AD-A044 905 STANFORD UNIV CALIF SYSTEMS OPTIMIZATION LAB F/G 9/2
THE DESIGN AND STRUCTURE OF A FORTRAN PROGRAM LIBRARY FOR OPTIM--ETC(U)
APR 77 P E GILL, W MURRAY, S M PICKEN N00014-75-C-0865
UNCLASSIFIED SOL-77-7 NL

1 OF 1
AD
A044905

END
DATE
FILMED
10-77
DDC

THE DESIGN AND STRUCTURE OF A FORTRAN PROGRAM LIBRARY FOR OPTIMIZATION

BY

PHILIP E. GILL, WALTER MURRAY, SUSAN M. PICKEN,

AND MARGARET H. WRIGHT

TECHNICAL REPORT SOL 77-7
APRIL 1977

ADA044905

DDC
OCT 5 1977

AD No.
DDC FILE COPY

Systems Optimization Laboratory

Department of Operations Research

Stanford University

Stanford California 94305

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

THE DESIGN AND STRUCTURE OF A FORTRAN PROGRAM LIBRARY FOR OPTIMIZATION

by

Philip E. Gill,* Walter Murray,* Susan M. Picken,* and Margaret H. Wright

Technical Report SOL 77-7

April 1977

DDC
RECEIVED
OCT 5 1977
C

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
Stanford University
Stanford, California

*Division of Numerical Analysis and Computing, National Physical Laboratory, Teddington, England.

Research and reproduction of this report were partially supported by the Office of Naval Research Contract N00014-75-C-0865; National Science Foundation Grant MCS76-20019; the U.S. Energy Research and Development Administration Contract EY-76-S-03-0326 PA #18 at Stanford University.

Reproduction in whole or in part is permitted for any purposes of the United States Government. This document has been approved for public release and sale; its distribution is unlimited.

408 765

TABLE OF CONTENTS

SECTION		PAGE
1	INTRODUCTION	1
	1.1 The evolving need for program libraries	1
	1.2 Issues in the design of a program library	2
	1.3 Purpose of this paper	4
2	OVERALL PHILOSOPHY	6
	2.1 Statement of problem and objectives of library design	6
	2.2 Choice of algorithms	7
	2.3 Algorithm standards	9
	2.4 Structure of implementations	10
	2.5 Structure vs program efficiency	12
3	DISCUSSION OF DESIGN DECISIONS	14
	3.1 Communication by parameter list	14
	3.2 Choice of data structures	16
	3.3 Local arrays	19
	3.4 Work space arrays	21
	3.5 Interface with the user	23
	3.6 Service routines	25
	3.7 Auxiliary routines	28
	3.7.1 Subroutines vs in-line code	28
	3.7.2 Flexibility of auxiliary routines	31
	3.8 Communication with user-supplied routines	33
	3.9 Documentation	36
4	A DETAILED ILLUSTRATION - IMPLEMENTATION OF A STEP-LENGTH ALGORITHM	40
	4.1 Need for a step-length algorithm	40
	4.2 Choice of a step-length algorithm	41
	4.3 Provision of user-defined parameters	42
	4.4 Control structure of a step-length algorithm	45
5	CONCLUSION	48
6	APPENDIX (an example of documentation)	49
7	ACKNOWLEDGEMENT	57
8	REFERENCES	58

ACCESSION for
NTIS
DDC
UNANNOUNCED
BY
DISTRIBUTION/AVAILABILITY CODES
A

Abstract

This paper discusses in substantial detail the design principles and structure of an existing Fortran program library whose primary application is to solve optimization problems. Such a discussion not only helps to clarify the scope of application for potential users of the library, but also is useful for workers on other software projects. The fundamental objectives of the present library have been to produce sound, careful implementations of reliable methods that represent the state of the art in numerical optimization. The general implications of these overall design aims are presented, as well as specific instances of the results of decisions to include particular desirable features.

1. Introduction

1.1 The evolving need for program libraries

The manner in which computer implementations of numerical algorithms are disseminated has changed dramatically since the early days of automatic computation. Initially, a user who required a computer routine to solve a particular numerical problem would typically consult research journals that might contain a theoretical description of an appropriate method, and then write his own code. Unfortunately, such personalized implementations were subject to a significant risk of unreliability, because of a lack of attention to details of programming or numerical analysis. Furthermore, as the complexity of numerical methods increased, it became impractical for individuals to write their own versions of all necessary algorithms, even given the will to do so.

Subsequently, it became the practice among some authors of new numerical methods to publish computer programs as well as theoretical descriptions. Although this development was a step in the right direction, in many ways the situation was even more complicated. The quality of the published programs varied enormously, and there was little uniformity of standards concerning programming structure and style. In addition, the published programs were often written only to test the proposed method on a small set of problems (usually well-behaved), and displayed serious errors of programming as well as the inability to detect and recover from numerical difficulties. Under these conditions, the user was obliged to undertake a search of the literature, among a large collection of published routines, with no guidelines to assist in making a good choice.

The inefficiency and confusion resulting from an uncontrolled proliferation of alternative routines led to an awareness of the need for program libraries. This term does not mean simply a collection of programs from varied sources, written in isolation, that are grouped together in one place with some uniform documentation. Rather, a program library is a set of routines that are conceived and written within a unified framework, to be available to a general community of users. Today there is wide recognition of the great value of program libraries designed to solve useful classes of numerical problems, since the latest developments in mathematical computing can thereby be made available to many users.

1.2 Issues in the design of a program library

Two aspects of the design and production of a program library will be considered here.

First, it is essential that each individual routine contained in a program library can be classified as "good" software. This requirement implies the need for analysis of the purpose of every routine, since the task of developing a sound and careful implementation of a numerical method is extremely difficult and time-consuming, even for experts (see the Preface to Wilkinson and Reinsch, 1971). The principles upon which good computer programs for numerical methods should be based have been discussed by numerous authors in varying contexts (see, for example, Byrne and Hindmarsh, 1975; Cody, 1974, 1975, 1976; Cox, 1974; de Boor, 1971a; Dixon, 1974; Ford and Hague, 1974; Hillstrom et al., 1976; Rice, 1971; Schonfelder, 1974; Shampine and Gordon, 1975; Smith et al., 1974a). Almost without exception, it is agreed that mathematical software should be designed to satisfy certain

criteria (see Rice, 1971, for a widely quoted list). However, it is further recognized that some of these desirable qualities are inherently contradictory (see, for example, Cox, 1974; Dixon, 1974; Hillstrom et al., 1976; Kahaner, 1971, 1976; Lyness and Kaganove, 1976; Rice, 1971). Consequently, the creation of any computer program must include decisions, implicit and explicit, concerning the relative weight and importance to be assigned to the possibly conflicting attributes.

Second, a uniform global design should be imposed on the entire program library, whose definition implies the existence of a collective purpose that transcends the aims of any particular routine. The significant assumption that a program library should be available and useful to a general-user community has several immediate consequences. In particular, the demands upon the library routines will necessarily reflect differing levels of interest and expertise, and hence will vary widely from user to user. For example, a user who is familiar with the details of an algorithm may wish the corresponding library implementation to contain sufficient flexibility to be "tuned" to a particular frequently occurring problem; a user who simply needs to solve a problem one time only may want to be able to use the library routines with a minimum of time and effort, without being required to learn any details of the underlying algorithms or software. Similarly, the criteria that are considered desirable in the library routines will change drastically with different applications. In one instance, a library program may be used to solve a problem for which the cost of computer time is negligible compared to the implications of failing to solve the problem or of finding a poor solution, so that the need for reliability dominates all other criteria. In another application of the same numerical

problem, however, the most important consideration may be speed of execution, even at the risk of inaccuracy or failure.

Clearly, a "perfect" library can never exist. Any particular routine, or the entire library, will inevitably be subject to criticism from some individuals for not satisfying their ideal requirements, since all desirable features cannot be achieved simultaneously. Therefore, the design decisions and the justification of those decisions should be specified explicitly for any program library. A presentation of the overall philosophy behind a library helps to clarify the scope of application for potential users, and allows an informed judgment of how the available routines may be used most effectively for any given problem. In addition, such a discussion is helpful for workers on other software projects because similar difficulties and questions often recur in efforts to create program libraries; a discussion of experience and alternatives in a particular library design can reveal non-obvious, or unfortunate, consequences of certain critical decisions, which might not otherwise become apparent until significant effort had been expended.

1.3 Purpose of this paper

Descriptions of collections of computer routines have been published, in varying degrees of detail, for several areas of numerical analysis: quadrature (see de Boor, 1971b); data fitting (see Cox, 1974); ordinary differential equations (see Byrne and Hindmarsh, 1975; Shampine and Gordon, 1975); nonlinear partial differential equations (see Sincovec and Madsen, 1975); eigensystems (see Smith et al., 1974b); and special functions (see Cody, 1976). In some of these cases, the motivation for the chosen implementation is presented, either explicitly or by implication. There are

also examples of documentation of routines for optimization (see, for example, Fletcher, 1972; James and Roos, 1975; Land and Powell, 1973; Lasdon et al., 1976; Mylander, Holmes, and McCormick, 1974; Rosen and Wagner, 1975). However, none of these latter instances includes a discussion of the reasons for the given design, nor qualifies as a "program library".

The purpose of this paper is to discuss in substantial detail the design principles and structure of an existing Fortran library whose primary application is to solve optimization problems. The detail is included to avoid general statements of principles with which everyone would agree. There is an immense gap between a theoretical description of intent in software writing — for example, "we emphasize modularity, good structure, and efficiency" — and an analysis of how well the chosen design in fact satisfies the proposed criteria. Only the latter process determines the quality of the resulting software.

This paper is not meant to serve as documentation for the library, but rather to provide a perspective on the complete design; however, specific examples will always be cited to illustrate relevant points. Detailed documentation of the routines is published separately (see, for example, Gill et al., 1975a, and the appendix).

2. Overall Philosophy

2.1 Statement of problem and objectives of library design

The optimization problems to be considered can be stated as follows:

P1: minimize $F(x)$, $x \in E^n$

$$\text{subject to } c_i(x) = 0, \quad i = 1,2,\ldots,m',$$

$$c_i(x) \geq 0, \quad i = m'+1,\ldots,m,$$

where F and $\{c_i\}$ are given real-valued functions. The function $F(x)$ is normally termed the "objective function," and the set $\{c_i\}$ is the set of "constraint functions."

