System design for the testing of mathematical software

A.G. Buckley

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 90

SYSTEM DESIGN FOR THE TESTING OF MATHEMATICAL SOFTWARE

Albert G. BUCKLEY

NOVEMBRE 1987

System Design for the Testing of Mathematical Software

Albert G. Buckley
INRIA
Rocquencourt
78153 Le Chesnay Cedex
(1) 39 63 55 11

Juin, 1987

au congé sabbatique de la
Department of Mathematics,
Statistics and Computing Science
Dalhousie University
Halifax, N. S. B3H 3J5
Canada

CONSTRUCTION D'UN SYSTEME DE TEST DES BIBLIOTHEQUES MATHEMATIQUES

Résumé

Cet article décrit un logiciel écrit pour tester les programmes d'optimisation sans contraintes. Néanmoins, les idées présentées dans ce logiciel l'ont été de façon à s'appliquer à d'autres bibliothèques mathématiques, comme par exemple résolution de systèmes non linéaires ou d'équations différentielles. Le logiciel est entièrement portable et est, par construction, suffisamment souple pour permettre de tester une large classe d'algorithmes d'optimisation. De nombreuses caractéristiques sont disponibles, comme la communication réciproque, différentiation numérique et test automatique de dérivées, définition dynamique de problèmes, un langage de commande souple, et interaction pendant l'exécution aussi bien que pendant les entrées-sorties. Ce logiciel devrait simplifier le processus de test des nouveaux programmes d'optimisation, et on pourrait l'utiliser pour améliorer la qualité des mesures de performance des algorithmes d'optimisation.

System Design for the Testing of Mathematical Software

ABSTRACT

This paper will describe software which has been written for testing unconstrained optimization software. The intention however is that the ideas presented in this software could have applicability to testing mathematical software in other areas, such as nonlinear equation solving or differential equations. The software is entirely portable and is, by design, sufficiently flexible to allow a wide class of optimization algorithms to be tested using the facilities available within the package. A number of useful features are available within the package, including reverse communication, automatic finite differencing and gradient testing, dynamic problem definition, a flexible control language, and control over both the running environment and input/output processes. This software should simplify the process of testing new optimization software, and it could be used to improve the quality of testing in the evaluation of minimization algorithms.

System Design for the Testing of Mathematical Software[1]

[1]Submitted to ACM Transactions on Mathematical Software. Also appears as Technical Report 1987CS-5, Department of Mathematics, Statistics and Computing Science, Dalhousie University, Halifax, N. S. B3L 3C4, Canada.

1 Introduction

In this paper we would like to discuss the process of testing algorithms which are written to solve particular mathematical problems. At the outset, we should make it clear that we are not going to be concerned with the evaluation of the relative merits of different algorithms. Rather we will discuss the design of general purpose software which can serve as a tool for researchers to *aid* in the process of evaluating algorithms.

One may look at the testing of algorithms from two quite distinct points of view. At one extreme there is an individual (or perhaps a small group) who has, through innovative ideas, designed and written a new algorithm for solving some particular class of problems. This individual would then like to test the new algorithm to see if it performs as well as he had perhaps hoped it would. The resulting testing is often quite limited (and is sometimes rather biased towards certain characteristics of the algorithm at hand). The reason of course is that testing is very time consuming and often rather a dull job, so that there is little motivation to become any more involved with it than is necessary.

On the other hand, there have been major projects undertaken to do a thorough job of evaluating a number of current algorithms, with a view to reaching some conclusions and making certain recommendations to the general community about which algorithms are best on which types of problems. Two well known examples of this are the Colville study [3] and the work of Schittkowski [15]. The difficulty with this process comes when a new algorithm is introduced. There is essentially no way to "redo" the study, incorporating the new algorithm. Thus the substantial amount of effort put into large evaluation projects is not available to the general user community, so that, as successive and possibly better algorithms appear, the results of the study may become more and more dated. This has certainly been the case with the Colville study.

Here we want to discuss the design of software which will attempt to bridge the gap between these two extremes. We want to indicate how one can design software which can be used to perform large numbers of tests in a convenient fashion and which will produce statistics upon which recommendations can be based. This software must be available to any researcher, who may then use it for evaluating the performance of new and/or current algorithms.

In what follows, we will not consider a process of quite such vague generality. Instead, we will pick a specific mathematical problem and consider algorithms for solving that particular problem. The intention however is that much of what is said and done can be extended and/or adapted to other types of problems in a manner which will be, conceptually at least, quite straightforward. We will see a number of features described in what follows, but these should be taken as indicative of the kinds of ideas which can be implemented into a general purpose testing package, rather than as features specific to the minimization problem to be considered.

One of our reasons for considering a particular problem is that it makes much of the ensuing discussion simpler. Another reason is that by considering the specific problem of minimizing a nonlinear function $f(x)$ of n real variables, we can orient our discussion to specific software which has been written for this problem. Notice that we will use the term "software" to refer to the set of routines which we have been discussing as being available to aid in the evaluation of specific algorithms. The term software can of course also be applied to the codes implementing the actual mathematical algorithms, but we will not use it in that sense here.

So let us now center our discussion on algorithms which, given a function f, generate in some unspecified manner a vector $\hat{x}$ which will be (normally) a good approximation to a local minimum x^* of f. And let us also focus our attention on a collection of routines called, for want of a better name, TESTPACK, or more briefly, TP. In broad terms, given the code MIN for a specific algorithm, TP enables a user to apply the algorithm MIN to a large group of

test problems and to obtain, with a minimum[2] of effort, a set of statistics which can then be examined to determine the efficacy of the algorithm MIN.

Before we go any further, we should make some reference to the work of Moré, Garbow and Hillstrom [12] at the Argonne National Laboratory. They considered this same problem of minimization (along with the related problems of nonlinear equation solving and nonlinear least squares) and published work in which they discussed approaches to testing and where they presented a collection of test problems which is now quite well known. The work to be described here goes further than their Argonne package, as will become clear. However, we have used much of what they have done, and we have included their test functions in this package, along with references which relate the work here to that which they published, so it is clear that we must express our gratitude to them for the part which the Argonne package has played in the development of this work.

As a final introductory note, we should say that we will not give detailed instructions on the use of TP here. However, one may obtain a report [2] on the use of TP directly from the author; it contains more detailed descriptions of all aspects of TP.

2 Objectives

Let us consider what the objectives should be in designing and writing a piece of software such as TP. There are several, and we will discuss them in what we feel is roughly their order of importance.

- *Versatility:* We have said that TP should be able to take almost any subroutine, say MIN, which computes an approximate local minimum of f, and apply that subroutine to minimize a large collection of test problems. Therefore one must be careful in designing TP to ensure that one does not, either implicitly or explicitly, impose any unnecessary restrictions on the routine MIN. Moreover, since MIN can have virtually any calling sequence (as determined by its developer), one should structure TP so that no direct call to MIN occurs. One can perhaps consider an analogy here. When a general purpose algorithm is being written for minimizing a function f, it is often the case that the user must supply a subroutine for evaluating f, and this subroutine must have a *specified* calling sequence in order that it can be linked to the minimization algorithm. Here, we are writing general purpose software for testing *the algorithm*, and we wish to be able to link an arbitrary *algorithm* into the software. However, a variety of algorithms for minimizing f will have a variety of calling sequences, so that here we can *not* impose a fixed calling sequence on the test algorithms. In addition, if TP is to be truly general purpose, it should not be necessary for the user to have to make any modifications to the source code in TP. How these points are handled is discussed in Section 6.

- *Portability:* The intention is that TP will be provided as a file of source code which can be sent to any site, installed there, and then used to test algorithms at that site. In other words, we are not interested in a testing project which is restricted to one site, such as a testing center. Clearly then the source code for TP must be portable. Note though that some interest has been expressed [1] in forming a centralized testing center for evaluation of algorithms. The portability of TP does not preclude the possibility of having TP available at such a site. In fact, one can view this paper and TP as representative of some ideas and one approach which could be used in the development of a testing center for mathematical software. Portability is taken up in Section 6.2.

[2] no pun intended.

- *Simplicity:* TP is intended to be a tool to aid in the evaluation of minimization software. It must be simple to use; otherwise it would be easier for each algorithm designer who needs to test an algorithm implementation to write *his own* programs to test his own algorithms, and the entire point to TP would be lost. It should be noted that TP offers a number of features which could not be duplicated by an individual without considerable effort, but that alone would not convince people to use the package if it were unduly difficult either to implement or use TP. Therefore it is imperative that TP, or any similar software, be simple to use.

- *Breadth:* A software package such as TP should offer a scope of testing which would not likely be available otherwise. This means that a reasonably large class of test problems should be available, and that features should be offered which would be beyond those which an individual would want to take the time to develop himself for testing just one algorithm. Combined with simplicity of use, this should provide good motivation to use the package. At the same time, it should be noted that power and simplicity are somewhat conflicting goals. When implementing features into TP, it is always a question whether or not the benefits obtained are worthwhile, if at the same time additional complexity is introduced into the package. We believe that TP currently offers a wide scope of useful features, whilst still remaining straightforward to use.

- *Consistency:* One of the difficulties often encountered in evaluating a new algorithm is that of doing a meaningful comparison with published results. The published results may be incomplete, or they may be presented in a fashion which is not amenable to comparison with the algorithm at hand. For example, the results may be given in terms of CPU time, with no way of repeating those results on a different machine. Or, the results might have used one specific stopping criterion, whereas what is desired is to repeat those runs with a modified criterion. All of these points become easier to deal with in the presence of "universal" testing software. In particular, by having a large, well-defined and permanent set of test problems as part of the testing package, it becomes much easier to deal with all of these questions of consistency. Runs—the *same* runs—can be repeated to obtain CPU times on a different machine, or with a new stopping criterion, and so on.

- *History:* Such a package should facilitate comparison to results published before the advent of the package. Therefore, in so far as is possible, the testing package should contain all previously published test problems, along with a careful tabulation of their sources and places where they appeared in the literature. TP includes a number of cross reference tables, as will be discussed in Section 5.6.

- *Modularity:* When designing software to serve a purpose as broad as has been suggested, it is clear that duplication of effort should be avoided. For example, if one wished to produce a package for testing algorithms which solve sets of nonlinear equations, it would be highly desirable if one could use large parts of the code from TP, which does after all treat a very similar problem. We think that TP has been well-designed, and it would indeed be possible to re-use much of the code. For example, TP includes a language for communicating with the user; the routine which decodes that language is completely independent of the minimization problem and thus could be used directly in the development of software for testing nonlinear equation solvers.

This discussion has stated in very general terms what we feel should be some of the main objectives in the design of such software. In what follows, we will discuss how we feel we have met these objectives with TP, or to what extent we could not meet them, in which case we will discuss those limitations inherent in current computing environments which made it difficult to

meet the desired objectives. We will also comment on some possible future solutions for these problems.

