CONF-7906140--1

ON THE DESIGN OF OPTIMIZATION SOFTWARE

by

Jorge J. More'

Prepared for

Conference on

Nonlinear Optimization and Applications

L'Aquila, Italy

June 18-20, 1979

gb

U of C-AUA-USDOE

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Operated under Contract W-31-109-Eng-38 for the

U. S. DEPARTMENT OF ENERGY

1. Introduction

MINPACK is a research project whose long term goal is the development of a systematized collection of quality optimization software. One of the results of this project is a package, MINPACK-1, for the solution of systems of nonlinear equations and nonlinear least squares problems. Section 2 of this paper provides an outline of this package and then some of the design decisions made during the production of this package are discussed.

The goal of MINPACK-1 is to minimize the amount of effort required of the user and the computer to solve a particular problem. This goal demands that close attention be paid to the ease of use, reliability and efficiency of MINPACK-1. Ease of use requires the careful design of the user documentation and user interface, while reliability and efficiency require that the algorithms have acceptable global and local convergence properties and that the implementations extend the domain of the algorithms as much as possible.

It is not my intention to provide a complete description of the design principles behind MINPACK-1, but rather to illustrate some of these principles by considering specific examples. The concepts discussed in this paper are robustness, scale invariance, interface routines, and reverse communication.

The concept of robustness has been used with great success in other areas of mathematical software, but it is frequently given too little attention in optimization software. In Section 3 we illustrate the importance of this concept by discussing the robustness of a very simple but important calculation: cubic interpolation. The importance of this calculation derives from its use as a basic step in many one-dimensional optimization routines.

Similarly, although almost every successful developer of optimization algorithms is aware of the importance of scale invariance, the use of this concept in the implementation of optimization software has been overlooked by many. Scale invariance is discusssed in Section 4 and there we note a connection between robustness and scale invariance and show how scale invariance can be used to decide between different versions of an algorithm.

Interface routines and reverse communication are concepts that have been used to facilitate the use of MINPACK-1. Some of the implications of these concepts are treated in Sections 5 and 6 by discussing the design of an interface routine for a nonlinear least squares algorithm, and the implementation of an algorithm for checking that the user-supplied derivation information is consistent with the function values.

An important topic that has been left out is the implications of these concepts to the testing of optimization software. For this topic the interested reader may consult [9] and [10].

Acknowledgements Many people have contributed to the development of MINPACK-1. Among them Jim Boyle spent a lot of time and effort in developing a version of TAMPR suitable for optimization software, Brian Smith and Jim Cody have been very generous with their advice on general software considerations, and Larry Nazareth and Dudley Goetschel contributed to the development of the software. The final product is, however, the responsibility of three people: Burt Garbow Ken Hillstrom, and myself.

2. MINPACK-1

To solve a system of nonlinear equations the user of MINPACK-1 is required to specify n functions $f_i : R^n \to R$. The algorithms are designed to solve the system

(2.1) $$f_i(x_1, \ldots, x_n) = 0 \quad , \qquad 1 \leq i \leq n$$

If $F : R^n \to R^n$ is the mapping whose i-th coordinate is the residual f_i then (2.1) can be written in vector form as

(2.2) $$F(x) = 0.$$

For a nonlinear least squares problem, the user is required to supply m functions $f_i : R^n \to R$. The algorithms are designed to solve the problem

(2.3) $$\min \left\{ \sum_{i=1}^{m} f_i^2(x) : x \in R^n \right\}$$

If $F : R^n \to R^m$ is the mapping whose i-th coordinate function is the residual f_i, then (2.3) is equivalent to the minimum ℓ_2-norm problem

(2.4) $$\min \left\{ \| F(x) \| : x \in R^n \right\} .$$

To solve problems (2.2) and (2.4) we have implemented modifications of Powell's hybrid algorithm and the Levenberg-Marquardt algorithm, respectively. Some of the necessary modifications to the Levenberg-Marquardt algorithm are described in [8], but at present there is no description of our modifications to Powell's hybrid algorithm. We hope to provide complete descriptions of the modified algorithms in the near future.

