Department of Operations Research
Stanford University
Stanford, CA 94305
SOFTWARE FOR OPTIMIZATION,
by
L. Nazareth

TECHNICAL REPORT, SOL 78-32
December 1978

Research and reproduction of this report were partially supported by
the Office of Naval Research Contract N00014-75-C-0267; the National
Science Foundation Grants MCS76-20019 A01 and ENG77-05/61 A01; and
the Department of Energy Contract EY-76-S-03-0326 PA #18.

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.
Abstract

Our aim in this paper is to provide the reader with:

- some feel for what quality software entails,
- an overview of various aspects of optimization software,
- information on solution techniques and available software in the form of a decision tree.
- An extensive bibliography so that the reader can further pursue specific topics of interest.

We concentrate upon linear programming, non-linear unconstrained optimization and related areas, and non-linear programming.

This paper is intended to supplement an earlier oral presentation at the Texas Conference on Mathematical Software entitled "State of Software for Optimization".
Acknowledgment

My grateful appreciation to Professor G.B. Dantzig and the many others who helped make my visit to the Systems Optimization Laboratory an interesting and valuable learning experience.

My thanks also to the Applied Mathematics Division, Argonne National Laboratory, who jointly with the Systems Optimization Laboratory supported my appointment at Stanford.
SOFTWARE FOR OPTIMIZATION

by

L. Nazareth

1. Introduction

This paper is intended to supplement an earlier oral presentation on developments in optimization software.* Since very many mathematical problems can be posed in terms of function optimization, and since software for each optimization area ranges from small scale pilot programs to the large scale systems of which commercial LP systems are the most familiar example, we are of necessity selective in our choice of subject matter. We concentrate upon the areas of linear programming, non-linear unconstrained optimization (and the related areas of non-linear least squares and systems of non-linear equations), and non-linear programming. In particular, the very important areas of discrete variable programming and dynamic programming are not covered here.

Our aim in this paper is to provide the interested reader with:

a) Some feel for what quality software entails.
b) An overview of various aspects of optimization software.

c) Information on solution techniques and available software for the above areas in the form of a decision tree.

d) An extensive bibliography, so that the reader can further pursue specific topics of interest.

The paper is organized as follows: Section 2 provides some historical background to the optimization areas covered. Section 3 gives an overview of the software development process, and discusses attributes of 'quality' mathematical software, illustrating these with specific examples. Section 4 deals with software primarily intended to aid algorithm and code development, and discuss the idea of a language for mathematical programming. Section 5 deals with available optimization software for the areas covered here. A detailed decision tree is given. Section 6 discusses the testing of software and the bibliography is given in the final section.

We make no claims to being complete, but the author would welcome feedback on important omissions and inaccuracies, particularly with regard to material in Section 5.
2. Background

Table 1 is designed to give the reader a time frame for developments in optimization. Some of the important theoretical and algorithmic references are listed along with a few parallel developments in software and computers, and the reader can superimpose his own set of favorite topics.

Table 2, adapted from Wolfe [1975b], shows how our ability to solve problems has increased substantially. The vertical axis represents complexity of the problem and the horizontal axes lists different areas of optimization.

For further historical details see the survey articles of Dantzig [1977], Orchard-Hays [1977] and Wolfe [1975b].
<table>
<thead>
<tr>
<th>Dates</th>
<th>Algorithms &amp; Theory</th>
<th>Software</th>
<th>Computers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1935</td>
<td>Newton (1727); Fourier (1823); Babbage (1840)</td>
<td>EDSAC Library</td>
<td>First Generation ENIAC, UNIVAC, EDVAC, ACE, EDSAC</td>
</tr>
<tr>
<td>1940</td>
<td>Motzkin (1936)-Inequality Theory; Kantorovitch (1939)-L.P.</td>
<td>LP 10 rows (1952)</td>
<td>SEAC at N.B.S.</td>
</tr>
<tr>
<td>1945</td>
<td>VonNeumann (1944)-Game Theory</td>
<td>LP 100 rows (1954)</td>
<td>IBM 701</td>
</tr>
<tr>
<td>1950</td>
<td>Dantzig (1947)-L.P., Simplex Method</td>
<td>LP 256 rows (1956)</td>
<td>IBM 704</td>
</tr>
<tr>
<td>1960</td>
<td>Gomory (1958)-Integer Programming; Wolfe (1959)-Q.P.</td>
<td>LP 90/94-1000 rows</td>
<td>Third Generation IBM 360, CDC 6400</td>
</tr>
<tr>
<td>1965</td>
<td>Davidson (1959)-Variable Metric</td>
<td></td>
<td>Parallel Computers</td>
</tr>
<tr>
<td>1975</td>
<td>Fiacco &amp; McCormick (1964/6)-Penalty</td>
<td>SUMT</td>
<td></td>
</tr>
<tr>
<td>1975</td>
<td>Han (1975), Powell (1977)-Variable Metric for Constrained Optimization</td>
<td>NAG Project, IMSL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPSX/370</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NPL Optimization Library</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MINPACK (Argonne)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MINOS CODE (Murtagh &amp; Saunders (1977))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NL2SOL (Dennis et al. (1977))</td>
<td></td>
</tr>
</tbody>
</table>
Table 2