3 Features

Let us continue by summarizing some of the major features of TP. Certainly anyone who has done any algorithm testing will have programmed and used some of these features. However we believe that no other portable package has offered as many as are available here.

Generality: We have already stated that TP has been designed so that a wide class of codes for unconstrained minimization can be tested with TP.

Defaults and Tailoring: The package is designed so that all control quantities have default values. At the same time, considerable scope is given for each user to tailor TP to his own needs and computing environment. One way this is done is to use a PROLOGUE file, which is read at the beginning of executing TP, and which may contain those commands which are needed to adapt TP to a particular installation, e.g. see CC= in Section 7. Of course, the file PROLOGUE need only be set up once, and a version suitable for many environments is provided as part of TP. This file is also useful for defining user specific names to replace generic names; see Section 6.5. Many readers will no doubt recognize this as analogous to the processing of a standard "log-in" or "start-up" file.

Control Language: TP uses a language for user communication which allows maximum latitude in what may be entered. The rules defining the control language are quite flexible, and TP will attempt to interpret anything reasonable. An echo is available to ensure ambiguities are correctly resolved.

Reverse Communication: There is a difference in philosophy amongst algorithm writers as to the method whereby function values should be obtained by a minimization algorithm. TP allows either direct calls to a function evaluation routine or reverse communication to the calling routine.

Finite Differences: The user of TP may indicate that derivative calculations are to be done using finite differences, and these computations will be done automatically. TP may also be used to test the accuracy of gradient evaluation codes by selecting *test* mode, in which both analytic gradients and finite differences are computed and then compared.

Problem Specification: A very careful enumeration of test problems is included with TP. This includes a file in which the problems are defined in a machine-readable fashion, as well as a more human-readable document in which all details relating to each problem are given. All problems known from the literature are included, and a number of cross reference tables are given in [2] which facilitate access to these problems.

I/O Control: In so far as is feasible, TP allows the user to control the input and output processes. File names may be interactively defined, output can be turned on or off, summaries can be made long or short, and so on.

Environment Control: Although FORTRAN 77 is supposed to be a portable language, that does not alter the fact that very different things can take place under different operating systems or on different hardwares, even within the standard language. For example, should carriage

control characters be issued as in VMS[3], or not, as in NOS, when writing to a terminal? Is the run batch or interactive? Is the terminal a screen or hardcopy device? Should one avoid certain characters which have a special meaning to this operating system? TP allows the user to specify how many of these situations should be handled.

Distribution Tape: TP is distributed on a tape containing a number of files. The main one is of course all of the source in portable FORTRAN 77. A suitable PROLOGUE file is included, which contains the definitions of all of the test problems. For the very few (and very small) routines into which all machine dependencies have been isolated, alternate versions can be obtained, one of which will suit most systems. Also included is a step by step set of instructions of how to implement and use TP.

The remainder of this paper will consider TP in more detail.

4 Demonstration

This section is devoted to a demonstration of an interactive session using TP. Of course, the commands chosen here are for the purpose of demonstrating some of the features of TP, and hence do not represent a very useful session.

The section will consist of two parts. First there will be a set of comments. These pertain to the actual demonstration session which follows, and which constitutes the second part of this section. Each comment is preceded by a number, which refers to the line with the same number in the terminal output which follows. Only referenced lines are numbered. Note that lines in the slanted typewriter font are those which were given as input. The symbol "␣" indicates a blank line.

Considerably more output may be generated when running TP; however this is stored in several files which will not be shown here. We now begin with the comments on the terminal session, which will then follow.

$\star\star$*Line* 1 $\Rightarrow$ A session starts with a request for a title. This helps to identify work at a later time. It is added to all permanent file output generated by TP.

$\star\star$*Line* 2 $\Rightarrow$ Initially, several input steps are done. This is as a reminder of the types of control which are available with TP. Later we will see that this can be dispensed with. Any command can be entered at any step.

$\star\star$*Line* 3 $\Rightarrow$ The problems to be tested may be sorted by a variety of criteria, either in descending or ascending order.

$\star\star$*Line* 4 $\Rightarrow$ This is a reminder to change any of the user parameters defined with the RENAME command described in Section 6.5. At this particular time, none were changed.

$\star\star$*Line* 5 $\Rightarrow$ One may input a list of problems to be solved.

$\star\star$*Line* 6 $\Rightarrow$ Each problem may be specified by name or by number.

$\star\star$*Line* 7 $\Rightarrow$ Before beginning execution, a list of the current problems to test is printed. This may be suppressed.

$\star\star$*Line* 8 $\Rightarrow$ Before beginning the actual minimization of the test problems, a pause occurs to give the user the chance to change any parameters, or to give any further commands which he may have forgotten.

[3]In this document, VMS will refer to the operating system provided by Digital Equipment Corporation as I have run it on a VAX 11/780 at the University of Arizona and on micro-Vaxes in the School of Business at Dalhousie University. Similarly, NOS will refer to Control Data's operating system as I am familiar with it at Concordia University on a Cyber 825/835 and on a Cyber 170/730 at Dalhousie University; UNIX will mean Berkeley 4.2 or 4.3 UNIX as running on a VAX 11/785 in the Department of Mathematics, Statistics and Computing Science at Dalhousie University, and CMS will refer to IBM's VM/CMS. Although I will attempt to be accurate with respect to statements made regarding these systems, I disclaim responsibility for misunderstandings I may have regarding the operations of one or more of these systems.

***Line* 9 ⇒ Here the command specifies that the list of problems is to be echoed by name, rather than by number. The pause is then repeated.
***Line* 10 ⇒ Any positive response, such as "yes", "OK" or "run" will cause execution to begin.
***Line* 11 ⇒ After the problems have been solved, a brief summary appears.
***Line* 12 ⇒ We now begin the next round of testing, first with a request for a new title.
***Line* 13 ⇒ Commands may be abbreviated; MED is short for MEDIUM. This has the effect of increasing the amount of output to the terminal.
***Line* 13 ⇒ DOSUMMARY indicates that the summary output is to be repeated, without recomputation of the solutions. More output will occur because of the command MED.
***Line* 14 ⇒ At the MEDIUM level, we see the values of all parameters which can effect the execution of the test problems. It is dated so that it may be kept for future reference.
***Line* 15 ⇒ The summary of the same run appears; the information given is somewhat different. One can also see the problems are indeed sorted in descending order by their dimensions.
***Line* 16 ⇒ We now give a new title for the next set of problems to run.
***Line* 17 ⇒ Here we change several parameters.
***Line* 17 ⇒ ACC specifies the accuracy desired in the solution.
***Line* 17 ⇒ MIN indicates that we wish to go back to briefer output; that is more suitable for this demonstration.
***Line* 17 ⇒ ITERS (see line 18) says that we want to change some of the parameters concerned with output of iterates to the terminal. Currently there is no such output. PR=10 then indicates that every 10th iterate should be printed interactively on the terminal.
Line* 17 ⇒ DER=FIRST turns on the code which will compute both finite differences and analytic derivatives and compare them, seeking to find errors in the coding of the derivative evaluation. FIRST indicates that this is just to be done on the first call to compute a derivative value. EVTRAC indicates that the computations done during this testing should be traced. All trace output goes by default to a file called TRACES. @Line* 18 ⇒ An error occured on the previous line of input. All that needs to be retyped is the string which is indicated as being in error. The remainder of the line will be processed, after the error correction as entered. Note that ITERS should have been ITERATES.
***Line* 19 ⇒ Here we change one of the parameters associated with the particular minimization routine BBLNIR. The name QUADIN is defined in the PROLOG file with the RENAME command described in Section 6.5.
***Line* 20 ⇒ RESET causes the list of problems to be solved to be emptied. Then AD= causes problems 1 and 12 to be added to the list. Note that in line 6, we omitted the keyword ADD. The same could be done here.
***Line* 21 ⇒ Here we change our mind, and decide not to include problem 12 after all. It is for such that step (5) was intended.
***Line* 22 ⇒ Here we see the iterates generated as a result of the commands ITERATES and PR in line 17.
***Line* 23 ⇒ When derivative testing is on, a statement appears of the number of figures of agreement between the analytic and finite difference calculations.
***Line* 24 ⇒ We now assume a more experienced user, who prefers something briefer than 5 requests for input. BRIEF turns off the 5 requests; NOREQ turns off the requirement to always specify a title. We also keep this demonstration fairly short by turning off printing of the iterates and derivative testing. Note that execution of a set of tests begins as soon as the commands on the line have been read.
***Line* 25 ⇒ One can control a number of trace flags. Here we turn on just the first one. The effect of the trace flags depends on the use made of them by specific minimization algorithms. We also specify that trace output should appear directly on the terminal.
***Line* 25 ⇒ REVERSE specifies that reverse communication is to be used.