Certain kinds of numerical problems have the property that the algorithms to solve them are finite decision procedures, or "reliable exact arithmetic algorithms" (see Lyness and Kaganove, 1976, for a further discussion of ways to characterize numerical problems). To create a good computer implementation of such an algorithm is in itself a difficult task, but at least the software designer need not be concerned about the merits of the underlying theoretical procedure.

Such finite decision procedures do not exist for most optimization problems of the form P1, and it is necessary to make some assumptions about the properties of F and $\{c_i\}$ in order to have any hope of solving these problems with an automatic routine. The creation of an optimization library thus involves not only programming considerations, but also definition of a model of the problems to be solved and selection of appropriate algorithms.

The objective of the present library is to provide sound, careful implementations of reliable methods for solving useful categories of optimization problems. The routines are written in a portable subset of ANSI Fortran (ANSI, 1966), so that they may be used with a minimum of change on several machine ranges. The library is continually modified to represent as closely as possible the prevailing "state of the art" in numerical optimization.

2.2 Choice of algorithms

The efficient solution of optimization problems by a single, all-purpose, method is not possible, and the structure of the library reflects the variety of theoretical algorithms. This diversity is not surprising, in view of the complexity of optimization problems. Although solving the eigenvalue problem is based in most cases on a finite decision procedure, the EISPACK set of eigensystem software (Smith et al., 1974b) contains 58 distinct explicitly called subroutines and 12 driver subroutines, the choice of which depends on properties of the given problem; the number of alternative routines for optimization problems will necessarily be even larger.

Optimization algorithms are designed to solve particular categories of problem, where each category is defined by properties of the objective and constraint functions, as illustrated below.

Properties of $F(x)$	Properties of $\{c_i(x)\}$
Linear	No constraints
Sum of squares of linear functions	Upper and lower bounds
Quadratic	Linear
Sum of squares of nonlinear functions	Sparse linear
Nonlinear	Nonlinear

The choice of algorithm depends not only on the type of problem, but also on the available information about the problem functions, the dimension of the problem, the cost of evaluating the problem functions, and so on.

The choice of which algorithms to include in the library is complicated, in part because of the difficulty of assessing the relative merits of different algorithms that solve the same category of problems. Any comparison among algorithms inherently includes a merit function to be maximized, and for optimization problems there is no universal agreement as to what considerations should be included in the merit function. Even for relatively straightforward problems in linear algebra, such as solving a linear system, it is possible to create examples for which the generally accepted "best" algorithms are non-optimal. For the more difficult problem of nonstiff ordinary differential equations, Shampine, Watts and Davenport (1976) provide a lengthy and detailed discussion of criteria for measuring the performance of ODE codes, and repeatedly emphasize that the decision about which code is "best" depends on the problem to be solved and on the qualities that are most important to the particular user. A similar statement can be made concerning optimization problems; the determination of the "best"

algorithm for most problems depends on a complicated analysis of the user's needs, as well as on problem-dependent information, much of which may be unknown. Accordingly, the present optimization library is designed to include implementations of the algorithms that have performed most successfully in extensive testing on problems that are believed to be typical of those encountered in a general environment (where "success" is measured by some reasonable criterion; see Gill and Murray, 1974b, Chapter 9, for additional comments).

2.3 Algorithm standards

All theoretical algorithms to be included in the library are required to be reliable and robust (see Cody, 1975, for definitions of these terms). Any algorithm to be implemented for general use should be able to test whether the assumptions upon which it is based seem to be satisfied, and to deal satisfactorily with situations not satisfying those assumptions. As an illustration of this standard, a widely discussed issue in unconstrained optimization has been the procedure that should be followed in a Newton-type algorithm when the Hessian matrix at the current point is not positive definite (see Murray, 1972, Chapter 4, for a detailed discussion). Any Newton-type routine in the library must be able to detect an indefinite Hessian and carry on in a reasonable and stable manner. Similar safeguards should be included in the definition of every theoretical algorithm proposed for the library.

An additional algorithmic consideration is that many alternative computational procedures exist which, although theoretically equivalent, display widely differing numerical behavior. The distance between the statement of a mathematical process and its implementation in finite precision

has been emphasized by several authors (see, for example, Smith et al., 1974a). In the present library, it is considered essential to verify that every procedure to be carried out has satisfactory numerical properties. For example, in all library routines for quasi-Newton methods for unconstrained minimization, the Hessian approximation is represented in terms of its Cholesky factorization, to allow strict control over whether the matrix is numerically positive definite (Gill and Murray, 1972); this facility does not exist if an approximation to the inverse Hessian is maintained.

2.4 Structure of implementations

The structure of the library has been chosen to reflect as much as possible the ideas of good design that are sometimes called "structured programming" or "step-wise" algorithm design (see Dijkstra, 1972). In essence, an abstract algorithm is decomposed into steps, each of which is successively sub-divided until the nature of the algorithm is revealed by a highly structured combination of concise and well-defined computations. After this analysis has been carried out for the theoretical algorithm, the computer implementation is designed to display the same structure. In our view, such a systematic analysis of algorithms is essential to obtain good numerical software, and should be completed before a single line of code is written.

The process of dividing an algorithm into components of decreasing complexity has been called many names — for example, the term "modularization" is sometimes applied in this way. Unfortunately, "modularization" is often used to imply an analysis of the components of an "algorithm"

that occurs after a computer implementation already exists – the reverse of the procedure described above (see Minkoff, 1975). Therefore, we wish to avoid the term "modularization", because an after-the-fact analysis of computer programs is contrary to the spirit of the desired approach. We prefer instead to define the algorithms and related implementations in the present library as "usefully structured" (Dijkstra, 1972, p. 5).

One of the benefits of this kind of analysis is that it is possible to identify closely related or identical sub-steps in a variety of algorithms; hence, the same subroutine can be used in numerous distinct contexts. A further advantage is that this approach may reveal structural similarities in algorithms that were considered initially to be quite different, because their underlying nature was disguised by notation or computational detail (see the examples given by Lawson, 1976).

Well-structured programs are vital if extensions, modifications, or generalizations are to occur without undue complication. Since active research is continuing in all areas of optimization, it is probable that improvements will be made to various aspects of every algorithm. If the structure of the underlying algorithms has been accurately reflected in the computer implementations, it should be easy to extend the algorithmic capabilities of the library. For example, a new quasi-Newton update formula can be included as a minor alteration of an existing quasi-Newton routine.

Finally, careful structuring facilitates the changes that might result from a more powerful programming language, or from a generalized data structure provision. For example, if it became possible to store symmetric matrices in minimal storage with efficient double indexing, many schemes

devised to avoid wasting storage under the current limitations of Fortran (see Section 3.2) would become unnecessary; the relevant routines should be designed so that such a change would have a localized and well-understood effect. Good structuring can avoid the serious danger of "thinking in Fortran" rather than perceiving the overall nature of an algorithm; a concern with the details of Fortran often leads to primitive programming practices, which then perpetuate the restrictions of present software tools and make it virtually impossible to take advantage of any improvements.

2.5 Structure vs program efficiency

A further principle of the library design is that, if a choice is necessary, good structure rather than "efficiency" will be emphasized. The term "efficiency" is taken here to apply only to programming considerations; the efficiency of the numerical procedures to be followed is included in the factors that determine the choice of algorithm.

The first point to be stressed is that the efficiency of a software design should be viewed within the full context of the total amount of effort required to develop the routines as well as solve the given problems, and not (as it often is) defined solely in terms of one-time execution speed. The efficiency of a library design involves not only the cost of an isolated execution of a few subroutines, but also the tasks of verification, testing, and modification. It is widely accepted that the need to create a manageable and reliable set of routines should dominate a concern with run-time efficiency:

"Premature emphasis on efficiency is a big mistake which may well be the source of most programming complexity and grief .... we should strive most of all for a program that is easy to understand and almost sure to work" (Knuth, 1974, p. 294).

A second consideration is that the process of writing "efficient" Fortran code is not at all straightforward. Given that library routines are written in a portable subset of ANSI Fortran, to be run with a minimum of change on different computers, it is impossible in many cases to write code that is efficient on all machines. The surprising examples cited by Parlett and Wang (1975) offer a decisive illustration that the effort to write "efficient" Fortran requires a detailed _a priori_ knowledge of the compiler and machine to be used. In fact, someone who has made an attempt to write "optimized" Fortran may find that the code written by a less sophisticated programmer actually runs faster.

3. Discussion of Design Decisions

3.1 Communication by parameter list

A fundamental early decision concerns the mechanism by which the various library subroutines communicate information to one another and to the user. Because the routines are written in standard Fortran, the two possible means of communication are: (1) through the formal parameter list; and (2) through COMMON storage. Some previous optimization codes have chosen alternative (2) (Lasdon et al., 1976; Mylander, Holmes and McCormick, 1974; Rosen and Wagner, 1975), while others have chosen (1) (Fletcher, 1972); however, the reasons for the decision were not specified in any of these cases.

For the present library, it was decided to restrict communication among routines entirely to the parameter list mechanism. COMMON storage is sometimes viewed as the Fortran equivalent of global variables in a block-structured language, but the use of COMMON suffers from many defects not present with the facilities of true global variables. We shall give three reasons for the total omission of the use of COMMON by library routines:

(1) With COMMON, it is impossible to include arrays of variable dimension. In most optimization applications, the lengths of the program arrays depend conceptually on the dimension of the problem – for example, the vector of independent variables is represented as a floating-point vector of length n . Large fixed-length arrays in COMMON, of sufficient size to solve most problems, would result in a serious waste of storage, particularly since two-dimensional arrays are often involved. Although the user could be asked to re-dimension all arrays to correspond to the size of his problem, such a requirement makes an unreasonable demand on

the user, and also rules out the possibility for the library routines to be pre-compiled.

(2) It has been our experience that often the user has already set up the COMMON area within his program before he is ready to use the library routines. In such cases, the necessity to alter COMMON storage throughout the program is extremely inconvenient, and frequently leads to obscure and complicated programming errors.

