings for **_opalgr**, **_opstep**, and **_opshess** are equivalent to **__opstmth** = "BFGS STEPBT HESS".

_opusrch scalar. If 1, and if the selected line search method fails, then **OPTMUM** enters an interactive line search mode. Default = 0.

_opusrgd scalar, pointer to user-supplied procedure for computing a numerical gradient. This procedure has two input arguments, a pointer to the procedure computing the function to be minimized and an NP × 1 vector of coefficients. It has one output argument, a 1 × NP row vector of derivatives of the function with respect to the NP coefficients.

_opusrhs scalar, pointer to the user-supplied procedure for computing a numerical Hessian. This procedure has two input arguments, a pointer to the procedure computing the function to be minimized and an NP × 1 vector of coefficients. It has one output argument, an NP × NP matrix of the second derivatives of the function with respect to the NP coefficients.

__output scalar, determines printing of intermediate results.

0	nothing is written.
1	serial ASCII output format suitable for disk files or printers.
2	(NOTE: DOS version only) output is suitable for screen only. ANSI.SYS must be active.

Default = 2.

Remarks

OPTMUM can be called recursively.

Source

`optmum.src`

Library

`optmum`

Purpose

Resets **OPTMUM** global variables to default values.

Format

OPTSET;

Input

None

Output

None

Remarks

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

OPTSET calls **GAUSSET**.

Source

`optmum.src`

Library

optmum

Purpose

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

Format

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

Input

x	NP×1 vector, parameter estimates.
f	scalar, value of function at minimum.
g	NP×1 vector, gradient evaluated at *x*.
retcode	scalar, return code.

Output

Same as Input.

Globals

__title	string, title of run. By default, no title is printed.

Remarks

The call to **OPTMUM** can be nested in the call to **OPTPRT**:

```
{ x,f,g,retcode } = optprt(optmum(&fct,x0));
```

This output is suitable for a printer or disk file.

Source

optmum.src

Library

optmum

Purpose

Computes the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure. Single-sided (forward difference) gradients are computed.

Format

d = **GRADFD(&**f**,***x0***);**

Input

&*f* procedure pointer to a vector-valued function:

$$f : R^K \rightarrow R^N$$

It is acceptable for $f(x)$ to have been defined in terms of global arguments in addition to x, and thus f can return an N×1 vector:

```
proc f(x);
    retp( exp(x*b) );
endp;
```

x0 NP×1 vector, point at which to compute gradient.

Output

g N×NP matrix, gradients of f with respect to the variable x at *x0*.

Globals

_grdh scalar, determines increment size for numerical gradient. By default, the increment size is automatically computed.

Remarks

GRADFD returns a ROW for every row that is returned by f. For instance, if f returns a 1×1 result, then **GRADFD** returns a 1×NP row vector. This allows the same function to be used where N is the number of rows in the result returned by f. Thus, for instance, **GRADFD** can be used to compute the Jacobian matrix of a set of equations.

To use **GRADFD** with **OPTMUM**, put

```
#include gradient.ext;
```

at the top of the command file, and

```
_opusrgd = &gradfd;
```

somewhere in the command file after the call to **OPTSET** and before the call to **OPTMUM**. For example,

```
library optmum;
#include optmum.ext;
#include gradient.ext;
optset;

start = { -1, 1 };

proc fct(x);
    local y1,y2;
    y1 = x[2] - x[1]*x[1];
    y2 = 1 - x[1];
    retp (1e2*y1*y1 + y2*y2);
endp;

_opusrgd = &gradfd;
{ x,fmin,g,retcode } = optprt(optmum(&fct,start));
```

OPTMUM uses the central difference method when **_opgdmd** = 1, thus using **GRADFD** as it is written is redundant. Its use would be advantageous, however, if you were to modify **GRADFD** for a special purpose.

Source

gradient.src

Library

optmum

Purpose

Computes the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure. Central difference gradients are computed.

Format

d = **GRADCD(&***f***,***x0***);**

Input

&*f* procedure pointer to a vector-valued function:

$$f : R^{NP} \rightarrow R^N$$

It is acceptable for $f(x)$ to have been defined in terms of global arguments in addition to x, and thus f can return an N×1 vector:

```
proc f(x);
    retp( exp(x*b) );
endp;
```

x0 NP×1 vector, points at which to compute gradient.

Output

g N×NP matrix, gradients of f with respect to the variable x at *x0*.

Globals

_grdh scalar, determines increment size for numerical gradient. By default, the increment size is automatically computed.