<table>
<thead>
<tr>
<th>Complexity</th>
<th>1978</th>
<th>1955</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECOMPOSABLE</td>
<td>LP</td>
<td>LP</td>
</tr>
</tbody>
</table>

U.M. — Unconstrained minimization, QP—Quadratic Programming
LC/NLO — Linear constraints, non-linear objective, NLC/NLO—non linear constraints and objectives
Complexity — (number of rows + number of variables)
‡ — indicates the figure given is only a rough approximation between the bounds indicated
3. Quality Software

3.1. Overview of Software Development

The design of an item of mathematical software depends very much upon the intended use of the software. We wish to stress the distinction between implementations of an algorithm designed primarily for studying the behavior of an algorithm, and implementations designed primarily for solving problems. The former are called algorithm/code oriented versions and the latter user/problem oriented versions. The distinction is, of course, not clear cut, since algorithm/code oriented versions can and should be used to solve practical problems, and user/problem oriented versions can and should be used to study the encoded algorithm. However an implementation will usually place emphasis on one of these two goals; and often an algorithm/code oriented version will be developed as a prelude to a user/problem oriented version.

An algorithm/code oriented version should not be construed to mean a hastily thrown together version. Rather it indicates a version in which emphasis is placed upon the goals of flexibility, generality and modifiability, even if this results in a sacrifice of efficiency. In a user/problem oriented version, increasing emphasis is placed upon efficiently solving a wide class of problems and providing a wide range of options. This may call for substantial reformulation and reorganization of the calculations to reduce overhead and circumvent numerical difficulties.
We feel that insufficient attention has been paid to developing tools to aid the implementation of algorithm/code oriented versions and this has contributed to the proliferation of untested algorithms which abound in the literature. This will be discussed further in Section 4. In contrast, a number of software aids have been developed to aid the process of tailoring a code to a particular compiler/machine configuration, i.e., to develop portable versions. For further details see Boyle [1976]. Since such aids can be applied to most items of mathematical software and are not specialized to optimization, we shall not discuss them further here.

For a more detailed discussion of the process of mathematical software development see Nazareth [1978a].

3.2. Attributes of Quality Software

Both algorithm/code oriented and user/problem oriented software should meet certain standards. What is it that characterizes 'quality' software?

Recent efforts to develop good mathematical software, Rice [1971], Smith et al. [1974], Ford and Hague [1974], identify several attributes. We quote these and illustrate them with specific examples.

(a) Robustness refers to the ability of a computer program to detect and gracefully recover from abnormal situations without unnecessary interruption of the computer run. In situations when a calculation
does fail, the code should fail gracefully. Robustness involves, for example, the filtering out of improper arguments, the avoidance of destructive overflows, and the reorganization of a calculation to minimize the effect of rounding error.

Example 2: Cody [1976], Avoiding both destructive overflows and non-destructive underflows in the computation of \( \|x\| = \left( \sum_{i=1}^{n} x_i^2 \right)^{1/2} \).