For each of the algorithms there are two versions. One of the versions only requires the user to provide the function F, while in the other version the user is required to provide the function F and the Jacobian matrix

$$F'(x) = \left(\frac{\partial f_i(x)}{\partial x_j} \right)$$

of F. The advantage of providing the Jacobian is increased reliability; for example, the algorithms are then much less sensitive to noisy functions. The disadvantage of providing the Jacobian is that this is an error-prone task. For this reason, MINPACK-1 also contains a program to check that the Jacobian is consistent with the function values.

There is one additional program in MINPACK-1. Prompted by minicomputer users we have provided a version of the Levenberg-Marquardt algorithm which only requires the storage of an n by n matrix. This algorithm is suitable for nonlinear least squares problems with a large amount of data but a moderate number of variables.

3. Robustness

A robust implementation extends the domain of the algorithm so that it copes with as many problems as possible without a serious loss of efficiency.

There are many aspects to robustness. For linear algebra software these aspects have been discussed by Smith, Boyle and Cody [12], for special functions see Cody's work [2, 3], and for optimization see my discussion in [10]. In what follows we illustrate one of these aspects by considering a very simple calculation: Hermite interpolation by a cubic polynomial. This calculation is a basic step in many one-dimensional linear search routines, and yet, to my knowledge, the problems noted below have not been discussed before.

Cubic interpolation. Given distinct points α_1 and α_2, function values f_1 and f_2, and derivative values g_1 and g_2, let Q be the cubic interpolant such that

$$Q(d_i) = f_i \quad , \quad Q'(d_i) = g_i , \quad i = 1, 2 .$$

If

$$g_1(\alpha_2 - \alpha_1) < 0$$

and

$$g_1 \cdot g_2 < 0 \quad \text{or} \quad f_1 \leq f_2 ,$$

then Q has a minimizer α^* between α_1 and α_2.

Almost any book on optimization has the formula for the calculation of α^*; for example [6, 13]. A straightfoward use of this formula leads to the following implementation.

Implementation

```
IF  $(g_1 \cdot g_2 < 0$   OR   $f_1 \leq f_2)$
    $\theta = 3 \left(\frac{f_1 - f_2}{\alpha_2 - \alpha_1}\right) + g_1 + g_2$
```

$$\gamma = \text{sign}\,(\alpha_2 - \alpha_1)\left[\theta^2 - g_1 \cdot g_2\right]^{\frac{1}{2}}$$

$$\alpha^* = \alpha_1 + (\alpha_2 - \alpha_1)\left[\frac{\gamma - g_1 + \theta}{2\gamma - g_1 + g_2}\right]$$

This implementation is mathematically correct; in particular α^* lies in the interval with endpoints α_1 and α_2. Moreover, this implementation attempts to reduce the round-off error in the evaluation of α^* (see [4]). We now show that this implementation is not robust unless the input values f_1, f_2, g_1, and g_2 are severely restricted.

To make the above claim precise, we first need to discuss three machine parameters: the machine precision, the smallest positive magnitude and the largest magnitude.

The machine precision is the smallest floating point number ϵ_M such that

$$1 + \epsilon_M > 1$$

in working precision, and thus specifies when a floating point number is negligible in an additive operation. In particular,

$$y + x = x \implies |y| \leq \epsilon_M |x| .$$

The smallest positive magnitude DWARF and the largest magnitude GIANT provide bounds on the allowable (working precision) floating point numbers; an attempt to calculate a floating point number x with

$$0 < |x| < \text{DWARF}$$

causes an underflow, while an attempt to calculate a floating point number x with

$$|x| > \text{GIANT}$$

causes an overflow.

