# A Modular System of Algorithms for Unconstrained Minimization

Robert B. Schnabel
John E. Koontz
Barry E. Weiss

CU-CS-240-82

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

# A MODULAR SYSTEM OF ALGORITHMS FOR UNCONSTRAINED MINIMIZATION

by

Robert B. Schnabel*,
John E. Koontz**,
and Barry E. Weiss***

CU-CS-240-82 (revised)    June, 1985

* Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, 80309 and Scientific Computing Division, National Bureau of Standards, Boulder, Colorado 80303. This research supported in part by ARO contracts DAAG 29-79-C-0023 and DAAG 29-81-K-0108

** Statistical Engineering Division, National Bureau of Standards, Boulder, Colorado, 80303.

*** Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, 80309. Current address: AT&T Information Systems, 11900 N. Pecos St., Denver, Colorado 80234.

## Abstract

We describe a new package, UNCMIN,for finding a local minimizer of a real valued function of more than one variable. The novel feature of UNCMIN is that it is a modular system of algorithms, containing three different step selection strategies (line search, dogleg, and optimal step) that may be combined with either analytic or finite difference gradient evaluation, and either analytic, finite difference, or BFGS Hessian approximation. We present the results of a comparison of the three step selection strategies on the problems in More', Garbow, and Hillstrom in two separate cases: using finite difference gradients and Hessians, and using finite difference gradients with BFGS Hessian approximations. We also describe a second package, REVMIN, that uses optimization algorithms identical to UNCMIN but obtains values of user supplied functions by reverse communication.

## 1. Introduction

This paper describes UNCMIN, a modular system of FORTRAN subroutines for the solution of the unconstrained minimization problem

$$\min_{x \in R^n} f(x) : R^n \to R. \tag{1.1}$$

They are intended for the case when $f$ is at least twice continuously differentiable, although the derivatives do not have to be available analytically. The algorithms may sometimes solve problems where $f$ is only once continuously differentiable. No restriction is made on the number of variables $n$, but since the algorithms store one $n \times n$ matrix and solve a system of $n$ linear equations in $n$ unknowns at each iteration, they are intended mainly for problems of small to moderate size, with $n$ for example between 2 and 100. UNCMIN will successfully solve problems with $n = 1$, although perhaps not as efficiently as algorithms intended specifically for this case.

There are a number of other FORTRAN routines available for solving the unconstrained minimization problem, including algorithms in the Harwell, IMSL, MINPACK, and NAG libraries (see reference section), and the packages by Shanno and Phua [1980] and Gay [1983]. This paper emphasizes the ways in which ours is distinctive. Books that describe unconstrained minimization algorithms include Fletcher [1980], Gill, Murray, and Wright [1982], and Dennis and Schnabel [1983].

The distinguishing feature of UNCMIN is that it is a modular *system* of subroutines. This means that the user can build a variety of minimization algorithms with UNCMIN, by selecting from among several options for the step selection process, and for the evaluation or approximation of $\nabla f$ and $\nabla^2 f$. The possibility of selecting among alternative strategies for derivative evaluation, namely user-supplied analytical computation, finite differences, or BFGS updates, is found in several other packages. The provision of alternative step selection strategies that can be used interchangeably with the remainder of the routine, namely a line search, double dogleg, and a locally constrained optimal step (hereafter referred to as "hookstep"), is rare; in fact we know of no code that provides both line search and trust region alternatives. The combination of all these options is believed to be unique. In our experience, these options have proved useful to users who wish to find conveniently the best method to use on a particular class of problems. The modular structure also makes the system very useful as a testing and research tool, because alternative strategies can be compared in a controlled environment, and new approaches for various parts of the minimization algorithm can be tested readily by substituting one or more new modules for the corresponding existing modules. Section 2 discusses the modular structure of our system in more detail.

The FORTRAN subroutines in UNCMIN correspond closely, though not always exactly, to the pseudo-code in Appendix A of Dennis and Schnabel [1983], and were designed initially as a companion to this book. The methods used for solving the unconstrained minimization problem were not intended to be new, but rather a selection of the best existing algorithms. Inevitably, various original features were introduced, some of which are discussed in Section 3. We also paid careful attention to several mundane aspects of the algorithms that usually are important to users only when they malfunction, namely stopping criteria, selection of finite difference stepsizes, and treatment of badly scaled problems. These topics also are included in Section 3.

We tried to pay careful attention to the user interface with our package. An important feature is the provision of a choice between an easy-to-use calling sequence, in which the user provides only $n$, $f$, and the initial estimate $x_0$, and all tolerances and algorithms options are automatically selected, and a more complicated call in which the user nonetheless needs to set only those options desired. Also provided are checking of user-supplied derivatives, and a variety of output options. These features are discussed briefly in Section 4, and more fully in Weiss [1980] and in a forthcoming paper. In Section 5 we discuss briefly the portability and storage requirements of the code, as well as suggestions for adapting it to small computers.

We have also developed a reverse communication version of our code, REVMIN. This version is algorithmically identical to UNCMIN; the difference is that whenever a function or analytic derivative evaluation is required, the reverse communication version returns to a dummy driver REVDRV inserted between the calling program and REVMIN, evaluates the function or derivative, and then re-calls REVMIN, which resumes the optimization algorithm at the proper place. This capability is required whenever the evaluation of $f(x)$ requires additional data from the calling program, and it is inconvenient to pass this information through FORTRAN COMMON. For example, this is the situation in the three time series codes in the National Bureau of Standards statistical library STARPAC that use REVMIN. The need for reverse communication and the conversion of UNCMIN to REVMIN is discussed in Section 6.

Finally, in section 7 we present the results of a comprehensive set of tests using the algorithms in UNCMIN on the test problems in Moré, Garbow, and Hillstrom [1981]. In particular, we compare the three step selection strategies, line search, dogleg, and hookstep, in the case when $\nabla f$ and $\nabla^2 f$ both are approximated by finite differences and again when $\nabla^2 f$ is approximated instead using BFGS updates. To our knowledge, these are the first such comprehensive and controlled comparative tests of these global strategies for unconstrained minimization to be reported. Gay [1983] compares two different BFGS codes, one using a line search and the other a dogleg.

## 2. Modular Structure

The advantages of modular design to the organization, development and testing of any large computer software system are well known. The modular organization of UNCMIN has another important advantage that we discuss in this section. This advantage is that we can organize the unconstrained minimization algorithm into sections that are functionally independent, such as derivative calculation, step selection, and checking stopping criteria. We may then supply alternative modules for each of these sections, so long as they have the same input and output parameters and perform the same function. We take advantage of this directly by supplying alternative step selection and derivative calculation modules in UNCMIN. Different combinations of these then provide the user with a variety of possible algorithms, which is advantageous since no one algorithm seems to be best for all problem classes. This structure permits a user to compare the effectiveness of the alternative strategies for particular sections of the algorithm, and determine which is best suited for a particular class of problems. In addition, an algorithm developer may develop and test a new version of any module by substituting it for the existing module and comparing the performance of the code using the new and old modules.