(3) The final justification for communication solely by parameter list, which underlies the two previous reasons, is that this restriction is a consequence of the ideas of good structure discussed in Section 2.4. The ability to understand a given subroutine is greatly enhanced when all the quantities upon which its execution depends are in the parameter list. In addition, the limitation to parameter list communication means that the internal logic of each subroutine can be made independent of the external environment into which it is to be placed. Since all information is passed through strictly controlled channels, there is no worry that the subroutine will produce unwanted side-effects (for example, by overwriting a portion of COMMON because of an accidental coincidence of variable names).

The dimensions of any variable-length array are always included in the parameter list along with the array name. Fortran allows an array parameter, say "B", to be declared within a subroutine by "DIMENSION B(1)," regardless of the true length of the actual parameter; but such a short cut is not allowed by some compilers — for example, the WATFIV compiler (see Cress et al., 1970) — and it may also disguise the nature of the arrays so declared. All library routines therefore include the correct dimensions as parameters in the calling sequence.

3.2 Choice of data structures

The careful selection of data structures is an essential part of the design of computer programs. A good high-level programming language allows the programmer to define a data structure so that its manipulation within the program resembles its original role in the theoretical algorithm. Ideally, the programmer should not be required to transform convenient conceptual data structures to an inconvenient machine-based representation; but the highly limited capabilities of Fortran necessarily cause a departure from this ideal.

In numerical applications such as optimization, vectors and matrices are the primary conceptual data structures; it might seem straightforward to transform these into one- and two-dimensional Fortran arrays, respectively. Such a transformation is convenient in some cases — for example, the vector of independent variables can be represented in an obvious way as a one-dimensional floating-point array. In other instances, however, the choice may be complicated by programming considerations.

Some Fortran programmers represent matrices as one-dimensional arrays, because true double indexing does not exist in Fortran. The element A(I,J) is obtained as element [(J-1)*(declared row dimension of A) + I] of the block of contiguous storage locations representing the matrix A, and thus there would seem to be an inherent overhead cost associated with accessing two-dimensional arrays. The additional cost of processing a two-dimensional array compared to a one-dimensional array depends on several complicated factors, which include the hardware features, the compiler, the array dimensions, the loop structure of the code, and the particular indexing strategy chosen.

To obtain some estimates of the relative execution times for theoretically equivalent manipulations of one- and two-dimensional arrays, numerous experiments were run on an IBM 370/168 computer, with the Fortran H compiler, opt = 0 (unoptimized) and opt = 2 (optimized). As expected, the largest increase in running time was displayed for an unoptimized double do-loop that initializes all elements of a two-dimensional array to a constant, since the loop and index processing then dominate the trivial calculation within the loop; the worst case (a 50% increase over the one-dimensional loop) occurred when the index of the outer loop was 50 times larger than for the inner, because of the repeated overhead in setting up the inner loop. With the opt = 0 compiler, there were a few other instances when the combination of unoptimized code and an unfortunate choice of loop structure led to a noticeable (10 - 15%) increase in execution time for a particular section of code involving the two-dimensional array. However, in all remaining cases, which involved a selection of typical array calculations, the difference in execution time for processing the two-dimensional array was negligible (often less than the fluctuation in timing estimates), especially with the optimizing compiler. This result confirms the inability to predict the relative efficiencies of sections of Fortran code without knowledge of the compiler and the particular data to be processed.

In view of this uncertainty, it seems sensible to use the most straightforward data structures, especially since the representation of a matrix as a one-dimensional array often causes confusion and complexity, both for the user and the library programmers. In the library least-squares routines, for example, if the user were required to store the components of the Jacobian matrix in a one-dimensional array according to some non-trivial

indexing strategy, there would be a high probability that the information would be jumbled. The discarding of double indexing is in most cases an excessive concession to the limitations of current Fortran in quest of the dubious possibility of marginally faster execution speed.

There are instances, however, when a theoretical two-dimensional structure is represented in the library routines by one-dimensional arrays. A frequent question in implementations of linear algebraic techniques concerns a suitable program representation of a symmetric matrix (similar considerations also apply for a triangular matrix). Conceptually, a symmetric matrix has a two-dimensional character; but if it is represented by a two-dimensional array, almost half the storage is "wasted" in the sense that duplicated information is stored that could be found by a simple interchange of indices.

If unlimited storage were available, it would be feasible to retain the natural two-dimensional representation for a symmetric matrix. In most applications, however, it is unreasonable to use $O\left(\frac{n^2}{2}\right)$ storage locations to contain easily accessible duplicated information, and thus symmetric matrices are often represented by a combination of one-dimensional arrays. For example, in the modified Newton routines for unconstrained optimization, the symmetric Hessian matrix is stored in two one-dimensional arrays – the diagonal elements in a vector of length n, and the off-diagonal elements in a vector of length $n(n-1)/2$ (stored by rows). This representation is extremely convenient, because the (possibly) modified Hessian matrix $\bar{G}$ is factorized into the form

$$\bar{G} = LDL^T ,$$

where D is a diagonal matrix, and L is a unit lower triangular matrix. Hence, the vectors that represent the original matrix can be overwritten with the vectors that represent the Cholesky factors. The resulting data structure is not entirely straightforward, since a non-trivial transformation of indices is necessary to obtain the general off-diagonal element $\{\bar{G}_{ij}\}$ or $\{L_{ij}\}$ from the one-dimensional array. However, the effect of this data structure is confined to a few routines, and the benefit of significant savings in storage is considered to outweigh the localized loss of clarity.

3.3 Local arrays

Optimization algorithms typically involve numerous temporary arrays whose dimensions are related to the dimension of the problem to be solved. The issue to be considered in this section is the mechanism by which storage is allocated to such local arrays. In Algol, a procedure may contain declarations of variable-length local arrays, where the dimensions depend on parameters of the procedure; but in Fortran, locally declared arrays must be of fixed dimension. The choices for providing temporary arrays in Fortran subroutines thus are:

(1) declare the arrays locally to each subroutine, with fixed dimensions large enough for most anticipated problems. The user must modify the source text of every such routine if the declared dimensions are insufficient;

(2) declare the space for any local arrays outside the subroutine. The vectors that contain this space are then included in the parameter list of the subroutines that require temporary arrays, along with the appropriate

integers that define the dimensions (see Section 3.1).

Alternative (1) has the same disadvantages discussed in Section 3.1 with respect to the use of COMMON storage. There is a significant waste of storage if large arrays of fixed dimension are declared in many subroutines, especially since the locally declared storage cannot be shared by several subroutines; and it is inconvenient to require the user to modify local array declarations throughout the library routines. For some types of numerical problems, the waste of storage due to fixed-size locally declared arrays may not be serious enough to justify consideration of alternatives. For example, in the ordinary differential equation code DE (Shampine and Gordon, 1975), fixed local arrays are acceptable because the dimensions of the arrays are known in general to be of reasonable size, and the maximum amount of unused storage is relatively insignificant compared to the program size (since all arrays except one are one-dimensional). In optimization routines, however, large two-dimensional arrays are frequently required, and the problem of wasted storage cannot be ignored. Consequently, alternative (2) is used in the present library.

The decision to avoid declaring local arrays whose conceptual dimension is problem-dependent introduces into the computer implementations a deviation from the straightforward framework of the original algorithms, since the parameter list of a given subroutine will include not only quantities that represent its input and output, but also quantities whose role is purely to provide storage for temporary intermediate results. This disadvantage is nontheless believed to be outweighed by the increase in flexibility and the savings of storage.

3.4 Work space arrays

Library subroutines include all variable-size temporary arrays and their dimensions in the calling sequence (see Section 3.3). If such local arrays are declared individually, the growth in the length of some parameter lists can become extreme. There is no inherent objection to long parameter lists, if each parameter has a significant role in the definition of the input or output of the subroutine. However, this qualification does not imply that there must necessarily be a separate parameter name for every variable-length temporary array, which has a purely internal significance. The EISPACK subroutine TSTURM has six temporary vectors in its calling sequence, and this number may not seem excessive; but some optimization routines have well over 20 distinct temporary arrays. The clutter of a long sequence of temporary array names will tend to distract from the insight into the nature of a subroutine that the elements of the formal parameter list should give.

An alternative to including individual work-space parameters in the calling sequences is provided by partitioning larger arrays. It is acceptable in standard Fortran to call a subroutine whose k-th formal parameter is an array name with an element of an array as the k-th actual parameter. This device is consistent because an array is completely defined in Fortran by the address of its first element and an indexing rule. Thus, an array parameter is defined within a subroutine as the set of contiguous memory locations whose first element is at the address specified by the corresponding actual parameter. This convention for specifying an array means that one large array can be partitioned into several smaller sub-arrays. The location of the first element of each partition can then

serve in a subroutine call as the actual parameter corresponding to an array of appropriate length, since the execution of the subroutine generates accesses only to elements of the desired sub-vector.

The flexibility provided by work space arrays can be exploited if the library routines are developed and extended. For example, if a new method is devised for solving a given optimization problem, which requires more intermediate calculations than the original, the form of the existing library module can be retained while implementing the new algorithm, by simply using additional elements of the already present work space arrays.

The technique of partitioning one array into many can lead to a lack of clarity in the resulting code, since the names of the conceptual arrays of the original algorithm do not appear. The library routines apply this solution only with caution, and in every instance the code is surrounded by detailed comments explaining the partitioning.

The difficulty may eventually be resolved by the use of pre-processors that make intelligent text substitutions. With such a pre-processor, all textual references to the desired local array name could be replaced before compilation by references to suitable elements of the work space array, i.e., if the vector P , of length N , began in location N+1 of the vector W , references to "P(I)" could be replaced by "W(N+I)". Alternatively, the recent "MAP" proposal to modify the ANSI Fortran standards (IFIP Working Group 2.5, 1976) provides a similar means of gaining such flexibility.

3.5 Interface with the user

In designing library software, it must be decided explicitly how much of the theory and complexity of a numerical algorithm the user needs to understand before making use of the associated routines. The controversial statement that a possible purpose of mathematical software is to "relieve [the user]... of any need to think" (Davis and Rabinowitz, 1967, p.315, in the context of automatic quadrature) is confirmed as the ideal of many users by other authors (see Dixon, 1974; Kahaner, 1971; Kuki, 1971), and by the experience of anyone who has ever acted as a consultant concerning numerical problems. Because many users found it burdensome to have to call several (even two) subroutines that represent the conceptual steps of an eigensystem computation, the EISPACK system now provides driver routines which call all the related subroutines required for a particular problem. Many people refuse to use routines that require more than a marginal effort to understand, and wish to know nothing of the possible subtleties involved in solving the problem. A consequence of this attitude among users is that many software designers try to keep all calling sequences as short as possible (see Kahaner, 1971; Shampine and Gordon, 1975), because users are reluctant to try subroutines with long calling sequences and may even prefer an inferior routine with a shorter parameter list. This situation leads to the feeling that software designers should try to remove any hint of difficulty for the user.