The usual FORTRAN calculation proceeds as follows:

```
SUM = 0.0 DO
DO 10 I = 1, N
   SUM = SUM + X(I)**2
10 CONTINUE
XNORM = DSQRT(SUM)
```

SUM can overflow even though XNORM may be a machine representable member. In order to avoid this, the calculation can be done as follows, where we assume for convenience, that the largest element, in absolute value, is \( x(1) \).

```
SUM = 1.0 DO
DO 10 I = 2, N
   A = X(I)/X(1)
   SUM = SUM + A*A
10 CONTINUE
XNORM = DABS(X(I))*DSQRT(SUM)
```
Now $x(I)/x(I)$ can underflow (non-destructively) leading to troublesome interrupt messages. To avoid this the computation can be further reorganized as follows:

```
SUM = 1.0 DO
B = DABS(X(I))
DO 10 I = 2, N
A = 0.0 DO
IF (B + DABS(X(I)) .NE. B) A = X(I)/X(I)
SUM = SUM + A*A
10 CONTINUE
XNORM = DABS(X(I))*DSQRT(SUM).
```

Example 2: Reorganizing calculation to minimize effect of rounding error. When variable metric methods were first suggested for solving the problem \( \min_{x \in \mathbb{R}^n} f(x) \), the calculation was stated in terms of updating an approximation to the inverse Hessian \( H \) of \( f(x) \). Given a step \( \Delta x \hat{=} x^* - x \) and the associated change of gradient of \( f(x) \), \( \Delta g \hat{=} \Delta f(x^*) - \Delta f(x) \hat{=} g^* - g \), a new approximation \( H^* \) is developed, for example, by the BFGS (see Broyden [1970]) update

\[
H^* = H + \frac{1}{\Delta x \Delta g} \left[ \rho \Delta x \Delta x^T - \Delta x \Delta g^T H - H \Delta g \Delta x^T \right]
\]  

(3.1)

where

\[
\rho \hat{=} 1 + \frac{\Delta g^T H \Delta g}{\Delta x^T \Delta g}
\]  

(3.2)
In theory \( H > 0 \) (i.e., positive definite) \( \Rightarrow H^* > 0 \) whenever 
\( \Delta g^T \Delta x > 0 \). However rounding error in the computation of \( H^* \) can
destroy this property. Another difficulty is that even when \( H^* > 0 \)
but ill-conditioned, rounding error in computing the next direction of
search \( d^* \triangleq -H^* g^* \) can result in \( d^*_c g^*_c > 0 \), where \( d^*_c = -f_l(H^*g^*) \) is
the computed search direction. See Gill, Murray and Pitfield [1972].

The above difficulties can be circumvented by reorganizing
the calculation following the suggestions of Gill and Murray [1972].
They suggest working with an approximation to the Hessian \( B \) which is
maintained in the factored form \( B = LDL^T \), where \( L \) is lower triangu-
lar and \( D \) diagonal. In this case we can ensure positive definite-
ness by keeping \( D > 0 \). In addition a bound on the condition number
of \( B \) can be improved by modifying \( D \). See also the Example 2 under
Reliability.

(b) **Reliability** refers to the ability of an item of software
to perform a calculation both efficiently and accurately and to
reflect the basic characteristics of the algorithm e.g., its scale
invariances.

**Example 1:** Estimating gradients of \( f(x) \) by finite differences. In
theory each component \( g_j \) can be estimated by first order finite
differences

\[
g_j = \frac{[f(x + h e_j) - f(x)]}{h} \tag{3.3}
\]

where \( h \) is an infinitesimal step.
In finite precision arithmetic however this is a difficult computation. A good routine must be designed with considerable care and we state some of the issues which arise in designing such a routine.

-- Choice of step length, h.
- Should h vary with each component?
- Should h be chosen to balance rounding and truncation?

In this latter case estimates of second derivatives are needed to estimate truncation error. How are these obtained? Should h be estimated at every iteration or should it be only periodically recomputed in a separate subroutine and held fixed in between calls to this subroutine.

-- Should a switch to central differences be made when forward differences are insufficiently accurate?

-- Should the increment h be relative to $|x_j|$ or should it be absolute? In the former case $g_j$ is invariant under a simple scaling of variables $x_j + \alpha x_j$, whilst in the latter case $g_j$ is invariant under a translation of variables $x + x + c$.

-- Should gradients be estimated in a transformed space of variables? i.e., consider the function

$$f(z) = f(x) + g^T(z - x) + \frac{1}{2}(z - x)^T B(z - x)$$

where $B = (JJ^T)^{-1}$ and non-singular.
Contours of $f(z)$ are illustrated in Figure 3.