To illustrate this design, consider the basic structure of an iteration of our algorithm. In very general terms, it is:

Given $x_c \in R^n$, the best current estimate of the minimizer;

$g_c = \nabla f(x_c)$ or a finite difference approximation to it;

$H = \nabla^2 f(x_c)$ or a finite difference or BFGS approximation to it:

1. Calculate the Newton step $p = -H^{-1} g_c$, or a variation if $H$ is not positive definite.
2. Using $p$, determine the next iterate $x_+$. ("Step selection strategy")

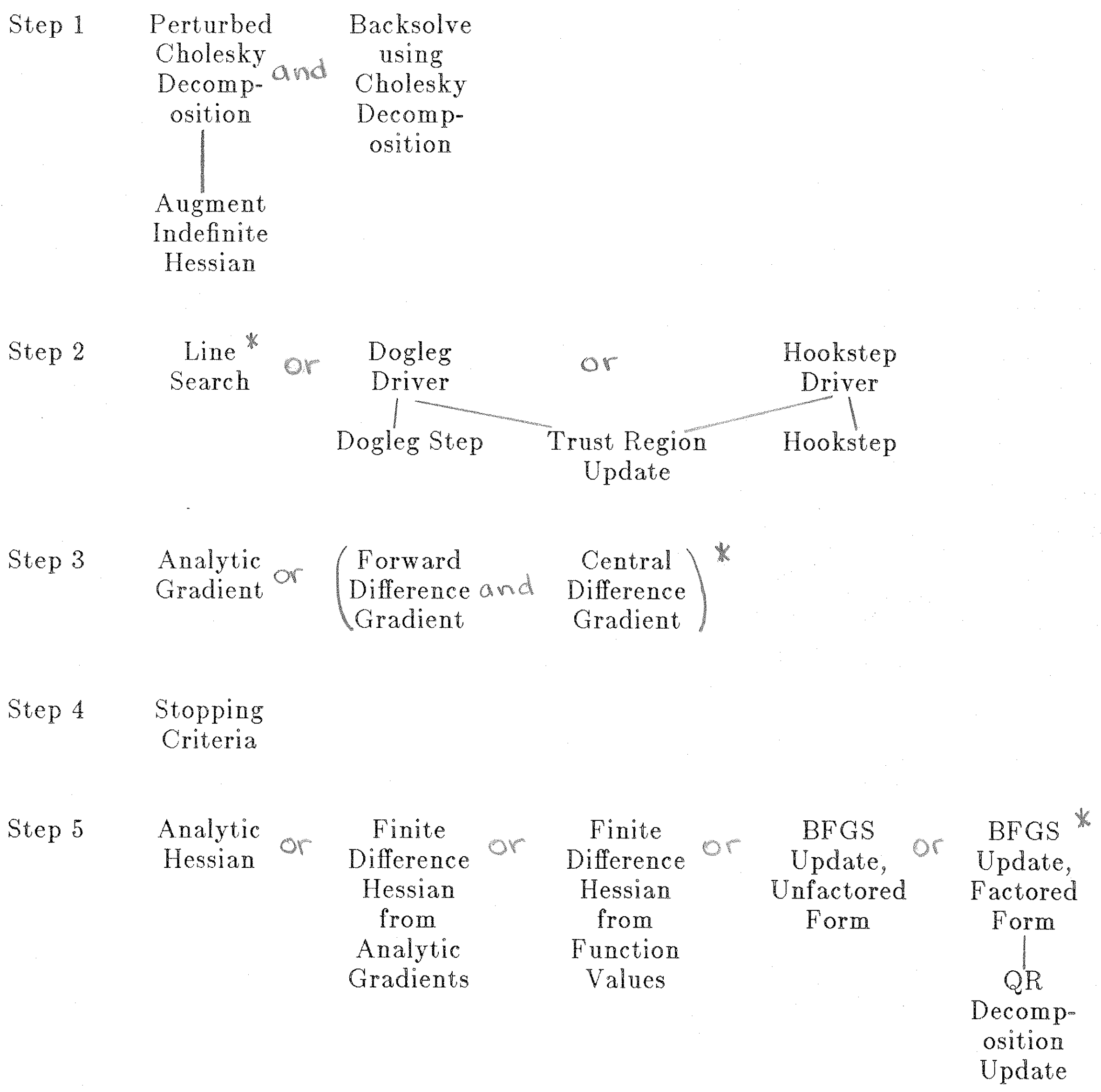
3. Evaluate $g_+ = \nabla f(x_+)$ or an approximation to it.
4. Decide whether to stop. If not:
5. Evaluate $H = \nabla^2 f(x_+)$ or an approximation to it.
6. Set $x_c \leftarrow x_+$, $g_c \leftarrow g_+$ and return to step 1.

In UNCMIN, these steps are carried out by the modules listed in Figure 2.1.

There are eighteen possible algorithmic combinations that a user may select: the cross product of any one of the three step selection strategies (line search, dogleg, or hookstep) with any one of the two gradient calculation methods (analytic or finite difference) and with any one of the three Hessian calculation methods (analytic or finite difference or BFGS update). Some of the other choices shown in Fig. 2.1 are controlled wholly by the software. When using finite difference gradients, the routine starts using forward difference approximations, and switches to central differences only if forward differences seem to have become too inaccurate at some iteration because very small steps are being taken in an uphill direction. When finite difference Hessian approximation is requested, the approximation is made from analytic gradients if they are available, from function values otherwise. When BFGS approximations to the Hessian are used, the factored form of the update (see e.g. Goldfarb [1976]) is used if the step selection strategy is the line search or dogleg; this causes each iteration to require only $O(n^2)$ operations. The factored form does not interface well with the hookstep strategy, so an unfactored update is used with this step selection method.

Of the eighteen user-controlled algorithmic possibilities, three probably are unrealistic, the combinations involving analytic Hessians and finite difference gradients. Any of the other fifteen combinations might be used in practice. The default choice (see Section 5) is the line search with finite difference gradients and BFGS approximation to the Hessian, but our users have used many other combinations. A

Figure 2.1 Modules for Unconstrained Minimization


Step 1
Perturbed Cholesky Decomp-osition
and
Backsolve using Cholesky Decomp-osition
Augment Indefinite Hessian
Step 2
Line Search *
or
Dogleg Driver
or
Hookstep Driver
Dogleg Step
Trust Region Update
Hookstep
Step 3
Analytic Gradient
or
Forward Difference Gradient
and
Central Difference Gradient
*
Step 4
Stopping Criteria
Step 5
Analytic Hessian
or
Finite Difference Hessian from Analytic Gradients
or
Finite Difference Hessian from Function Values
or
BFGS Update, Unfactored Form
or
BFGS Update, Factored Form *
QR Decomp-osition Update

* -- default choice

computational comparison of the three step selection strategies is given in Section 7.

The modular structure of UNCMIN also has been helpful in our development and testing of new strategies for unconstrained minimization, because we can often form the desired new algorithm by replacing just one module in UNCMIN. Then we can compare the performance of the two codes which are identical in all other respects. We have used the modular system to test new secant updates (replacing the BFGS module in step 5), to test new step selection strategies (inserting a new option in step 2), and to test strategies for handling indefinite Hessians (replacing the first part of step 1). Using modular replacement also significantly decreases the time required to construct test codes.

Another advantage of our modular design is that several of the modules can also be used in a similar system of algorithms for solving simultaneous systems of nonlinear equations. Most importantly, all the step selection algorithms of step 2 can be used without change. This would significantly reduce the time to construct additional software for solving systems of nonlinear equations. The pseudo-code in the appendix of Dennis and Schnabel [1983], which is intended for both unconstrained minimization and nonlinear equations, also shares modules in this fashion.

Figure 2.1 does not list all the modules in UNCMIN; in particular, modules involved in the initialization phase of the algorithm and several service modules have been omitted. UNCMIN contains a total of 38 subroutines; a pared down version is described in Section 4.

## 3. Interesting Algorithmic Features

The methods implemented in UNCMIN are described in detail in Dennis and Schnabel [1983]. Our intention in this book was to collect the best existing algorithms rather than to propose new ones. Inevitably, some new features were introduced. In this section, we mention briefly the algorithms available in UNCMIN, and some of our innovations. We assume that the reader of this section is familiar with modern algorithms for unconstrained minimization; some comprehensive references are Fletcher [1980], Gill, Murray, and Wright [1981], and Dennis and Schnabel [1983].

Three step selection strategies, a line search, a dogleg, and a hookstep (locally constrained optimal step), are used. The line search is a backtracking line search using safeguarded quadratic interpolation for the first backtrack and safeguarded cubic interpolation for any subsequent backtracks at each iteration. It terminates when the condition

$$f(x_+) \le f(x_c) + 10^{-4} \nabla f(x_c)^T (x_+ - x_c) \tag{3.1}$$

is satisfied for the first time. A second common line search condition

$$\nabla f(x_+)^T (x_+ - x_c) \ge \beta \, \nabla f(x_c)^T (x_+ - x_c), \quad \beta \in (10^{-4}, 1) \tag{3.2}$$

is not enforced explicitly by the code. Our practical experience is that (3.2) is virtually always satisfied by the first step to satisfy (3.1), and that when it is not, continuing the line search to enforce (3.2) (with, say, $\beta = 0.9$) makes virtually no difference in the ultimate efficiency of the algorithm. Also, our line search algorithm is globally convergent without requiring (3.2) due to the safeguards in the backtracking strategy (see Shultz, Schnabel, and Byrd [1985]), except in the BFGS case where no general global convergence result for a line search algorithm exists.