An opposing view is that the user should be forced to become at least minimally educated and informed about the numerical procedures to be carried out by a library routine. It can be argued that, by catering to the user's

desire for simplicity, the authors of mathematical software are performing a disservice if the form of the software gives the false impression that a problem is easy and straightforward to solve. In particular, an attempt to enforce short parameter lists to please users may compromise the performance of an optimization algorithm, because the adjustment of selected tolerances and convergence criteria can have a significant influence on the efficiency, and even the success, of a given method applied to an optimization problem. The most suitable values for these parameters typically depend on knowledge that only the user has, which would seem to imply that for optimum performance the library routines should contain all such parameters in the calling sequence.

The dilemma in library design is that the creation of superb routines will be unproductive if no one uses them because of their excessive complexity; on the other hand, overzealous simplification to satisfy some users may lead to the loss of highly desirable flexibility for others. Rather than compromise between flexibility and ease of use, the present library has been designed to satisfy both objectives by providing two routines — a general and a "simple" — for each purpose. A similar idea is seen in the EISPACK collection (Smith et al., 1974b) and in the ODE routines described by Shampine and Gordon (1975). If a particular parameter can be chosen automatically (given that certain assumptions are satisfied), the uncertain or uninterested user may choose a "simple" or "easy-to-use" routine, for which only a minimum number of parameters need to be specified. However, the possibility is retained that a well-informed user can enhance the performance of an algorithm by exploiting his knowledge of the problem

to select the appropriate parameters of the general routine. (All users are not required to think, but the knowledgeable user is at least given the opportunity of thinking.)

Although this scheme would appear to double the amount of code, the easy-to-use routines are simply "front ends", which call the more general routines; thus, the increase in code is not significant. Similarly, the amount of documentation is not doubled, since one of the objectives in providing easy-to-use programs is to reduce the amount of information that the user must assimilate before he can solve his problem.

It should be noted that a development in programming language control structures may soon make the concern about long calling sequences unnecessary. This development is the "keyword" parameter communication mechanism, described in Hardgrave (1976) and Zahn (1975). The idea is that the actual parameters in a subroutine call are not specified by position, but rather by keyword. The user thus needs to include in the actual calling sequence only the parameters that he wishes to specify, and the others are set by default, so that a flexible routine can also serve as "easy-to-use". In fact, the keyword parameter mechanism would allow a range of alternatives, from specifying no optional parameters to a full calling sequence.

3.6 Service routines

The "simple" routines described in Section 3.5 have been provided in response to user demand for short calling sequences. However, the remaining parameters, which are masked from the user of the simple routine, must nonetheless be chosen. There are two ways to automate the process of parameter selection.

First, a reasonable *a priori* choice for some parameters is possible if the problem to be solved satisfies a specified set of assumptions. For example, the choice of termination criteria depends on the problem's scaling; if the problem is assumed to be "well scaled" in a given sense, then the convergence criteria can be based on the machine precision. Similarly, a "sensible" value of the parameter that controls termination of the step-length algorithm can be selected based on wide computational experience. The parameters chosen in this way are considered to be "safe", in the sense that the assumed properties of the model problem upon which their selection is based will correspond fairly well to those of most problems to be solved.

However, certain other parameters are considered to be too sensitive to the given problem to allow an *a priori* choice. Consequently, a second, more complicated, service to the user is provided by library routines that automate problem-dependent decisions. The procedures for parameter selection that depend on properties of the specific problem functions can usually be automated to some extent, and a carefully designed automatic routine should give a better result than the random guess that might well come from an uninformed user. For example, to solve an unconstrained optimization problem with a quasi-Newton method based on finite-difference approximations to gradients, a finite-difference interval must be provided for each variable. The values of the difference intervals are often critical in the performance of the algorithm, but many users have no idea how to choose this set of intervals. It has therefore been considered worthwhile to provide a library routine to determine an initial sensible set of finite-difference intervals; this routine (FRMDEL; see Gill et al., 1975c) calls the user-provided function, and chooses the intervals based on examining the magnitudes of

cancellation and truncation errors in the derivative approximation. There is no guaranteed technique for difference-interval selection, because the values obtained at one point may be unreasonable at a different point if the problem-function's behavior alters drastically. If, however, the scaling of the problem does not change too much as the minimization proceeds (a situation that usually holds in practice), the initial set of intervals remains a good choice. Such "service" routines in the library represent the view that it is better to automate parameter selection in a sensible way if the user cannot provide the information, rather than incur the risk of significantly reduced efficiency.

Other library subroutines provide a further service to the user — a consistency check. One of the most common difficulties in use of optimization routines is that the user's subroutines incorrectly evaluate the relevant partial derivatives. Because exact gradient information normally enhances efficiency in all areas of optimization, the user should be encouraged to provide analytic derivatives whenever possible. However, mistakes in the computation of derivatives can result in serious and obscure run-time errors, as well as complaints that the library routines are incorrect. Consequently, library routines are provided to perform an elementary check on the user-supplied gradients. Because the checking procedure will not be cost-effective if it requires too many evaluations of user-supplied functions, the present library contains subroutines that compute finite-difference approximations to two projections of first and/or second derivatives (it would usually be considered too expensive to compute finite differences along all the coordinate directions). The approximated values are compared with the supposedly exact values, and should display

reasonable agreement. If they do not, then either the user has made an error or the problem is very badly scaled — in both cases, the user should take corrective action. Of course, as with the routine to select the finite-difference intervals, such a procedure can not guarantee correctness; even if the quantities display some agreement, it may simply be due to a fortuitous starting point. However, the simple process described above detects the vast majority of user errors in computing partial derivatives.

3.7 Auxiliary routines

An "auxiliary routine" is a subroutine or function that is not called directly by the user, but only by other library routines.

3.7.1 Subroutines vs in-line code

The process of analysis mentioned in Section 2.4 leads to the description of an algorithm as a structured and logically coherent collection of computations, which must be transformed into Fortran code. The topic to be discussed here is the process of deciding whether a particular computation or set of computations should be implemented in the library as a separate subroutine, or as part of the code body of another routine.

Some people favor in-line code as a general practice, based on the argument that efficiency (in the restricted sense of run-time speed) is degraded by the overhead cost associated with a large number of subroutine calls. In order to achieve maximum clarity, however, the division of the library into subroutines would be based directly on the structure of the theoretical algorithms, with a one-to-one relationship between the conceptual steps of an algorithm and those of the corresponding implementation.

This latter policy does not significantly decrease efficiency in most cases because the execution time for subroutines tends to be dominated by the actual computations (arithmetic operations and memory accesses), and the time required to set up the calling mechanism is negligible by comparison. However, in some instances there is a marginal balance between the structural benefits of creating an auxiliary routine and the consequent loss of run-time speed.

The following experiments were designed to investigate the execution time associated with a subroutine call. First, code was written to compute in-line the scalar product of the vectors A and B, of length N , and assign the value to a variable. A subroutine DOT1 (N, A, B, DPROD) was written to perform the same calculation, and assign the result to the variable DPROD. On an IBM 370/168, with the Fortran H compiler, opt = 0 and opt = 2 , the in-line code and calls to DOT1 were executed 3000, 4000, and 5000 times, for N = 1,2,10,50,75,100, with each run repeated twice to observe the fluctuation in timing estimates.

Table 1 gives the average ratio of the execution times of DOT1 to those of the in-line code.

N	1	2	10	50	75	100
opt = 0	2.68	2.09	1.22	1.00	1.00	1.00
opt = 2	4.72	3.61	1.77	1.13	1.07	1.05

Table 1

Second, in-line code and a subroutine DOT2(N,A,B,DPROD,X1,X2,...,X12) were written, to compute the scalar product of the vectors A and B as before, and also to assign the values of some simple arithmetic expressions to the scalar variables X1,X2,...,X12. The in-line code and calls to DOT2 were executed repeatedly, exactly as in the previous case. Table 2 gives the average ratio of execution times for DOT2 to those of the equivalent in-line code.

N	1	2	10	50	75	100
opt = 0	1.89	1.78	1.34	1.03	1.00	1.00
opt = 2	2.10	2.04	1.68	1.21	1.14	1.10

Table 2

It should be stressed that the values in Tables 1 and 2 were computed in a multi-programming environment, and are inevitably subject to a lack of high precision; however, the ratios for the various cases fluctuated only in the hundredths place, so that they are reasonably accurate for this machine. One must always be cautious in drawing general conclusions from the results of a particular timing experiment, which depend on the compiler as well as the machine. Nonetheless, the results correspond to *a priori* expectations, and allow at least a rough measure of the desired phenomenon.

For these examples, the increase in execution time attributable to a subroutine call is very substantial for small N , noticeable for moderate N,

and essentially negligible for large N. The significance of these differences depends on the role of the given calculation within the library. If the computations of the entire library were primarily a sequence of inner products for small or moderate values of N, it would probably not be worthwhile to create an auxiliary dot-product subroutine, since the repeated subroutine calls would clearly have a serious impact on execution speed. However, in nonlinear optimization most subroutines perform order (N^2) or order (N^3) computations, and the user-supplied routines are typically even more time-consuming. A scalar product represents the simplest calculation that might be isolated into a subroutine within a general optimization library, so that these experiments indicate the worst degradation in execution time that could possibly occur as the result of a decision to create an auxiliary routine. Consequently, the principle stated earlier of dividing into subroutines whenever possible to reflect the underlying algorithmic structure should cause a negligible loss of execution speed, and a great gain in clarity.

3.7.2 Flexibility of auxiliary routines

It is typical of optimization algorithms that identical, or closely related, computations must be carried out in methods to solve different problems, in alternative methods to solve the same problem, and in different steps of a single method. If in-line code, or different subroutines, for similar computations can be replaced by a call to a single, flexible auxiliary routine, the amount of source code in the library can be reduced.

The following difficulties must be considered in deciding whether to create a single auxiliary routine to carry out related but slightly different computations:

(a) the length of the calling sequence may increase;

(b) the source code body of the routine may be larger;

(c) the execution time of the routine may increase because of additional logic or computation to provide flexibility (e.g., testing of parameter values).