If we make the transformation $z = x + Jy$, then $f(z)$ transforms to $\tilde{f}(y)$

$$\tilde{f}(y) = f(x) + k^T y + \frac{1}{2} y^T y$$

where $k = J^T g$. The contours of $\tilde{f}(y)$ are illustrated in Figure 4.

$k_i$ are estimated by

$$k_i = \frac{f(x + J_i h) - f(x)}{h} - \frac{h}{2}$$

and $g$ is then given by $g = (J^T)^{-1} k$. 

Figure 3

Figure 4
Example 2: Invariance w.r.t. transformations of variables. It is well known that apart from the initial choice of the approximation, the variable metric algorithm is invariant w.r.t. transformation of the variables. However, if the algorithm is modified to ensure that the search direction \( d_k = -H_k g_k \) satisfies

\[
|d_k^T g_k| \geq \varepsilon \|g_k\| \|d_k\|,
\]

for \( \varepsilon \) a small constant, (3.4)

(for example by modifying \( H_k \) suitably), then this destroys scale invariance, since (3.4) is not invariant. For a fuller discussion, see Powell [1976b].

(c) **Structured** refers to whether the program is designed along the principles of good programming, i.e., whether it has a top to bottom flow of control, is formatted to display its structure and so on. See Dahl et al. [1972] and Kerninger and Plauger [1974].

(d) **Usability** refers to the ease with which a user can choose a program and apply it to his problem. For example, how well designed are the calling sequences and documentation. See Gill et al. [1977].

(e) **Validity** refers to the existence of evidence that the software has performed well in a particular computer environment, and to the existence of testing aids which demonstrate that the present installation of software is performing as expected. We discuss testing further in Section 6.
(f) **Transportability** refers to whether an item of software can be moved from one computer installation to another without degradation of performance and with minimal change.

**Example:** Features of COMMON statement in FORTRAN which can hinder transportability. A very complete discussion of difficulties which arise in transporting FORTRAN programs is given in Smith [1976]. When using the COMMON statement some of the difficulties which arise are:

---

-- The order of variables affects portability, e.g., some IBM machines require that variables in a COMMON statement which use two storage units, begin on an even word boundary, else alignment errors or a degradation in efficiency can result.

-- Variables in labelled COMMON may become undefined upon execution of a RETURN (or END) statement, unless there is a COMMON statement for that block, in at least one of the higher level program units in the chain of active programs.

-- Other inconveniences associated with COMMON are that variable dimensioned entities cannot be used in COMMON, and that the size of a labelled COMMON block may be required to be the same in each program unit in which the COMMON block occurs.
4. Software Designed to aid Algorithm/Code Development

Implementing an optimization algorithm is a difficult and a time consuming task. For this reason it is essential that the algorithm or software developer be provided with suitable tools which facilitate his task. We can distinguish three approaches:

Approach 1. Develop a high level language, for example along the lines of the Mathematical Programming Language (MPL) of Dantzig et al. [1970]. The aim is to design a language in which highly readable programs can be written and which parallels the venacular of applied mathematics. This would make it possible to write programs quickly and easily and would serve as a means of communicating ideas precisely. In the early creative stages of algorithm development such a language is an invaluable aid since one's intuition can now be supplemented by hard computational results. This can then in turn lead to new ideas. Such a language is also a valuable educational aid. However, once the main features of an algorithm have been laid out, a fundamental difficulty remains, namely that numerically sound procedures are difficult to write in any language, no matter how convenient. What is then needed is a good library of procedures, tailored to optimization, from which an optimization algorithm can be built. This is particularly useful when one wants to test out a new algorithm on real life problems. Building optimization algorithms from a library of procedures also makes for a more uniform comparison of algorithms since their implementations can be made to differ only
in the essentials and test results are thus less subject to variations in programming style. It is difficult for a new language to gain wide acceptance and for compilers to be made available on a wide range of machines. Often therefore, we have to fall back upon FORTRAN, although other high level languages, e.g., PL-1 and ALGOL-68 are making some headway.

**Approach 2.** Use individual components of a user/problem oriented implementation (or optimization system) which has a modular design. Examples in the area of Linear Programming are discussed in Nazareth [1978]. For examples of such systems in the area of non-linear programming see Muralidharan and Jain [1975], Hillstrom [1976]. The main difficulty with this approach is that one has usually very limited flexibility. Each component in a user/problem oriented system is usually designed within the context of the overall system and often utilizes a common data structure. Using a component on a stand alone basis usually requires that it be substantially modified.