The dogleg strategy implemented is the double dogleg of Dennis and Mei [1979]. The hookstep algorithm is a minor modification of the Levenberg-Marquardt algorithm of Moré [1978] for nonlinear least squares. Both the dogleg and the hookstep algorithms can assume that the model Hessian matrix is positive definite for reasons indicated below. The trust region updating strategy is fairly closely related to Moré's, and is documented in Dennis and Schnabel [1983]. Both trust region algorithms satisfy the conditions of Shultz, Schnabel, and Byrd [1985] for global convergence.

The formulas for finite difference derivative approximations are the standard ones. Stepsizes are calculated automatically according to the following rules. For forward difference approximations to the gradient (or Hessian approximation using analytic gradients), the stepsize used to perturb the $i^{th}$ component $x_i$ of the current vector $x_c$ is

$$h_i = \mathrm{sign}(x_i) * 10^{-NDIGITS/2} * \max\{|x_i|, typx_i\} \tag{3.3}$$

where *NDIGITS* is the number of accurate digits in the objective function $f(x)$ and $typx_i$ is a typical magnitude of the ith component of $x$. If the user does not supply values for *NDIGITS* and $typx_i$, the default values $NDIGITS = -\log_{10}(macheps)$ -- corresponding to full accuracy in $f(x)$ -- and $typx_i = 1$ are used. For justification of (3.3), see for example Dennis and Schnabel [1983]. For central difference approximation of the gradient, or for Hessian approximation from function values, the stepsize is (3.3) with *NDIGITS/2* replaced by *NDIGITS/3*. In our experience, these stepsizes are quite satisfactory unless the user assumes the default value for *NDIGITS* when in fact far fewer digits of $f(x)$ are accurate. In this case, entirely inaccurate derivative approximations may result. If the user does not know the approximate value of *NDIGITS*, it may be estimated easily and accurately by the routine of Hamming [1971] given in Gill, Murray, and Wright [1981].

When finite difference gradient approximation is selected, our software starts with forward difference approximation, and switches to central differences if at some iteration the steplength or trust region becomes so small that $||x_+ - x_c||$ is within the stopping tolerance, even though $f(x_+)$ does not satisfy (3.1). In this case, the iteration is restarted using a central difference gradient, and central difference gradients are used thereafter. In our experience, this switch is invoked occasionally at the final iterations of the algorithm, especially if the stopping tolerance for the gradient is so stringent that it cannot be satisfied using forward difference gradient approximations.

When BFGS Hessian approximations are requested, the initial approximant is a scaled version of the identity matrix. (Scaling is discussed shortly.) All the approximants are positive definite; in the rare case that

$$(x_+ - x_c)^T\,(\nabla f(x_+) - \nabla f(x_c)) \le (\text{machine epsilon})^{1/2}\,||x_+ - x_c|| \cdot ||\nabla f(x_+) - \nabla f(x_c)||$$

the update at that iteration is skipped. The possibility of skipping (or modifying) updates is necessary because in a trust region BFGS code there is no guarantee that a positive definite secant update will exist. (In spite of this potential drawback, the test results of Section 7 seem to indicate that a trust region BFGS code is a useful addition to a suite of unconstrained optimization algorithms.) In addition, the omission of condition (3.2) can cause our line search BFGS code not to update the Hessian. While this could cause our line search BFGS algorithm to reduce to steepest descent for several iterations when started in an indefinite region, in our experience the skipping of updates is a rare occurrence that has not caused difficulties on test or real-world problems.

A difficult problem in unconstrained minimization is what to do when analytic or finite difference Hessians are used and the current value, say $H$, is not positive definite at some iteration. Various strategies have been proposed, see for example Gill and Murray [1974], Gay [1981], Sorensen [1982], Moré and Sorensen [1983], and Shultz, Schnabel, and Byrd [1985]. Many of these were introduced after we developed our code. In both our line search and trust region algorithms, therefore, we use a fairly simple approach related to the approaches of Gill and Murray [1974] (see also Gill, Murray, and Wright [1981], p.111) and to the hookstep algorithm. A Cholesky factorization of $H$ is attempted; it results in the factorization

$$LL^T = H + D$$

where $D$ is a non-negative diagonal matrix that is zero if $H$ is positive definite. The factorization algorithm is similar to Gill, Murray, and Wright's. Then, if $D \neq 0$, the Cholesky factorization of

$$\overline{H} = H + \min\{sdd, \|D\|_1\} * I$$

is calculated, where $sdd$ is that smallest positive number such that $H + sdd*I$ is "safely" positive definite. Then $\overline{H}$ replaces $H$ for the remainder of the line search or trust region iteration. This approach is more expensive than Gill and Murray's, but it is well justified by its relation to the optimal step approach, and it assures global convergence. It is possible that subsequently developed strategies which deal more directly with indefinite Hessians, such as those in Moré and Sorensen [1983] and Shultz, Schnabel, and Byrd [1985], will perform better in the presence of indefiniteness.

We paid careful attention to two mundane but important aspects of minimization algorithms, scaling and stopping criteria. The software package is coded so that if the user inputs the typical magnitude $typx_i$ of each component of $x$, the performance of the package is then equivalent to what would result from redefining the independent variable in the user's function with

$$x_{scaled} = \begin{bmatrix} 1/typx_1 & & \\ & \cdots & \\ & & 1/typx_n \end{bmatrix}$$

and running the package without scaling. The default value for each $typx_i$ is 1. In our experience, users can usually supply appropriate values for $typx_i$, and most badly scaled problems can be solved successfully using this approach. Strategies in which the code estimates $typx$ at each iteration are still not well established, but such a strategy could be incorporated into UNCMIN merely by adding a rescaling module called once at the start of each iteration.

There are five stopping criteria in UNCMIN: 1) $\nabla f(x_+) \cong 0$; 2) $x_+ \cong x_c$; 3) the package could not satisfy (3.1) at the last iteration; 4) iteration limit exceeded; and 5) divergence suspected ($f(x)$ unbounded below or $f(x)$ approaches a finite value asymptotically from above). We attempted to make the first two tests as scale independent as possible; see Dennis and Schnabel [1983] for details. In our experience, when the code stops due to $\nabla f(x_+) \cong 0$, it is almost always near a local minimizer. When it stops because $x_+ \cong x_c$ it is usually near a solution; however this tolerance should be set quite small since these algorithms sometimes take very small steps while still far from the solution. When the algorithm stops because the last iteration could not satisfy (3.1), it is sometimes near a solution and unable to achieve additional accuracy due to finite precision effects; this occurs most often when finite difference gradients are used, and the accuracy requested is too high. Divergence is tested by imposing a very large maximum step size; if five consecutive steps are at least 99% of this size, divergence is suspected.

## 4. User Oriented Features

We have attempted to make UNCMIN helpful and easy to use. This section discusses three features of UNCMIN included for this reason : alternative calls to UNCMIN, automatic checking of user-supplied analytic derivatives, and various levels of printed output.

UNCMIN may be called either with a very simple calling sequence, or with a more complex sequence in which the user still chooses the amount of information supplied and relies upon defaults for the remaining variables. The simple calling sequence is

CALL OPTIF0 (*NR*, *N*, *X*, *FCN*, *XPLS*, *FPLS*, *GPLS*, *ITRMCD*, *A*, *WRK*).