The treatment of underflows and overflows depends on the installation, the computer and the compiler. In many systems a quality which underflows is just set to zero and the computation continues. The treatment of overflows is much more varied, but it is reasonable to assume that a quantity which overflows is set to GIANT (with the appropriate sign) and that the user is allowed a certain number of overflows, say 10, before the computation is terminated by the system. In the remainder of this paper it is assumed that overflows and underflows are treated as described above.

In any system it is clearly desirable to avoid overflows and underflows. In general it is not possible to avoid all overflows and underflows without a serious effect on efficiency, so MINPACK-1 implementations only try to avoid destructive overflows and underflows: that is, any overflow or underflow which damages the accuracy of the computation. One of the implications of this design decision is that any underflow in a MINPACK-1 implementation should be ignored, while an overflow signals an unusual condition; namely, that the desired quantity is out of the range of the machine.

It is now natural to ask if the above implementation avoids destructive overflows and underflows. To answer this question in the negative, we show that the computation of $g_1 \cdot g_2$ and γ may lead to destructive overflows and underflows. For example if

$$g_1 = -g_2 = \tfrac{1}{2}\,(\text{DWARF})^{\frac{1}{2}}$$

then $g_1 \cdot g_2$ underflows and thus α^* is not calculated by the interpolation formula. This may not seem like a serious error, but it is nevertheless a deficiency of the implementation and moreover, an unnecessary deficiency. To fix this problem it is sufficient to replace $g_1 \cdot g_2$ by

$$\left(\frac{g_1}{|g_1|}\right) \cdot g_2$$

in the test. A serious error can occur if γ underflows or overflows. For example if

(3.1) $$\alpha_1 < \alpha_2 \ , \qquad g_1 < 0$$

(3.2) $$\frac{f_2 - f_1}{\alpha_2 - \alpha_1} \geq g_2 > 0$$

(3.3) $$\frac{f_2 - f_1}{\alpha_2 - \alpha_1} > \tfrac{1}{2} (\text{GIANT})^{\frac{1}{2}} \ ,$$

then it is straightforward to verify that

$$- \theta > (\text{GIANT})^{\frac{1}{2}} \ .$$

As a consequence, the computation of θ^2 overflows and forces

$$\gamma = (\text{GIANT})^{\frac{1}{2}} \ .$$

This error is disastrous because it now leads to

$$\gamma - g_1 + \theta < 0 \ ,$$

while $2\gamma - g_1 + g_2$ is clearly positive. Thus

$$\alpha^* < \alpha_1 < \alpha_2 \ .$$

This type of failure can also occur if the computation of γ underflows, so it is clear that this computation is not robust. Fortunately it is not difficult to fix matters. If we let

(3.4) $$\sigma = \max \left\{ |\theta| \ , \ |g_1| \ , \ |g_2| \right\}$$

and compute γ as

(3.5) $$\gamma = \text{sign}\,(\alpha_2 - \alpha_1)\, \sigma \left[\left(\frac{\theta}{\sigma}\right)^2 - \left(\frac{g_1}{\sigma}\right) \left(\frac{g_2}{\sigma}\right) \right]^{\frac{1}{2}} \ ,$$

then it is clear that all destructive overflows are avoided. How about underflows? It can be shown that all underflows are non-destructive unless the following two conditions hold:

(3.6) $$\min\left\{\frac{|g_1|}{\sigma}, \frac{|g_2|}{\sigma}\right\} \quad \text{underflows}$$

(3.7) $$\left(\frac{\theta}{\sigma}\right)^2 \quad \text{underflows or is zero .}$$

Although (3.6) will hold from time to time, it is unlikely that (3.7) will hold. To see this, first note that rounding errors usually guarantee that