***Line* 26 ⇒ This is the trace output produced by the program BBLNIR in response to turning on trace 1.
***Line* 27 ⇒ Here we alter the brief mode so that several lines of input may be given before executing the test problems.
***Line* 28 ⇒ Turn off the trace flag. We have shown what it does.
***Line* 29 ⇒ Go back to forward communication, the more normal case.
***Line* 30 ⇒ Change to using a different minimization algorithm, namely CONMIN due to Shanno and Phua [16]. Note that it does not support reverse communication, which is why we changed back to forward.
***Line* 31 ⇒ Limit the memory available for the minimization, and set the variable NMETH required by CONMIN.
***Line* 32 ⇒ Clear the problem list, put in problem 64, and reset the accuracy requirement. Note that an accuracy request of the form ACC=8 will be understood to mean ACC=1.e-8.
***Line* 33 ⇒ Now we are ready so we begin execution of the test with the explicit command RUN.
***Line* 34 ⇒ Go back to subroutine 1, which happens to be BBLNIR.
***Line* 35 ⇒ Now define a new problem, i.e. one which is not predefined in the PROLOGUE file. USE=12 says that the new problem is defined to be the same as problem number 12, at least to start. The SCALE value for the new problem is to be changed to 2 however, and the new problem will be named NEWHELIX. Note that it is automatically assigned the first free number.
***Line* 36 ⇒ VERIFY mode causes all commands to be echoed, with abbreviations expanded.
***Line* 36 ⇒ The command SEPARATOR changes the character which is recognized as separating commands; from now on it a "|".
***Line* 37 ⇒ This is the echo generated by VERIFY.
***Line* 38 ⇒ This add command with "/F"causes the newly defined problem NEWHELIX to be added to the list to be solved, along with *all* test problems which use the test *function* ROSENB.
***Line* 39 ⇒ We now define 2 accuracy levels. The set of test problems will be done twice, first with an accuracy of 10^{-6}, then with 10^{-30}.
***Line* 40 ⇒ Here we change the termination criterion. We turn off the first three criteria, and turn on the fourth. This can be either that $|f(x_i)| \leq \epsilon$ or that $|f(x_i)| \leq \epsilon \times |f(x_0)|$. The command FABSOLUTE specifies the first of these; i.e. it should *not* be relative to the initial value of f. Here we have used the new separator defined in line 36.
***Line* 41 ⇒ This will change the format of the output summary.
***Line* 42 ⇒ Here we see the result with ACC=1e-6. Observe also that the distance DX to the nearest known analytic solution is given. Note also the long precision for f resulting from the command LONG in line 41.
***Line* 43 ⇒ Here we see the result with ACC=1e-30. Note that one problem is flagged with an error, since the desired termination criteria was not achieved within the limit on the number of function evaluations.
***Line* 44 ⇒ This says that we are ready to quit.
***Line* 45 ⇒ And this makes sure that we meant it.

```
1. (1) TITLE LINE? (OR "END")
   Demonstration of TestPack
2. (2) CONTROL INFORMATION?
3. sort=dimension;descending
4. (3) USER PARAMETERS?
   ␣
5. (4) WHICH PROBLEMS?
6. 1,helix,30,47,149
   ␣
7. PROBLEMS:      47 149  30  12   1
```

```
 8. (5) READY?
 9. check=names
    ␣
    PROBLEMS:   CHNRSNGM CHNRSH10 BROWND   HELIX    ROSENB
    (5) READY?
10. y
11. PROB    #_IT #_FNC #_GRD FUNCTION GRADIENT ER
      47      37     40     40 0.10E+01 0.87E-03  0
     149      33     35     35 0.34E-10 0.47E-04  0
      30      29     31     31 0.86E+05 0.42E-02  0
      12      29     33     33 0.13E-10 0.68E-04  0
       1      40     48     48 0.82E-20 0.39E-08  0
    TOTALS:  168    187    187;   RATIO =   1.11
    ␣
     0 ERRORS
12. (1) TITLE LINE? (OR "END")
13. med;dosummary
14. Demonstration of TestPack
    TEST BEING EXECUTED AT 10:47 A.M., WEDNESDAY, JUNE  3, 1987
    NORM=  2 (EUCLIDEAN )
    TERMINATION TESTS ON:
      GRADIENT  ->F; STEPSIZE  ->T; SHANNOXG  ->T; FUNCTION  ->F;
    RELATIVE TEST ON F IS T; RELATIVE TEST ON G IS T
    DERV=  1 (ANALYTIC  )
    MEMORY:  5350 D.P. VALUES; SUBROUTINE:  1 (BBLNIR   ); ACCURACY =  0.100E-04
    FACTOR:       1.0000
    USER PARAMETER VALUES:
     0.000E+00  0.100E+01  0.100E+01  0.100E+01  0.100E+01  0.200E+00  0.100E+01
     0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.100E+01  0.000E+00  0.100E+01
     0.100E+01  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00
    ␣
15. RUN SUMMARY
    PR# FN#  NAME  DIM ITS FNS GRS   FVALUE  GVALUE  MSECS  FSECS ER NUP RST FRC
     47  9 CHNRSNGM 25  37  40  40 0.10E+010.87E-03  4.193  0.239  0 101   0   0
    149  9 CHNRSH10 10  33  35  35 0.34E-100.47E-04  0.867  0.077  0 241   0   0
     30 36 BROWND    4  29  31  31 0.86E+050.42E-02  0.515  0.282  0 533   0   0
     12  2 HELIX    3  29  33  33 0.13E-100.68E-04  0.239  0.027  0 667   0   0
      1  1 ROSENB   2  40  48  48 0.82E-200.39E-08  0.215  0.023  0 890   0   0
           TOTALS:     168 187 187 (RATIO   1.11)    6.029  0.648
    ␣
      0 PROBLEMS WERE FLAGGED WITH ERRORS.
    (1) TITLE LINE? (OR "END")
16. Continuing the Demonstration
    (2) CONTROL INFORMATION?
17. acc=1e-12;min;iters;pr=10;der=first;evtrace
18.  DECODE ERROR : UNRECOGNIZED WORD BEFORE SEPARATOR.
     STRING IN ERROR IS : iters.
    ERROR CORRECTION IS :
    iterates
    (3) USER PARAMETERS?
19. quadin=2
    (4) WHICH PROBLEMS?
20. reset;ad=1,12
    ␣
    PROBLEMS:   HELIX    ROSENB
    (5) READY?
21. drop 12
```

```
PROBLEMS:   ROSENB
(5) READY:
y
22. PT #  0; F=  24.200000     (#  1) !!G!!=.23E+03
POINT X: -1.2       1.0
PT # 10; F=  2.7121466     (# 12) !!G!!=.49E+01
POINT X: -.65       .41
PT # 20; F=  .61311742     (# 25) !!G!!=.17E+01
POINT X:  .22      0.41E-01
PT # 30; F= 0.93987805E-02(# 38) !!G!!=.60E+00
POINT X:  .90       .82
PT # 40; F= 0.81727526E-20(# 48) !!G!!=.39E-08
POINT X:  1.0       1.0
PT # 42; F= 0.84717220E-31(# 50) !!G!!=.35E-14
POINT X:  1.0       1.0
PROB    #_IT #_FNC #_GRD FUNCTION GRADIENT ER
   1      42    50    50 0.85E-31 0.35E-14  0
TOTALS:   42    50    50;   RATIO =   1.19
23. GRAD TEST:  0.506E-09 9.323     1     1     1
␣
 0 ERRORS
(1) TITLE LINE? (OR "END")
24. brief;noreq;noevtrac;pr=0;der=analytic
COMMANDS; IMMEDIATE EXECUTION.
25. traces=T;traceun=terminal;reverse
26. **** BBLNIR ENTERED AND INITIALIZATION COMPLETE ****
␣
DIMENSION  =     2                    MEMORY AVAILABLE IS    5344
ACCURACY REQUESTED =   0.1000000E-11   STATUS ON ENTRY IS  1
EXPECTED REDUCTION IN F EQUALS INITIAL FUNCTION VALUE OF    24.20000
INTEGER CONTROL SETTINGS  METH  QUADIN ALPIS1 SCGAMM HTEST  UPDATT
                            0      2      1      1      1      1
REAL CONTROL VALUES  RO =    .2000000     BETA =    1.000000
LOGICAL CONTROL VALUES   FQUAD  SCDIAG SHANNO FROMRS
                           F      F      F      F
                         FORCEF FLETSC  RELF   RELG
                           T      F      T      T
␣
THE FOLLOWING HAVE BEEN SET DURING INITIALIZATION
LMSTQN (F); CG (F); USESHN (T); ONEUPD (F)
MACHINE RELATIVE ACCURACY EPS = 0.11E-17
TERMINATION RELATIVE TO   24.20000    (F);   232.8677    (G)
STORAGE OF      3 SUFFICIENT; USING QN ALGORITHM.
␣
ON ENTRY, VALUES WERE DEFINED FOR F AS    24.200000000000000
              AND FOR THE NORM OF G AS             232.86769
PROB    #_IT #_FNC #_GRD FUNCTION GRADIENT ER
   1      42    50    50 0.85E-31 0.35E-14  0
TOTALS:   42    50    50;   RATIO =   1.19
␣
 0 ERRORS
COMMANDS; IMMEDIATE EXECUTION.
27. noimmediate
COMMANDS; THEN "RUN".
28. traces=F
COMMANDS; THEN "RUN".
```

```
29. forward
    COMMANDS; THEN "RUN".
30. use=CONMIN
    COMMANDS; THEN "RUN".
31. memory=600;nmeth=0
    COMMANDS; THEN "RUN".
32. reset;add=64;acc = 8
    COMMANDS; THEN "RUN".
33. run
    PROB    #_IT #_FNC #_GRD FUNCTION GRADIENT ER
      64       3     9     9 0.74E+01 0.15E-07  0
    TOTALS:    3     9     9;   RATIO =   3.00

    ␣
     0 ERRORS
    COMMANDS; THEN "RUN".
    reset
    COMMANDS; THEN "RUN".
34. use=1
    COMMANDS; THEN "RUN".
35. defin;use=12;scale=2;prob=NEWHELIX;end
    PROBLEM ASSIGNED PROBLEM #      152.
    COMMANDS; THEN "RUN".
36. verify;separator=|
37. SEPARATO=|
    COMMANDS; THEN "RUN".
38. add=newhelix,rosenb/F
    ADD     =newhelix,rosenb/F
    COMMANDS; THEN "RUN".
39. ac=6,30
    ACCURACY=6,30
    COMMANDS; THEN "RUN".
40. term=FFFT|fabsolute
    TERMINAT=FFFT
    FABSOLUT
    COMMANDS; THEN "RUN".
41. long
    LONGF
    COMMANDS; THEN "RUN".
    run
    START   =RUN
42.   NAME   #IT #FN      FUNCTION (GRADIENT)->NORM      MIN       DF        DX   SOL
    NEWHELIX  63  64 0.74147283375302971E-06 0.16E-02 0.10E-03 0.00E+00 0.18E-01  4
    ROSNB5    26  33 0.27659608374405930E-06 0.43E-02 0.23E-02 0.00E+00 0.12E-02  3
    ROSNB4    49  62 0.41411629128050669E-06 0.89E-02 0.35E-02 0.00E+00 0.14E-02  3
    ROSNB3    34  43 0.21860569091466111E-07 0.25E-02 0.12E-02 0.00E+00 0.31E-03  3
    ROSNB2    42  47 0.58344619383091569E-06 0.32E-01 0.15E-01 0.00E+00 0.58E-03  3
    ROSENB    36  44 0.65645901289590805E-06 0.26E-01 0.12E-01 0.00E+00 0.12E-02  3
    ROSNB6    29  37 0.63144006916217924E-07 0.11E-01 0.50E-02 0.00E+00 0.10E-03  3
    TOTALS:  279   330   330;   RATIO =   1.18

    ␣
     0 ERRORS
43.   NAME   #IT #FN      FUNCTION (GRADIENT)->NORM      MIN       DF        DX   SOL
    NEWHELIX 149 150 0.22414746025301112E-24 0.11E-16 0.42E-17 0.00E+00 0.64E-06  4
    ROSNB5    32  39 0.54542937878154140E-35 0.95E-16 0.43E-16 0.00E+00 0.21E-17  3
    ROSNB4    55  68 0.29112575211108970E-31 0.20E-14 0.78E-15 0.00E+00 0.37E-15  3
    ROSNB3    40  49 0.34089336173846338E-34 0.25E-15 0.11E-15 0.00E+00 0.44E-17  3
    ROSNB2    48  53 0.41560895817107879E-32 0.28E-14 0.13E-14 0.00E+00 0.26E-16  3
```

```
   ROSENB     42   50 0.84717219586925800E-31 0.35E-14 0.13E-14 0.00E+00 0.63E-15   3
   ROSNB6     34   42 0.00000000000000000E+00 0.00E+00 0.00E+00 0.00E+00 0.00E+00   3
   TOTALS:   400    451    451;    RATIO =    1.13
   ␣
    1 ERRORS
   COMMANDS; THEN "RUN".
44. quit
   QUIT
   FOR SURE(Y)?
45. y
   ␣
   TOTAL TIME TAKEN WAS       27.5465 SECONDS.
   ␣
   STOP RUN OK
```

5 Test Problems

A fundamental part of any attempt to evaluate minimization algorithms is the specification of a set of test problems. In this section, we will not describe actual test problems which are available in TP, but rather the mechanisms by which test problems may be defined and subsequently accessed. This is in keeping with our purpose of illustrating the kind of things one can do in a general purpose package. We observe that it will be equally easy to define and test a single problem, or to select and test a collection containing any number of problems. Problem definition can be either static or dynamic. Keep in mind that a predefined set of problems is provided with TP, so that it is not necessary to understand the problem definition mechanisms unless new problems are to be added.