The user supplies the problem dimension $N$, the matrix and vector work arrays $A$ and $WRK$, the row dimension $NR$ of $A$, the starting value $X$, and the objective function $FCN$. (The other four parameters are output parameters discussed below.) OPTIF0 then calls the subroutine DFAULT to assign default values to all algorithmic option parameters (method of derivative approximation and step selection strategy), stopping tolerances, scaling information, level of output, and several miscellaneous tolerances. Using these defaults it calls subroutine OPTIF9 which does the minimization. To obtain control of any or all of these parameters, the user instead writes a driver that first calls DFAULT to set all input parameters to their default values, then changes only those parameters for which non-default values are desired, and finally calls OPTIF9, using the calling sequence

CALL OTPIF9 (*NR*, *N*, *X*, *FCN*, *D1FCN*, *D2FCN*, *TYPX*, *TYPF*, *METHOD*, *IEXP*, *MSG*, *NDIGIT*, *ITNLIM*, *IAGFLG*, *IAHFLG*, *IPR*, *DLT*, *GRADTL*, *STEPMX*, *STEPTL*, *XPLS*, *FPLS*, *GPLS*, *ITRMCD*, *A*, *WRK*).

This allows the user to be concerned with only the minimum number of parameters necessary. For further details, see Weiss [1980].

Automatic checking of derivatives is provided because in our experience, incorrectly coded derivatives are a common cause of failure of optimization routines. When an analytic gradient or Hessian is supplied, UNCMIN automatically compares its value at the starting point to a finite difference approximation. If any component of an analytic derivative differs by more than 1% from the corresponding finite difference component (with safeguarding for near-zero values), UNCMIN returns with an error termination code. The user may cause derivative checking to be skipped by supplying an appropriate input parameter value.

The package returns a termination code *ITRMCD*, its best approximation to the minimizer *XPLS*, and the function and gradient values *FPLS* and *GPLS* at *XPLS*. The default level of printed output consists of reporting the input parameters, a cause of termination message, and parameter, function and gradient values at the initial and final iterations. By varying the input parameter *MSG*, the user may suppress printed output entirely, or may cause the results after each intermediate iteration to be printed. Even more detailed output useful mainly for algorithm development and testing may be obtained by activating output statements that are imbedded as comments in the code. For more details, see Weiss [1980].

## 5. Computer Environment Considerations

This section discusses the language, storage and floating point environment requirements of UNCMIN.

The entire package is coded in ANSI 1966 standard FORTRAN, and elicits only one objection from the PFORT verifier (Ryder [1974]). In the subroutine FSTOFD, a formal parameter which is an $n \times m$ matrix is allowed to correspond in some calls to an actual parameter which is an $n$-vector. This allows FSTOFD to be used in calculating both finite difference gradients and Hessians. The exception is acceptable to virtually all FORTRAN compilers and the violation was made deliberately on this basis. The package also is acceptable to at least some FORTRAN 77 compilers, though it is not standard due to the use of Hollerith constants.

The real variables in UNCMIN are all single precision, suitable for the equipment of vendors like Cray or CDC whose single precision numbers have 14 or more base 10 places. However the single precision values of IBM or DEC machines, roughly 7 base

10 places, sometimes are insufficient for unconstrained optimization problems. A double precision version of UNCMIN will be available for such machines. The only machine dependent constants used by the code are functions of the machine epsilon, which is calculated by the code to be the smallest power of 2, $2^{-k}$, for which $1 + 2^{-k} > 1$.

The code contains approximately 3200 lines, 55% of which are comments. On some small machines, the object code produced is too long and it is preferable to load only those modules which are being used. For example, a subset consisting of only the modules required for the default algorithmic options (finite difference gradients, secant Hessian approximations, line search) has only 2200 lines. We have found that our users easily construct such pared down versions from our package.

The code requires that the user supply one matrix of size at least $n \times n$ and 9 vectors of size at least $n$ for work space, where n is the problem dimension. The matrix and one of the vectors are used to store one $n \times n$ symmetric matrix and one $n \times n$ lower triangular matrix. It would be possible to implement a few of the methods available in the package using only $n^2/2 + O(n)$ storage, but we have not done this since it is incompatible with the modular structure of our code.

## 6. Reverse Communication

The term reverse communication has been used by several authors (Krogh [1969]; Gill, Murray, Picken, and Wright [1979]; Gay [1980, 1983]; Moré [1980, 1982]; Dennis, Gay, and Welsch [1981]) to describe a program structure in which control is returned to the top level of the package or to the calling program on every occasion that a new function value is required. This capability was required to perform unconstrained minimizations within several time series codes in the National Bureau of Standard's STARPAC library. For this reason, we produced a reverse communication version of UNCMIN called REVMIN.

The concept and use of reverse communication are quite old. Two early uses of which we are aware were in Subroutine VD01 of the Harwell Subroutine Library (see references), written in 1964 by M. J. D. Powell to minimize a nonlinear function of one variable, and in a contour plotting subroutine written in 1965 by C. Lawson, N. Block, and R. Garrett [1965] at Jet Propulsion Laboratory. Little has been written about the reverse communication process, however, especially about the transformation of a non-reverse communication package into a reverse communication package. Therefore, this section discusses in some detail the need for REVMIN and the transformation of UNCMIN into REVMIN.

In simplified terms, the time series applications have the following form. The user calls a time series modeling routine, say TIME, passing into TIME a large amount of data, say *TDATA* (see Fig. 6.1). One task the time series package then performs is to set up and solve a maximum likelihood problem of the form

$$\min_{x \in R^n} \text{TIMEFN}\,(x,\ TDATA)$$

where TIMEFN is a function that it constructs. The difficulty this presents to UNCMIN or any standard minimization routine is that *TDATA* must be available to TIMEFN at each place where TIMEFN is called within UNCMIN. One way to accomplish this is to pass *TDATA* through UNCMIN to each subroutine that contains a call to the objective function, by adding the parameter *TDATA* to each subroutine argument list along the path to each such calling routine. This is undesirable since it would require a separate source and object code version of UNCMIN each time *TDATA* changed form. (We had three different time series applications, each with a different version of *TDATA* consisting of several large vectors and matrices.)