(3.8) $$|\theta| \geq \epsilon_M \max\left\{|g_1|, |g_2|\right\}$$

where ϵ_M is the machine precision. In fact, about the only way in which (3.8) can fail to hold is if $\theta = 0$. Now, if (3.7) holds then

$$\sigma = \max\left\{|g_1|, |g_2|\right\}$$

and thus (3.8) implies that

$$\left|\frac{\theta}{\sigma}\right| \geq \epsilon_M .$$

However, ϵ_M^2 does not underflow on any of the major machines (although on at least two machine ranges ϵ_M^2 is fairly close to DWARF), so (3.7) does not hold. It is best not to depend on (3.8) and thus we recommend the addition of the test

$$\text{IF } (\alpha^* \leq \min\left\{\alpha_1, \alpha_2\right\} \text{ OR } \alpha^* \geq \max\left\{\alpha_1, \alpha_2\right\})$$

$$\alpha^* = \alpha_1 + \frac{\alpha_2 - \alpha_1}{2} .$$

It could be argued that with this test the above modifications are unnecessary. Our claim is that this test makes the cubic interpolation algorithm robust even in those rare cases where (3.6) and (3.7) hold. On the other hand, the overflow and underflow problems which prompted the above modifications can occur quite frequently during the initial stages of a search where large function and derivative values are likely, or during the final stages where small derivatives are expected.

4. Scaling

MINPACK-1 algorithms must be scale invariant. To be more precise, we require that if the algorithm is applied to functions $\hat{F}$ and F related by the change of scale

$$\begin{aligned} \hat{F}(x) &= \alpha F(Dx) \\ \hat{x}_0 &= D^{-1} x_0 \end{aligned} \tag{4.1}$$

where α is a positive scalar and D is a diagonal matrix with positive diagonal entries, then the algorithm must generate iterates which satisfy

$$\hat{x}_k = D^{-1} x_k , \qquad k > 0 . \tag{4.2}$$

This is a very natural requirement which can have a significant effect on the implementation and performance of the algorithm.

To illustrate some of the consequences of scale invariance, consider the cubic interpolation algorithm of the previous section. For this algorithm it is easy to verify that α^* is unchanged if it is applied to the scaled data.

$$\mu f_1 , \mu f_2 , \mu g_1 , \mu g_2 \tag{4.3}$$

for any $\mu > 0$, and thus we also would like the implementation to satisfy this requirement. However, if the original data satisfies (3.1) and (3.2), then the scaled data also satisfies (3.3) for suitable μ, and we have already seen that a naive implementation fails if (3.1), (3.2) and (3.3) holds. Thus scale invariance requires robust implementations.

As another illustration of the consequences of scale invariance, consider the problem of determining an approximation to the Jacobian matrix. If $\hat{F}$ and F are related by the change of scale (4.1) then

$$\hat{F}'(\hat{x}_0) = \alpha F'(x_0) D ,$$

and thus we would like our approximation to retain this property. If we compute an approximation $A(x)$ to $F'(x)$ by forward differences then

$$A(x)e_j = \frac{F(x + \eta_j e_j) - F(x)}{\eta_j}$$

for some non-zero scalar η_j. If ϵ_F specifies the relative errors in the function F, then our choice of η_j is

$$\eta_j = \epsilon_F^{\frac{1}{2}} |x_j| \tag{4.4}$$

unless this results in a zero η_j and in this case η_j is set to $\epsilon_F^{\frac{1}{2}}$. Two other choices that have been proposed in the literature are

$$\eta_j = \epsilon_F^{\frac{1}{2}} \tag{4.5}$$

and

$$\eta_j = \epsilon_F^{\frac{1}{2}} (|x_j| + 1) . \tag{4.6}$$

It is now straightforward to verify that if $\hat{A}(x)$ is the forward difference approximation to $\hat{F}'(x)$ then

$$\hat{A}(\hat{x}_0) = \alpha A(x_0) D \tag{4.7}$$

for choice (4.4), but not for choice (4.5) or (4.6).