Concerning (a), the following experiment, similar to that described in Section 3.7.1, was designed to obtain some measure of the increase in execution time due to longer calling sequences. A subroutine DOT3(N,A,B,DPROD,X1,X2,...,X12), with 16 parameters, was written to compute the scalar product of the vectors A and B, and to assign this value to the variable DPROD. The scalar variables X1,X2,...,X12 are unaltered by DOT3, and thus the ratio of execution time for DOT3 to that of DOT1 (Section 3.7.1) should indicate the effect on execution time of 12 additional formal parameters.

Exactly as for the other experiments, the calls to DOT1 and DOT3 were executed repeatedly for various values of N. Table 3 gives the average ratio of execution times for calls to DOT3 to those of DOT1.

N	1	2	10	50	75	100
opt = 0	1.29	1.25	1.12	1.00	1.00	1.00
opt = 2	1.34	1.31	1.20	1.06	1.04	1.02

Table 3

A comparison of these results with Tables 1 and 2 indicates that the increase in execution time associated with additional parameters in the calling sequence is less significant than the increase due to calling a very simple subroutine instead of performing the calculations in-line (of course, the cautions mentioned in Section 3.7.1 about timing tests apply here as well). The overall conclusion is that long calling sequences for auxiliary routines will generally have a negligible effect on execution speed, and the difficulty suggested by (a) is not serious.

An analysis of difficulties (b) and (c) will depend on the particular case under consideration. In the present library, flexible auxiliary routines are preferred to in-line code or several similar subroutines because of the simplification of library maintenance, providing that the following conditions are satisfied:

(1) no increase in the number of calls to user-supplied routines results from the added flexibility;

(2) the increase in the source code body of the routine is less than (approximately) 10%;

(3) the increase in execution time because of the extra flexibility is at least an order of magnitude smaller than the total execution time for the routine. For example, if $O(n^3)$ operations are performed in the routine, then $O(n^2)$ operations would be tolerated to increase flexibility.

3.8 Communication with user-supplied routines

For optimization problems of the form P1 (Section 2.1), the user must supply code to evaluate the objective function, the constraint functions,

and (possibly) the appropriate derivatives. Therefore, the library routines need to contain a mechanism for obtaining from the user the information required to proceed with the computation, and it must be considered how this communication should take place.

An initial problem is the actual control structure that connects the routines of the library and the user. In most similar applications (such as quadrature or ordinary differential equations), the user supplies subroutines that define the problem functions, and these subroutines are called by the library routines; this control structure is used in the present library. An alternative approach, used by Lawson and Krogh (see Krogh, 1969), is termed "reverse communication", where the library routine returns control to the user whenever information is required. A closely related strategy is included in one aspect of the library design (see Section 4.4), but not at the outermost level.

The next question that arises is at what level the library routines should communicate with the user. Because the library subroutines will call the user-supplied routines, the latter must be defined with a fixed calling sequence. The composition of this calling sequence, and the means of interface with the library routines, pose some interesting issues, which are best illustrated by a detailed example.

Consider the case of the library subroutine QNMDER (Gill et al., 1975a), an implementation of a revised quasi-Newton method for unconstrained optimization that uses exact gradients (see Gill and Murray, 1972). The parameter list of QNMDER contains a formal parameter SFUN, a subroutine which is called within QNMDER whenever information concerning the function to be minimized or its gradient is required. It was decided that the same

subroutine should calculate both the function and gradient, since there is typically a significant overlap in computing these values.

The logic of QNMDER requires SFUN to provide different levels of function and gradient information, because the user is allowed to select either a step-length algorithm based on using only function values, or a step-length algorithm that requires both the function and gradient (see Section 4.2). The call to SFUN must therefore allow the return of the following combinations: both the function and gradient, only the function value, or only the gradient. The required flexibility is supplied by an integer parameter (IFLAG) in the calling sequence of SFUN, which is set by QNMDER before each call to SFUN. In particular, IFLAG=0 means that only the function value is to be computed, IFLAG=1 means that only the gradient vector is computed, and IFLAG=2 means that both the function and gradient should be returned. The user of QNMDER must accordingly provide a subroutine to serve as the actual parameter corresponding to SFUN, such that the correct logic is followed for each value of IFLAG.

However, it was felt that such flexibility is inappropriate for the user of the "easy-to-use" routines, who should not be required to program anything but an absolutely straightforward routine. Consequently, the "easy-to-use" routine that calls QNMDER illustrates a different relationship with the user-supplied subroutines. The "easy-to-use" subroutine always chooses the step length algorithm that requires both function and gradient values; thus IFLAG=2 in all calls to SFUN, so that IFLAG is redundant. Requiring the user of the "easy-to-use" routine to include a useless parameter in the calling sequence of his problem function subroutine introduces undesirable complications; on the other hand, the

subroutine QNMDER should not be altered simply to make it compatible with the "easy-to-use" version. Hence, a library subroutine, named SFUN2, is provided to serve as the parameter corresponding to SFUN in the "easy-to use" call of QNMDER. The subroutine SFUN2 then calls the user-supplied routine, which now must have not only a fixed calling sequence, but also a designated name, FUN2. The calling sequence of FUN2 is then of the minimal form FUN2(N,X,F,G) , where N is the number of variables, X is the vector of variables, F is the function value and G the gradient vector evaluated at X .

Similar conventions are observed throughout the library in order to communicate with the user. The existence of two levels at which the user's routines are called — directly (with fixed calling sequence) and indirectly (with fixed calling sequence and designated names) — allows both flexibility in the general case and simplicity in the "easy-to-use" case. The requirement of a designated name in the "simple" case makes it less convenient for the user to call the "easy-to-use" library routines to minimize different functions during the same run, but this sacrifice was believed to be insignificant in light of the extreme complication of providing "easy-to-use" routines without such a restriction.

3.9 Documentation

The importance of good documentation is always stressed in discussions of mathematical software (see, for example, Shampine, Watts and Davenport, 1976).

Documentation for the present library routines occurs in two forms. External documentation provides the user of an algorithm with information necessary for the efficient selection and use of the individual routines. In-line documentation appears within the source test of the program and is intended to enable programmers who may not have been involved in the original design to modify the routine in light of any new developments.

As already explained in Section 2.2, the structure of optimization problems requires that the library contain a variety of routines. The user must utilize his knowledge of the problem (e.g., the objective function is a sum of squares), and the amount and type of available information concerning the problem functions (e.g., analytic first derivatives can be computed) to select the appropriate routine. Unfortunately, to the inexperienced user it may appear that there are a bewildering number of apparently similar routines. In order to assist the user in selecting the appropriate routine, algorithm selection charts are provided with the library; this type of decision procedure occurs in program libraries for solving various sorts of problems (for example, the EISPACK collection). The selection chart illustrated in Figure 1 concerns the unconstrained optimization of a function $F(x)$ whose vector of first derivatives is $g(x)$ and whose matrix of second derivatives is $G(x)$. The user must decide whether to choose a routine that computes $F(x)$, or $F(x)$ and $g(x)$, or $F(x)$, $g(x)$ and $G(x)$. The basic principle is to utilize as much derivative information as can be efficiently computed, because methods using the higher derivatives are more robust and give more information about the quality of the solution obtained. Each path through the chart leads to the name of the subroutines that should be used.

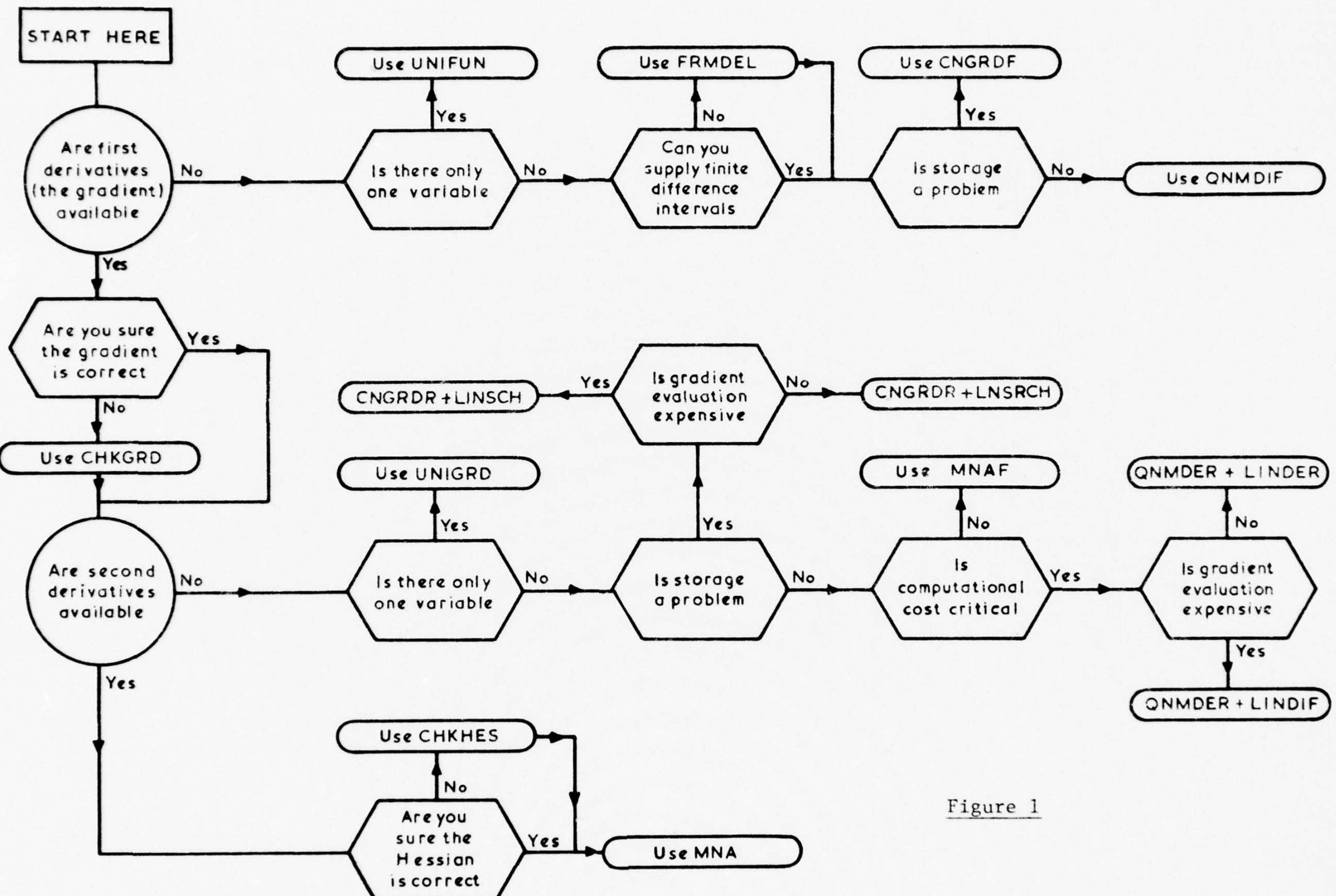

Figure 1

Each library subroutine is described in detail by an individual report (see, for example, Gill et al., 1975a), in which an effort has been made to provide an explanation of the theoretical algorithms as well as the computer implementations. The in-line documentation includes a block of explanatory comments at the beginning of every subroutine, containing a discussion of the numerical procedure, references, and a description of each parameter. In addition, comments surround every non-obvious block of code.