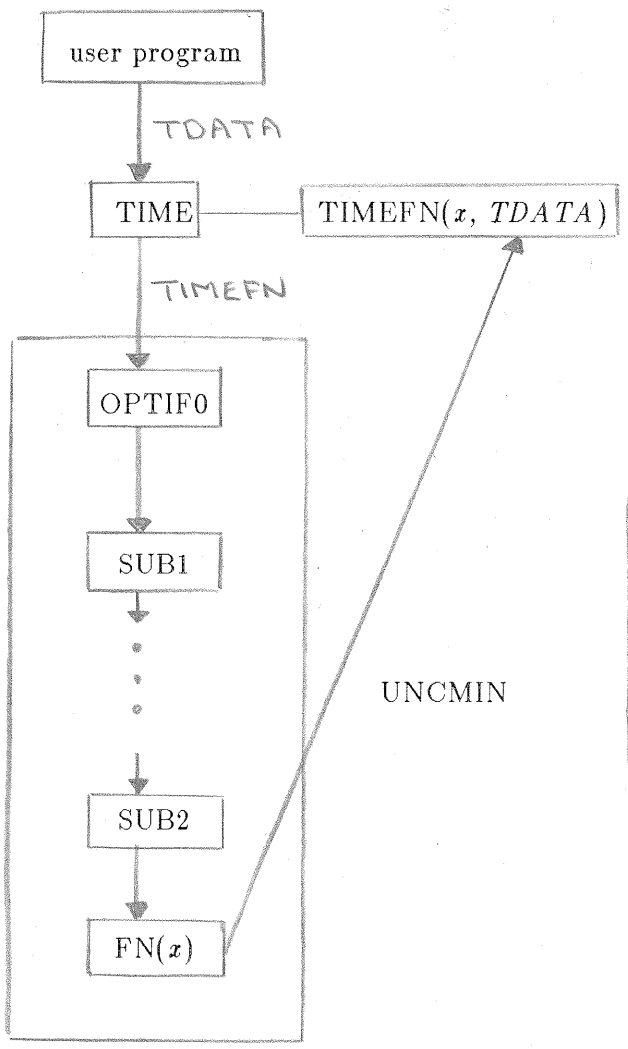


Figure 6.1 --
Non-reverse communication
example

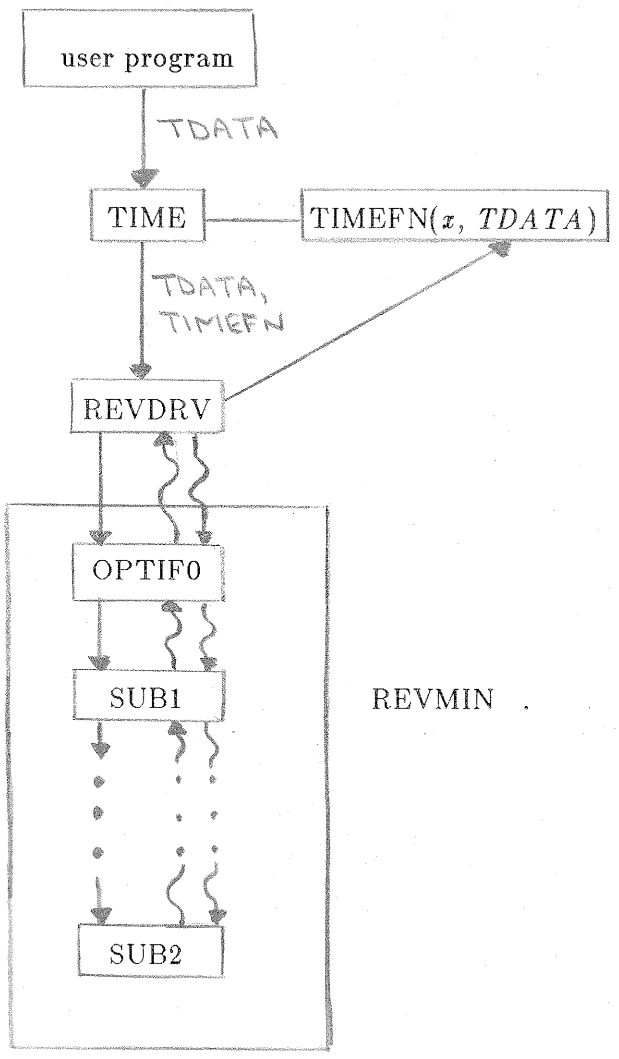


Figure 6.2 --
Reverse communication version
of same example

A second solution is to pass *TDATA* directly from TIME to TIMEFN via a labeled COMMON block. However since *TDATA* is a parameter to TIME, it cannot be defined by TIME to lie in COMMON and would need to be copied into a work array in COMMON instead. Since *TDATA* typically contains vectors with several thousand data values in our applications, this doubling of storage is undesirable.

A third solution is to have the user's program, which constructs *TDATA*, pass it to both TIME and TIMEFN via labeled COMMON. This avoids the need to copy *TDATA* and no extra storage is used. A disadvantage of this solution is that many FORTRAN software packages try to avoid passing arguments through COMMON because of potential for name conflicts between COMMONs. In addition, the fact that COMMON blocks may not contain dynamically dimensioned arrays severely limits the applicability of this solution.

The reverse communication solution is indicated in Figure 6.2. Each time a subroutine in UNCMIN needs a value of a TIMEFN (or an analytic derivative, if supplied by the user) the package records its state, returns up through UNCMIN to a dummy driver, REVDRV, between UNCMIN and TIME, calls TIMEFN using *TDATA* (which has been passed as a parameter to REVDRV), and returns with the function value down to the point in UNCMIN that requested it. To accomplish this, UNCMIN is converted by the process described below into a reverse communication package REVMIN that implements the identical optimization algorithm but obtains values of user-supplied functions by reverse communication. If *TDATA* changes form, the only changes required to REVDRV are to change its formal parameter list, its declaration of *TDATA*, and its call to TIMEFN; REVMIN is unaffected by the form of *TDATA*. We emphasize that the normal and reverse communication versions of a numerical algorithm perform identical calculations, and differ only in the method of obtaining values of user-supplied functions.

To accomplish the bubbling up and down through the package (the wiggly arrows in Fig. 6.2), a RETURN statement is inserted in place of each call to a user-supplied function, say the one in SUB2 in Fig. 6.1. In addition, a local variable (say *SPOT*) is added to SUB2 to identify the location in SUB2 of a request for a function value; it will be used to direct the flow of control to this statement at the end of the bubbling down phase. Also, one new parameter (say *REVPAR*) is added to SUB2 to indicate to the level above whether this is a normal return ($REVPAR = 0$) or a return to obtain a function value ($REVPAR > 0$); if *REVPAR* is positive, its value indicates to REVDRV whether the function, gradient, or Hessian should be evaluated. Finally, the preservation of the values of all local variables in SUB2 is ensured by placing them in a labeled COMMON block that is shared with REVDRV. (In FORTRAN 77 this can be accomplished by means of the SAVE statement instead.) While this may seem to contradict the avoidance of COMMON mentioned above, the difference is that the user does not use COMMON. (Another solution is to pass all the local variables to REVDRV through the parameter lists.) Then all subroutines and subroutine calls on the path from REVDRV down to SUB2, in our example only SUB1, are modified similarly. The statement "IF (REVPAR .GT. 0) RETURN" is inserted following each subroutine call leading down the path, and the local variable *SPOT* is used to identify this subroutine call, to which SUB1 will return control on the way back down. In addition, all the local variables in SUB1 are saved, and *REVPAR* is added to the parameter list so that it may be passed on up.

In this manner, the flow of control is returned to REVDRV. Here, *REVPAR* is used by REVDRV to select the user-supplied function to evaluate, with the proper parameters. After the function call, REVDRV calls REVMIN, which must return with the function value to the statement in SUB2 following the point where the function evaluation was requested. This is accomplished by a GO TO statement conditional on the values of the parameter *REVPAR* and the local variable *SPOT* which is placed at the beginning of each subroutine on the wiggly path between REVDRV and SUB2. If $REVPAR > 0$, meaning that the bubbling down phase of reverse communication is in process, a computed GO TO, based on the value of $SPOT$, takes each subroutine between REVDRV and SUB2 immediately to the subroutine call that will send it to the next subroutine down along the wiggly path. A similar statement in SUB2 sends it to the statement following the point where the function invocation occurred. If $REVPAR = 0$, meaning reverse communication is not in process, the subroutine starts with its original first statement. (The details of passing the normal function arguments up and the function value down have been omitted here.)

The transformation from a normal system of subroutines to a reverse communication version may be accomplished manually; it is facilitated by using a tool such as DAVE (Osterweil and Fosdick [1976]) to identify all the paths in the system to calls of user-supplied functions. We believe it would also be possible to accomplish the transformation by an automatic source to source transformation tool. A problem that must be recognized, however, is the complications that can arise due to the aliasing of function names when they are passed as parameters. This creates the problem that in a statement like CALL FN (...) in SUB2 in Fig. 6.1, FN may refer to any one of the user-supplied functions whose names were input by the user, depending perhaps on the execution path taken to reach SUB2. For example, this occurs in UNCMIN where the first order forward difference routine FSTOFD includes the input parameter $FCN$ and several calls of $FCN$. Here $FCN$ may be the name of either the user-supplied objective function or the user-supplied gradient function, depending, respectively, on whether FSTOFD was called by the driver to compute a gradient or by the Hessian approximation algorithm to compute a Hessian. In a case like this, the transformation of UNCMIN into REVMIN must arrange for each function parameter to be replaced by an integer parameter identifying the function. Additional complications arise from the need to ensure that all arguments to which the optimizer might apply each function are accessible in REVDRV. Furthermore, each distinct set of arguments to which a reverse communication function may be applied represents a distinct call required in REVDRV. Thus, the three subroutine parameters passed to UNCMIN (the function *FCN*, gradient *D1FCN*, and Hessian *D2FCN*) are replaced by seven distinct calls in REVMIN, since the function is evaluated on two different sets of arguments, the gradient on four, and the Hessian on one.

