

# CUTEr and SifDec: A Constrained and Unconstrained Testing Environment, revisited

NICHOLAS I. M. GOULD  
Rutherford Appleton Laboratory  
DOMINIQUE ORBAN  
Northwestern University  
and  
PHILIPPE L. TOINT  
University of Namur

---

The initial release of CUTE, a widely used testing environment for optimization software, was described by Bongartz, et al. [1995]. A new version, now known as CUTEr, is presented. Features include reorganisation of the environment to allow simultaneous multi-platform installation, new tools for, and interfaces to, optimization packages, and a considerably simplified and entirely automated installation procedure for UNIX systems. The environment is fully backward compatible with its predecessor, and offers support for Fortran 90/95 and a general C/C++ Application Programming Interface. The SIF decoder, formerly a part of CUTE, has become a separate tool, easily callable by various packages. It features simple extensions to the SIF test problem format and the generation of files suited to automatic differentiation packages.

Categories and Subject Descriptors: G.1.6 [**Numerical Analysis**]: Optimization—*constrained optimization, unconstrained optimization*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*sparse, structured, and very large systems*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; G.4 [**Mathematical Software**]—*certification and testing*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*portability*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*user interfaces*; D.2.6 [**Software Engineering**]: Programming Environment—*programmer workbench*

General Terms: Algorithms, Experimentation, performance, Reliability, Verification

Additional Key Words and Phrases: Nonlinearly constrained optimization, testing environment, shared filesystems, heterogeneous environment, SIF format

---

This work was supported by the MRNT grant for joint thesis support.

Authors' addresses: N. I. M. Gould, Rutherford Appleton Laboratory, Computational Science and Engineering Department, Chilton, Oxfordshire, OX11 0QX, England; email: n.gould@rl.ac.uk; D. Orban, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208-3118; email: orban@ece.northwestern.edu; P. L. Toint, University of Namur, 61 rue de Bruxelles, B-5000 Namur, Belgium; email: Philippe.Toint@fundp.ac.be

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2003 ACM 0098-3500/03/1200-0373 \$5.00

## 1. INTRODUCTION

The CUTE testing environment for optimization software and the associated test problem collection originated from the need to perform extensive and documented testing on the LANCELOT package [Conn et al. 1992]. Because the large set of test problems and testing facilities produced in this context were useful in their own right, they were extended to provide easy interfaces with other commonly used optimization packages, gathered in a coherent multi-platform framework and made available on the World Wide Web and via anonymous FTP, to the research community. The article [Bongartz et al. 1995] provides an overview of the environment, and full documentation of the available tools and interfaces at the time.

Since 1993, the CUTE environment and test problems have been widely used by the community of optimization software developers [Bielschowsky and Gomes 1998; Breielfeld and Shanno 1994, 1996; Byrd et al. 2000; Coleman and Yuan 1995; Conn et al. 1999a, 1999b; Dennis et al. 1999; Diniz-Ehrhardt et al. 1997, 1998; Fletcher and Leyffer 2002; Gert 1999; Gomes et al. 1999; Gould et al. 1999; Gould and Nocedal 1998; Lalee et al. 1998; Marazzi and Nocedal 2000; Sargent and Zhang 1998; Sartenaer 1997; Shahabuddin 1996; Ulbrich and Ulbrich 1999; Xiao 1996; Xiao and Chu 1995; Yamashita et al. 1997]. Such widespread use has inevitably led to a clearer awareness of the deficiencies of the original design, and also created a demand for new tools and new interfaces. The environment has evolved over time by the addition of new test problems and minor updates to a number of tools. The present article aims to describe its next major evolution: CUTER, in which we revisit the original CUTE design. This new release is characterized by

- a set of new tools, including a unified facility to report the performance of the various optimization packages being tested,
- full backward compatibility with CUTE,
- a set of new interfaces to additional optimization packages,
- some Fortran 90/95 support, and
- an integrated C/C++ Application Programming Interface.

The SIF optimization test-problem decoder, which used to be a constituent part of the CUTE environment, has been isolated into a separate package named SifDec. Any software that could require the decoding of a SIF file may now rely on it, as a package in its own right. It is characterized by

- the definition and support of an extension to SIF (the Standard Input Format) allowing for easier input of quadratic programs and for casting the problem against a selection of parameters, such as the problem size, and
- the ability to generate input files suited to automatic differentiation tools, such as the HSL [2002] AD01 and AD02 packages [Pryce and Reid 1998].

Both CUTER and SifDec have the following features:

- Completely new organization of the various files that make up the environment, now allowing concurrent installations on a single machine and shared installations on a network, and

- a new simplified and automated installation procedure, but
- the restriction of the environment to UNIX systems.

The last of these items is the reason why the rest of the article only considers directory structures and/or file names in a style typically found on UNIX systems. To some, the restriction to UNIX systems might seem a retrograde step because CUTE offered VMS and some DOS support, but this merely reflects our current expertise.

This article is intended to supersede the parts of Bongartz et al. [1995] that are obsolete in CUTER, to cover the new features, and to describe the new SifDec environment. It is organized as follows. Section 2 discusses the new organization of the CUTER environment files. Section 3 documents the new tools and discusses application program interfaces. Section 4 documents the new interfaces to additional optimization packages and Section 5 covers the isolated SIF decoder environment, the extension of the SIF description language to quadratic programs, and its support of user-changeable parameters. Section 6 describes the new installation procedures. Details of how the packages may be obtained are given in Section 7, and concluding comments are presented in Section 8.

## 2. A NEW FLEXIBLE ORGANIZATION

One of the defects of CUTE is that it was not designed to support a multiplatform environment; that is, instances of the environment that could be used simultaneously from a central server on several, possibly different, machines, with their own dialects of UNIX. Moreover, using CUTE on a single machine in conjunction with several different compilers (a case that frequently occurs when new software is tested) is extremely cumbersome. Likewise, handling different instances of the environment corresponding to different *sizes* of the tools (that is the size of the test problems that they can handle) is problematic. The reason for these difficulties is that the structure of the CUTE files, as described in Bongartz et al. [1995], does not lend itself to such use, because it only contains a single subtree of objects files. If we call the combination of a machine, operating system, compiler and size of the tools an *architecture*, the obvious solution to such a defect is then to allow several such subtrees in the installation, one for each architecture.

However, as soon as the possibility of using architecture-dependent subtrees is raised, the proper identification of the parts (scripts, programs) of the environment that are independent of the architecture also becomes an issue. Since it would be inefficient to store copies of these independent scripts and programs in each subtree, it is natural to store them in a data structure that is itself disjoint from the dependent subtrees. Finally, the propagation of subtrees containing sometimes very similar yet vitally different data makes the maintenance of the environment substantially more complicated, and therefore requires enhanced tools and a clear distinction between the parts of the environment that are related to testing optimization software and those related to its own maintenance.

The directory organization chosen for CUTER, shown in Figure 1, reflects these considerations. We now briefly describe its components.