The need to extend or modify library routines is so prevelant that it must be included as a consideration from the outset. A significant aid to extension (and portability) is the observance of rigid textual and programming conventions concerning declarations, the ordering of formal parameters, indentation, specification of comments, placement of machine- or precision-dependent quantities, etc. Such policies also allow a text editor to make straightforward changes to alter the code for various machines and precision ranges. The rules of indentation allow a visual perception of program structure, and assist modification by indicating the control block level of every statement. The appearance of a library program should reflect the same principles of clear structure that guide the analysis of the methods and creation of the implementations.

4. A Detailed Illustration — Implementation of a Step-Length Algorithm

4.1 Need for a step-length algorithm

A step-length algorithm can be defined in general terms as a procedure that solves the following problem: given an initial point, x , and a direction, p , find a positive scalar step α along p , such that at $x + \alpha p$ certain conditions are satisfied with respect to those conditions at x . In all the cases to be considered here, the desired step can be viewed as an approximation in some sense to α^* , a local minimum of a specified function, say $\phi(x)$, along the given direction.

A step-length algorithm is involved as a sub-step in almost all unconstrained optimization algorithms. This point is emphasized because there exists a misunderstanding that a step-length algorithm, or "line search", is equivalent to a procedure to locate a highly accurate estimate of α^* for the given ϕ . For example, an advantage of Davidon's "optimally conditioned" quasi-Newton algorithm (Davidon, 1975) is said to be that it requires "no line searches" (Hillstrom, 1976; Nazareth, 1976). However, any method that does not alter the search direction during an iteration, yet requires a decrease in some $\phi(x)$ at each iteration, must contain a step-length algorithm; such a method could be described as "without line searches" only if it required one function/gradient evaluation per iteration without exception.

A step-length algorithm appears not only in methods for general unconstrained optimization, but in many other applications as well. In linearly constrained problems, the step length is frequently chosen

either to satisfy some termination criteria with respect to the objective function, or as the maximum feasible step. For nonlinearly constrained problems, the termination of the step-length algorithm may be based on some modified function that involves the original objective and constraint functions. The step-length algorithms in the library should, therefore, be implemented with the intention that they will be required in widely varying contexts.

4.2 Choice of a step-length algorithm

In this section we note the results in the library design of a particular decision to retain flexibility.

Good step-length algorithms are typically based on safeguarded parabolic or cubic approximations to the behavior of the given function along the search direction (see Gill and Murray, 1974a); the cubic procedure utilizes function and gradient values at every trial point, while the parabolic procedure uses only function values except possibly at the initial point. The choice of a parabolic or cubic method is sometimes straightforward. For example, within a Newton-type algorithm based on exact or differenced second derivatives, the slightly more robust cubic step-length algorithm is always used, since the cost of evaluating the gradient at every trial point in addition to the function is assumed to be reasonable. On the other hand, within an algorithm based on finite-difference approximations to first derivatives, the parabolic algorithm is chosen because of the excessive cost of the n function evaluations required to estimate the gradient as the step-length algorithm proceeds.

However, with a quasi-Newton method that uses analytic gradients, the best choice of step-length algorithm ("best" in the sense of "with fastest execution time") depends on the expense involved in computation of the gradient vector relative to a calculation of the function value, since the parabolic method will normally lead to faster execution time if calculation of the gradient is significantly more expensive than evaluation of the function. When developing the library routines for this case, it was felt that the choice of step-length algorithm should not be made *a priori* for the knowledgeable user (in contrast to the "easy-to-use" case; see Section 3.6).

The inclusion of this flexibility has several implications with regard to the programming detail within the relevant routines. First, the name of a step-length subroutine must be included as a formal parameter in the calling sequence of the library subroutines within which the choice is permitted. Second, because either method can be chosen, the calling sequences of the subroutines that carry out both the parabolic and cubic procedures must be the same. Finally, special logic needs to be included in the calling subroutines to test which method was selected, so that the necessary gradients can be computed upon termination of each line search if the parabolic method was used. Throughout the library, there are similar instances where non-obvious consequences follow an apparently simple design decision.

4.3 Provision of user-defined parameters

The calling sequences of several general optimization subroutines include two parameters that define in part the termination criteria of

the step-length algorithm. Here we consider why these two specific levels of flexibility have been provided.

The first parameter, η (ETA), $0 \leq \eta < 1$, controls the accuracy to which the step-length procedure locates a local minimum of the given function along the search direction. Let ϕ denote the function to be minimized, and let $\bar{g}$ denote its projected gradient along the search direction. For the cubic case, the step-length algorithm normally terminates with α if:

$$|\bar{g}(x+\alpha p)| < \eta \; |\bar{g}(x)| \quad .$$

For the quadratic case, the algorithm normally terminates when a local minimum of ϕ along p is known to be contained in the interval [a,b] and

$$\left| \frac{\phi(x+\alpha p) - \phi(x+ap)}{\alpha - a} \right| < \eta \; |\bar{g}(x)| \quad ,$$

i.e., when the linearized approximation to the projected gradient at $x+\alpha p$ satisfies the same test as $\bar{g}(x+\alpha p)$ in the cubic case.

The computation time required to solve a given problem will vary with the choice of η ; reducing η tends to decrease the number of iterations, but increase the number of evaluations of the problem functions, with the reverse effect from increasing η . The "best" value of η (that allows the problem to be solved in the shortest time) depends on the problem. The variation in the optimal η is small for most problems, and there is usually little decrease in efficiency if η is set to an averaged optimal value based on extensive computational

experimentation (as in the easy-to-use routines). However, on some categories of problems, the optimal value of η may differ significantly from this average. In such instances, a considerable improvement results by allowing the user the freedom to adjust η to suit the problem, and consequently the parameter ETA is retained in the calling sequence.

The second parameter to be considered, which strongly influences the performance of the step-length algorithm, is λ (STEPMX), an upper bound on the norm of the step to be taken during any iteration (i.e., the final α must satisfy $\|\alpha p\| \leq \lambda$). There are several reasons for including this parameter in the step-length algorithm:

(1) to prevent overflow in the user-supplied problem function subroutine;

(2) to increase efficiency by evaluating the problem functions only at "sensible" values of x;

(3) to prevent the step-length algorithm from returning an inordinately large step because no smaller step satisfies the relevant convergence criteria. It is not always appreciated that, even if a function is unimodal in E^n, there may exist directions along which it is monotonically decreasing;

(4) to attempt to force convergence to the local minimum nearest to the initial estimate.

Because the user should be in the best position to specify a good choice of λ, this parameter is also included in the outermost calling sequence. For the easy-to-use routines, however, λ is automatically set at $\max\{10, \|x^{(0)}\|\}$, where $x^{(0)}$ is the initial value of x. This

choice of λ is dependent on the problem scaling, and hence the performance of the easy-to-use routines may be degraded on badly scaled problems.

4.4 Control structure for a step-length algorithm

A step-length algorithm should be implemented with the knowledge that it will appear almost universally throughout the library (see Section 4.1). Conceptually, the execution of a given step-length procedure is identical in any context, and one might hope to implement the parabolic and cubic step-length algorithms as invariant subroutines; this section considers the complications of such an implementation, which initially seems straightforward.

One significant element of a step-length algorithm involves the retention and communication of information. In the unconstrained case, the values of the objective function and its gradient vector at the best point must be stored as the linear search proceeds, to be communicated to the outer iteration upon termination. Because the best point found is not necessarily the most recent point at which the function was evaluated, the step-length routine also requires storage to contain the most recent function and gradient values. In this context, the information to be retained during execution of each linear search involves only the function upon which the termination criteria of the step-length algorithm are based (hereinafter referred to as the "intermediate function"). However, in other instances the information associated with a trial point is not restricted to the value and gradient of the intermediate function. For

example, in the nonlinear least squares case, the Jacobian matrix should be retained; in nonlinearly constrained problems, the data to be stored include at the least the values and gradients of the original objective and constraint functions. It is difficult to imagine an invariant step-length subroutine that would be suitable for all situations, without resort to programming contortions involving complex storage arrangements. The necessity of communicating varying sets of information eliminates the desirability of developing a universal step-length module.

However, it is possible to isolate a second element of a step-length algorithm that is not dependent on the information connected with each point; this aspect is the calculation of the next trial point, which is based on a safeguarded polynomial approximation procedure, and utilizes quantities relevant only to the intermediate function. Consequently, the library contains fixed subroutines to implement this second, problem-independent, portion of the step-length algorithms.

The fixed subroutines are NEWPTC (Gill et al., 1976b) and NEWPTQ (Gill et al., 1976a), which carry out a single iteration of a step-length procedure; NEWPTC computes the next trial point based on safeguarded cubic approximation, while NEWPTQ uses safeguarded quadratic approximation. The step-length procedure for a given application involves repeated calls to one or the other of these modules by an outer controlling subroutine, which is tailored to the particular context; the outer routine retains the necessary information as the iteration proceeds, and provides NEWPTC and NEWPTQ with the value (and possibly projected gradient) of the intermediate function at each new point. In this way, all calls to user-supplied or library

routines that define the problem functions occur outside NEWPTC and NEWPTQ. Such a control structure was suggested by the "reverse communication" mechanism of Lawson and Krogh, displayed in the differential equation solver DVDQ (see Krogh, 1969).