In obtaining (4.7) we assumed that the relative errors in the function are not affected by the change of scale (4.1); this is certainly the case if α and the elements of D are powers of the base of the machine. In general the appropriate choice of ϵ_F is difficult, but if the functions are not subject to large errors then ϵ_F should be of the order of the machine precision.

We emphasize that although (4.4) leads to a scale invariant determination of the forward difference approximation, this does not mean that choice (4.4) is good. Experience has shown, however, that scale invariance in an important property which should not be given up lightly, and that unless other considerations prevail a scale invariant algorithm is to be preferred. Thus scale invariance gives the edge to (4.4) over (4.5) and (4.6). Also note that (4.5) is sometimes inappropriate since

$$x_j + \epsilon_F^{\frac{1}{2}} = x_j$$

for sufficiently large $|x_j|$, and in this case the j-th column of $A(x)$ is zero.

5. Interface Routines

It is almost a truism that the user of any piece of mathematical software prefers a short calling sequence. The main reason for this is that a subprogram with a short calling sequence is easier to use than a subprogram with a longer calling sequence. On the other hand, a subprogram with a short calling sequence may not provide the necessary flexibility.

This conflict between flexibility and ease of use is usually resolved by providing an interface routine. The construction of these interface routines requires some design decisions which are of particular importance to the testing and comparison of the software. The reason for this is that the testing and comparison of algorithms is only reasonable if the number of parameters which affect the behaviour of the algorithm are limited as much as possible and thus, testing and comparisons are usually carried out on these interface routines. It is therefore important that these interface routines do not impose a heavy burden on the efficiency and reliability of the main routines. If the underlying routine is well-designed this is easy to do, but otherwise the interface routine can be unreliable and inefficient on very reasonable problems.

To illustrate the process by which these interface routines are constructed, consider the MINPACK-1 program for the solution of nonlinear least squares problems with a user-supplied Jacobian. This program (LMDER) has 24 parameters:

FCN, M, N, X, FVEC, FJAC, and LDFJAC are the parameters associated with the user-supplied subroutine. FCN is the name of the subroutine, M and N are the number of equations and variables, respectively, X is the vector of variables, FVEC is the vector of function values, and FJAC is the (LDFJAC,N) array for the Jacobian matrix.

FTOL, XTOL and GTOL are three kinds of tolerances.

MAXFEV is a limit on the number of function evaluations.

DIAG and MODE specify the type of scaling that is desired.

FACTOR is used to determine a limit on the initial step

NPRINT specifies the amount of printing to be done.

INFO returns information to the user on the reason for the termination of the iteration.

NFEV and NJEV are the number of function and Jacobian evaluations, respectively, used by the algorithm.

IPVT, QTF, WA1, WA2, WA3, and WA4 are various arrays used by LMDER.

To obtain an easy to use interface routine we simplify the calling sequence as follows:

FTOL, XTOL, GTOL are replaced by TOL. LMDER is then called with FTOL = TOL, XTOL = TOL, and GTOL = 0.

MAXFEV is set to 100*(N+1).

MODE is set to 1. This specifies automatic scaling and does not require DIAG to be specified on input.

FACTOR is set to 100.

NPRINT is set to 0. This specifies no printing.

NFEV and NJEV are not returned to the user.

QTF, WA1, WA2, WA3, and WA4 are replaced by the parameters WA, LWA.

This interface (LMDER1) has 12 parameters. It is possible to reduce this number further by eliminating FCN, TOL, FJAC, and LDFJAC, but in the case of FCN and TOL it was judged that the flexibility provided by these parameters justifies their appearance in the calling sequence. In the case of FJAC and LDFJAC it just seemed somewhat unnatural for FJAC not to appear in the calling sequence of LMDER1 and yet to appear in the calling sequence of FCN. Other decisions are certainly possible and for example, Gill et al. [5] decided not to include FCN or TOL in the calling sequence of their interface routines.