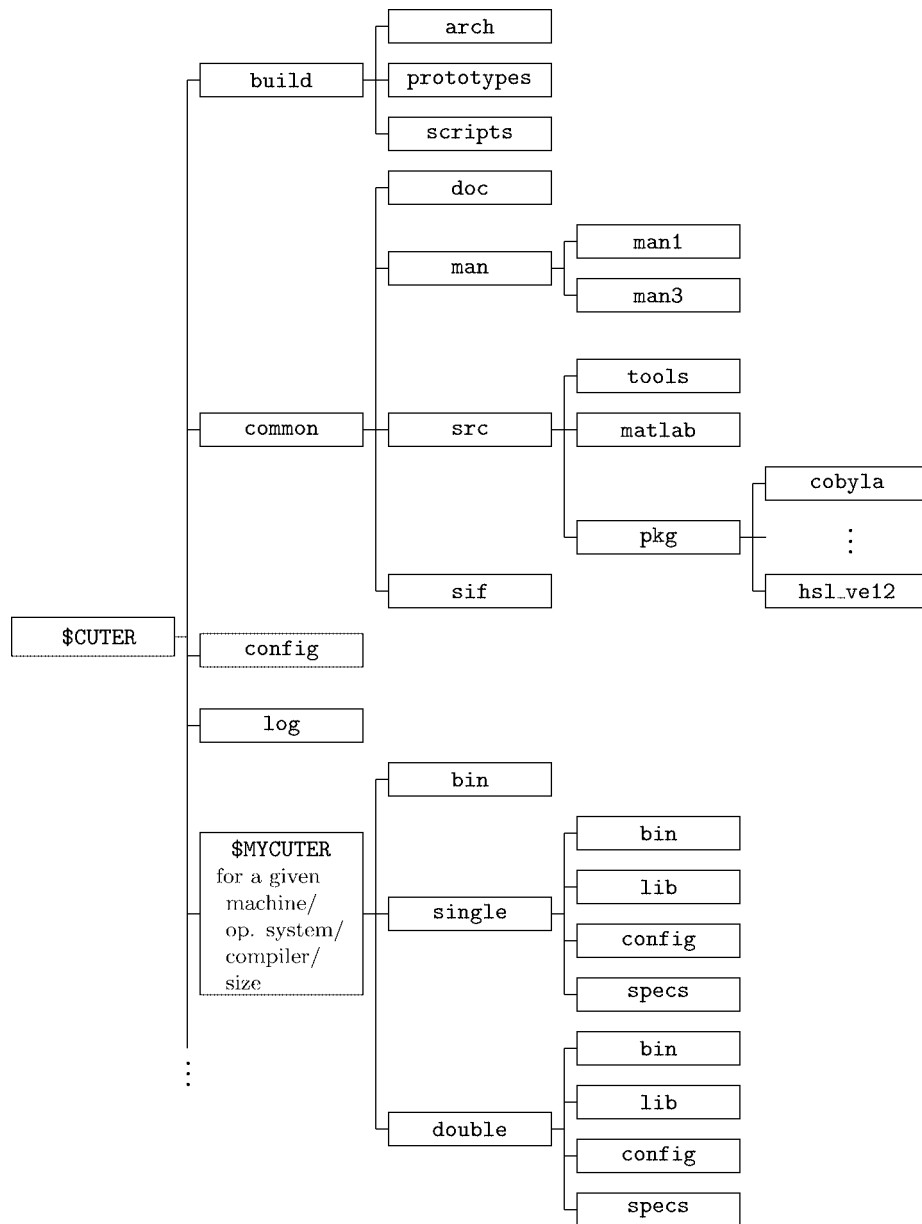


Fig. 1. Structure of the CUTER directories.

Starting from the top of the figure, the first subtree under the main `$CUTER` directory (the root of the CUTER environment) is `build`, which essentially contains all the files necessary for installation and maintenance. Its `arch` subdirectory contains the files defining all possible architectures that are currently supported by CUTER, allowing users to install new architecture-dependent subtrees as they are required, depending on the testing needs and the evolution

of platforms, systems and compilers. The `prototypes` subdirectory contains the parts of the environment that have to be specialized to one architecture before they can be used. We call such files *prototypes* and the process of specializing them to a specific architecture *casting*. The prototype files include a number of tools and scripts whose final form typically depends on compiler options and the chosen size of the tools. Finally, the remaining subdirectory of `build`, named `scripts`, contains the environment maintenance tools and various documentation files.

The second subtree under `$CUTER` is called `common` and contains the environment data files that are relevant for the purpose of testing optimization packages, but are independent of the architecture. Subdirectory `doc` contains a number of documentation files concerning the environment, such as a description of its structure and procedures to follow for interfacing the supported optimization packages. The CUTER tools and scripts themselves are documented in the `man` subdirectory, and, as is common on UNIX systems, its `man1` and `man3` subdirectories. The `src` subdirectory contains the source files for many of the environment utilities disseminated in a number of subdirectories. We briefly describe them in turn. `tools` contains the sources of the Fortran tools used in users' programs. `matlab` contains all the "m-files" that provide a MATLAB interface to the environment. `pkg` holds information related to the various optimization packages for which CUTER provides an interface—these being stored separately in their own subdirectory of `pkg`; Figure 1 represents those for COBYLA and HSL\_VE12. Each of these package-specific subdirectories typically includes an algorithmic specification file and a suitable README description of how to build an interface between CUTER and the package. The last subdirectory, `sif`, contains a few test problems in SIF format.

The next subdirectory under `$CUTER` is called `config` and contains all the configuration and rules files needed to generate the platform-specific *Makefiles*.

The `log` subdirectory of `$CUTER` contains a history of the various installations—and, possibly, subsequent un-installations—of the environment for the various architectures.

The remaining subdirectories of `$CUTER` are all architecture dependent: each corresponds to the installation of CUTER in a specific machine, for a given operating system and compiler and for a given tool size. The figure only represents one, but the leftmost continuation dots indicate that there might be more than one. Although these directories have been symbolically represented as subdirectories of `$CUTER` in the figure to reflect their dependence upon `$CUTER`, they may be located anywhere on the host system, including on remote machines over the local network. The names of these directories are (by default) automatically chosen at installation, but a user of one of these subtrees would typically give it a symbolic name, like `$MYCUTER`, to distinguish the version of CUTER currently in use. Each architecture-dependent subtree is divided into its single precision and double precision instances (`single` and `double`, respectively), each of these containing four subdirectories. The first, `bin`, contains the object files corresponding to the driving programs for the optimization packages and, if relevant, for the package codes themselves. If applicable, it should also contain the Fortran 90/95 module information files (those usually suffixed

by `.mod`). The second, `lib`, contains the library of CUTEr tools and, if relevant, any object libraries associated with the interfaced optimization packages. The `config` subdirectory contains the architecture-dependent files that were used to build the current `$MYCUTER` subtree (they are reused when a tool or optimization package is added or updated), while `specs` contains the algorithmic specification files for the optimization packages that are architecture dependent, if any. Finally, `$MYCUTER/bin` contains scripts that depend on the architecture but not the precision.

The fact that the CUTEr tools are now stored in the form of libraries (while they were stored as a collection of individual object files in CUTE), is another new feature. This allows a much simpler design of new optimization package interfaces, because the interface no longer needs to specify the exact list of tools that need to be loaded with the package.

A final new feature of the environment organization is that the documentation is available via the usual `man` command for the scripts and tools, and in ASCII, postscript and pdf formats for the rest.

### 3. NEW FEATURES

This section describes innovations in CUTEr, from the point of view of optimization software and the manipulation of data decoded from problems. Section 3.1 describes recently added CUTEr tools for handling data to be used by optimization software, while Section 3.2 concentrates on programming languages support.

#### 3.1 New Tools

The CUTEr tools for unconstrained and constrained optimization are presented in Tables I and II respectively, accompanied by a brief description. Storage of the Hessian matrix of either the objective or the Lagrangian function may be either dense or sparse. Unless otherwise specified, sparse storage occurs in coordinate format [Duff et al. 1986, §2.6]. Explicit mention is made whenever the storage scheme is instead finite-element format [Duff et al. 1986, §10.5]. Besides the general CUTEr documentation, `man` pages describing all supplied tools and their calling sequence are included in the distribution.