The displeasing feature of the chosen arrangement is that a parameter of NEWPTC and NEWPTQ must indicate whether the current call is the first within an iteration, and whether certain other special conditions hold. The concept of control defined by a co-routine (see Dahl and Hoare, 1972) would be a more accurate reflection of the nature of the step-length algorithm, but does not exist in current Fortran.

The overwhelming advantage of the given design is the separation of the two distinct tasks of obtaining the next trial point and maintaining the associated information. The implementation of the inner logic of the step-length algorithms — safeguarded cubic or parabolic approximation — thus is independent of the dimensionality and data structures of the particular optimization problem within which a step-length procedure is required.

5. Conclusion

The design principles underlying an optimization program library have been presented, in order to indicate the kinds of considerations and compromises involved in such a major software project. It is impossible to achieve a "perfect" library, but at least every decision in the present system has been made based on an analysis of the same set of aims. There will inevitably be unforeseeable future developments and modifications; in fact, many changes have already occurred since the first routines were produced in 1972. However, it is hoped that the care put into the library design will continue to allow consistent and well-structured extensions.

6. Appendix (an example of documentation)

SUBROUTINE UCNDQ1

An easy-to-use quasi-Newton algorithm to find the unconstrained minimum of a function of N variables using function values only.

Philip E. Gill, Walter Murray, Susan M. Picken, Margaret H. Wright and Hazel M. Barber

Ref. No. E4/10/0/Fortran/11/75

NPL ALGORITHMS LIBRARY Fortran SUBROUTINE UCNDQ1

1. Purpose

To minimize a function F(x1,x2,...,xN) of the N independent variables x1,x2,...,xN, using a quasi-Newton method. The subroutine UCNDQ1 is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work if the derivatives have occasional discontinuities).

The user must supply an initial estimate of the position of a minimum and a subroutine which will calculate F(x) at any point x = (x1,x2,...,xN). It is essential that the subroutine supplied by the user has the name FUN1 and has the correct specification.

SUBROUTINE UCNDQ1 is an easy-to-use version of SUBROUTINE QNMDIF, NPL ALGORITHMS LIBRARY Ref. No. E4/03/F and is intended for users who have absolutely no knowledge of either optimization or the

behavior of their problem. It must be emphasised that a minimization can always be performed more efficiently if the SUBROUTINE QNMDIF is used and the parameters are chosen to suit the problem being solved.

The user should also be aware of the alternatives to SUBROUTINE UCNDQ1. The SUBROUTINE UCFDQ2, NPL ALGORITHMS LIBRARY Ref. No. E4/09/F, is available which requires the user to calculate function and gradient values. There is also SUBROUTINE UCFDN2, NPL ALGORITHMS LIBRARY Ref. No. E4/08/F, which requires the user to calculate function and gradient values, and SUBROUTINE UCSDN2, NPL ALGORITHMS LIBRARY Ref. No. E4/07/F, a similar routine, except that it also requires the user to calculate second derivatives.

2. Description

From a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of $F(\underline{x})$. These points are generated using estimates of the gradient and curvature of the objective function. Finally, an attempt is made to verify that the final point is a minimum.

3. Specification

```
SUBROUTINE UCNDQ1(N, X, F, IFAIL, W, LW)
INTEGER N, IFAIL, LW
REAL F
REAL X(N), W(LW)
```

4. Parameters

Input parameter

N - INTEGER, a constant which must be set on entry to the number of independent variables in the function to be minimized.

Input and output parameter

X - REAL array, of length N.

On entry to UCNDQ1, X(J) must have been set by the user to an estimate of the Jth component of the position of the minimum (J = 1,2,...,N).

After a normal exit, X(J) contains the Jth component of the position of the minimum.

Output parameters

F - REAL. On exit, F gives the value of F($\underline{x}$) corresponding to the final point stored in X.

IFAIL - INTEGER. On exit, IFAIL = 0 indicates a successful call of the subroutine UCNDQ1. For other exit values of IFAIL and their meanings see Error Indicators. <u>N.B. The user must test the value of IFAIL to determine whether the subroutine has run successfully</u>.

Workspace parameters

W - REAL array of length LW, used as workspace by UCNDQ1. Elements of W used by UCNDQ1 are W(J), J=1,2,...,10*N+N*(N-1)/2 or J=1,2,...,11 if N=1.

LW - INTEGER, a constant which gives the actual length of the array W as declared in the program unit from which UCNDQ1 is called. This number must be at least 10*N + N*(N-1)/2 or 11 if N = 1.

5. User supplied subroutine

FUN1 - It is essential that the subroutine supplied by the user to calculate the value of the function has the name FUN1 and has the specification:-

```
SUBROUTINE FUN1(N, X, F)
INTEGER N
REAL F
REAL X(N)
```

The function value at the point defined by the array X(I), I = 1,2,...,N must be stored in F.

The elements of the array X and the integer N must not be altered in FUN1.

6. Error Indicators

IFAIL = 1, parameter outside expected range. This failure will occur if, on entry, N<1, LW<(10*N + N*(N-1)/2) or LW<11 if N=1.

IFAIL = 2, there have been 400*N function evaluations, yet the algorithm does not seem to be converging. The calculations can be restarted from the final point held in X. The error may also indicate that $F(\underline{x})$ has no minimum.

IFAIL = 3, the conditions for a minimum have not all been met but a lower point could not be found and the minimization has failed.

IFAIL = 4, an overflow has occurred during the computation. This is an unlikely failure, but if it occurs the user should restart at the latest point given in X.

IFAIL = 5, the conditions for a minimum have not all been met but the minimization has probably worked.

IFAIL = 6, the conditions for a minimum have not all been met but the minimization has possibly worked.

IFAIL = 7, the conditions for a minimum have not all been met and the minimization is unlikely to have worked.

IFAIL = 8, the conditions for a minimum have not all been met and the minimization is very unlikely to have worked.

If the user is unsatisfied with the solution it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. If persistent trouble occurs and the gradient can be calculated it may be advisable to use subroutine UCFDQ2 or UCFDN2 (see Purpose).

7. Auxiliary Routines

Routines named SFUN1, SPRNT1, DOT, LDLTSL, MDCOND, NMDCHL, MODCHL, QNMDIF, APPGRD, FRMDEL and LINSCH are called by UCNDQ1. Auxiliary routine NEWPTQ is called by LINSCH.

The texts of the first two, together with UCNDQ1, are given in

this document. The texts of the others are given in NPL ALGORITHMS LIBRARY documents, reference numbers E4/02/F, E4/03/F, E4/06/F and E4/15/F.

8. Storage

There are no internally declared arrays.

9. Timing

The number of iterations required depends on the number of variables, the behaviour of F(x) and the distance of the starting point from the solution. The number of operations performed in an iteration of UCNDQ1 is roughly proportional to N**2. In addition, each iteration makes at least N+1 calls of FUN1. So, unless F(x) can be evaluated very quickly, the run time will be dominated by the time spent in FUN1.

10. Accuracy

When a successful exit is made then, for a computer with a wordlength of t decimals, one would expect to get about $t/2 - 1$ decimals accuracy in x and about $t - 1$ decimals accuracy in F, provided the problem is reasonably well scaled.

11. Further Comments

Ideally, the problem should be scaled so that the minimum value of F(x) is in the range (-1, +1), and so that at points a unit distance away from the solution F is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth

trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that UCNDQ1 will take less computer time.

12. Keywords

Easy-to-use, function-only method, mathematical programming, maximization, minimization, optimization, quasi-Newton method, SUBROUTINE QNMDIF, unconstrained.

IMPLEMENTATION INFORMATION

This program (except for the first line) has been written in ANSI Fortran and has been tested on a CDC6500 computer.

The machine-dependent constant EPSMCH must be set in the first instruction in SUBROUTINE UCNDQ1, in SUBROUTINE QNMDIF, in SUBROUTINE APPGRD, in SUBROUTINE NMDCHL, and in SUBROUTINE FRMDEL. The machine-dependent constant RMAX must be set at the beginning of SUBROUTINE QNMDIF and of SUBROUTINE MODCHL. The constant EPSMCH is the smallest positive real number such that 1.0 + EPSMCH > 1.0, and RMAX is the largest positive real number such that both RMAX and -RMAX can be held in the machine. The example program has set EPSMCH = 2.0E+0**(-47), and RMAX = 1.0E+294 which are the correct values for the CDC 6500 computer.

EXAMPLE

Minimize the function

F(x1,x2,x3,x4) = (x1 + 10.0*x2)**2 + 5.0*(x3 - x4)**2 + (x2 - 2.0*x3)**4 + 10.0*(x1 - x4)**4,

starting from the initial estimate $\underline{X}$ = (3.0, -1.0, 0.0, 1.0).

This problem has the solution $F(\underline{x}) = 0.0$ at $\underline{x} = (0.0, 0.0, 0.0, 0.0)$, and is rather difficult in that the problem has a long valley along which the function is slowly varying.

PROCEDURES AND EXAMPLE PROGRAM TEXT

Now follows the text of

a) the program for the example problem,

b) a user specified subroutine FUN1, calculating the function value of the example problem,

c) the subroutine UCNDQ1,

d) two auxiliary subroutines SFUN1 and SPRNT1.

The other auxiliary routines required may be found in NPL ALGORITHMS LIBRARY documents reference numbers E4/02/F, E4/03/F, E4/06/F and E4/15/F.

This program has been run successfully on a CDC 6500 computer.

7. Acknowledgement

We thank the Energy Research and Development Administration, at the Stanford Linear Accelerator Center, under Contract No. E(04-3)-515, for providing the computer time.

8. References

American National Standards Institute (1966). Report USASI X 3.9 - 1966 (USA Standard Fortran).

Byrne, G. D. and Hindmarsh, A. C. (1975). A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations, ACM Trans. Math. Software, Vol. 1, pp. 71-96

Cody, W. J. (1974). The Construction of Numerical Subroutine Libraries, SIAM Review, Vol. 16, pp. 36-46.

Cody, W. J. (1975). The FUNPACK Package of Special Function Subroutines, ACM Trans. Math. Software, Vol. 1, pp. 13-25.