5.1 Test Functions.

Specifying a test function is a key part of constructing a test problem. TP currently contains over 65 distinct test functions, including those from the Argonne study [12], plus most of those which could be gleaned from the literature where numerical testing has been done. Each function has a unique name and number which can be used to access it, as will be discussed in Section 5.2. All functions, along with their corresponding gradient values, are evaluated in a single subroutine ZZFNS which is distributed as part of TP. Several of these functions have arguments; these can easily be set by the user, as in `F7ARG = 13.7`. One of several nonlinear scalings can be applied to any given test function; this both reduces the number of functions required, and gives flexibility in adapting test functions to examine particular characteristics of minimization algorithms. Both the existence of ZZFNS and the nonlinear scalings contribute to the consistency of testing, for exactly the same code, scaling and arguments for the test functions can be used by all. For complete details on the functions available, one must consult the document [2].

5.2 The Problem Definitions.

The distribution tape for TP includes the file PROLOGUE, which contains a collection of test problems; this is the static approach to problem definition. Since this file is distributed as part of TP, we have again furthered the objective of consistency, by having a fixed, well-defined and permanent set of test problems. Each test problem uses a particular test function, but other data must be included as well. A typical problem definition looks like:

```
DEFINITION; PROBLEM = HELIX, 12; FUNCTION = HELIX, 2
  DESCRIPTION = "This is the Helical Valley Function by Fletcher & Powell."
```

```
MAX = 150; N = 3; X = -1.0, 0.0, 0.0
SOLUTION = 3, 0, 1, 0, 0
```

The meaning of this definition is quite simple. Later in this section we will see how more complicated data can be specified.

The keyword DEFINITION is mandatory to signal the start of the definition of a new problem. The field "PROBLEM = HELIX, 12" specifies a name and number for this problem, either of which may be used to access this specific problem when TP is run. Each test problem uses one of the test functions, in this case the one whose number is 2 and whose name is HELIX. Thus the test function for the problem HELIX is known by the same name HELIX (which need not be and is not always the case). The string "This is the Helical Valley Function by Fletcher & Powell." provides a longer description of the problem; this will appear with all output for information.

The next field "MAX = 150" defines the maximum number of points at which the function may be evaluated. Thus, if an algorithm has not succeeded in minimizing the HELIX function within 150 function evaluations, it is considered to have failed, and execution of that test problem terminates. The field "N = 3" specifies the dimension of the problem, so there are 3 variables in the HELIX problem. More will be said about the dimension of problems later. The field "X=" specifies the starting point which will be provided to each test algorithm. Thus the initial guess at the minimum for the HELIX function is taken to be $x_0 = (-1,0,0)$, assuming FACTOR is 1 (see Section 7). Finally, the known solution to this problem is given: taking the fields in order, for the problem of dimension 3 (the only one here, although some problems have varying dimension), the solution has $f(x^*) = 0$, with $x^* = (1,0,0)$. The definition of a problem ends when the keyword END is found, or when the keyword DEFINITION is encountered, in which case definition of another problem begins.

For most of the problems defined in TP, the definition of the problems is as simple as that of HELIX; what we have described to this point is very standard. Of course, it does not really matter, since the set of test problems is provided as part of TP, and is not something that a user must provide. However, it is worth noting this structure, for TP allows the user to add his own test problems.

5.3 Accessing Problems.

With a collection of test problems defined, the question becomes how one may access them, i.e. how one may specify that a certain problem is to be run.

As we have seen in Section 4, this is quite easily done. TP provides three keywords, namely ADD, DROP and RESET, along with the related command SORT. TP maintains a list of problems to be executed. Initially this list is empty. Whenever the keyword ADD occurs, the set of problems following ADD is added to the list. Duplicates are of course removed. For example, ADD=3,HELIX,12 would add the two problems 3 and 12 to the list; HELIX is the same as 12. To remove one or more problems from the list, the command DROP is used. Alternatively, one may start constructing a new list by first specifying RESET, which empties the current list, and then using ADD to rebuild a list. The list is maintained from one run to the next.

The command SORT is used to control the order in which the specified problems are to be executed. The default is that they are executed in the order given, with each ADD adding the given problems onto the end of the list. This can be altered so that the problems are sorted in order of their dimension, with the command SORT=DIMENSION, or by problem name, and so on.

In the next part of this section, we will see a mechanism by which TP facilitates access to collections of problems.

5.4 Defining Problem Sets.

It is common to wish to test an optimization algorithm on a collection of test problems. We will look here at how TP facilitates this process.

First, consider the dimension of the test problem. In many standard test functions, the dimension can vary, and one sometimes wishes to test those problems with a variety of values for the dimension. In this case, one may define a set of dimensions to be used. For example, if "N = 6, 8, 13" is specified in the definition of problem P, then whenever one selects problem P, three tests will be run, one for each of the dimensions 6, 8 and 13. One must ensure of course that the dimensions given are suitable for the particular test function chosen.

It is appropriate here to introduce a notational convenience provided in TP. Suppose one wished to use dimensions 4, 7, ..., 100. This would make quite a long list to enter. On the other hand, the values follow an obvious pattern. This set of dimensions could be specified by setting N = 4:100:3, meaning that the dimension should vary from 4 to 100, incrementing by 3, just as in the common DO loop. The character : here signals that these three values specify a sequence, and *not* just the three dimensions 4, 100 and 3. Alternatively, a sequence of dimensions can be constructed geometrically instead of arithmetically, by using the character * in place of :. Hence, N = 4*128*2 yields test dimensions 4, 8, 16, 32, 64 and 128. The * and : notations can be used anywhere a list of values is acceptable with a keyword.

In summary, one simple mechanism has been provided which allows a user to define several tests to be run. The same technique can be used to define several tests using several values for the scaling or for the function arguments. Now we wish to describe two different ideas.

There are three types of named entities: *problems*, *functions* and *groups*; groups will be explained shortly. To indicate that a particular *problem* P is to be solved, the command ADD=P is given; P may be the name or number of a problem. If problems P, Q and R are to be added, the command would simply be ADD = P, Q, R. However, quite large sets of problems are sometimes desired.

Consider the case where there are several test problems, all of which are based on the same test function, say FNC. To test all of these, the command required is just ADD = FNC, where the *function name* is specified in place of a problem name; it is equivalent to ADD=P1,P2,...,Pt, where P1,...,Pt is the entire set of problems based on function FNC. Alternately, if the collection of problems desired is not related in this way, one may define a group of problems. A natural place to do this is in the PROLOGUE file. The commands "DEFIN; GROUP = NEAT; ADD=P, Q, R;END" define a set of problems consisting of P, Q and R, which can be referred to by the name NEAT. To solve these problems, one just enters "ADD = NEAT".

We should note that names are not required to be unique across name-types, so the same name could occur as a problem, as a function and as a group (although in practical situations the names will generally be different). To identify which kind of entity is intended when a conflict is possible, one simply adds a suffix to the name. Thus the command "ADD=ROSENB/F" would cause all of the problems using the function named ROSENB to be added to the current list of problems to be solved, whereas ADD=ROSENB would just add the single problem named ROSENB. One may similarly add the obvious suffices "/P" or "/G" to stipulate a problem or group name, resp. If no suffix is added, then TP first tries to find a problem of the given name; if it fails it looks next for problems using a function of that name and finally, if both have failed, for a group with the given name. The suffix is only needed to resolve ambiguities and to override this natural search order.

We have noted that the file PROLOGUE is part of the distributed test package. In addition to containing definitions of problems, this file can also contain group specifications. If a group includes a problem which itself specifies more than one test (as in the discussion of dimensions above), then all such tests are part of the group.

Thus we see that there is a way of taking a current list of problems to be tested and adding to it either a single problem, selected problems, or all problems using a particular function or all problems in a predefined group. Of course the same technique applies for dropping entries from the current list of test problems.

5.5 Dynamic Definition.

So far we have described problems which are predefined and provided in a file distributed with TP. Of course, it is highly likely that some users will want to define their own problems. This may certainly be done by adding definitions to the file PROLOGUE. This is not completely satisfactory though, for on most systems that cannot be done while TP is running, but must be done in advance, independently of TP, using some sort of file editor. So TP provides a way of defining new problems or groups during execution. Indeed, one may even specify modifications to problems predefined in the file PROLOGUE.

The actual commands used to do this are exactly the same as those used to define a problem in the file PROLOGUE. However, problems defined in this way have what could be termed a semi-permanent status. They will be saved in a (direct access unformatted) file called DAUF (see Section 6.4), and they will remain there after execution of TP terminates. Upon the next invocation of TP, this file DAUF will normally be available to TP, and so any problems which were saved in DAUF will be accessible. However, should the PROLOGUE file be modified, the file DAUF will be reconstructed, and only those problems defined in PROLOGUE will appear in the new copy of DAUF. This would not automatically include problems temporarily stored in DAUF. Exactly the same comments apply to groups which are defined interactively.

We should note one particular feature which makes it easy to define problems dynamically. In the PROLOGUE file, all information is provided which is necessary to define a test problem. This generally requires definition of several fields. A common situation is that one wishes to test one of the problems provided, but with a small change. In this case, one can define a new problem by modifying an old one. For example, the statements `DEFINITION; PROBLEM= MODHELIX; USE=HELIX; XO= -3,-1,-1; END` would define a new problem MODHELIX which would differ from HELIX only in the initial guess at the solution. Or, if an algorithm failed to minimize a particular function within the specified limit of function evaluations, say in problem TOUGH, one could repeat the problem with a higher limit, as in `DEFIN;PROB=temp;use=TOUGH;max=400;end;ADD=temp`. It is expected that this feature will be quite useful.

5.6 Problem Definition Sheets.

In an attempt to be as complete as possible regarding the test problems available with TP, especially their history, we have included a collection of what we will call PDS's, for Problem Definition Sheets. There is one for each test function, and we have attempted to include all information pertinent to that test function on its PDS. We would now like to look at this information in some detail. For ease of discussion, we have included three sample PDS's, for HELIX, KOWOSB and WATSON in Figures 1, 2 and 3.