In addition to providing an interface, LMDER1 could also perform other tasks. For example, it could check for consistency of the Jacobian with the function values. If an inconsistency is found then the computation could be terminated and the user informed of the inconsistency. This course of action assumes that the check for consistency does not fail, and this can be a very risky assumption. An alternative course of action is to allow the user to turn off the consistency check, but this complicates the interface. It seems best to limit the role of LMDER1 to that of an interface.

It is important that the automatic choice of parameters made by LMDER1 does not degrade the efficiency and reliability of LMDER. For example, FACTOR is used to determine a bound on the length of the initial step taken by the algorithm. A severe underestimate for FACTOR can lead to a decrease in efficiency. However, since the stepbound is continually revised and if necessary doubled at each iteration, a small value for FACTOR does not have a serious effect on efficiency. Some algorithms have a parameter which allows a user to limit the steplength at <u>every</u> iteration; for these algorithms the distance travelled by the algorithm is bounded by a linear function of the number of iterations, and thus a severe underestimate for the initial stepbound has a disastrous effect on efficiency.

6. Reverse Communication

To solve a system of nonlinear equations or a nonlinear least squares problem, a user of MINPACK-1 must write a subroutine FCN which provides function values and if needed, the Jacobian matrix. In the case of the nonlinear least squares solver LMDER, the calling sequence of FCN is

```
FCN (M, N, X, FVEC, FJAC, LDFJAC, IFLAG) .
```

If IFLAG = 1 the user must calculate the functions at X and return this vector in FVEC, while if IFLAG = 2 then the Jacobian matrix at X must be returned in the (LDFJAC, N) array FJAC. Requiring the user to specify FCN can have some disadvantages which would disappear if LMDER were to communicate with the user by reverse communication. In the remainder of this section we elaborate on this remark.

The main disadvantage of requiring the user to write FCN is that it makes the sharing of information with FCN difficult. For example, it is often the case that the Jacobian matrix $F'(x)$ has some terms which already appear in $F(x)$, and in these cases it may be important to share this information. For instance, if the i th residual is of the form

$$f_i(x) = g(x)^3$$

for some function g, then the gradient of f_i is

$$\nabla f_i(x) = 3\, g(x)^2\, \nabla g(x) ,$$

and in this case the evaluation of the i th row of $F'(x)$ would be simplified if the value of $g(x)$ were available. Note, however, that sharing information between function and Jacobian usually requires COMMON and is therefore not always convenient.

The above problem does not exist if the user is not providing the Jacobian, but a related problem is that in some cases the information necessary to compute $F(x)$ or $F'(x)$ is naturally available in the user's main program. For example, if the residuals of a nonlinear least problem are of the form

$$f_i(x) = \varphi_i(x) - y_i$$

for some functions φ_i and data y_i, and if the data y_i is calculated in the main program, then this information must be passed to FCN. This requires the use of COMMON.

The concept of reverse communication is best explained by considering the following problem: Write a subroutine CHKDER which checks that the user-supplied derivative information is consistent with the function values. CHKDER must check Jacobians of nonlinear equations, Jacobians of nonlinear least squares, gradients, and Hessians.

As a first step, note that all of the problems mentioned above are special cases of checking that the gradients of m nonlinear functions are consistent with the function values, or equivalently, that the Jacobian of a mapping $F : R^n \rightarrow R^m$ is consistent with the function values. For example, if checking gradients then $m = 1$ while if checking Hessians then $m = n$ and

$$F(x) = \nabla f(x)$$

for some function $f: R^n \rightarrow R$.

To implement CHKDER we will use the following outline:

<u>Algorithm</u>

1. Determine a suitable vector p.
2. Evaluate $F(x)$, $F(x+p)$, and $F'(x)$.
3. Compare $F(x+p) - F(x)$ with $F'(x)p$.

If the implementation of CHKDER requires that the user supply the function values and Jacobian matrix by a subroutine FCN, then the resulting implementation would have all of the problems mentioned at the beginning of this section. Moreover, the user must now write an interface routine between FCN and the routines which provide function and derivative information.