Cody, W. J. (1976). "An Overview of Software Development for Special Functions," in Numerical Analysis: Dundee 1975, Lecture Notes in Mathematics No. 506, pp. 38-48, Springer-Verlag, Berlin and New York.

Cox, M. G. (1974). "A Data Fitting Package for the Non-Specialist User," in Software for Numerical Mathematics (D.J. Evans, ed.), pp. 235-251, Academic Press, London and New York.

Cress, P., Dirksen, P., and Graham, J. W. (1970). Fortran IV with WATFOR and WATFIV, Prentice-Hall, Englewood Cliffs, New Jersey.

Dahl, O. J. and Hoare, C.A.R. (1972). "Hierarchical Program Structures," in Structured Programming, by Dahl, O.-J., Dijkstra, E. W., and Hoare, C.A.R., pp. 175-220, Academic Press, London and New York.

Davidon, W. C. (1975). Optimally Conditioned Optimization Algorithms without Line Searches, Math. Prog., Vol. 9, pp. 1-30.

Davis, P. J. and Rabinowitz, P. (1967). Numerical Integration, Blaisdell, London.

de Boor, C. (1971a). "On Writing an Automatic Integration Algorithm," in Mathematical Software (J. R. Rice, ed.), pp. 201-209, Academic Press, New York and London.

de Boor, C. (1971b). "CADRE: An Algorithm for Numerical Quadrature," in Mathematical Software (J. R. Rice, ed.), pp. 417-449, Academic Press, New York and London.

Dijkstra, E. W. (1972). "Notes on Structured Programming," in Structured Programming, by Dahl, O.-J., Dijkstra, E. W., and Hoare, C.A.R., pp. 1-81, Academic Press, London and New York.

Dixon, V. A. (1974). "Numerical Quadrature — A Survey of the Available Algorithms," in Software for Numerical Mathematics (D.J. Evans, ed.), pp. 105-137, Academic Press, London and New York.

Fletcher, R. (1972). Fortran Subroutines for Minimization by Quasi-Newton Methods, Rept. AERE-R7125, Harwell.

Ford, B. and Hague, S. J. (1974). "The Organization of Numerical Algorithms Libraries," in Software for Numerical Mathematics (D. J. Evans, ed.), pp. 357-372, Academic Press, London and New York.

Gill, P. E. and Murray, W. (1972). Quasi-Newton Methods for Unconstrained Optimization, J. Inst. Math. Appl., Vol. 9, pp. 91-108.

Gill, P. E. and Murray, W. (1974a). Safeguarded Steplength Algorithms for Optimization using Descent Methods, Rept. NAC-37, National Physical Laboratory.

Gill, P. E. and Murray, W. (eds.) (1974b). Numerical Methods for Constrained Optimization, Academic Press, London and New York.

Gill, P. E., Murray, W., Picken, S. M., Wright, M. H., and Graham, S. R. (1975a). Subroutine QNMDER, Rept. No. E4/02/0/Fortran/11/75, National Physical Laboratory.

Gill, P. E., Murray, W., Picken, S. M., Wright, M. H., and Barber, H. M. (1975b). Subroutine UCFDQ2, Rept. No. E4/09/0/Fortran/11/75, National Physical Laboratory.

Gill, P. E., Murray, W., Picken, S. M., Wright, M. H., and Barber, H. M. (1975c). Subroutine FRMDEL, Rept. No. E4/06/0/Fortran/11/75, National Physical Laboratory.

Gill, P. E., Murray, W., Picken, S. M., Barber, H. M., and Wright, M. H. (1976a). Subroutine LINSCH, Rept. No. E4/15/0/Fortran/02/76, National Physical Laboratory.

Gill, P. E., Murray, W., Picken, S. M., Barber, H. M., and Wright, M. H. (1976b). Subroutine LNSRCH, Rept. No. E4/16/0/Fortran/02/76, National Physical Laboratory.

Hardgrave, W. T. (1976). Positional versus Keyword Parameter Communication in Programming Languages, ACM SIGPLAN Notices, Vol. 11, pp. 52-58.

Hillstrom, K. E. (1976). A Simulation Test Approach to the Evaluation and Comparison of Unconstrained Nonlinear Optimization Algorithms, Rept. No. ANL-76-20, Argonne National Laboratory.

Hillstrom, K. E., Nazareth, L., Minkoff, M., Moré, J., and Smith, B. T. (1976). Progress and Planning Report, MINPACK Project, Argonne National Laboratory.

IFIP Working Group 2.5 (1976). MAP Statement in Fortran to Assist in the Portability of Numerical Software.

James, F. and Roos, M. (1975). Minuit — A System for Function Minimization and Analysis of the Parameter Errors and Correlations, Computer Physics Communications, Vol. 10, pp. 343-367.

Kahaner, D. K. (1971). "Comparison of Numerical Quadrature Formulas," in Mathematical Software (J. R. Rice, ed.), pp. 229-259, Academic Press, New York and London.

Kahaner, D. K. (ed.) (1976). Los Alamos Quadrature Workshop, Transcript of the Panel Discussion, ACM SIGNUM Newsletter, Vol. 11, pp. 6-25.

Knuth, D. E. (1974). Structured Programming with go to Statements, ACM Computing Reviews, Vol. 6, pp. 261-301.

Krogh, F. T. (1969). VODQ/SVDQ/DVDQ — Variable Order Integrators for the Numerical Solution of Ordinary Differential Equations, Section 314 Subroutine Write-Up, Jet Propulsion Laboratory, Pasadena, Calif.

Kuki, H. (1971). "Mathematical Function Subprograms for Basic System Libraries — Objectives, Constraints and Trade-Off," in Mathematical Software (J. R. Rice, ed.), pp. 187-199, Academic Press, New York and London.

Land, A. H., and Powell, S. (1973). Fortran Codes for Mathematical Programming, John Wiley and Sons, London and New York.

Lasdon, L. S., Waren, A. D., Jain, A., and Ratner, M. (1976). Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming, Technical Rept. SOL 76-3, Department of Operations Research, Stanford University.

Lawson, C. L. (1976). "On the Discovery and Description of Mathematical Programming Algorithms," in Numerical Analysis: Dundee 1975, Lecture Notes in Mathematics No. 506, pp. 157-165, Springer-Verlag, Berlin and New York.

Lyness, J. N., and Kaganove, J. J. (1976). Comments on the Nature of Automatic Quadrature Routines, ACM Trans. Math. Software, Vol. 2, pp. 65-81.

Minkoff, M. (1975). A Modularized Package of Dual Algorithms for Solving Constrained Nonlinear Programming Problems, in ACM 75 Proceedings of the Annual Conference (J. White, ed.), ACM, New York, pp. 159-162.

Murray, W. (ed.) (1972). Numerical Methods for Unconstrained Optimization, Academic Press, London and New York.

Mylander, W. C., Holmes, R. L., and McCormick, G. P. (1974). A Guide to SUMT - Version 4, Rept. RAC-P-63, Research Analysis Corporation, McLean, Virginia.

Nazareth, L. (1976). On Davidon's Optimally Conditioned Algorithms for Unconstrained Optimization, Tech. Memo. No. 283, Applied Mathematics Division, Argonne National Laboratory.

Parlett, B. N. and Wang, Y. (1975). The Influence of the Compiler on the Cost of Mathematical Software - in Particular on the Cost of Triangular Factorization, ACM Trans. Math. Software, Vol. 1, pp. 35-46.

Rice, J. R. (1971). "The Challenge for Mathematical Software," in Mathematical Software (J. R. Rice, ed.), pp. 27-41, Academic Press, New York and London.

Rosen, J. B. and Wagner, S. (1975). The GPM Nonlinear Programming Subroutine Package Description and User Instructions, Technical Rept. 75-9, Department of Computer and Information Sciences, University of Minnesota.

Schonfelder, J. L. (1974). "Special Functions in the NAG Library," in Software for Numerical Mathematics (D. J. Evans, ed.), pp. 285-300, Academic Press, London and New York.

Shampine, L. F. and Gordon, M. K. (1975). Computer Solution of Ordinary Differential Equations, W. H. Freeman and Co., San Francisco.

Shampine, L. F., Watts, H. A., and Davenport, S. M. (1976). Solving Nonstiff Ordinary Differential Equations — the State of the Art, SIAM Review, Vol. 18, pp. 376-411.

Sincovec, R. F. and Madsen, N. K. (1975). Software for Nonlinear Partial Differential Equations, ACM Trans. Math. Software, Vol. 1, pp. 232-260.

Smith, B. T., Boyle, J. M., and Cody, W. J. (1974a). "The NATS Approach to Quality Software," in Software for Numerical Mathematics (D. J. Evans, ed.), pp. 393-405, Academic Press, London and New York.

Smith, B. T., Boyle, J. M., Garbow, B. S., Ikebe, Y., Klema, V. C., and Moler, C. B. (1974b). Matrix Eigensystem Routines — EISPACK Guide, Lecture Note Series in Computer Science, Vol. 6, Springer-Verlag, New York.

Wilkinson, J. H. and Reinsch, C. (1971). Handbook for Automatic Computation, Volume II, Linear Algebra, Springer-Verlag, New York.

Zahn, C. T., Jr. (1975). "Structured Control in Programming Languages," in AFIPS Conference Proceedings, Vol. 44, 1975 National Computer Conference, pp. 293-295, AFIPS Press, Montvale, New Jersey.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SOL 77-7	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design and Structure of a Fortran Program Library for Optimization		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Philip E. Gill, Walter Murray, Susan M. Picken and Margaret H. Wright		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0865
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Operations Research--SOL Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-143
11. CONTROLLING OFFICE NAME AND ADDRESS Operations Research Program, Code 434 Office of Naval Research Arlington, VA 22217		12. REPORT DATE April 1977
		13. NUMBER OF PAGES 64
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

optimization; minimization; program library; mathematical software; mathematical programming; software design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see attached

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SOL 77-7: The Design and Structure of a Fortran Program Library for Optimization

ABSTRACT

This paper discusses in substantial detail the design principles and structure of an existing Fortran program library whose primary application is to solve optimization problems. Such a discussion not only helps to clarify the scope of application for potential users of the library, but also is useful for workers on other software projects. The fundamental objectives of the present library have been to produce sound, careful implementations of reliable methods that represent the state of the art in numerical optimization. The general implications of these overall design aims are presented, as well as specific instances of the results of decisions to include particular desirable features.