Each PDS includes a definition of the function of course, and includes any data values which are needed to define the function, as for KOWOSB. Formulae for the components of the gradient are also included.

Each PDS is headed by the name of the function, as it is known to TP, along with the dimension n of the function, which can either be fixed, as in HELIX, or variable. In the latter case, the limits on the dimension are stated, so we see that WATSON can only be used for n between 2 and 31.

In order to put TP in correct historical perspective, references to the source of the function are given (with complete bibliographic references appearing in [2]). In the "source" one can

Function name	*Dimension*	*Source*	*References*
HELIX	3	[5]	[12,4,6,8]

FUNCTION: $f(x) = 100\left((x_3 - 10T)^2 + (r-1)^2\right) + x_3^2$

where $r = \sqrt{x_1^2 + x_2^2}$

and $T = \begin{cases} \frac{1}{2\pi}\tan^{-1}\frac{x_2}{x_1}, & x_1 > 0 \\ \frac{1}{2} + \frac{1}{2\pi}\tan^{-1}\frac{x_2}{x_1}, & x_1 < 0 \end{cases}$

GRADIENT: $g_1(x) = 200\left(\frac{10x_2}{2\pi r^2}(x_3 - 10T) + x_1(1 - \frac{1}{r})\right)$

$g_2(x) = 200\left(\frac{-10x_1}{2\pi r^2}(x_3 - 10T) + x_2(1 - \frac{1}{r})\right)$

$g_3(x) = 200(x_3 - 10T) + 2x_3$

ANALYTIC SOLUTION: $f^* = 0$ at $x^* = (1,0,0)$

Problems Defined		
Number	*Name*	*Data*
#12	HELIX	$x_0 = (-1,0,0)$

Computed Solutions							
Number	Iterations	Func. Count	Acc Req.	$f(\hat{x})$	$\|g(\hat{x})\|$	$\min_i\{\|g_i(\hat{x})\|\}$	$\|\hat{x} - x^*\|$
#12	35	39	1_{-15}	$.92969_{-37}$	$.11_{-16}$	$.15_{-35}$	$.35_{-18}$

Description: This function has a helical valley. See [12] #7, [7] #2, [8] #34 and [9] #10.

Figure 1: PDS for the HELIX Function

Function name	Dimension	Source	References
KOWOSB	4	[10]	[12,11]

FUNCTION: $$f(x) = \sum_{i=1}^{11} \left\{ y_i - \frac{x_1(u_i^2 + u_i x_2)}{u_i^2 + u_i x_3 + x_4} \right\}^2 \equiv \sum_{i=1}^{11} r_i^2$$

DATA

i	y_i	u_i
1	.1957	4.0000
2	.1947	2.0000
3	.1735	1.0000
4	.1600	.5000
5	.0844	.2500
6	.0627	.1670
7	.0456	.1250
8	.0342	.1000
9	.0323	.0833
10	.0235	.0714
11	.0246	.0625

GRADIENT:
$$g_1(x) = -2 \sum_{i=1}^{11} \left\{ r_i \frac{u_i^2 + u_i x_2}{u_i^2 + u_i x_3 + x_4} \right\}$$
$$g_2(x) = -2x_1 \sum_{i=1}^{11} \left\{ r_i \frac{u_i}{u_i^2 + u_i x_3 + x_4} \right\}$$
$$g_3(x) = 2x_1 \sum_{i=1}^{11} \left\{ r_i u_i \frac{u_i^2 + u_i x_2}{(u_i^2 + u_i x_3 + x_4)^2} \right\}$$
$$g_4(x) = 2x_1 \sum_{i=1}^{11} \left\{ r_i \frac{u_i^2 + u_i x_2}{(u_i^2 + u_i x_3 + x_4)^2} \right\}$$

ANALYTIC SOLUTION: NONE (*but see below*)

Problems Defined

Number	*Name*	*Data*
#31	KOWOSB1	$x_0 = (.25, .39, .415, .39)$; [12]#15
#102	KOWOSB2	$x_0 = (0, 0, 0, 0)$; [11]"Enzyme"
#103	KOWOSB3	$x_0 = (.25, .4, .4, .4)$; [11]"Enzyme"

Computed Solutions

Number	Iters	Fnc. Ct.	Acc Req.	$f(\hat{x})$	$\|g(\hat{x})\|$	$\min_i\{\|g_i(\hat{x})\|\}$	$\|\hat{x} - x^*\|$
#31	36	41	1_{-12}	$.30750\,56038\,49237_{-3}$	$.26_{-12}$	$.11_{-13}$	-1
#102	57	58	1_{-11}	$.30750\,56038\,49237_{-3}$	$.48_{-11}$	$.87_{-12}$	-1
#103	33	39	1_{-16}	$.30750\,56038\,49237_{-3}$	$.38_{-16}$	$.45_{-17}$	-1

Description: This is a least squares data fitting problem discussed in [10]. Note that there is an additional solution at ∞: $f \approx .102734_{-2}$ at $\hat{x} \approx (+\infty, 4.07, -\infty, -\infty)$.

Figure 2: PDS for the KOWOSB Function

find, as far as we know, the earliest definition of the problem. Thus HELIX first appeared in the well known paper of Fletcher and Powell [5], whilst the least squares problem which leads to the WATSON problem is described in a paper by Osbourne [13]. The "references" are to papers where a significant amount of testing took place using this function. It would be appreciated if we could be informed of any inaccuracies or incompleteness in this information.

Known analytic minima are specified. This includes both the point at which each minimum occurs, along with the function value at that point. In KOWOSB, we see that some solutions are in fact unbounded.

For each test function, there may be just a single test problem defined, as in HELIX, or several may appear. In WATSON, there are three test problems. A 6-dimensional problem may be run by referring to TP problem number 61, whose name is WATSON6, or one may choose a 9- or 12-dimensional problem. For KOWOSB, 3 tests are defined, each with a different starting point. We note that these 3 problems come from different sources; number 31 has a starting point which appeared in the Argonne test set [12], while the other two starts were used in Kowalik and Osbourne Citekoos, where the problem was referred to as the "Enzyme" problem. We have attempted to include, for each test function, all different problems using that function which have appeared in the literature.

For each test problem, we have given the best approximate solution $\hat{x}$ which we have found, using either a quasi-Newton or conjugate gradient code. It is hoped to refine these at some future time using a Newton code. Although the gradient is of course 0 at each minimum, we have also tabulated the value of the norm and least component of the gradient at $\hat{x}$. This information is often useful when running tests, as it gives some idea of the scaling of the problem and of a reasonable stopping criterion. The notation r_t is a shorthand for $r \times 10^t$. Many problems are like HELIX with a single solution, but some have several, as for WATSON, where the solution depends on n, and so an approximate solution is given for each. Some indication of the difficulty of each problem is shown by including the number of iterations and function evaluations needed to solve the problems to the requested accuracy. The distance to the nearest analytic solution is given (or -1 is shown if none is known).

Finally, each PDS includes a "Description". Here, a brief statement is made of the mathematical origin of the problem. Thus, both WATSON and KOWOSB originate as least squares data fitting problems. Others come from the solution of integral equations or ordinary differential equations. Special properties are noted, such as being quadratic or a sum of squares. We also indicate things like known misprints in the literature; these can cause terrible confusion when looking at comparative test problems.

In summary, the PDS's contain all information needed to run the tests, to evaluate the results and to make comparisons to other test results.

6 Design

In writing software such as this, several decisions have to be made. We would like to comment on a few of these here, in particular on those which have a significant effect on the potential usefulness of such a package.

6.1 Language

The first decision to justify is of course the language used, for there is certainly a wide choice available today. The most obvious reason for choosing FORTRAN is simply that, in spite of many efforts to the contrary, it is still by far the most widely used programming language of the scientific community. Maybe that should not be so, but it is. We have used the 1977 Standard [14], even though at the time this project was begun, that standard was not widely

Function name	*Dimension*	*Source*	*References*
WATSON	$2 \le n \le 31$	[13]	[12,6,7,11]

$$\text{FUNCTION:} \quad f(x) = \sum_{i=1}^{29} \left\{ \sum_{j=2}^{n} (j-1) x_j \left(\frac{i}{29}\right)^{j-2} - q_i^2 - 1 \right\}^2 + x_1^2 + (x_2 - x_1^2 - 1)^2$$

$$= \left(\sum_{i=1}^{29} r_i^2 \right) + x_1^2 + (x_2 - x_1^2 - 1)^2.$$

$$\text{Here} \quad q_i = \sum_{j=1}^{n} x_j \left(\frac{i}{29}\right)^{j-1}$$

$$\text{GRADIENT:} \quad g_1(x) = 2x_1[1 - 2(x_2 - x_1^2 - 1)] - 2\sum_{i=1}^{29} r_i q_i$$

$$g_2(x) = 2\left[\sum_{i=1}^{29} \{ r_i (1 - \frac{2i}{29} q_i) \} + (x_2 - x_1^2 - 1) \right]$$

$$k \ge 3, \quad g_k(x) = 2\sum_{i=1}^{29} \left[r_i (\frac{i}{29}^{k-2} ((k-1) - \frac{2i}{29} q_i) \right]$$

ANALYTIC SOLUTION: NONE

Problems Defined		
Number	*Name*	*Data*
#61	WATSON6	$n = 6, \quad x_0 = (0,0,\ldots)$
#127	WATSON9	$n = 9, \quad x_0 = (0,0,\ldots)$
#128	WATSON12	$n = 12, \quad x_0 = (0,0,\ldots)$

Computed Solutions							
Number	Iters	Fnc. Ct.	Acc Req.	$f(\hat{x})$	$\|g(\hat{x})\|$	$\min_i\{\|g_i(\hat{x})\|\}$	$\|\hat{x} - x^*\|$
#61	57	60	1_{-11}	$.22876\,70053\,55243_{-2}$	$.43_{-11}$	$.34_{-12}$	-1
#127	124	126	1_{-09}	$.13997\,60138\,09902_{-5}$	$.70_{-09}$	$.24_{-10}$	-1
#128	185	189	1_{-09}	$.26642\,48766\,90175_{-8}$	$.28_{-09}$	$.37_{-12}$	-1

Description: The solutions are given in [6]. The problem arises from using $p(t) = \sum_{i=1}^{n} x_i t^i$ to obtain a least squares solution to $y' = 1 + y$ with $y(0) = 0$ on $[0,1]$. Hessians for $n = 6$ and $n = 9$, respectively, are about 8.6×10^4 and 1.7×10^9 at x^*. The polynomial form chosen for the solution is not really suitable so the problem is fairly difficult. It is mildly ill-conditioned and has an even eigenvalue distribution. Note that the problem in [11] is slightly different. WATSON6 is #6 in [6]. WATSON6 and WATSON9 appear as [7], #5. WATSON6, WATSON9 and WATSON12 appear as [12] # 20.