To avoid these problems, consider an implementation of CHKDER with the calling sequence

CHKDER (M, N, X, FVEC, FJAC, LDFJAC, XP, FVECP, MODE, ERR)

The user must call CHKDER twice, first with MODE = 1 and then with MODE = 2. The first call to CHKDER executes the first step of the algorithm by setting XP to $x + p$. Before the second call to CHKDER the user must set FVEC to $F(x)$, FVECP to $F(x+p)$, and FJAC to $F'(x)$. The second call to CHKDER executes the third step of the algorithm; on exit ERR(I) is set to a normalized estimate of the consistency between the I th function and the I th gradient.

Although the use of reverse communication in the above problem provided a very neat solution, the use of reverse communication on a general optimization problem has its problems. One of the problems is that reverse communication increases the complexity of the code and is thus not as easy to use. For example, the use of reverse communication requires saving the values of certain variables between calls. If these variables are saved by placing them in the calling sequence, then this increases the complexity of the calling sequence. Another possibility is to use COMMON, but this requires that the user insert the appropriate COMMON block in his program. Note, however, that the SAVE statement of Fortran 77 [1] would eliminate this particular problem. Another problem is that reverse communication requires repeated entries (exits) into (from) different parts of the program, and thus the resulting code is harder to write and understand.

In a particular application the advantages of using reverse communication may outweigh the disadvantages. This was the case in the above problem of checking for consistency between function and derivative values. A more sophisticated use of reverse communication is described by Mallin-Jones [7]. It also seems that reverse communication will be very useful in the solution of sparse optimization problems (Reid [11]).

References

1. Brainerd, W. (editor), Fortran 77, CACM 21 (1978), 806-820.

2. Cody, W.J., The construction of numerical subroutine libraries, SIAM Review, 16 (1974), 36-46.

3. Cody, W.J., An overview of software development for special functions, in Numerical Analysis: Dundee 1975, G.A. Watson, ed., Lecture Notes in Mathematics 506, Springer-Verlag, 1976.

4. Davidon, W.C., Solution of problem 74-3: Davidon's cubic interpolation, by S.K. Park and T.A. Straeter, SIAM Review, 17 (1975), 170-171.

5. Gill, P.E., Murray, W., Picken, S.M., and Wright, M.H., The design and structure of a Fortran program library for optimization, National Physical Laboratory Report NAC82, Teddington, England, 1977.

6. Luenberger, D.G., Introduction to linear and nonlinear programming, Addison-Wesley, 1973.

7. Mallin-Jones, A.K., Nonlinear algebraic equations in process engineering calculations, in Numerical Software - Needs and Availability, D. Jacobs, ed., Academic Press, 1978.

8. Moré, J.J., The Levenberg-Marquardt algorithm: implementation and theory, in Numerical Analysis, G.A. Watson, ed., Lecture Notes in Mathematics 630, Springer-Verlag, 1977.

9. Moré, J.J., Garbow, B.S., and Hillstrom, K.E., Testing unconstrained optimization software, Applied Mathematics Division Technical Memorandum 324, Argonne National Laboratory, 1978.

10. Moré, J.J., Implementation and testing of optimization software, Department of Applied Mathematics and Theoretical Physics Report DAMTP 79/NA4, University of Cambridge, England, 1979.

11. Reid, J.K., Software for sparse matrices, in Numerical Software - Needs and Availability, D. Jacobs, ed., Academic Press, 1978.

12. Smith, B.T., Boyle, J.M. and Cody, W.J., The NATS approach to quality software, in Software for Numerical Mathematics, D.J. Evans, ed., Academic Press, 1974.

13. Wolfe, M.A., Numerical methods for unconstrained optimization, Van Nostrand, 1978.