**Approach 3.** The idea behind the third approach has already been mentioned. Here one seeks to develop a carefully specified set of modules which can be viewed as being the 'primitives' or 'basic operators' of a language for building optimization algorithms. Two efforts along these lines are discussed in Nazareth [1977] and [1978]. The former describes a pilot system based upon a set of algorithms developed by the author in the area of non-linear unconstrained optimization. Building upon this experience, a software organization and
development effort was undertaken in the area of Linear Programming, as described in Nazareth [1978]. It is important to emphasize that the development of a code requires careful craftsmanship and should not be viewed as the mere stringing together of modules. However if such modules are carefully designed and correctly implemented, they can greatly ease the task of implementing an algorithm and perhaps they should be viewed as a way of developing an "artists sketch" of a code, which can then be further refined. They also serve as a valuable means of cooperation and communication between different researchers. Finally they are a useful educational aid.
5. Optimization Software

In this section we give a detailed decision tree of the major categories of optimization methods together with references to some recent implementations and/or algorithms in each category. References are given to journal articles or technical reports and implementations are identified by a symbol indicating their source.

Available software varies widely in quality and we do not set out here to make any value judgements. Clearly such a compact presentation is also far from complete. Our more modest aim is to provide the reader with some selected information on individual items of software. Other surveys e.g., Dennis [1976], Dixon [1973], Fletcher [1976], Wolfe [1975a], Wright [1978] should also be consulted.

5.1. Some Major Sources of Optimization Software (Alphabetical)

[a] Argonne National Laboratory, Applied Mathematics Division([hp]-
Hillstrom's Package [1976] and [m]-MINPACK-1).

[b] Bell Telephone Laboratory, Murray Hill, New Jersey, (PORT Library).

Science and Systems Division, Harwell, England.

[ibm] IBM Mathematical Subroutine Library (SL-MATH).

[ims] International Mathematical and Statistical Libraries, Inc.

[w] Computer Center, University of Wisconsin, Madison.

[nber] National Bureau of Economic Research, Cambridge, Massachusetts
(now part of M.I.T.).

[noc] Numerical Optimization Center, Hatfield College of Technology,
Hatfield, Herts., England.

[npl] National Physical Laboratory, Division of Numerical Analysis
and Computing, Teddington, England. (The NPL Optimization
Library is the most comprehensive collection of optimization
software currently available.)

[s] Computer Science Department, Stanford University.

[sol] Systems Optimization Laboratory, Department of Operations
Research, Stanford University.

The symbol associated with each establishment in the above
list is used to identify the establishment in the Decision Tree.
Note that software referred to in this manner is not necessarily
available for general distribution. Conversely when an algorithm
in the Decision Tree does not have a symbol associated with it or
when the symbol [au] is used, an implementation may be available from
the author(s) of the cited reference.
Decision Tree

UNCONSTRAINED OPTIMIZATION

\[ \min f(x) \quad (1) \]
\[ x \in \mathbb{R}^n \]

SMALL/MEDIUM

\[ f(2) \]

DIRECT SEARCH

(CONJUGACY BASED

(GILL & MURRAY [1972a][np1],
DAVIDON & NAZARETH [1977a,b][au])

(VARIABLE METRIC

(GILL, MURRAY & PICKEN [1972])(np1),
FLETCHER & FREEMAN [1975], MORE & SORENSEN [1977])

(LARGE

(MODIFIED NEWTON

(GILL & MURRAY [1972a][np1],
DAVIDON & NAZARETH [1977a,b][au])

(NON-DIFFERENTIABLE

(WOLFE [1974])

(CONJUGATE GRADIENT³

(Powell [1975][h], SHANNO [1977])

(NON-DIFFERENTIABLE

(WOLFE [1974])

(Powell [1975][h], Shanno [1977])

(Powell & Toint [1978])

NON-LINEAR LEAST SQUARES

\[ \min \sum_{i=1}^{m} [f_i(x)]^2 \]
\[ x \in \mathbb{R}^n \]

DIRECT SEARCH

(SPENDLEY [1969])

(QUASI-NEWTON UPDATES TO JACOBIAN

(Powell [1971a][h][hp])

(SMALL RESIDUAL -- GAUSS-NEWTON VARIANTS

(Levenberg-Marquardt [1963][m])