Remarks

GRADCD returns a ROW for every row that is returned by f. For instance, if f returns a 1×1 result, then **GRADCD** returns a 1×NP row vector. This allows the same function to be used where N is the number of rows in the result returned by f. Thus, for instance, **GRADCD** can be used to compute the Jacobian matrix of a set of equations.

To use **GRADCD** with **OPTMUM**, put

```
#include gradient.ext;
```

at the top of the command file, and

```
_opusrgd = &gradcd;
```

somewhere in the command file after the call to **OPTSET** and before the call to **OPTMUM**. For example,

```
library optmum;
#include optmum.ext;
#include gradient.ext;
optset;

start = { -1, 1 };

proc fct(x);
    local y1,y2;
    y1 = x[2] - x[1]*x[1];
    y2 = 1 - x[1];
    retp (1e2*y1*y1 + y2*y2);
endp;

_opusrgd = &gradcd;
{ x,fmin,g,retcode } = optprt(optmum(&fct,start));
```

OPTMUM uses the central difference method when **_opgdmd** = 1, thus using **GRADCD** as it is written is redundant. Its use would be advantageous, however, if you were to modify **GRADCD** for a special purpose.

Source

gradient.src

Library

optmum

Purpose

Computes the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure. Single-sided (forward difference) gradients are computed, using Richardson Extrapolation.

Format

d = **GRADRE(&**f**,**$x0$**);**

Input

&f procedure pointer to a vector-valued function:

$$f : R^{NP} \rightarrow R^N$$

It is acceptable for $f(x)$ to have been defined in terms of global arguments in addition to x, and thus f can return an N×1 vector:

```
proc f(x);
    retp( exp(x*b) );
endp;
```

$x0$ NP×1 vector, points at which to compute gradient.

Output

g N×NP matrix, gradients of f with respect to the variable x at $x0$.

Globals

__grnum integer, determines the number of iterations algorithm produces. Beyond a certain point, increasing **__grnum** does not improve accuracy of result; on the contrary, round error swamps accuracy and results become significantly worse.

__grsca scalar, between 0 and 1. By reducing **__grsca**, algorithm may arrive at an acceptable result sooner, but this may not be as accurate as a result achieved with larger **__grsca** and which might take longer to compute. Generally, an **__grsca** much smaller than 0.05 does not improve results significantly.

__grstp	scalar, should be less than 1. The best results seemed to be obtained most efficiently when **__grstp** is between 0.4 and 0.8. Changing **__grstp** and **__grsca** may have positive effects on results of algorithm.

Remarks

The settings for the global variables, for a reasonably well-defined problem, produces convergence with moderate speed. If the problem is difficult and doesn't converge then try setting **__grnum** to 20, **__grsca** to 0.4, and **__grstp** to 0.5. This slows down the computation of the derivatives by a factor of 3 but increases the accuracy to near that of analytical derivatives.

GRADRE returns a ROW for every row that is returned by f. For instance, if f returns a 1×1 result, then **GRADRE** returns a 1×NP row vector. This allows the same function to be used where N is the number of rows in the result returned by f. Thus, for instance, **GRADRE** can be used to compute the Jacobian matrix of a set of equations.

The algorithm, Richardson Extrapolation (see **Numerical Analysis**, by Lee W. Johnson and R. Dean Riess, page 319) is an iterative process which updates a derivative based on values calculated in a previous iteration. This is slower than **GRADP**, but can, in general, return values that are accurate to about 8 digits of precision. The algorithm runs through n iterations. **_grnum** is a global whose default is 25.

```
#include gradient.ext;

proc myfunc(x);
    retp( x.*2 .* exp( x.*x./3));
endp;
x0 = { 2.5, 3.0, 3.5 };
y = gradre(&myfunc,x0);
print y;

82.98901642      0.00000000       0.00000000
 0.00000000    281.19752454       0.00000000
 0.00000000      0.00000000    1087.95412226
```

To use **GRADRE** with **OPTMUM**, put

```
#include gradient.ext;
```

at the top of the command file, and

```
_opusrgd = &gradre;
```

somewhere in the command file after the call to **OPTSET** and before the call to **OPTMUM**. For example,

```
library optmum;
#include optmum.ext;
#include gradient.ext;
optset;

start = { -1, 1 };

proc fct(x);
    local y1,y2;
    y1 = x[2] - x[1]*x[1];
    y2 = 1 - x[1];
    retp (1e2*y1*y1 + y2*y2);
endp;

_opusrgd = &gradre;
{ x,fmin,g,retcode } = optprt(optmum(&fct,start));
```

Source

gradient.src

Index

P

Q

R

S

T

U