There also are minor technical problems in the transformation to a reverse communication system. If the function call is inside a DO loop (this is common, for example in finite difference routines), the loop must be rewritten in the primitive way since most versions of FORTRAN do not permit re-entry into the body of the loop. Analogous problems may occur with other control structures, for example a call embedded in a logical IF statement.

The additional costs of using a reverse communication version of a system of numerical algorithms are a small amount of additional storage, a fairly small increase in the size of the source and object code (in our case, the number of non-comment source lines increased 15%, from 1670 to 1917), and some addition to the cost of executing the algorithm. Using the rough timing data available on our CYBER 170/750, the increase in CPU time when running the same test problems using UNCMIN and REVMIN averaged between 25% and 50% for problems where the function is inexpensive to evaluate, although occasionally the change was outside this range. The additional cost was at the higher end of this range when both the gradient and Hessian were approximated by finite differences, since this requires many function calls. It was at the lower end when only the gradient was approximated by finite differences and the Hessian by secant updates, and would have been lower yet if analytic gradients had been supplied. Our test problems had dimension between 2 and 20; when using finite difference gradients, the additional cost of reverse communication increases with the dimension of the problem. Of course, if the objective function is at all expensive to evaluate, as it is in many real-world problems such as our time series example, the additional execution cost due to reverse communication quickly becomes negligible.

Finally, we mention that the module capability that has been proposed as a part of FORTRAN 8X (see Wagener [1984]) may eliminate the need for reverse communication. This is because modules would allow the easy and convenient specification and use of global data. In our example, *TDATA* would be defined in a global data module, and TIMEFN would contain the statement USE /TDATA/. Then the non-reverse communication version of the optimizer (Fig. 6.1) would suffice.

## 7. Comparative Testing

The provision of alternative modules for derivative evaluation and step selection in UNCMIN affords an excellent controlled environment for comparative testing. In this section, we summarize the results of three comparative tests we conducted using UNCMIN with the test problems from Moré, Garbow, and Hillstrom [1981].

The first test compared the performance of our algorithms using analytic gradients and Hessians to the performance of the same algorithms using finite difference gradients and Hessians, on a small number of test problems. Table 7.1 shows that in almost all cases, there is very little or no difference in the number of iterations or function evaluations required by a particular method on a particular problem (if the extra function evaluations used in the finite difference approximations are excluded). This is true whether the step selection strategy is the line search, dogleg, or hookstep. Occasionally there is a substantial difference; ordinarily this is due to the sensitivity of the test problems to small changes in the sequence of iterates. In two cases a larger difference occurred because the finite difference method switched to central difference gradients. (The code switches to central difference gradients when it detects that a step using a forward difference gradient is in an uphill direction. This decision usually only occurs after 10 to 20 backtracks and evaluations of $f(x)$ at one iteration have failed to decrease the current function value, thus causing large differences in Table 7.1.)

In our experience, the results of Table 7.1 are fairly typical; as long as the gradient stopping tolerance is within the accuracy obtainable using finite differences gradients, a routine will usually perform about the same using analytic or finite difference derivatives. For this reason and because it is more indicative of how minimization routines are used in practice, we used finite differences rather than analytic derivatives in the subsequent tests. On rare occasions, we noticed that our results were impaired because the automatic stepsizes provided by rules like (3.3) were unsatisfactory.

The other two tests compared the three step selection strategies, line search, dogleg, and hookstep, in an otherwise identical algorithm. The first test compared these strategies when using finite difference gradients and Hessians, while the second compared the same three strategies when using finite difference gradients and secant (BFGS) approximations to the Hessian. In both cases, the test problem set was most of the problems in Moré, Garbow, and Hillstrom [1981].

We present only a brief simple summary of our test results in this paper. The reason for this is that we conclude from our tests that, while the number of iterations, function, and derivative evaluations required by the line search, dogleg, or hookstep methods to solve a particular problem using the same derivative information may differ significantly, the behavior of the three step selection strategies averaged over all the test functions is very close. This is true whether comparing the three strategies using finite difference Hessians, or using secant approximation Hessians. Furthermore, each method is best, and each is worst, on some problems. A more comprehensive reporting of our test results would merely reinforce this conclusion. Such a report can be found in Weiss [1980]; these test results are from an earlier version of our code and differ occasionally, but not significantly, from our final results.

Table 7.2 summarizes the comparison between line search, dogleg, and hookstep when using finite difference gradients and Hessians. It compares the number of functions evaluations required by each method to solve each test problem, including the function evaluations required for the finite difference derivatives. (This statistic is sometimes called "equivalent function evaluations".) If we compared instead the number of iterations required, or separated derivative evaluations from function evaluations, the results would appear very similar. (We comment on run times separately below.) For each test problem, we assign a "1" to the method requiring the smallest number of function evaluations (call this number *probmin*); to the other two methods we assign the number of function evaluations they required divided by *probmin*. If a method failed to solve the problem in 500 iterations or gave up, an F (for failure) is recorded. The final column contains *probmin*. For example, if the line search, dogleg, and hookstep require 11, 10, and 12 function evaluations respectively, the row of numbers would be 1.1, 1, 1.2, 10. The last column allows the table to show absolute as well as relative data. The bottom three lines of Table 7.2 contain the mean, variance, and standard deviation of the first three columns. Our method of reporting results obscures the fact that occasionally one method finds the solution more accurately than another, due to the variety of stopping conditions. Such differences were rare and would not affect our conclusions; moreover, the difference in final values of $f(x)$ in such cases were never more than one order of magnitude, which is small since the optimal objective function value is zero for all test problems.

The main conclusion we draw from Table 7.2 was stated above: there is very little overall difference between the three step selection strategies. Each method solved most but not all of the test problems; the difference in the number of failures probably is insignificant. The standard deviations of the three columns suggest that the differences between the three means are statistically insignificant. Indeed, different implementations of the three strategies might lead to a different ordering of the means.

Table 7.3 summarizes in identical fashion the comparison between line search, dogleg, and hookstep using finite difference gradients and secant (BFGS) Hessian approximations. The bottom lines again indicate no significant difference between the three approaches. It may be significant that the line search method had no failures in this case while the other two methods had several. There were several cases when one method required more than the minimum number of function evaluations but got a significantly more accurate answer (more than two orders of magnitude difference in the final $f(x)$ -- recall that the minimum objective function value is zero in all cases); in these cases, this method also is given a score of 1 in Table 7.3, with the actual rating given in parentheses.

The set of test problems in Tables 7.2 and 7.3 is very similar to the set used by Gay [1983]. We attempted to run each minimization problem in Moré, Garbow, and Hillstrom [1981] from the standard starting point $x_0$, from 10 $x_0$, and from 100 $x_0$. In some cases the latter runs are omitted. Most often this is because the function overflowed at the starting point and occasionally because the problem was too expensive to run or because no method could solve it. On some problems, erroneous finite difference derivative calculations cause all our methods to fail; an example is Brown's badly scaled function, where the scales of the starting and optimal points differ by six orders of magnitude. While the Moré, Garbow, and Hillstrom test set is widely used in unconstrained optimization, it should be noted that all its test problems are zero residual nonlinear least squares problems, i.e. the objective function is a sum of squares and the minimum function value is zero. It is possible, therefore, that our conclusions would differ it a test set with different structural characteristics were used.

Tables 7.2 and 7.3 do not indicate the CPU times required by the various methods. This information is reported in Weiss [1980] and was accumulated for our final tests as well. For the lower dimensional test problems, these times give a rough indication of the overhead cost per iteration of each method. For the larger problems, the times become dominated by the evaluations of the test function, since due to our use of finite differences there are many function evaluations.

As might be expected, in general the line search method requires less time per iteration than the dogleg, and the dogleg less than the hookstep. The differences, however, are not very large. The dogleg occasionally takes as much as 20-25% more time per iteration than the line search, but usually the difference is 10% or less. The discrepancy probably is mainly due to the modular structure that causes the dogleg to require more subroutine calls per iteration than the line search. The hookstep typically requires 20-30% more time per iteration than the line search. An additional cost in the hookstep is the extra matrix factorizations that are sometimes required. On most practical problems we have helped to solve, the objective function was sufficiently expensive to evaluate that these differences in algorithmic overhead were incidental.