Users of the previous versions of CUTE will notice a number of new tools for both unconstrained—or bound-constrained—and constrained problems. We note the `uvarty` and `cvarty` tools, whose purpose is to determine the type of each variable, which may be continuous, binary (0-1) or integer. For constrained problems, the tool `cdimen` determines the number of variables and constraints involved. The tools `cdimse` and `cdimsh` determine the number of nonzero entries in the Hessian of the Lagrangian when using (respectively) finite-element or general sparse matrix storage, and thus allow users to set appropriate array sizes in advance, while `cdimsj` does the same for the constraint Jacobian. The tool `cscifg` is now obsolete and replaced by `ccifsg`. For backward-compatibility reasons, the former is included but simply calls the latter as a subroutine. Programs that ran under earlier versions of CUTE will therefore still run under CUTEr. Similarly, for unconstrained problems, the new tools `udimen`, `udimse` and

Table I. The Unconstrained Optimization CUTeR Tools

Tool name	Brief description
ubandh	extract a banded matrix out of the Hessian matrix
udh	evaluate the Hessian matrix in dense format
udimen	get the number of variables involved
udimse	determine the number of nonzeros required to store the sparse Hessian matrix in finite-element format
udimsh	same as udimse, in sparse format
ueh	evaluate the sparse Hessian matrix in finite-element format
ufn	evaluate function value
ugr	evaluate gradient
ugrdh	evaluate the gradient and Hessian matrix in dense format
ugreh	evaluate the gradient and Hessian matrix in finite-element format
ugrsh	evaluate the gradient and Hessian matrix in sparse format
unames	obtain the names of the problem and its variables
uofg	evaluate function value and possibly gradient
uprod	form the matrix-vector product of a vector with the Hessian matrix
usetup	set up the data structures for unconstrained optimization
ush	evaluate the sparse Hessian matrix
uvarty	determine the type of each variable
ureprt	obtain statistics concerning function evaluation and CPU time used

udimsh determine the number of variables involved, and the numbers of nonzeros in the Hessian in finite-element and sparse formats respectively. Finally, the ureprt and creprt tools produce statistics about a particular run on (respectively) an unconstrained or constrained test problem, reporting data such as total CPU time, number of iterations, function and constraints evaluations (if appropriate), number of evaluations of their derivatives, and the number of Hessian matrix-vector products used.

All the external package drivers supply report data using the ureprt and creprt tools. These drivers have filenames matching the `*ma.f` or `*ma.f90` expression. They may be found in `$CUTeR/common/src/tools` before compilation and under the name `$MYCUTeR/precision/bin/*ma.o` after compilation. For example, the hypothetical source package `pak.f` needs to be compiled into `$MYCUTeR/precision/bin/pak.o`. All the object files and the relevant libraries are subsequently linked by the corresponding interface, following the procedure described in Section 4.

### 3.2 New Application Programming Interfaces

In this section, we comment on the possibility of hooking optimization software written in Fortran 90/95 or C/C++ to CUTeR.

CUTeR makes provision for optimization or linear algebra codes written in standard Fortran 90/95 by providing explicit interface blocks to all tools present in the library and delayed compilation in case module information is not present at installation time. As guidelines to writing new Fortran 90/95 interfaces, a generic interface `gen90ma` and a real interface `ve12ma` for the HSL code HSL\_VE12 are already part of the CUTeR distribution.

Optimization and linear algebra software increasingly use the flexibility and generality of object-oriented languages like C++ and, more generally, enjoy the

Table II. The Constrained Optimization CUTer Tools

Tool name	Brief description
ccfg	evaluate constraint functions values and possibly gradients
ccfsg	same as ccfg, in sparse format
ccifg	evaluate a single constraint function value and possibly gradient
ccifsg	same as ccifg, in sparse format
cdh	evaluate the Hessian of the Lagrangian in dense format
cdimen	get the number of variables and constraints involved
cdimse	determine number of nonzeros to store the Lagrangian Hessian in finite-element format
cdimsh	determine number of nonzeros to store the Lagrangian Hessian in coordinate format
cdimsj	determine number of nonzeros to store the matrix of gradients of the objective function and constraints, in sparse format
ceh	evaluate the sparse Lagrangian Hessian in finite-element format
cfn	evaluate function and constraints values
cgr	evaluate constraints gradients and objective/Lagrangian gradient
cgrdh	same as cgr, plus Lagrangian Hessian in dense format
cidh	evaluate the Hessian of a problem function
cish	same as cidh, in sparse format
cnames	obtain the names of the problem and its variables
cofg	evaluate function value and possibly gradient
cprod	form the matrix-vector product of a vector with the Lagrangian Hessian
cscfg	evaluate constraint functions values and possibly gradients in sparse format
cscifg	same as cscfg, for a single constraint
csetup	set up the data structures for constrained optimization
csgf	evaluate constraints and objective/Lagrangian function gradients
csgreh	evaluate both the constraint gradients, the Lagrangian Hessian in finite-element format and the gradient of the objective/Lagrangian in sparse format
csgfsh	same as csgreh, in sparse format instead of finite-element format
csh	evaluate the Hessian of the Lagrangian, in sparse format
cvarty	determine the type of each variable
creprt	obtain statistics concerning function evaluation and CPU time used

benefits and portability of the C language. As a response to those interests, CUTer provides a C/C++ Application Programming Interface to every tool available in the environment. This interface is transparent in the sense that users need not worry about architecture or compiler-dependent details as these are treated internally.

At the time of this writing, only the most popular combinations of Fortran and C compilers have been considered, but inevitably further support will be provided in the future.

#### 4. NEW INTERFACES

CUTer contains a number of additional interfaces to existing packages (as well as interfaces to newer versions of previously supported packages) beyond those offered with CUTE. The purpose of providing these interfaces is to allow researchers to run a variety of solvers on a consistent set of test examples, and thus to assess the respective merits of each for solving classes of related problems, and practitioners to solve simplified versions of their problems or similar



problems by established solvers and cross-compare the results. The newly supported packages are:

**filtersQP.** This nonlinear programming package combines filter and trust-region methods to globalize an SQP iteration [Fletcher et al. 2002; Fletcher and Leyffer 1998, 2002]. The filtersQP package may be obtained from Roger Fletcher (fletcher@maths.dundee.ac.uk) and Sven Leyffer (leyffer@mcs.anl.gov).

**GALAHAD.** This is a library of Fortran 90 packages for solving a variety of optimization problems [Gould et al. 2002]. There are CUTEr interfaces to each of the linear and non-convex quadratic programming packages LSQP, QPA and QPB, as well as a direct SIF interface to the nonlinear programming package LANCELOT (see Section 5). GALAHAD was written by Nick Gould, Dominique Orban and Philippe Toint, and may be downloaded from <http://galahad.rl.ac.uk/galahad-www>.

**HRB.** To convert matrices (for example, Hessians, Jacobians, and KKT augmented system matrices) derived from SIF problem data into Harwell-Boeing [Duff et al. 1999, 1992] or Rutherford-Boeing [Duff et al. 1997] sparse matrix formats. HRB was written by Nick Gould, and is unique in CUTEr in that the interface requires no external package.

**HSL\_VE12.** This package finds critical points of nonconvex quadratic programming problems using an interior-point trust-region algorithm [Conn et al. 2000]. HSL\_VE12 is part of HSL [2002] and was written by Nick Gould and Philippe Toint.