Figure 3: PDS for the WATSON Function

used, and few vendors had compilers which would accept it. A major reason was that it would have been extremely difficult (or inconvenient at the very least) to write efficient portable code for all of the character processing done within TP, especially decoding the user language. As well, FORTRAN 77 does provide at least the basic control structure IF – THEN – ELSE.

Pascal has two main shortcomings. One is the lack of an acceptable standard, for we did not consider the Wirth document [17], which does not even allow passing of varying length arrays to a procedure, to be an acceptable standard. The ISO standard is only recently available, and it has not even been accepted as an American National Standard. A further point is that procedures can not be compiled separately, and repeated recompilation of 20,000 or so lines of code would not be acceptable.

Due to personal preferences, PL/I was never even considered. APL is of course not suitable for a project such as this. Algol 68 has some desirable features but it is little used in North America. ADA could be appropriate in the future, but such is not the case at this time, even though it too has some desirable features.

One point that is worth noting is that the proposals for the next standard for FORTRAN include a number of the most desirable features from, say, Algol 68 or ADA. In fact, the code for TP has been written so that it would not be difficult to implement some of the best of these. We have in mind the definition of "global" variables and the use of external "definition and declaration modules", whatever name they might go by. This further strengthens the case for sticking with FORTRAN.

6.2 Portability

We have restricted ourselves to the ISO[4] defined standard for the FORTRAN language, and Tp is intended to be fully portable. One must nonetheless be quite careful with respect to features whose implementations are not portable across operating systems. A few features in TP are in fact system dependent, but these are small, isolated and carefully documented. Indeed, there are just 7 routines, most with just a few lines of code, into which all dependencies are consolidated. Versions exist of each of these routines which will be suitable for most systems presently in use. The distributed code for TP will normally be ready to run on a particular system, since the appropriate version of these routines will be selected before distribution. Generic versions are also available.

We feel it is appropriate to comment on some of the most important dependencies here, for we have taken care to see that TP should run on a large variety of batch or interactive systems. In what follows, a reference to "the standard" will mean the ISO Standard for FORTRAN, as defined in [14].

- Both single and double precision versions are available. All real constants are defined in parameter statements.

- Direct access unformatted I/O is standard, but the length of unit is processor dependent. Since unformatted direct access data transfer is much more efficient for temporary data storage, we have chosen to use it for the file DAUF (see Section 6.4). However, in order to make the code portable, one must set 4 integer constants in a routine ZZRECL.

- The OPEN statement is standard, but restrictions are imposed by certain operating systems. For example, UNIX forbids opening "the terminal" for I/O, but on NOS it is mandatory in order to be able to accept null lines. Therefore, all file opening is handled by a single routine ZZOPEN.

[4] We comment that what is generally referred to as "ANSI-standard Fortran '77" is in fact an internationally recognized standard [14].

- Character sets are always a problem, for systems such as NOS are still with us. UNIX users expect to function in lower case, whilst that is difficult for most NOS users when within the ISO standard. A system dependent routine ZZLCUC is available to convert lower to upper case. The user may specify that all input is to be converted to upper case by giving the command UPPERCASE; UNIX users need not bother and TP will happily accept lower case. Incidentally, internal dictionaries are in upper case, but conversion to upper case is automatic when searching a dictionary.

- There are 3 routines available which act as intermediaries to suitable system routines for obtaining the current date and time and CPU usage.

- A routine ZZMPAR is available which contains hard-coded constants which define the numerical characteristics of a wide variety of machines. One only need select an appropriate few statements in order to obtain a version suitable for a particular machine. These values are used in the estimation of finite difference derivatives.

6.3 Linking

Let us suppose that you wish to use TP to test a minimization routine. It is clear that you wish neither to have to change the calling sequence of the routine you have devised, nor to modify the source code of TP. Assuming that TP is implemented on your system, the following steps, which in most cases should require very little coding, are required. All changes are isolated to the single routine ZZLINK.

The routine ZZLINK is called by ZZTP (the driver of TP). This is the fundamental link between the test package and a specific user routine; the primary responsibility of ZZLINK is to call the user's test algorithm MIN. That may involve breaking up two work vectors IWORK and RWORK, which are provided in ZZLINK, into whatever smaller working vectors are appropriate for MIN. Specific instructions appear in the descriptive section of ZZLINK; there are just a few tasks to be done. These are illustrated in Figure 4. The required tasks are, briefly:

- If it is not done in MIN itself, check that the amount of working storage is sufficient for the requirements of the algorithm MIN, as in "IF (I5 .LT. 0) THEN".

- Define indices, such as I1,I2,..., which have the effect of breaking up the work arrays RWORK and IWORK into appropriate subvectors which meet the requirements of the call to MIN.

- Do any special initialization required for MIN, as in the call to BBLSET. This may involve the use of the special values passed in the array USER; these are defined through the generic user parameters described in Section 6.5. Since it has no effect on the rest of TP, the user may wish to pass some of these values to MIN in COMMON. Be sure proper account is taken of STATUS if reverse communication is to be used.

- Call the routine MIN with the correct calling sequence. Note the use of the integer variable SUBR for the case where more than one routine is being tested, as here, where both BBLNIR and CONMIN can be used.

- Assign the correct value to STATUS (if necessary) before returning to the calling routine. For convenience, a number of named parameters are provided for returning values to ZZTP; these are explained within ZZLINK.

The calling sequence to ZZLINK is a bit long, for it contains many of the variables from ZZTP which the user may wish to access in his own routine, by including them in his calling sequence.

```
      GOTO ( 1000, 2000 )  SUBR

C-----CALL BUCKLEY-LENIR ROUTINE BBLNIR.

 1000 IF ( STATUS .NE. RCRPT .AND. STATUS .NE. RCNOFG ) THEN

C     INITIALIZE, EXCEPT FOR A REVERSE COMMUNICATION RE-ENTRANCE.

         I1 = 1
         I2 = I1 + N
         I3 = I2 + N
         I4 = I3 + N
         I5 = LR - 3*N

         IF ( I5 .LT. 0 ) THEN

            STATUS = NOSTOR
            GOTO 90000

         ENDIF

         CALL BBLSET ( NINT(USER(1)), NINT(USER(2)), NINT(USER(3)),
     -                 NINT(USER(4)), NINT(USER(5)),
     -
     -                 USER(6), USER(7),
     -
     -                 USER(8)  .NE. ZERO , USER(9)  .NE. ZERO ,
     -                 USER(10) .NE. ZERO , USER(11) .NE. ZERO ,
     -                 USER(12) .NE. ZERO , USER(13) .NE. ZERO ,
     -                 NINT(USER(14)),      RELF, RELG,
     -                 TRACUN,  TRACES                          )

      ENDIF

      DECRF = ZERO
      CALL BBLNIR ( ZZFNS, N, X, F, DECRF, G, ACC, STATUS, ZZINNR,
     -     RWORK(I1), RWORK(I2), RWORK(I3), RWORK(I4),  I5, IW, RW, DW )

         IF      ( STATUS .EQ. -1 ) THEN
            STATUS = RCFG
         ELSE IF ( STATUS .EQ.  0 ) THEN
            STATUS = DONE
         ELSE IF ( STATUS .EQ.  1 ) THEN
            STATUS = XSFUNC
         ELSE IF ( STATUS .EQ.  2 ) THEN
            STATUS = NOSTOR
         ELSE IF ( STATUS .EQ.  3 ) THEN
            STATUS = USERV-1
         ELSE IF ( STATUS .EQ.  4 ) THEN
            STATUS = USERV-2
         ELSE IF ( STATUS .EQ.  5 ) THEN
            STATUS = USERV-3
         ELSE IF ( STATUS .EQ.  6 ) THEN
            STATUS = RABORT
         ELSE IF ( STATUS .EQ.  7 ) THEN
            STATUS = IPUNDF
         ENDIF

      GOTO 90000

C-----CALL SHANNO'S CG ALGORITHM CONMIN.

 2000 CONTINUE

      CALL CONMIN(ZZFNS,N,X,F,G,ACC,STATUS,RWORK,LR,NINT(USER(15)),
     -            TRACES, TRACUN, NTR )

      IF      ( STATUS .EQ.  0 ) THEN
         STATUS = DONE
      ELSE IF ( STATUS .EQ.  1 ) THEN
         STATUS = XSFUNC
      ELSE IF ( STATUS .EQ.  2 ) THEN
         STATUS = USERV - 1
      ELSE IF ( STATUS .EQ.  3 ) THEN
         STATUS = USERV - 2
      ENDIF
```

Figure 4: Code from ZZLINK

In general, most of them can be ignored. These variables are explained within the descriptive section of ZZLINK.

6.4 Direct Access I/O

In order to speed the processing of test problems, TP maintains a considerable amount of required information in the form of an unformatted direct access file. Now, the PROLOGUE file contains the definitions of all of the test problems available to users of TP, but, since this must be read and perhaps modified by TP users, it is a sequential access formatted file. However, the reading of such a file is quite time consuming, for FORTRAN is notoriously slow in the processing of formatted data, particularly on certain systems. For this reason, TP generates the file DAUF, in which the data from PROLOG is unformatted and stored as direct access records, so that TP can process it much more efficiently.

What in fact happens is the following. Upon beginning execution, TP first attempts to read the file DAUF. If it succeeds in opening DAUF, it totally ignores the PROLOGUE file. This will mean much faster start up in general, for unformatted FORTRAN I/O is more efficient. If TP cannot open DAUF, it then reads the file PROLOGUE. When PROLOGUE has been completely read, the DAUF file is created and saved, with all the data from PROLOGUE, and it is then immediately available on subsequent invocations of TP. This will be quite satisfactory, since in normal circumstances, PROLOGUE will not change very often. So, in most cases, TP will immediately read the file DAUF, and execution will begin with little delay.

If, however, changes are made to PROLOGUE, then, before executing TP again, the file DAUF must be purged. It would be desirable if TP could automatically check if the PROLOGUE file had been modified, but we know of no convenient way to do this in a system independent fashion.

Because of this handling of direct access data in a semi-permanent form, problems which are defined interactively by the user are neither totally temporary (in that they do not disappear as soon as execution of TP ends, nor completely permanent, in that any change to PROLOGUE will cause DAUF to be recreated, so that all problems in DAUF will disappear. It is hoped to revise this in a future version. This is also discussed in Section 5.5.

6.5 Redefining Generic Names

If you have an algorithm to test, it is almost certain that there will be some variables in the algorithm whose values influence the manner in which the algorithm executes. Line search accuracy and the choice of line search strategy are obvious examples of such user parameters. It is clear that TP should have a facility for setting or changing such user parameters, but it is equally clear that it could be rather awkward to do so, for we can hardly predict and allow for every user parameter for any algorithm that might be tested some day. We think however that TP has a reasonable means of handling this situation.

There are two parts to the way TP does this. First, a block of "generic" user parameters is available. It is stored internally in an array USER(NU), where NU is an integer constant which defines the number of such generics. These generic parameters can be accessed in the following way.