(LARGE RESIDUAL

(DENNIS ET AL. [1977][nber], GILL & MURRAY [1976b][np1], NAZARETH [1975])

(SEPARABLE

(Golub & Porenyra [1973][s])

20
## Quadratic Programming

\[
\begin{align*}
\text{min } & \mathbf{x}^T \mathbf{C} \mathbf{x} + \mathbf{b}^T \mathbf{x} \\
\text{such that } & \mathbf{A} \mathbf{x} = \mathbf{b} \\
& \mathbf{x} \geq 0
\end{align*}
\]

**Small**: Active Set Strategies

**Convex/Indefinite**: Based upon Simplex Method

**Large**: Primarily Convex

**Based upon Complementarity**

*(Fletcher [1970][h], Gill & Murray [1977a][np1])

*(see e.g., Cottle [1977])

*(Tomlin [1976][sol])

## Linearly Constrained Optimization

\[
\begin{align*}
\text{min } & f(x) \\
\text{subject to } & \mathbf{A} \mathbf{x} \leq \mathbf{b}
\end{align*}
\]

**Projected/Reduced Gradient/VARIABLE METRIC**

**MODIFIED NEWTON**

*(Buckley [1975][h], Rosen & Wagner [1975], Gill & Murray [1976a][np1])

*(Gill & Murray [1976a][np1])

*(Murtagh & Saunders [1977][sol])

## Nonlinear Programming

\[
\begin{align*}
\text{min } & f(x) \\
\text{subject to } & c(x) \leq 0
\end{align*}
\]

**Penalty/Barrier**

**AUGMENTED LAGRANGIAN**

**PROJECTED LAGRANGIAN**

**PROJECTED LAGRANGIAN**

**PROJECTED LAGRANGIAN**

**SMALL/MEDIUM REDUCED GRADIENT**

**LARGE REDUCED GRADIENT**

*(Fiacco & McCormick [1964], Mylander et al. [1971][au], Lootsma [1970], Rosen & Kreuser [1971][w])

*(Fletcher [1975][h], Gill & Murray [1977b][np1])

*(Rosen [1977][au])

*(Wilson [1963], Murray [1969], Biggs [1972][noc], Han [1975], Powell [1977][au], Murray & Wright [1978])

*(Abadie & Guigou [1969][1971][au], Lasdon et al. [1976][au])

*(Jain, Lasdon & Saunders, see Jain [1976])
Footnotes to Decision Tree

1 A particular case of this problem is that of 1-D optimization, both stand alone and for use within n-dimensional optimization routines. See e.g., Brent [1973a], Gill and Murray [1974].

2 The symbols f, g, B and J are used to indicate the type of information about the function that is usually required when

\[ f \] stands for function value
\[ g \] stands for gradient
\[ B \] stands for Hessian
\[ J \] stands for Jacobian.

Thus e.g., \( f/(f,g) \) means function value or (function value and gradient).

3 Currently a very active research area. For an overview see Nazareth and Nocedal [1978].

4 The distinction between small, medium and large scale is as follows. In small scale L.P. no account is taken of the sparsity of the L.P. matrix, i.e., it is assumed that the matrix is dense and is usually stored as a 2-D array. In medium scale L.P. it is assumed that the L.P. matrix will fit in core provided only non-zeros are stored in packed form, e.g., as a column list/row index data structure. Finally, large scale systems e.g., MPSX/370 make extensive use of secondary storage.

5 For a good overview see Dantzig [1968], Geoffrion [1970].

6 Terminology of Murray and Wright [1978].
6. **Testing**

Evaluating optimization routines is a difficult task, and one which requires both qualitative and quantitative measures of performance. A fundamental requirement is that the testing environment simulate an actual environment of use since, if it did not, the evaluation would be valid but in all likelihood, irrelevant. Furthermore, the overall quality of a code can only be gauged after investigating a broad range of issues, for example, efficiency, robustness, usability, usefulness of documentation, ability of fail gracefully in the presence of user abuse, rounding error difficulties or violation of underlying assumptions. A testing method usually concentrates on efficiency and robustness, evaluating these by exercising the code on a set of well chosen and hopefully realistic problems.

To date the most common method of evaluating optimization routines has become known as 'battery' or 'simulation' testing.