The stopping tolerances used in the tests reported in Tables 7.2 and 7.3 are not overly stringent. All the successful runs in Tables 7.2 and 7.3 stop because they satisfy either

$$\max_{1 \le i \le n} \left\{ \frac{|(\nabla f(x_+))_i| \;*\; \max\{|(x_+)_i|,\ typx_i\}}{\max\{|f(x_+)|,\ typf\}} \right\} \le gradtol \tag{7.1}$$

or

$$\max_{1 \le i \le n} \left\{ \frac{|(x_+)_i - (x_c)_i|}{\max\{|(x_+)_i|,\ typx_i\}} \right\} \le steptl, \tag{7.2}$$

where the scaling parameters $typx_i$ and $typf$ have the default value 1 in all cases. The stopping tolerance values were $gradtol = 10^{-5}$ and $steptol = 10^{-10}$, which are typical of the tolerances we see used in practice. Gay [1983] in his tests used much tighter tolerances. We also ran the problems in Tables 7.2 and 7.3 with $gradtol = 10^{-10}$. Since we are using finite difference gradients, this tolerance often is near the limit of the attainable accuracy, and significantly more iterations often were required. The only noticeable difference in the comparative results, however, is that the performance of the hookstep often deteriorated more than the performance of the other two methods. This performance indicates that the methods were not (all) taking Newton steps in their final iterations, due presumably to the inaccuracy in the finite difference gradient; we conjecture that the hookstep method deteriorated most because it makes the most use of the (inaccurate) gradient.

Finally, it is interesting to compare the finite difference Hessian method with the secant approximation method when the same step selection strategy is utilized. In 78 of 102 cases (34 problems with 3 step selection strategies each), the finite difference Hessian method required fewer iterations than the secant method. This confirms the conventional wisdom that second derivative methods are usually, but not always, less expensive than secant methods if function evaluation is sufficiently inexpensive that algorithmic overhead is the overriding cost. On the other hand, the secant method required fewer total function evaluations (including the function evaluations used by finite differences) in all but 16 of the 102 cases. The 16 include all 9 runs of the Brown-Dennis problem and only 7 other cases. This again confirms the conventional wisdom that secant methods usually are more efficient than finite difference Hessian methods on problems where function evaluation is expensive and analytic Hessians are not available.

Table 7.1 -- Comparison between performance of algorithms using analytic vs. finite difference derivatives

| Function | Method of derivative evaluations | Iterations / Function evaluations by | | |
|---|---|---|---|---|
| | | Line Search | Dogleg | Hookstep |
| Rosen-brock | Analytic gradient and Hessian | 24 / 33 | 21 / 25 | 22 / 27 |
| | Finite diff. gradient and Hessian | 23 / 30 | 21 / 25 | 21 / 25 |
| | BFGS, analytic gradient | 23 / 30 | 44 / 64 | 40 / 60 |
| | BFGS, finite diff. gradient | 23 / 30 | 43 / 60 | 41 / 63 |
| Powell | Analytic gradient and Hessian | 15 / 16 | 15 / 17 | 15 / 16 |
| | Finite diff. gradient and Hessian | 15 / 16 | 15 / 17 | 15 / 16 |
| | BFGS, analytic gradient | 48 / 54 | 42 / 51 | 41 / 48 |
| | BFGS. finite diff. gradient | 44 / 70 | 44 / 53 | 41 / 48 |
| Wood | Analytic gradient and Hessian | 58 / 92 | 36 / 40 | 40 / 46 |
| | Finite diff. gradient and Hessian | 56 / 90 | 37 / 41 | 43 / 52 |
| | BFGS, analytic gradient | 32 / 40 | 42 / 57 | 41 / 52 |
| | BFGS, finite diff. gradient | 32 / 40 | 42 / 51 | 46 / 68 |

* switched to central difference gradients

Table 7.2 -- Comparative test results using finite difference gradients and Hessians

| Function | $n$ | Starting point | Multiple of minimum function evaluations | | | Minimum function evaluations |
|---|---|---|---|---|---|---|
| | | | Line Search | Dogleg | Hookstep | |
| Beale | 2 | $x_0$ | 1 | 1.28 | 1.40 | 60 |
| | | $10x_0$ | F | 1 | F | 222 |
| Helical valley | 3 | $x_0$ | 1.44 | 1 | 1.2 | 135 |
| | | $10x_0$ | 1.48 | 1.45 | 1 | 187 |
| | | $100x_0$ | 1 | 1.02 | 1 | 186 |
| Gaussian | 3 | $x_0$ | 1 | 1.06 | 1 | 17 |
| Box 3D | 3 | $x_0$ | 1.64 | F | 1 | 192 |
| Wood | 4 | $x_0$ | 1.55 | 1 | 1.17 | 711 |
| | | $10x_0$ | 1.34 | 1 | 1.08 | 864 |
| | | $100x_0$ | 1 | 1.10 | 1.06 | 888 |
| Brown - Dennis | 4 | $x_0$ | 1 | 1.14 | 1.01 | 138 |
| | | $10x_0$ | 1 | 1.08 | 1.08 | 252 |
| | | $100x_0$ | 1 | 1 | 1 | 366 |
| Biggs Exp | 6 | $x_0$ | F | F | F | 17008 |
| Watson | 9 | $x_0$ | F | F | 1 | 2384 |
| Extended Rosen-brock | 10 | $x_0$ | 1.10 | 1 | 1 | 1610 |
| | | $10x_0$ | 1.49 | 1 | 1.09 | 3671 |
| | | $100x_0$ | F | 1.26 | 1 | 11672 |
| Extend Powell Singular | 8 | $x_0$ | 1 | 1 | 1 | 804 |
| | | $10x_0$ | 1 | 1 | 1 | 1069 |
| | | $100x_0$ | 1 | 1 | 1 | 1387 |
| Penalty I | 10 | $x_0$ | 1.24 | 1.63 | 1 | 2294 |
| | | $10x_0$ | 1.16 | 1 | 1 | 2901 |
| | | $100x_0$ | 1.12 | 1 | 1.02 | 3280 |
| Penalty II | 10 | $x_0$ | 1.15 | 1 | 1.08 | 6708 |
| | | $10x_0$ | 1.10 | 1 | 1.05 | 7166 |
| | | $100x_0$ | 1.11 | 1 | 1.06 | 7623 |
| Variable Dimension | 10 | $x_0$ | 1 | 1 | 1 | 1151 |
| | | $10x_0$ | 1 | 1 | 1 | 1379 |
| | | $100x_0$ | 1.08 | 1.04 | 1 | 1918 |
| Trigono-metric | 10 | $x_0$ | 1 | 1.88 | 1 | 695 |
| | | $10x_0$ | 1.20 | 1 | 1.20 | 1880 |
| | | $100x_0$ | F | 1.20 | 1 | 1075 |
| Chebyquad | 9 | $x_0$ | 1.06 | 1.06 | 1 | 2252 |
| Failures | | | 5 | 3 | 2 | |
| Successes | | | 29 | 31 | 32 | |
| Mean of successes | | | 1.147 | 1.103 | 1.047 | |
| Variance of successes | | | 0.0368 | 0.0411 | 0.00738 | |
| Standard deviation of successes | | | 0.195 | 0.206 | 0.0873 | |

Table 7.3 -- Comparative test results using finite difference gradients and BFGS Hessian approximations

| Function | $n$ | Starting point | Multiple of minimum function evaluations | | | Minimum function evaluations |
|---|---|---|---|---|---|---|
| | | | Line Search | Dogleg | Hookstep | |
| Beale | 2 | $x_0$ | 1.04 | 1 | 1.02 | 51 |
| | | $10x_0$ | 1.10 | 1 | 1.10 | 136 |
| Helical valley | 3 | $x_0$ | 1 (1.11) | 1.04 | 1 | 117 |
| | | $10x_0$ | 1.01 | 1 | 1.05 | 126 |
| | | $100x_0$ | 1 | 1.09 | 1.13 | 123 |
| Gaussian | 3 | $x_0$ | 1.20 | 1 | 1.45 | 20 |
| Box 3D | 3 | $x_0$ | 1.25 | 1 | 1.11 | 118 |
| Wood | 4 | $x_0$ | 1 | 1.34 | 1.65 | 172 |
| | | $10x_0$ | 1 | 1 (1.52) | 1.43 | 302 |
| | | $100x_0$ | 1.16 | 1.09 | 1 | 511 |
| Brown - Dennis | 4 | $x_0$ | 1 | 1.03 | 1.03 | 169 |
| | | $10x_0$ | 1 | 1.01 | 1.01 | 308 |
| | | $100x_0$ | 1 | 1 | 1 | 458 |
| Biggs Exp | 6 | $x_0$ | 1 | 1.14 | 1.07 | 309 |
| Watson | 9 | $x_0$ | 1.08 | 1.01 | 1 | 1302 |
| Extended Rosen-brock | 10 | $x_0$ | 1 (1.30) | 1.01 | 1 | 495 |
| | | $10x_0$ | 1 (1.65) | 1.34 | 1 | 680 |
| | | $100x_0$ | 1 (1.33) | 1 | 1.16 | 1746 |
| Extend Powell Singular | 8 | $x_0$ | 1.22 | 1 (1.08) | 1 | 384 |
| | | $10x_0$ | 1 (1.37) | 1.01 | 1 | 828 |
| | | $100x_0$ | 1.01 | 1 | 1.50 | 1298 |
| Penalty I | 10 | $x_0$ | 1 | F | F | 1756 |
| | | $10x_0$ | 1.18 | 1 | 1.12 | 1753 |
| | | $100x_0$ | 1 | F | F | 2259 |
| Penalty II | 10 | $x_0$ | 1.08 | 1 | 1.03 | 271 |
| | | $10x_0$ | 1.76 | 2.12 | 1 | 2531 |
| | | $100x_0$ | 1.07 | 1 | 1.05 | 5293 |
| Variable Dimension | 10 | $x_0$ | 1.13 | 1.06 | 1 | 171 |
| | | $10x_0$ | 1.06 | 1 (1.58) | 1 | 544 |
| | | $100x_0$ | 1 | [1] | F | 1708 |
| Trigono-metric | 10 | $x_0$ | 1.03 | 1 | 1.03 | 298 |
| | | $10x_0$ | 1 | 1.18 | 1.18 | 965 |
| | | $100x_0$ | 1.66 | [1.07] | 1 | 450 |
| Chebyquad | 9 | $x_0$ | 1.14 | 1 | 1 | 234 |
| Failures | | | 0 | 0 [+2] | 3 | |
| Successes | | | 34 | 32 [-2] | 31 | |
| Mean of successes | | | 1.094 | 1.082 | 1.101 | |
| Variance of successes | | | 0.0293 | 0.0452 | 0.0281 | |
| Standard deviation of successes | | | 0.174 | 0.216 | 0.170 | |

| | | | |
|---|---|---|---|
| Number of times found significantly lower function value than other method but required more iterations | 5 | 3 | 0 |
| Means if parenthesized values used | 1.145 | 1.122 | 1.101 |
| Variance | 0.0396 | 0.0576 | 0.0281 |
| Standard deviation | 0.202 | 0.244 | 0.170 |

[--] -- Close to, but not at, solution
Excluded from summary statistics

## Acknowledgements

We thank Chuck Lawson of Jet Propulsion Laboratory and John Reid of Harwell Laboratory for informing us about the early history of reverse communication. We also thank the referees and associate editor for their many helpful comments.

## 8. References

J. E. Dennis Jr., D. M. Gay, and R. E. Welsch [1981], "An adaptive nonlinear least-square algorithm", *ACM Transactions on Mathematical Software* 7, pp. 348-368.

J. E. Dennis Jr. and H. H. W. Mei [1979], "Two new unconstrained optimization algorithms which use function and gradient values", *Journal of Optimization Theory and its Applications* 28, pp. 453-482.

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

D. M. Gay [1980], "Some tips for writing portable software", Technical Report TR-14, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology.

D. M. Gay [1981], "Computing optimal locally constrained steps", *SIAM Journal on Scientific and Statistical Computing* 2, pp. 186-197.

D. M. Gay [1983], "Subroutines for unconstrained minimization using a model/trust-region approach", *ACM Transactions on Mathematical Software* 9, pp. 503-524.

P. E. Gill and W. Murray [1974], "Newton-type methods for unconstrained and linearly constrained optimization", *Mathematical Programming* 28, pp. 311-350.

P. E. Gill, W. Murray, S. M. Picken, and M. H. Wright [1979], "The design and structure of a Fortran program library for optimization", *ACM Transactions on Mathematical Software* 5, pp. 259-283.

P. E. Gill, W. Murray, and M. H. Wright [1981], *Practical Optimization,* Academic Press, London.

D. Goldfarb [1976], "Factorized variable metric methods for unconstrained optimization", *Mathematics of Computation* 30, pp. 796-811.

R. W. Hamming [1971], *Introduction to Applied Numerical Analysis* McGraw-Hill, New York.

Harwell Subroutine Library, A Catalogue of Subroutines (M. J. Hopper, ed.), Computer Science and Systems Division, A.E.R.E. Harwell, Oxon., England.

IMSL Library Reference Manual, International Mathematical and Statistical Libraries, Houston, Texas.

F. T. Krogh [1969], "VODQ/SVDQ/DVDQ--variable order integrators for numerical solution of ordinary differential equations", Section 314, Subroutine Write-Up, Jet Propulsion Laboratory, Pasadena, California.

C. Lawson, N. Block, and R. Garrett [1965], "Fortran IV subroutines for contour plotting", Section 314, Technical Memo No. 106, Jet Propulsion Laboratory, Pasadena, California.

MINPACK Documentation, Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois.

J. J. Moré [1978], "The Levenberg-Marquardt algorithm: implementation and theory", in *Numerical Analysis, Dundee 1977, Lecture Notes in Mathematics 630,* G. A. Watson, ed., Springer-Verlag, Berlin, pp. 105-116.

J. J. Moré [1980], "On the design of optimization software", in *Otimazzazione Nonlineare e Apllicazioni,* S. Incerti and G. Treccani, eds., Pitagora Editrice, Bologna, Italy.

J. J. Moré [1982], Notes on optimization software", in *Nonlinear Optimization 1981,* M. J. D. Powell, ed., Academic Press, London, pp. 339-352.

J. J. Moré B. S. Garbow, and K. E. Hillstrom [1981], "Testing unconstrained optimization software", *ACM Transactions on Mathematical Software* 7, pp. 17-41.

J. J. Moré and D. C. Sorensen [1983], "Computing a trust region step", *SIAM Journal on Scientific and Statistical Computing* 4, pp. 553-572.

NAG Fortran Library Manual, The Numerical Algorithms Group (USA), Downers Grove, Illinois.

L. J. Osterweil and L. D. Fosdick [1976], "DAVE -- A validation error detection and documentation system for Fortran programs", *Software -- Practice and Experience* 6, pp. 473-486.

B. G. Ryder [1974], "The PFORT verifier", *Software -- Practice and Experience* 4, pp. 359-377.

G. A. Shultz, R. B. Schnabel, and R. H. Byrd [1985], "A family of trust region based algorithms for unconstrained minimization with strong global convergence properties", *SIAM Journal on Numerical Analysis,*22, pp. 47-67.

D. C. Sorensen [1982], Newton's method with a model trust region modification", *SIAM Journal on Numerical Analysis* 19, pp. 409-426.

J. L. Wagener [1984], "Status of work towards revision of programming language Fortran", *SIGNUM Newsletter* 19, No. 3, July 1984.

B. E. Weiss [1980], "A modular software package for solving unconstrained nonlinear optimization problems", M. S. thesis, Department of Computer Science, University of Colorado at Boulder.