To begin, each of them has a default value; specifically 0. To set one of them to a different value, there is a standard name provided in the keyword dictionary. For USER(i), the name is UiPARAM. Thus, to set the third parameter, you could enter U3 = 7, an abbreviation for U3PARAM = 7.

These generic parameters are available in ZZLINK (Section 6.3) and can be passed to the test algorithm in its calling sequence.[5] For example, to set BETA for the user routine BBLNIR, assume

[5] You can also use COMMON if you prefer.

that its value is specified by the 7th parameter. To set BETA to 1.0, specify U7PAR = 1.0. Then in ZZLINK, include USER(7) in the calling sequence for BBLNIR in the position corresponding to BETA in the definition of the calling sequence for BBLNIR. See for example the call to BBLSET in Figure 4.

The second part of handling user parameters is through the RENAME keyword. In the description above, it is rather unfortunate that you must type U7 = 1.0 during the input, when you would really like to type BETA = 1.0. The RENAME feature allows you to change the name to BETA (at run time); then you can indeed change BETA by specifying BETA=1.

The command is

```
RENAME = generic name = new name { = default value}
```

For example, you may type

```
RENAME = U7PAR = BETA = 1.0
```

Here "generic name" refers to the name of the keyword originally in the dictionary and "new name" is the name which will replace it in the dictionary. Note that these are not synonyms in that you can not alternately use either name; the new name actually replaces the original name. The default value is optional. If it is not included, the default value for the new name is that which was originally provided for the appropriate value USER(i). If it is included, it replaces the original default value. Notice that in the example above, the default value for BETA, i.e. the default value for USER(7), becomes 1.0. Note that it does not change the current value.

It is convenient to use the renaming feature; simply predefine all user specific names in PROLOGUE. This is illustrated best by considering a sample from the PROLOGUE file:

```
RENAME = U7PAR  =   BETA    = 1.0
RENAME = U1PAR  =   METHOD  =  1
RENAME = U3PAR  =   ALPIS1  = FALSE
RENAME = U12PAR =   FORCEF  = TRUE
```

All generic parameters are REAL (or double precision if appropriate). One may handle integer or logical arguments in the following way. If the value should be integer, then set the value to an integer constant, as illustrated for METHOD. Internally, USER(2)—i.e. METHOD—will be stored as the real value 1.0. When using USER(2) to pass a value to a minimization subroutine, simply pass NINT(USER(2)). For logical values, use 0.0 for TRUE and 1.0 for FALSE and pass "USER(3) .EQ. 0.0" as the logical argument. Note that FALSE and TRUE are predefined literal-integers whose values are simply 1 and 0, resp. Again, see figure 4.

Of course this is a "one-shot" job. Once you have renamed the keywords in PROLOGUE, they are there (or in DAUF) and automatically invoked every time that you wish to use TP.

6.6 User Routines

There are two points which we would like to discuss here; first, what changes one might wish to make to the user routine MIN when using TP, and second, the use of routines which are available with TP for encapsulating special purpose user oriented code.

First consider the few changes which one might choose to make to MIN. These are recommended in order to provide consistent procedures for testing and to take advantage of the facilities which have been provided. Put briefly, this means modifying MIN to use the routines ZZPRNT, ZZEVAL and ZZTERM. Each of these is described carefully in its initial descriptive section. Here we indicate simply what is involved in using each of them.

ZZPRNT is used to print appropriate values at the current point. At the end of each iteration, call ZZPRNT, and that's it. The calling sequence for ZZPRNT is short and is described within its

listing. Output will occur every K^{th} iteration, where K is set during input (see Section 7) by the statement PRINT = K. Details of what is printed when K is positive, zero or negative are given in ZZPRNT and Section 7.

ZZEVAL should be used whenever the function and/or gradient is to be evaluated. No action is required except to call ZZEVAL whenever you need a function and/or gradient value. The calling sequence for ZZEVAL is also brief and is given with its listing. Note the return code when the function evaluation count reaches its maximum (as set in the problem definition; see 5.2); The user really should obtain function values by using ZZEVAL for it is in ZZEVAL that function and gradient calls are counted and timed and where such tasks as scaling and derivative estimation are done. If MIN is designed to work with reverse communication as a method of obtaining function and/or gradient values, then it is not necessary to use ZZEVAL, and instead the user routine MIN may return to its calling routine (namely ZZTP, via ZZLINK) which will take care of obtaining the desired values. Notice though that the variable STATUS must be correctly set on return to ZZTP, as mentioned in Section 6.3.

ZZTERM may be used to test the accuracy condition to see if the algorithm should be terminated. If ZZTERM is used, it will make your program easier to modify if you wish to change the termination criterion to produce results which are compatible with those which have been published elsewhere. Note that ZZTERM does not attempt to deal with abnormal termination. See Section 7 and the description of ZZTERM for details. If ZZTERM is not used then the keywords NORM, FRELATIVE, FABSOLUTE, GRELATIVE, GABSOLUTE, and TERMINATION will have no effect.

Each of these routines provides facilities in addition to the basic tasks described. For further information, one must consult the routines themselves.

In addition, two routines ZZBFOR and ZZAFTR are provided for the convenience of the user of TP. In the distributed version of TP, both of these are dummy routines; they do almost nothing. They are called by the main driver ZZTP just *before* and just *after* the call is made to ZZLINK to perform a minimization test. This allows the user to provide code which is highly specific to his test routine without altering the main code in ZZTP. A sample of its use is given in the distributed code.

6.7 Output

TP allows considerable flexibility in the output which is obtained. There are keywords provided which allow the user to control the amount of output which is obtained, and to change the allocation of the files where this output will appear. Examples of these are given in Section 7.

However, we would like to comment on the structuring of the output. It is important that TP be able to provide summary data, and yet at the same time it must allow for output which may be generated as part of the user's algorithm. What TP does for each run of test problems is the following.

All output produced by TP itself is simply dumped in free format mode into a temporary file. This will include information about the problems solved, solutions found, errors encountered and so on. That is, TP will dump all the information it needs in order to produce its summary statistics. Each such line of output will be preceded by a key. Interspersed with this output can be any data produced by the user's algorithm. When a set of runs is complete, TP rewinds the temporary file and processes the data therein. Data produced by TP is identified by the key and summary statistics are accumulated. Lines produced by the user's routine are echoed into the permanent record file, if desired. Then a summary of the run is printed on the terminal and/or into the permanent file, according to the output required.

The advantage of this approach is that it avoids any problem of mixing because of output from the user's routine. It also completely separates the functionality of producing summary data from the process of doing the testing. Thus it would be straightforward to modify the

summary routine to produce different summary information. Also, the routines ZZAFTR and ZZBFOR mentioned in Section 6.6 can also be used to incorporate special summary data into the output file.

6.8 Interprogram Communication

The structure of TP creates one programming problem which must be avoided to keep the package simple to use. To explain, the main routine ZZTP will have to call the user's test algorithm MIN at some point. This is done through the intermediate routine ZZLINK (see Section 6.3). Thus ZZTP calls ZZLINK; then ZZLINK calls MIN. Now, in many test algorithms, MIN will call the function evaluation routine, i.e. ZZEVAL, where the test functions are evaluated. There is quite a lot of information which must be available both to ZZTP and to routines such as ZZEVAL. But at run time, ZZEVAL will be separated from ZZTP by calls to ZZLINK and, especially, the user routine MIN. We have said before that the use of TP must be kept simple, and that the user should not have to alter the calling sequence to his routine. Therefore, values set in ZZTP and needed by ZZEVAL should not be transmitted by including them in the call to ZZEVAL, for then they would have to be in the call to MIN as well. This is particularly true because of the number of control variables which need to be passed in this way. We have used the entry point as the solution[6]. For example, whenever the user (through the command language) causes the current derivative evaluation mode to change, a call is made to an entry point in ZZEVAL so that the new mode may be recorded there. This may seem a nuisance, but we feel it is important, for it makes it invisible to the user. Finally, in ADA and the proposals now being put forward for the next standard of FORTRAN, it will be possible to define "modules" (which may go under some other name) in which it will be possible to define parameters and variables which will be globally available. The current structure of TP would make transition to such a language very easy.

6.9 Use of Parameters

Although TP has been designed so that it is not necessary to alter any of its internal code, it is nonetheless quite possible that someone will want to make modifications, perhaps to add extended features or to adapt TP to other types of problems. In order to facilitate this, extensive use was made of the FORTRAN 77 PARAMETER statement. The main routine ZZTP in TP begins with a section in which a large collection of PARAMETER and default declarations appear. These define all default values for any of the quantities which the user can access through the command language. All array sizes are set with named constants, and so on. These are all designed to simplify modification.

Those which are required in more than one of the 50 or so routines which constitute TP are in fact maintained on a host system as include files. This will simplify the conversion of TP to the next FORTRAN standard, which will include a MODULE definition facility. For the present, the include files are actually textually incorporated into all required routines before TP is distributed, so that the distributed code will be standard conforming.

[6] In order to avoid the use of COMMON; see [2]

This also greatly improves some of the code within TP. Suppose we take gradient evaluation for example. There are 3 possibilities: analytic evaluation, finite difference estimation or a comparison of both values. Clearly, one must have an internal flag to identify the current choice; an integer with one of three values is appropriate. Thus, in ZZTP, the three parameter codes are defined as `PARAMETER ( ANAL = 1, DIFF = 2, TEST = 3 )`. This creates a difficulty however. At least two routines need to act on the current setting, so that if the 3 integer values 1, 2, and 3 are used in one routine, then the same values must be used in all other routines. Having to maintain the same PARAMETER values in different routines is awkward. Using `include` files avoids this difficulty.

7 Additional Features

In this section, we would like to demonstrate some further features of TP. This will be done by considering some of the commands available, and discussing their effect. The complete syntax of the command input is described in [2]. Here we will generally use the keywords in their full form; however, many abbreviations are possible.

There are two input *modes*, one for specifying commands which control the processing, and one for defining new problems or groups of problems. This section deals with processing commands; definitions were discussed in Section 5.2.

ACCURACY◇ This defines the accuracy required in the solution. In other words, this sets the value which determines when to terminate each minimization. For example, one may enter `ACCURACY = 1.e-8`. The value is passed to ZZLINK as ACC; see Section 6.3. It is possible to rerun a set of problems with varying values for ACCURACY, as in `ACC= .005, .00001`, or, by using the loop facility mentioned in Section 5.4, as in `ACC=1.e-8 * 1.e-12 * .1`, or, equivalently, `ACC=8 * 12 * 1`.

ARITHMETic◇ This redefines the character which specifies an arithmetic loop. If it is inconvenient on a particular system to specify a sequence with, say `X0=1:99:2`, one may redefine the character which triggers interpretation of a set of values as a implied loop. For example, if one specified `ARITHMETIC =&`, one could subsequently write `X0 = 1&99&2`. *Any* character which has a special meaning in the decoding of commands in TP may be similarly redefined. This would most naturally be done in the PROLOGUE file.