Comprehensive studies along these lines are described in Colville [1968], Hillstrom [1977], Himmelblau [1972]. Battery testing has two basic components, namely a set of test problems and a set of measures of performance. The approach is subject to limitations which sometimes make a clear ranking of methods difficult to discern. For example, it is difficult to know how much confidence should be attached to a particular measure of performance when slight variation of starting point or geometry of test problem leads to a substantial variation in the measure of performance. For a discussion of these difficulties see Nazareth and Schlick [1976]. This has motivated
the approach which employs "problem families" or "parametrically defined test problems" introduced originally into the evaluation of routines for numerical quadrature by Lyness and Kaganove [1976]. See also Dembo and Mulvey [1976]. A careful a priori experimental design and the use of statistical sampling theory and analysis are implicit in this approach, which is sometimes referred to as 'performance profile' testing to differentiate it from 'battery' testing.

A second distinction which it is worth emphasizing is the distinction between algorithm and software evaluation. In particular testing an algorithm usually places most emphasis on efficiency whilst software evaluation attaches a great deal of importance to reliability and robustness.

Finally it is worthwhile making a distinction between decentralized and centralized testing of routines. The former is illustrated by the original study of Colville [1968], and the latter is illustrated by the study of Hillstrom [1977]. In decentralized testing a set of software tools are usually made available to developers of routines who then use them to develop information about how well their routines perform. For an example in the area of non-linear programming see Nazareth [1977 where the testing tools comprise:

(i) Subroutines which return function and/or gradient information for a set of different test functions.

(ii) Subroutines which return starting point and expected solution (if known), for each function.
(iii) Report writer whose features include:
   — Flexible and convenient way of specifying which functions to test.
   — Replaceable section of code for routine being tested.
   — Interface subroutines between user form of function call and subroutines in (i) above.
   — Output summaries and graphical display.

Systems Optimization Laboratories (see Dantzig et al. [1973]) are a natural environment for centralized testing, i.e., gathering and testing a number of routines at one particular site. This usually requires a substantial commitment of resources, but it makes for a much more uniform comparison and permits the use of much more stringent test problems, in particular problems arising from real life applications (see Dantzig and Parikh [1977]).

Until fairly recently, the development of a testing methodology for optimization routines has been sorely neglected. However, the crucial importance of the subject is now being recognized. For a description of some recent work see Bus [1977], Crowder, Dembo and Mulvey [1977], Nash [1975], More et al. [1978], and consult the minutes of meetings of the Committee on Algorithms of the Mathematical Programming Society.
Bibliography


27


Nash, J. (1975), Bibliography of Non-linear Least Squares (microfiche).


Powell, M.J.D. (1971a), A.E.R.E., Harwell Library Subroutine, VAO5A.


Staha, R.L. and Himmelblau, D.M. (1976), "Evaluation of Constrained Non-linear Programming Techniques," (manuscript), Department of Chemical Engineering, University of Texas, Austin.


**SOFTWARE FOR OPTIMIZATION**

**AUTHOR(s)**
L. Nazareth

**PERFORMING ORGANIZATION NAME AND ADDRESS**
Department of Operations Research — SOL
Stanford University
Stanford, CA 94305

**CONTROLLING OFFICE NAME AND ADDRESS**
Operations Research Program — ONR
Department of the Navy
800 N. Quincy Street, Arlington, VA 22217

**PROJECT OFFICE NAME AND ADDRESS**

**REPORT DATE**
December 1978

**NUMBER OF PAGES**
36

**DISTRIBUTION STATEMENT (of this Report)**
This document has been approved for public release and sale; its distribution is unlimited.

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

**SUPPLEMENTARY NOTES**

**KEYWORDS**
Mathematical Software
Software Bibliography
Unconstrained Optimization

**ABSTRACT**
See Attached

**DECLASSIFICATION/DEGRADING SCHEDULE**
Unclassified

**AFFECTED SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)**
Unclassified
SOL 78-32
SOFTWARE FOR OPTIMIZATION
L. Nazareth

Our aim in this paper is to provide the reader with:

a) Some feel for what quality software entails.
b) An overview of various aspects of optimization software.
c) Information on solution techniques and available software in the form of a decision tree.
d) An extensive bibliography so that the reader can further pursue specific topics of interest.

We concentrate upon linear programming, non-linear unconstrained optimization and related areas, and non-linear programming.

This paper is intended to supplement an earlier oral presentation at the Texas Conference on Mathematical Software entitled "State of Software for Optimization".