**IPOPT.** This package solves smooth nonlinear programs with a mixture of equality and inequality constraints using a primal-dual interior-point method, factorizations and a filter to promote global convergence. IPOPT [Wächter 2002] is maintained by Andreas Wächter (andreasw@watson.ibm.com) and is open-source software available as part of the COIN-OR (<http://www.coin-or.org>). IPOPT is authored by Andreas Wächter, Lorenz T. Biegler, Yi-Dong Lang and Arvind Raghunathan.

**KNITRO.** This minimizes a smooth nonlinear function subject to nonlinear equality and inequality constraints using an interior-point approach. The resulting barrier subproblems are treated using SQP. KNITRO [Byrd et al. 2000] is maintained by its developers Richard Waltz (rwaltz@ece.northwestern.edu) and Jorge Nocedal (nocedal@ece.northwestern.edu).

**LA04.** This package solves linear programs using a steepest-edge simplex method [Reid 2001]. LA04 is part of HSL [2002] and was written by John Reid (J.K.Reid@rl.ac.uk).

**L-BFGS-B.** This package [Zhu et al. 1997] treats unconstrained or bound constrained problems. It uses a limited memory BFGS quasi-Newton method and is available from Jorge Nocedal (nocedal@ece.northwestern.edu).

**LOQO.** A linesearch-based primal-dual interior-point code for nonlinear programming, using an SQP approach and direct factorizations [Vanderbei and Shanno 1997]. LOQO is written in C and maintained by its author Robert Vanderbei (rvdb@princeton.edu).

**NITSOL.** This package [Pernice and Walker 1998] solves systems of nonlinear equations treating Newton systems by a restarted `GMRES` and promoting global convergence using a backtracking linesearch. The package is maintained by its authors Michael Pernice (`pernice@lanl.gov`) and Homer Walker (`walker@wpi.edu`).

**PRAXIS.** This is Brent's algorithm, reimplemented in Fortran 77 by John Chandler, for unconstrained minimization without derivatives [Brent 1973]. It is available from John Chandler (`jpc@a.cs.okstate.edu`).

**SNOPT.** This package [Gill et al. 1998, 2002] minimizes a smooth linear or nonlinear function subject to bounds and sparse linear or nonlinear constraints using SQP. The package may be obtained from Philip Gill (`pgill@ucsd.edu`).

**TAO.** The Toolkit for Advanced Optimization focuses on the design and implementation of component-based optimization software for the solution of large-scale optimization applications. `TAO` is written in C++, is authored by Steve Benson (`benson@mcs.anl.gov`), Lois Curfman McInnes, Jorge Moré and Jason Sarich and may be obtained from <http://www.mcs.anl.gov/tao>.

The interface implementations differ slightly from those of past CUTE releases. If `pak` is a generic name for an interface, the scripts `sdpak` and `pak` are found under `$MYCUTER/bin`. The script `sdpak` sets a number of environment variables and calls `pak` with a command-line option to apply the SIF decoder (see Section 5) to an input problem. `pak` subsequently collects and compiles source and object files as necessary, links them together and executes the resulting program. The script `sdpak` is merely included for backward compatibility with earlier releases of CUTE.

Generic interfacing scripts—`sdgen` and `gen` for Fortran 77 packages, `sdgen90` and `gen90` for Fortran 90/95 packages and `sdgenc` and `genc` for C/C++ packages—may also be found in `$MYCUTER/bin`. These serve the purpose of helping users to design interfaces to new or currently unsupported packages. The corresponding prototype files may be found under the directory `$CUTER/build/prototypes`.

Besides the general CUTEr documentation, man pages describing all supplied interfaces are included in the distribution. Documentation for installing and using the package `pak` may be found in the directory `$CUTER/common/src/pkg/pak`. Note, however, that the supported packages are *not* supplied in CUTEr; rather, directions on how to obtain them are indicated on the official CUTEr website (see Section 7). Object files should be placed where CUTEr can find and link them—for instance in `$MYCUTER/precision/bin`, where `precision` is the working precision. The precision-independent specification files for the package `pak` are found in the directory `$CUTER/common/src/pkg/pak`, whereas if the options specification files depend on the working precision, they are found in `$MYCUTER/precision/specs`.

## 5. A SEPARATE SIF DECODER

In this section, we examine the new design of the SIF decoder. In contrast with earlier versions of CUTE [Bongartz et al. 1995], the SIF decoder is not embedded in CUTEr. We believe that this may be justified for a number of reasons. First,

while the decoder is used intensively by the CUTEr testing environment, there is no *a priori* reason why it should not also be useful in other contexts. As a prime example, the SIF decoder plays a vital role in the package LANCELOT B (an updated version of the LANCELOT package [Conn et al. 1992]) from the GALAHAD [Gould et al. 2002] optimization software library. It is thus more consistent to isolate the decoder and simply have any dependent packages call it as needed. Another reason for our decision is ease of maintenance, and consistency when upgrading the decoder—all the packages that refer to it are then guaranteed to use the same version. Finally, the SIF decoder may evolve in its own right and develop separately. For example, it has recently been extended to generate routines for function evaluation suited for input to the HSL automatic differentiation packages HSL\_ADO1 and its threadsafe counterpart HSL\_ADO2 [HSL 2002]. The resulting isolated decoder has been named SifDec.

### 5.1 A New Design

The design and contents of the SifDec directory tree are very similar to the new design of CUTEr described in Section 2 and reflect similar concerns. The design is summarized in Figure 2. Corresponding environment variables play corresponding roles; the root of the tree is called `$$SIFDEC` while the current instance of SifDec is referred to as `$$MYSIFDEC`. In addition, the `doc` subdirectory contains the complete SIF reference document.

### 5.2 Extensions to the SIF

**5.2.1 Quadratic Programs.** A longstanding wish among CUTE users has been for the SIF format to allow specifically for quadratic objective functions (although it was obviously possible to represent such functions via suitable nonlinear element functions). As a number of extensions to the MPS Linear Programming format from which SIF evolved are in use [IBM 1998; Maros and Mészáros 1998; Ponceleón 1990], we have chosen to extend the original SIF format to handle quadratic parts of the objective function in a more flexible manner. We now briefly describe this extension for the reader already familiar with the SIF format as specified in Conn et al. [1992]. The terminology we use is adopted from there.

In Conn et al. [1992], the objective function is represented as a *group partially separable function* consisting of several potentially nonlinear *groups*. The purpose of our extension is to allow one of the groups to be specified as a quadratic objective group, whose type of nonlinearity is immediately identified by its definition without the need for additional nonlinear group or element functions. More precisely, the objective function is now assumed to have the form

$$f(x) = \sum_{i \in I_0} g_i(\xi_i) + \frac{1}{2} \sum_{j=1}^n \sum_{k=1}^n h_{jk} x_j x_k, \quad \text{with} \quad \xi_i = \sum_{j \in J_i} w_{ij} f_j(\bar{x}_j) + a_i^T x - b_i,$$

where  $x = (x_1, x_2, \dots, x_n)$  and  $\bar{x}_j$  is either a small subvector of  $x$  or such that  $f_j$  has a large invariant subspace. The term  $\frac{1}{2} \sum_{j,k} h_{jk} x_j x_k$  is the *quadratic objective group* and constitutes an extension to the format proposed in Conn et al. [1992]; the leading  $\frac{1}{2}$  is present by convention.

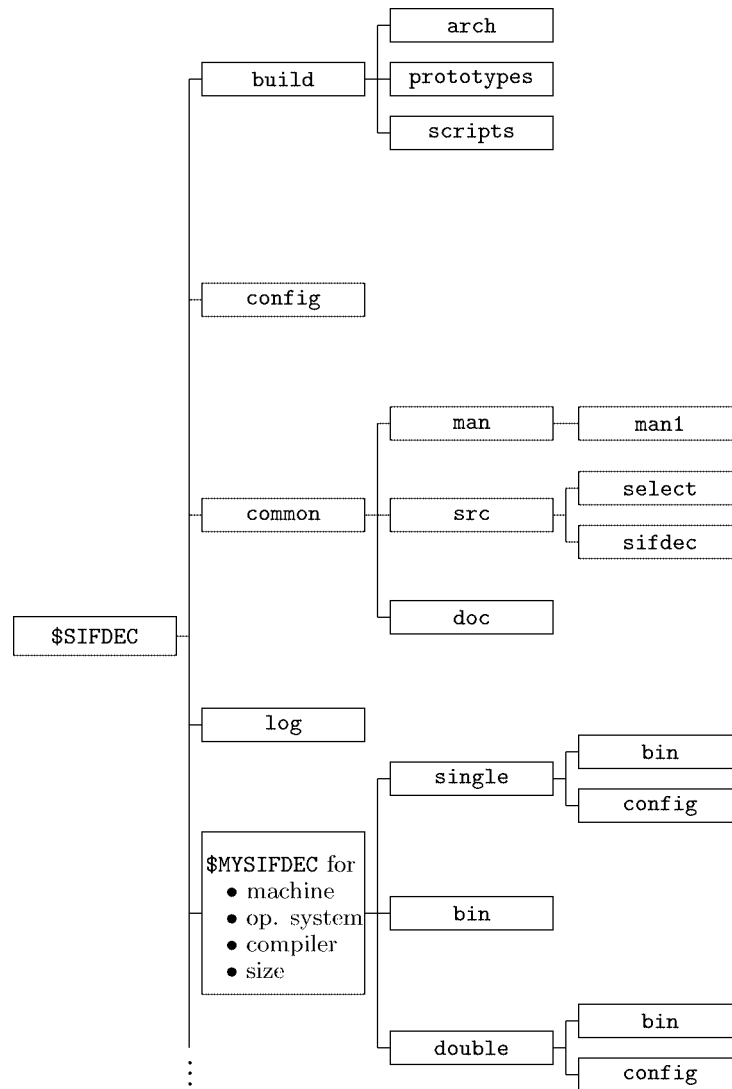


Fig. 2. Structure of the SifDec directories.

*Example 5.1.* In order to fix the ideas, let us consider the problem

$$\text{minimize } f(x_1, x_2) = e^{x_1^2} + x_2^2 + 4x_1x_2.$$

Its objective function then comprises two groups, the first of which ( $e^{x_1^2}$ ) uses a non-trivial nonlinear group function  $g(\alpha) = e^\alpha$ . The rest of the objective function may then be considered as a quadratic objective group and written as

$$\frac{1}{2}(h_{11}x_1x_1 + h_{12}x_1x_2 + h_{21}x_2x_1 + h_{22}x_2x_2),$$

where  $h_{11} = 0$ ,  $h_{12} = h_{21} = 4$  and  $h_{22} = 2$ .

Table III. Possible Data Cards for QUADRATIC, HESSIAN, QUADS, QUADOBJ or QSECTION

F.1	Field 2	Field 3	Field 4	Field 5	Field 6
3	←10→	←10→	←12→	←10→	←12→
QUADRATIC <i>or</i> HESSIAN <i>or</i> QUADS <i>or</i> QUADOBJ <i>or</i> QSECTION					
X	var name	var name	num value	var name	num value
Z	var name	var name	num value	arr name	num value
↑	↑	↑	↑	↑	↑
2	5	15	25	36	40
					50
					61

The quadratic objective group is specified in the SIF file by a new section starting with the keyword (or indicator *card*) QUADRATIC (the cards HESSIAN, QUADS<sup>1</sup>, QUADOBJ<sup>2</sup> and QSECTION<sup>3</sup> are treated as synonyms of QUADRATIC); this section must appear between the sections START POINT and ELEMENT TYPE (see Conn et al. [1992, §7.2.1]).

Within this new section, each line is used to specify at most two values of  $h_{ij}$  that share a common value of  $i$  or  $j$ ; any  $h_{ij}$  not recorded is assumed to have the value zero, only one of the pair  $(h_{ij}, h_{ji})$ ,  $i \neq j$ , should be given, and any repeated values will be summed. The syntax for data following these indicator cards is given in Table III.

The strings var name in data fields 2 and 3 (and optionally 2 and 5 for those cards whose field 1 does not contain Z) give the names of pairs of problem variables  $x_j$  and  $x_k$  for which  $h_{jk}$  is nonzero. All problem variables must have been previously set in the VARIABLES/COLUMNS section. Additionally, on a Z card, the name of the variable must be an element of an array of variables, with a valid name and index, while on an X card, the name may be either a scalar or an array name.

On cards whose field 1 is either empty or contains the character X, the strings num value in data fields 4 and (optionally) 6 contain the associated numerical values of the coefficients  $h_{jk}$ . On cards for which field 1 contains the character Z, the string arr name in data field 5 gives a real parameter array name. This name must have been previously defined and its associated value then gives the numerical value of the parameter.

In Example 5.1, if the variables  $x_1$  and  $x_2$  are named X1 and X2, the QUADRATIC section for this problem takes the form given in Table IV.

This extension to the SIF format has resulted in our including the Maros and Mészáros collection of quadratic programming test problems [Maros and Mészáros 1998] as an annex to the main CUTER collection. The complete

<sup>1</sup>For compatibility with Ponceleón's proposal [Ponceleón 1990].

<sup>2</sup>For compatibility with Maros and Mészáros' test set [Maros and Mészáros 1998].

<sup>3</sup>For compatibility with OSL [IBM 1998].

Table IV. The SIF File Specification for Example 5.1

F.1	Field 2	Field 3	Field 4	Field 5	Field 6
3	← 10 →	← 10 →	← 12 →	← 10 →	← 12 →
QUADRATIC					
	X2	X1	4.0	X2	2.0

test set may be downloaded from the location <http://cuter.rl.ac.uk/cuter-www/mastsif.html>.

**5.2.2 User-Changeable Parameters.** One of the less convenient features of SIF-encoded problems was that the decoding procedures in CUTE were not designed to recognize, nor alter, instance-dependent parameters such as problem dimensions or critical coefficients. Many real models, particularly those arising from some form of discretization, depend upon parameters that a user might wish to refine. With CUTE, a user wishing to change such a parameter was forced to edit the SIF file. This file was usually provided with a number of suggested values, all but one of which were “commented out.” To remove this inconvenience, SifDec makes provisions for both the definition and the altering of “variable parameters” from the problem-decoding scripts.

Any real or integer parameter definition containing the comment \$-PARAMETER in field 5 (i.e., in columns 40–49) defines that parameter to be a *variable* parameter. This is consistent with old-style SIF-encoded problems because strings starting with \$ in this field were previously treated as comments. Any characters after \$-PARAMETER are regarded as comments, and will be passed back to a user on request. All SIF files in the CUTE collection that previously contained variable parameters have been updated to take advantage of this new SifDec facility, but of course they are still consistent with CUTE.

Given this extra syntax, the SIF decoding scripts have been extended to support two new options, allowing users to select variable parameters in the SIF file. The first of these options, `-show`, prints all the variable parameters present in the SIF file, along with suggested values to which they may be set as well as any other provided comments. The second, `-param` allows users to choose, from the command line, which values to assign to these parameters. For instance, assuming that `N` and `THETA` have been marked as variable parameters of `SAMPLE.SIF` and that `N=400` and `THETA=3.5` are valid values, the command

```
sifdecode -param N=400,THETA=3.5 SAMPLE.SIF
```

(see Section 4 for a discussion of the related script `sdpak`, which also inherits these features) will decode `SAMPLE.SIF` into the appropriate subroutines and data files, setting `N` to 400 and `THETA` to 3.5.

These new features allow users to systematically solve a set of problems in all prescribed sizes. Default values are given in each SIF file, and we have taken the opportunity to raise these defaults to reflect the size of problems that we feel ought to be of current interest, given that many of the previous defaults were assigned over ten years ago and are rather small to challenge state-of-the-art solvers.

As an extension of the `-param` command-line option, users may force a problem to be solved using parameter values that may not have been pre-assigned in the SIF file. This is done using the `-force` option, as in

```
sifdecode -force -param N=1000,THETA=3.5 SAMPLE.SIF
```

where `SAMPLE.SIF` might not contain the parameter setting `N=1000`, or `THETA=3.5`. Omitting the `-force` option would result in an abort of the process, while specifying it results in the SIF decoder and the optimizer attempting to complete the solve using the specified values. Since nothing guarantees that these values are valid, the `-force` command-line option should be used carefully.

## 6. THE NEW INSTALLATION PROCEDURES

We now describe the CUTER installation procedure. It applies equally to SifDec, the only difference being the names of the procedures invoked, as we mention at the end of this section.

The installation of CUTER is driven by portability concerns and is performed by the script `install_cuter`, which prompts for information on the local architecture and environment and the desired compiler, and creates the appropriate directory structure but leaves the local installation to *Umakefiles*. *Umakefiles* can be considered as *Makefile* generators, or “meta *Makefiles*” in that they generate *Makefiles* suited to the local platform and architecture without user intervention. Their use is documented within CUTER and they are in fact a simplified flavour of *Imakefiles* [Dubois 1993]. They greatly ease the task of the user when it comes to modifying the size of the CUTER tools and rebuilding part of their instance of CUTER, as *Makefiles* rebuild only what needs to be rebuilt. The *Umakefiles* needed to build a complete instance of CUTER rely on a set of configuration files stored under `$MYCUTER/config`, where the details about the local architecture are contained. Should users need to modify local parameters, they can do so by editing two files: `Umake.tmpl` and the configuration file corresponding to their platform (for instance `sun.cf`, `linux.cf`, `ibm.cf`, etc.). The *Makefiles* then need to be re-generated and CUTER needs to be rebuilt using normal make commands.

The installation script searches the `$CUTER/build/arch/f.arch` file to present a list of possible Fortran compilers to the user. This does not imply that the corresponding compilers are actually installed on the local system but this list is meant to represent the most common compilers on that system. Details about each compiler in the list are found in the file `$CUTER/config/platform.cf` if it is platform-specific or in `$CUTER/config/all.cf` if it is available on all platforms. Similarly, the file `$CUTER/build/arch/c.arch` is searched for possible C/C++ compilers. Addition of a compiler can normally be achieved by modifying one “similar” to those provided. The files `$CUTER/config/platform.cf` and `$CUTER/config/all.cf` should be checked before the installation procedure is initiated.

The configuration files also provide basic system commands and definition of a temporary directory. Some or all of these files may need to be modified, although suitable settings are given for systems we have access to.

During installation, the option to choose between small, medium, large, or custom “sizes” for CUTER is provided. These sizes come pre-specified, but may be tuned by editing the `size.*` files in the directory `$CUTER/build/arch` and reissuing the install command.

The installation procedure works by casting *prototype files* against the system, compiler, precision and size information chosen by the user, casting the Fortran source files following the same pattern, and compiling and possibly linking the result. As before, the term *casting* refers here to the process of specializing the files to the host architecture and user selections. Each installation is logged, both for information purposes and with subsequent un-installation possibilities in mind. Un-installing an installed CUTER is carried out by the script `uninstall_cuter`, which also updates the log file. The CUTER tools, documentation, scripts, or other files may be updated by the script `update_cuter`, as updates and bug fixes become available.

The installation procedure for SifDec is identical, with the proviso that the names `install_script_cuter`, `install_cuter`, `update_cuter`, `uninstall_cuter`, `CUTER` and `MYCUTER` should instead be interpreted as `install_script_sifdec`, `install_sifdec`, `update_sifdec`, `uninstall_sifdec`, `SIFDEC`, and `MYSIFDEC` respectively.

## 7. OBTAINING CUTER AND SifDec

CUTER and SifDec are written in standard ISO Fortran 77, but CUTER also provides support for Fortran 90/95 and C/C++ packages. Single and double precision versions are available in a variety of sizes. Machine dependencies are carefully isolated and easily adaptable, making installation on heterogeneous networks possible. Automatic installation procedures are available for a variety of Unices, including LINUX. CUTER and SifDec can be downloaded from

<http://cuter.rl.ac.uk/cuter-www>, and  
<http://cuter.rl.ac.uk/cuter-www/sifdec>

respectively. Information on updates as well as indications on how to obtain the supported optimization and linear algebra software is available on the websites.

## 8. CONCLUSION AND PERSPECTIVES

This paper describes improvements and new features of CUTER, the latest release of the CUTE testing environment for optimization solvers, and of SifDec, the isolated SIF decoder. The purposes of CUTER are to

- provide a way to explore an extensive collection of problems,
- provide a way to compare existing solution packages,
- provide a way to use a large test problem collection with new packages,
- provide motivation for building a meaningful set of new interesting test problems,
- provide ways to manage and update the system efficiently, and
- do all the above on a variety of popular platforms.



SifDec has been isolated and designed in order to

- supply a consistent interface to any package that may require the decoder, such as CUTER and LANCELOT-B [Conn et al. 1992],
- ease its maintenance, upgrading, and addition of new capabilities,
- provide access to automatic differentiation packages.

The environments are currently only available for UNIX platforms, but it is possible to install both packages on shared-filesystem local networks, because machine dependencies have been carefully isolated. A number of previously unsupported optimization and linear algebra packages are now interfaced to CUTER, and corresponding driver programs are supplied. New tools for both constrained and unconstrained optimization have been added. Some support for automatic differentiation packages is now integrated into SifDec. Documentation now appears in different forms, including the usual UNIX manual pages describing the tools and interfaces, postscript and pdf general documentation covering installation, maintenance, and usage. Additional details will be provided on the dedicated Websites. It is hoped that installing CUTER and SifDec on currently unsupported UNIX platforms, as well as writing interfaces for additional optimization packages, will be found to be relatively easy.

In the future, we plan to merge the different CUTER tools so as to remove their dependency on whether the input problem is constrained or not, and have a single consistent set of tools. We also intend to use automatic memory allocation to remove the dependency of both the SIF decoder and the CUTER tools on preselected sizes. An intuitive graphical user interface (GUI) is under way to ease the installation phase, to manage the different local installations of CUTER and SifDec, and to enable the user to work in a unified environment. As already mentioned, the Websites will keep up-to-date information about new features for both packages, bug fixes, new documentation, and more.

## APPENDIX

### A. CALLING SEQUENCES FOR THE NEW EVALUATION TOOLS

In this section, we give the argument lists for those subroutines summarized in Tables I and II that are new to CUTER; the remaining subroutines are documented in the appendix to [Bongartz et al. 1995]. There are two sets of tools: one set for unconstrained and bound constrained problems, and one set for generally constrained problems. Note that these two sets of tools cannot be mixed and that C/C++ and modern Fortran compilers are case insensitive.

The superscript  $i$  on an argument means that the argument must be set on input. A superscript  $o$  means that the argument is set by the subroutine.

#### A.1 Unconstrained and Bound Constrained Problems

- Discover how many variables are involved in the problem:  
call `udimen( input $i$ , n $o$  )`
- Determine how many nonzeros are required to store the Hessian matrix of the objective function (when stored in a sparse format):  
call `udimsh( nnzh $o$  )`

- Determine how many nonzeros are required to store the Hessian matrix of the objective function (when stored as a sparse matrix in finite-element format):  
call `udimse( neo, nzho, nzirnho )`
- Obtain the type of each variable:  
call `uvarty( ni, ivartyo )`
- Obtain statistics concerning function evaluation and CPU time use:  
call `ureprt( ucallso, timeo )`

### A.2 Generally Constrained Problems

- Discover how many variables and constraints are involved in the problem:  
call `cdimen( inputi, no, mo )`
- Determine how many nonzeros are required to store the matrix of gradients of the objective function and constraints (when stored in a sparse format):  
call `cdimsj( nnzjo )`
- Determine how many nonzeros are required to store the Hessian matrix of the Lagrangian (when stored in a sparse format):  
call `cdimsh( nnzho )`
- Determine how many nonzeros are required to store the Hessian matrix of the Lagrangian (when stored as a sparse matrix in finite-element format):  
call `cdimse( neo, nzho, nzirnho )`
- Obtain the type of each variable:  
call `cvarty( ni, ivartyo )`
- Evaluate an individual constraint function and possibly its gradient (when this is stored in a sparse format):  
call `ccifsg( ni, ii, xi, cio, nnzsgco, lsgcii, sgcio, ivsgcio, gradi )`
- Obtain statistics concerning function evaluation and CPU time use:  
call `creprt( ccallso, timeo )`

### A.3 Argument Descriptions

The arguments in the above calling sequences have the following meanings:

- `ccalls` is an array whose components give counts for various activities during the current execution of the constrained tools. Components are:
- `ccalls(1)` number of objective function evaluations
  - `ccalls(2)` number of objective gradient evaluations
  - `ccalls(3)` number of objective Hessian evaluations
  - `ccalls(4)` number of Hessian-vector products
  - `ccalls(5)` number of constraint evaluations
  - `ccalls(6)` number of constraint Jacobian evaluations
  - `ccalls(7)` number of constraint Hessian evaluations
- `ci` is the value of the general constraint function `i` evaluated at `x`.
- `grad` is a logical variable, which should be set `.true.` if the gradient of the constraint function is required from `ccifsg`. Otherwise, it should be set `.false.`

<code>i</code>	is the index of the general constraint function to be evaluated by <code>ccifsg</code> .
<code>input</code>	is the unit number for the decoded data, i.e., from which <code>OUTSDIF.d</code> [Bongartz et al. 1995] is read.
<code>ivarty</code>	is an array whose $i$ -th component indicates the type of variable $i$ . Possible values are 0 (a variable whose value may be any real number), 1 (an integer variable that can only take the values zero or one) and 2 (a variable that can only take integer values).
<code>ivsgci</code>	is an array whose $i$ -th component is the index of the variable with respect to which <code>sgci(<math>i</math>)</code> is the derivative.
<code>lsgci</code>	is the declared dimension of <code>sgci</code> .
<code>m</code>	is the number of general constraints.
<code>n</code>	is the number of variables for the problem.
<code>ne</code>	is the number of elements in a finite-element representation of the Hessian for the problem.
<code>nzh</code>	is the dimension of the array needed to store the real values of the finite-element Hessian.
<code>nzirn</code>	is the dimension of the array needed to store the integer values of the finite-element Hessian.
<code>nnzh</code>	is the number of nonzeros in the Hessian.
<code>nnzj</code>	is the number of nonzeros in the constraint Jacobian.
<code>nnzsgc</code>	is the number of nonzeros in <code>sgci</code> .
<code>sgci</code>	is an array that gives the values of the nonzeros of the gradient of the general constraint function <code>i</code> evaluated at $x$ . The $i$ -th entry of <code>sgci</code> gives the value of the derivative with respect to variable <code>ivsgci(<math>i</math>)</code> of function <code>i</code> .
<code>time</code>	is an array whose components give CPU times (in seconds) for various activities during the current execution of the tools. Components are: <code>time(1)</code> CPU time for call to <code>usetup/csetup</code> . <code>time(2)</code> CPU time since last call to <code>usetup/csetup</code> .
<code>ucalls</code>	is an array whose components give counts for various activities during the current execution of the unconstrained tools. Components are: <code>ucalls(1)</code> number of objective function evaluations <code>ucalls(2)</code> number of objective gradient evaluations <code>ucalls(3)</code> number of objective Hessian evaluations <code>ucalls(4)</code> number of Hessian-vector products
<code>x</code>	is an array that gives the current estimate of the solution of the problem.

## ACKNOWLEDGMENTS

Thanks to the following people for providing interfaces: Philip Gill for `SNOPT`, Jorge Nocedal and Richard Waltz for `KNITRO`, Sven Leyffer and Roger Fletcher for `filtersQP`. We also wish to thank CUTE users for their comments, bug reports,

use, abuse and contributions. Finally, detailed comments by Michael Saunders and an anonymous referee have been most useful.

## REFERENCES

- BIELSCHOWSKY, R. H. AND GOMES, F. A. M. 1998. Dynamical control of infeasibility in nonlinearly constrained optimization. Presentation at the Optimization 98 Conference, Coimbra.
- BONGARTZ, I., CONN, A. R., GOULD, N. I. M., AND TOINT, PH. L. 1995. CUTE: Constrained and Unconstrained Testing Environment. *ACM Trans. Math. Soft.* 21, 1, 123–160.
- BREITFELD, M. G. AND SHANNO, D. F. 1994. Preliminary computational experience with modified log-barrier functions for large-scale nonlinear programming. In W. W. Hager, D. W. Hearn, and P. M. Pardalos, Eds. *Large Scale Optimization: State of the Art*, pages 45–66. Dordrecht, The Netherlands, Kluwer Academic Publishers.
- BREITFELD, M. G. AND SHANNO, D. F. 1996. Computational experience with penalty-barrier methods for nonlinear programming. *Annals of Operations Research*, 62, 439–463.
- BRENT, R. P. 1973. Algorithms for Minimization without Derivatives. Prentice-Hall, Englewood Cliffs, NJ.
- BYRD, R. H., GILBERT, J. CH., AND NOCEDAL, J. 2000. A trust region method based on interior point techniques for nonlinear programming. *Math. Prog.* 89, 1, 149–185.
- BYRD, R. H., NOCEDAL, J., AND WALTZ, R. A. 2000. Feasible interior methods using slacks for nonlinear optimization. Tech. Rep. 11, Optimization Technology Center, Argonne National Laboratory, Argonne, IL, USA.
- COLEMAN, T. F. AND YUAN, W. 1995. A new trust region algorithm for equality constrained optimization. Report TR95-1477, Department of Computer Science, Cornell University, Ithaca, NY.
- CONN, A. R., GOULD, N. I. M., ORBAN, D., AND TOINT, PH. L. 2000. A primal-dual trust-region algorithm for non-convex nonlinear programming. *Math. Prog.* 87, 2, 215–249.
- CONN, A. R., GOULD, N. I. M., AND TOINT, PH. L. 1992a. LANCELOT: A Fortran Package for Large-scale Nonlinear Optimization (Release A). Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, NY.
- CONN, A. R., GOULD, N. I. M., AND TOINT, PH. L. 1999a. A primal-dual algorithm for minimizing a nonconvex function subject to bound and linear equality constraints. In G. Di Pillo and F. Giannessi, Eds. *Nonlinear Optimization and Related Topics*, pages 15–50. Dordrecht, The Netherlands, Kluwer Academic Publishers.
- CONN, A. R., VICENTE, L. N., AND VISWESWARIAH, C. 1999b. Two-step algorithms for nonlinear optimization with structured applications. *SIAM J. on Optimization* 9, 4, 924–947.
- DENNIS, J. E., EL-ALEM, M., AND WILLIAMSON, K. A. 1999. A trust-region approach to nonlinear systems of equalities and inequalities. *SIAM J. Optimization* 9, 2, 291–315.
- DINIZ-EHRHARDT, M. A., GOMES-RUGGIERO, M. A., AND SANTOS, S. A. 1997. Comparing the numerical performance of two trust-region algorithms for large-scale bound-constrained minimization. *Revista Latino Americana de Investigación Operativa* 7, 23–54.
- DINIZ-EHRHARDT, M. A., GOMES-RUGGIERO, M. A., AND SANTOS, S. A. 1998. Numerical analysis of leaving-face parameters in bound-constrained quadratic minimization. Report 52/98, Department of Applied Mathematics, IMECC-UNICAMP, Campinas, Brasil.
- DUBOIS, P. 1993. *Software Portability with imake*. O'Reilly & Associates, Inc.
- DUFF, I. S., ERISMAN, A. M., AND REID, J. K. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, England.
- DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. 1999. Sparse matrix test problems. *ACM Trans. Math. Soft.* 15, 1, 1–14.
- DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. 1992. Users' guide for the Harwell-Boeing sparse matrix collection (Release 1). Report RAL-92-086, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.
- DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. 1997. The Rutherford-Boeing sparse matrix collection. Report RAL-TR-97-031, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.
- FLETCHER, R., GOULD, N. I. M., LEYFFER, S., TOINT, PH. L., AND WÄCHTER, A. 2002. Global convergence of a trust-region SQP-filter algorithm for nonlinear programming. *SIAM J. Optimization* 13, 3, 635–659.

- FLETCHER, R. AND LEYFFER, S. 2002. Nonlinear programming without a penalty function. *Math. Prog.* 91, 2, 239–269.
- FLETCHER, R. AND LEYFFER, S. 1998. User manual for filterSQP. Numerical Analysis Report NA/181, Department of Mathematics, University of Dundee, Dundee, Scotland.
- GERTZ, E. M. 1999. *Combination Trust-Region Line-Search Methods for Unconstrained Optimization*. PhD thesis, Department of Mathematics, University of California, San Diego, CA.
- GILL, P. E., MURRAY, W., AND SAUNDERS, M. A. 1998. User's guide for SNOPT 5.3: a Fortran package for large-scale nonlinear programming.
- GILL, P. E., MURRAY, W., AND SAUNDERS, M. A. 2002. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM J. Optimization* 12, 4, 979–1006.
- GOMES, F. A. M., MACIEL, M. C., AND MARTÍNEZ, J. M. 1999. Nonlinear programming algorithms using trust regions and augmented Lagrangians with nonmonotone penalty parameters. *Math. Prog.* 84, 1, 161–200.
- GOULD, N. I. M., LUCIDI, S., ROMA, M., AND TOINT, PH. L. 1999. Solving the trust-region subproblem using the Lanczos method. *SIAM J. Optimization* 9, 2, 504–525.
- GOULD, N. I. M. AND NOCEDAL, J. 1998. The modified absolute-value factorization norm for trust-region minimization. In R. De Leone, A. Murli, P. M. Pardalos, and G. Toraldo, Ed. *High Performance Algorithms and Software in Nonlinear Optimization*, pages 225–241. Kluwer Academic Publishers, Dordrecht, The Netherlands.
- GOULD, N. I. M., ORBAN, D., AND TOINT, PH. L. 2002. GALAHAD—a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. Report RAL-TR-2002-014, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.
- HSL. 2002. A collection of Fortran codes for large scale scientific computation.
- IBM. 1998. Optimization Solutions and Library. *QP Solutions User Guide*. IBM Corporation.
- LALEE, M., NOCEDAL, J., AND PLANTENGA, T. D. 1998. On the implementation of an algorithm for large-scale equality constrained optimization. *SIAM J. Optimization* 8, 3, 682–706.
- MARAZZI, M. AND NOCEDAL, J. 2000. Wedge trust region methods for derivative free optimization. Report 2000/10, Optimization Technology Center, Northwestern University, Evanston, IL.
- MAROS, I. AND MESZAROS, C. 1998. A repository of convex quadratic programming problems. *Optimization Methods and Software* 11–12, 671–681.
- PERNICE, M. AND WALKER, M. 1998. NITSOL: A Newton Iterative Solver for Nonlinear Systems. *SIAM J. Sci. Comput.* 19, 1, 302–318.
- PONCELEÓN, D. B. 1990. *Barrier Methods for Large-Scale Quadratic Programming*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA.
- PRYCE, J. D. AND REID, J. K. 1998. ADO1, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.
- REID, J. K. 2001. LAO4, a Fortran 77 code for steepest-edge simplex linear programming. Report RAL-TR-2001-000, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.
- SARGENT, R. W. H. AND ZHANG, X. 1998. An interior-point algorithm for solving general variational inequalities and nonlinear programs. Presentation at the Optimization 98 Conference, Coimbra.
- SARTENAER, A. 1997. Automatic determination of an initial trust region in nonlinear programming. *SIAM J. on Scientific Computing* 18, 6, 1788–1803.
- SHAHABUDDIN, J. S. 1996. *Structured Trust-Region Algorithms for the Minimization of Nonlinear Functions*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York.
- ULBRICH, S. AND ULBRICH, M. 1999. Nonmonotone trust region methods for nonlinear equality constrained optimization without a penalty function. Presentation at the First Workshop on Nonlinear Optimization, “Interior-Point and Filter Methods,” Coimbra, Portugal.
- VANDERBEL, R. J. AND SHANNO, D. F. 1997. An interior point algorithm for nonconvex nonlinear programming. Tech. Rep. SOR 97-21, Princeton University, NJ.
- WÄCHTER, A. 2002. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.
- XIAO, Y. 1996. *Non-Monotone Algorithms in Optimization and their Applications*. PhD thesis, Monash University, Clayton, Australia.
- XIAO, Y. AND CHU, E. K. W. 1995. Nonmonotone trust region methods. Report 95/17, Monash University, Clayton, Australia.

YAMASHITA, H., YABE, H., AND TANABE, T. 1997. A globally and superlinearly convergent primal-dual point trust-region method for large scale constrained optimization. Report, Mathematical Systems, Inc., Sinjuku-ku, Tokyo, Japan.

ZHU, C., BYRD, R. H., LU, P., AND NOCEDAL, J. 1997. Algorithm 778. L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization. *ACM Trans. Math. Soft.* 23, 4, 550–560.

Received March 2002; revised March 2003 and August 2003; accepted August 2003