CC◇ TP attempts to be sensitive to the environment of different computing systems. For example, CC determines whether or not a carriage control character is appended to the front of lines being sent to the user's terminal. Some systems (e.g. VMS) assume they are there when running FORTRAN jobs and therefore remove them, in which case one should specify `CC = PRESENT`. If the first character on each line is not stripped off, then no "leading character" should be issued, which is accomplished with `CC=MISSING`. Other idiosyncracies can also be controlled; suitable keywords are EOF, UPPERCASE and DEVICE.

DERIVATIVE◇ This selects the mode for evaluation of derivative values as either analytic or by finite differences, as for example with `DERIVATIVE=DIFFERENCE`. The finite difference computations are then done automatically. One may also select test mode, in which both analytic and finite difference derivatives are computed and then compared for agreement. In this case, the level of agreement will appear with the summary output. When the main purpose of a computation is to test the accuracy of derivative evaluations, ACCURACY should not be set too small; a value such as .001 is typically appropriate.

DOSUMMARY◇ TP allows the user to control the amount of output which will be generated, both interactively to his terminal, and to a permanent record of the run in a file, with

the commands SEE and SUMMARY. When doing a set of tests, generally no visible output is generated until the tests are all done, as discussed in Section 6.7. At that point, the data which has been temporarily stored is formatted and presented on the terminal and/or stored on a permanent file. The command DOSUMMARY causes the current summary data to be output. A particular use would be to do a set of problems with minimal output, to examine the output and then to increase the level of output desired with SEE=MEDIUM, and, with no further computation, to repeat the same output with the command DOSUMMARY. As soon as this command is recognized, TP will skip to the summary processing section of its code. It will then read whatever data is present, which will still be there from the previous set of tests, and it will prepare an output summary. This is illustrated in Section 4.

FACTOR◇ One frequently wishes to modify standard starting points. One way this is done is to multiply each of its components by some fixed constant. This may be specified with FACTOR. It is normally 1, but may be reset. For example, FACTOR = 1 * 125 * 5 would cause each problem to be run 4 times, with each starting point multiplied in turn by 1, 5, 25 and 125.

FTRACE◇ When this is set, it causes the function value to be printed every time a function value is computed. This is useful for tracing the progress of a minimization calculation. GTRACE causes a similar trace of gradient computations.

INPUT◇ One may not always wish to execute tests interactively from a terminal, either because of the time they may take, or because it involves repeating the same set of data several times with only minor changes. The command INPUT causes transfer of input control to a specified file. For example, to create a batch run, one could put the statement INPUT="BATCH.CONTROL" in the PROLOGUE file. Thus, as soon as PROLOGUE had been read, processing would continue by reading commands from the specified file BATCH.CONTROL. If mode was BATCH (as set with the command MODE), execution would terminate when an end-of-file was reached on that file; otherwise TP would read further control statements from the user's terminal. Alternatively, if the same set of commands is to be executed repeatedly, they can be put in a file, say CMDS, and then the command INPUT=CMDS can be given interactively. The statements in CMDS will be executed, and then control will return to the terminal. Any desired change to control parameters can then be made, and the command INPUT=CMDS repeated. The name TERMINAL is special and always causes input to continue from the terminal, with appropriate prompting.

MEMORY◇ In some algorithms, the amount of memory available is critical to their performance, so that one will want to control the available memory during testing. MEMORY specifies the amount of working storage to be used. For example, MEMORY=1000X (or MEMORY=4096) indicates that 1000 hexadecimal locations (of a REAL array) should be used.

NORM◇ When evaluating the termination criterion, norms of certain vectors are computed. This specifies whether the 1, 2 or supremum norm should be used.

PRINT◇ This specifies the amount of intermediate printing desired either to a file or on the terminal. The magnitude of PRINT indicates how often to print values such as the function value and iteration count, and the sign determines how much will be printed. The default value for the file is such that the first and last points will always be printed; on the terminal nothing appears. Printing may be turned off with PRINT=0.

RENAME◇ This allows for renaming of generic parameters; see Section 6.5.

RESULTS◇ This is used to redefine the name of the file onto which the final output is placed. It can be used to split the output onto several different files, provided the operating system

permits reallocation of the file attached to a particular FORTRAN unit number *within* a standard FORTRAN 77 program[7]. When a new name is specified, any current output file is closed and kept, and a new file with the same unit number but the new name is opened.

REVERSE◇ This indicates that the current algorithm is to use reverse communication to obtain function and gradient values. See Section 3 and Section 6.3.

SEE◇ This specifies the level at which you wish to see the summary on the terminal when in interactive mode. The least output, which is SEE = NONE, is obtained for level 0, while the FULLest occurs with level 3, which is the most complete, but only desirable if you have a terminal which is running at a reasonably high speed.

SUBROUTINE◇ In any realistic testing, there will certainly be more than one algorithm under consideration. TP permits several routines to be linked at the same time, and considers them to be numbered 1,2,... according to the code in the routine ZZLINK described in Section 6.3. One specifies which is to be used with the command USE; for example, USE = 3, 1 would run each test first with algorithm 3, then with number 1. For convenience, you may also refer to them by defining names for each subroutine, most naturally in the PROLOGUE file. For example, the statement SUBROUTINE=BBLNIR=1 indicates that subroutine number 1 may subsequently be referred to by the name BBLNIR, so that one could specify the same two algorithms with USE = 3, BBLNIR.

TERMINATE◇ As discussed under NORM, the user has certain control over how the termination criterion is applied. To select a particular type of termination test, use TERMINATE. Details are in [2].

TRACES◇ It is often convenient to build into a routine, particularly a new one which is undergoing testing, the ability to print certain information about execution, i.e. a trace. It is also nice to be able to turn this off when the routine has begun to execute properly, but it is even nicer to leave the optional trace statements in the code in case later problems arise. There are several trace variables defined within TP, which can be passed to the user's routine (see Section 6.3). These flags can be turned on or off by entering for example "TRACES=TTFT"; this would turn on only the flags 1, 2 and 4. The file to receive the trace output may also be set with the command TRACEUNIT.

VERIFY◇ As mentioned, it is possible to abbreviate input commands in a couple of ways. If you wish to verify that abbreviated commands are interpreted correctly, turn VERIFY on, and the decoder will echo each command in its full form as it was understood.

These commands illustrate many but not all of the features available in TP. For complete details, one must consult the document [2].

8 Conclusions

We have described a routine which is designed to facilitate the testing of algorithms for unconstrained minimization of smooth nonlinear functions. We have demonstrated the kind of facilities which can be built into such a package with few assumptions being made about the routines to be tested. Such a testing package should provide a useful environment in which to evaluate new routines which are proposed for computing approximate solutions to this problem. Indeed, we might suggest that testing with this or a similar package be mandatory for acceptance for publication of new algorithms for minimization. With the availability of a large

[7] Which we do not believe is the case with CMS.

number of test problems, and a mechanism whereby testing of an algorithm on these problems is quite straightforward, there would be little excuse for doing less than exhaustive testing on a new routine.

We would also like to emphasize that this work can also be viewed as having a scope beyond that of the minimization problem. The ideas presented here could also have application to any field of computation where algorithms are written to solve well defined mathematical problems.

9 Acknowledgements

I first wish to express my thanks to the Natural Sciences and Engineering Research Council of Canada whose support through grants A 8962, A 2694 and E 0011 has made this research possible. In addition, I would like to thank the Computing Center of Concordia University, the MIS Department of the University of Arizona, and INRIA, l'Institut National de Recherche en Automatique et en Informatique en France. These institutions have provided the computing facilities on which this work was primarily developed.

A number of students deserve recognition for the part they have done in coding parts of this package (particularly earlier versions, some of which are no longer in the code), and for the ideas they have contributed. These include Leung Pak Chiu (Patrick Leung), Ron Chan, Stephen Kerr and Stephen Vincent. Thanks are also due to Dave Shanno, David Gay, Philippe Toint and Jorge Moré for their comments and contributions. Mrs. Alexandra LeNir deserves thanks for being a guinea pig for an early version which was used to produce some of the results for her thesis.

References

[1] A. Buckley. A portable package for testing minimization algorithms. In John M. Mulvey, editor, *Evaluating Mathematical Programming Techniques*, pages 226–235, Committee on Algorithms, Mathematical Programming Society, Springer-Verlag, New York, 1982.

[2] A. Buckley. *TESTPACK.* Dalhousie University, Department of Mathematics, Statistics and Computing Science, Dalhousie University, Halifax B3H 3J5, Canada, December 1986.

[3] A.R. Colville. *A Comparative Study On Nonlinear Programming Codes.* Report 320–2949, IBM Corporation, Philadelphia Scientific Center, Philadelphia, Pennsylvania, June 1968.

[4] R. Fletcher. Function minimization without evaluating derivatives: a review. *The Computer Journal*, 8:33–41, 1965.

[5] R. Fletcher and M.J.D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 163–168, 1963.

[6] P. E. Gill and W. Murray. *Conjugate gradient methods for large-scale nonlinear optimization.* Technical Report SOL 79-15, Systems Optimization Laboratory, Stanford University, California, October 1979.

[7] P. E. Gill, W. Murray, and R. A. Pitfield. *The implementation of two revised quasi-Newton algorithms for unconstrained optimization.* Technical Report, National Physical Laboratory, Teddington, England, April 1972.

[8] D. M. Himmelblau. *Applied Nonlinear Programming.* McGraw-Hill, New York, 1972.

[9] D. M. Himmelblau. A uniform evaluation of unconstrained optimization techniques. In F. A. Lootsma, editor, *Nonlinear Methods for Nonlinear Optimization*, pages 69–97, 1972.

[10] J. Kowalik and M. R. Osborne. Analysis of kinetic data for allosteric enzyme reactions as a nonlinear regression problem. *Math. Biosciences*, 2:57–66, 1968.

[11] J. Kowalik and M. R. Osborne. *Methods for Unconstrained Optimization.* American Elsevier, 1968.

[12] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, March 1981.

[13] M. R. Osborne. Some aspects of nonlinear least squares. In F. A. Lootsma, editor, *Nonlinear Methods for Nonlinear Optimization*, pages 171–189, 1972.

[14] *Programming Language FORTRAN, International Standard ISO 1539-1980(E);ANSI X3.9-1978.*

[15] K. Schittkowski. *Non-linear Programming Codes—Information, Test, Performance. Lecture Notes in Economics and Mathematical Systems*, Springer-Verlag, Berlin, 1980.

[16] D.F. Shanno and K.-H.Phua. Algorithm 500: Minimization of unconstrained multivariate functions. *ACM Transactions on Mathematical Software*, 2(1):87–94, Mar 1976.

[17] N. Wirth. *Pascal User's Guide.* Springer-Verlag, Heidelberg, 1972.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique