CERFACS **Technical Report No. TR/PA/01/04**

Nicholas I. M. Gould, Dominique Orban and Philippe L. Toint

CUTEr (and SifDec), a Constrained and Unconstrained Testing Environment, revisited*

Abstract. The initial release of CUTE, a widely used testing environment for optimization software was described in [2]. The latest version, now known as CUTEr is presented. New 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 SIF decoder, which used to be 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.

Key words. Nonlinear constrained optimization, testing environment, shared filesystems, heterogeneous environment, SIF format

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 [9]. 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 paper [2] 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 [1,3,4,6,7,10,11,12,13,14,21,23,25,26,27,31,32,36,37,38,39,40,41,42]. However, such widespread use has inevitably lead 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 paper aims to describe its next major evolution: CUTEr, in which we revisit the original CUTE design. This new release is characterized by

N. I. M. Gould: Rutherford Appleton Laboratory, Computational Science and Engineering Departement, Chilton, Oxfordshire, England. e-mail: n.gould@rl.ac.uk

D. Orban: CERFACS, 42 Avenue Gaspard Coriolis, 31057 Toulouse Cedex 1, France. e-mail: Dominique.Orban@cerfacs.fr

Ph. L. Toint: Facultés Universitaires Notre-Dame de la Paix, 61, rue de Bruxelles, B-5000 Namur, Belgium. e-mail: Philippe.Toint@fundp.ac.be

* This work was supported by the MNRT Grant for joint Ph.D. Support.

- a set of new tools, including a unified facility to report the performance of the various optimization packages being tested,
- a set of new interfaces to additional optimization packages, and
- some fortran 90/95 support.

The SIF optimization test-problem decoder, which used to be a constitutent part of the CUTE environment, has been isolated into a separate package named SifDec. Any software which 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 [29] `AD01` and `AD02` packages [35].

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 paper uniquely 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 since CUTE offered VMS and some DOS support, but this merely reflects our current expertise.

The paper is intended to supersede the parts of [2] that are obsolete in CUTEr, to complement it in order 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 then documents the new tools, Section 4 the new interfaces to additional optimization packages, Section 5 covers the isolated SIF decoder environment, the extension of 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 simultaneously support a multi-platform 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) at the same time. Moreover, using CUTE on a single machine in conjunction with several different compilers (a case that frequently occurs when testing new software) is extremely cumbersome. Likewise, handling different instance 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 [2], does not lend itself to such use, since 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 used.

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 which is itself disjoint from the dependent subtrees. Finally, the multiplication 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 2.1, reflects these preoccupations. We now briefly describe its components.

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 which 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. Its first subdirectory, `doc`, contains a number of documentation files concerning the environment (such as a description of its structure and the description of procedure to follow for interfacing the supported optimization packages), but not a description of the CUTEr tools and scripts themselves. These are documented in the `man` subdirectory (and, as is common on UNIX systems, its `man1` and `man3` subdirectories). The `src` subdirectory contains a number of subdirectories that contain the source files for many of the environment utilities: `tools` contains the sources of the Fortran tools used in user's programs, while `matlab` contains all the "m-files" that provide a MATLAB interface to the environment, and `pkg` holds information related to the various optimization packages for which CUTEr provides an interface—the `pkg` subdirectory itself contains a number of subdirectories, one for each supported package (we have represented those for the COBYLA and `HSL_VE12` packages), typically including an algorithmic specification file and a suitable README description of how to build an interface between CUTEr and the package. The last subdirectory of `common`, `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 which are relevant to *imake*, which are needed when

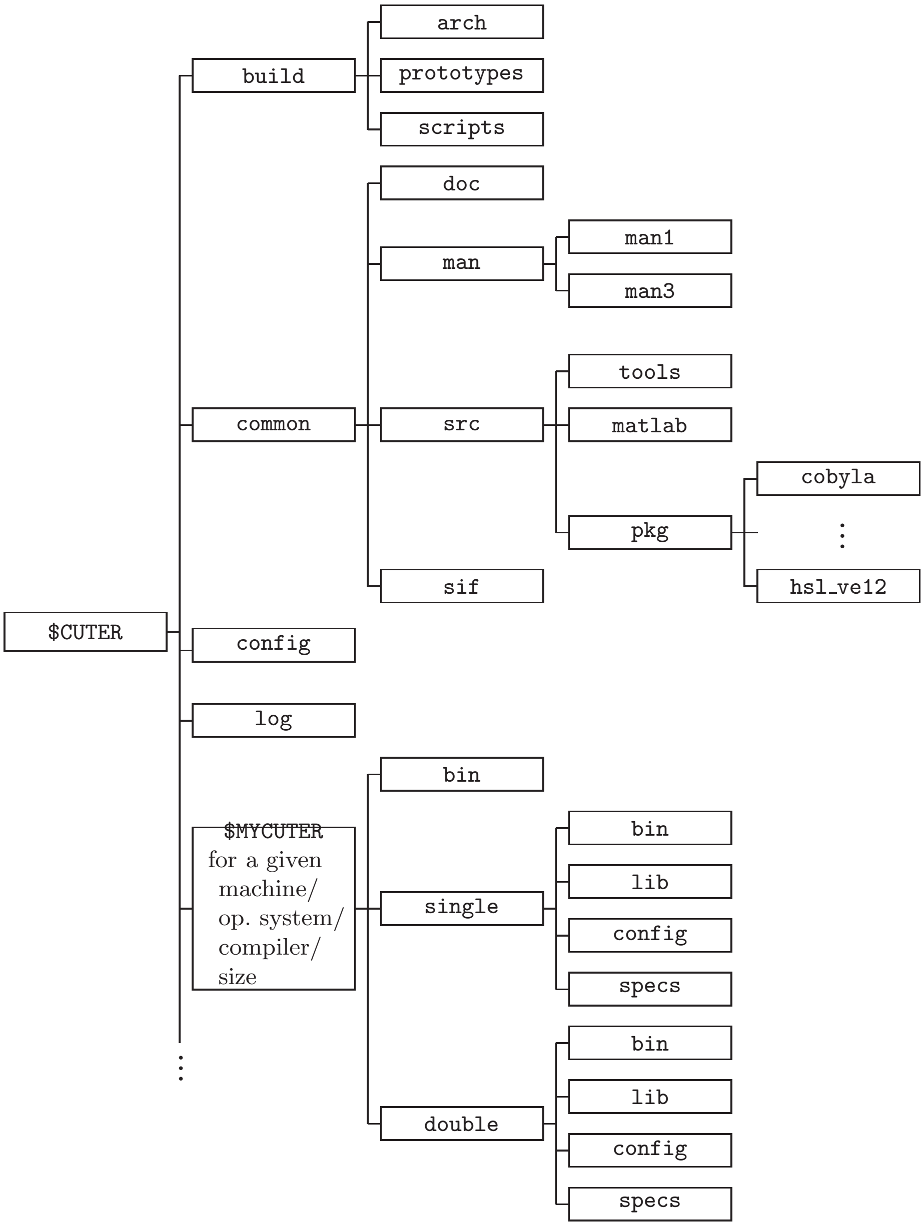

Fig. 2.1. Structure of the CUTEr directories

the latter is used to *bootstrap* the various *Imakefiles* in order to create the necessary *Makefiles*.

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

The remaining subdirectories of `$CUTER` are all architecture dependent: each corresponds to the installation of CUTEr on a specific machine, for a given operating system and compiler and for a given tool size. The figure only represents one, but the continuation dots at the bottom of the leftmost vertical line indicate that there might be more than one. Although these directories have been symbolically represented as subdirectories of `$CUTER` on the figure, to reflect their dependence upon `$CUTER`, they may be located anywhere on the host system, including on remote machines over the local network. The name of these directories are (by default) automatically chosen at installation, but a user of one of these subtrees would typically give it a symbolic name, like `$MYCUTER`, to distinguish the version of CUTEr currently in use. Each architecture-dependent subtree is divided into its single precison and double precision instances (`single` and `double`, respectively), each of these containing in turn four subdirectories. The first, `bin`, contains the object files corresponding to the driving programs for the optimization packages and, if relevant, of the package codes themselves. It should also contain the Fortran 90/95 module information files, if applicable (usually called `*.mod` files). 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 those scripts which are architecture, but not precision, -dependent.

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 novel feature. This allows a much simpler design of new optimization package interfaces, since the interface no longer needs specify the exact list of tools which need to be loaded together 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. It is hoped that this will make access to the relevant information more convenient for users.

3. New tools

CUTEr tools for unconstrained and constrained optimization are presented in Table 3.1 and Table 3.2 respectively, accompanied by a brief description. Whenever the description states that the Hessian matrix of either the objective or the Lagrangian function is in sparse format, it is implicitly understood that it is stored in coordinate format [16, §2.6]; explicit mention is made whenever this matrix is instead stored in finite-element format. 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, both for 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 `udimsh` determine the number of variables involved, and the numbers of nonzeros in the Hessian if 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 supplied 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. The corresponding package source, for example, `pak.f` (which is *not* supplied with CUTEr) needs to be compiled (but not linked) to `$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 §4.

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

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

Table 3.1. The unconstrained 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
csgr	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
csgrsh	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

Table 3.2. The constrained optimization CUTEr tools.

offered with CUTE. The purpose of providing these interfaces is to allow researchers and practitioners to run a variety of solvers on a consistent set of test examples, and thus to assess which algorithm is likely to be the most suitable for solving classes of related problems. The newly supported packages are:

Praxis. Praxis is Chandler's implementation of Brent's algorithms for minimization without derivatives. It is available from John Chandler, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma 74078, USA (`jpc@a.cs.okstate.edu`).

L-BFGS-B. This package is for unconstrained or bound constrained problems, and uses a limited memory BFGS quasi-Newton update. The package L-BFGS-B [43] is available from Jorge Nocedal, ECE department, Nothwestern University, Evanston IL 60208-3118, USA (`nocedal@ece.northwestern.edu`).

SNOPT. CUTEr interfaces the latest version, SNOPT 6.1 [24]. SNOPT minimizes a (smooth) linear or nonlinear function subject to bounds and sparse linear or nonlinear constraints using sequential quadratic programming. The package may be obtained from Philip Gill (`pgill@ucsd.edu`).

KNITRO. KNITRO minimizes a smooth nonlinear function subject to nonlinear equality and inequality constraints using an interior-point approach. The resulting barrier subproblems are treated using sequential quadratic programming. The KNITRO software [5] is maintained by Jorge Nocedal (`nocedal@ece.northwestern.edu`) and Richard Waltz (`rwaltz@ece.northwestern.edu`).

filterSQP. This nonlinear programming package uses the recent-proposed filter idea [20,21,22], is globalized with a trust region and solves a quadratic programming subproblem at each iteration. The filterSQP package is maintained by Roger Fletcher (`fletcher@maths.dundee.ac.uk`) and Sven Leyffer (`sven@maths.dundee.ac.uk`).

HRB. HRB converts matrices (for example, Hessians, Jacobians, and KKT augmented system matrices) derived from SIF problem data into Harwell-Boeing [17,18] or Rutherford-Boeing [19] 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 a interior-point trust-region algorithm [8]. `HSL_VE12` is part of HSL [29] and was written by Nick Gould and Philippe Toint.

The implementation of the interfaces differ slightly from that 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` applies the SIF decoder (see Section 5) to an input problem, sets a number of environment variables, collects and compiles source and object files as necessary, links them together and executes the resulting program. The script `pak` is similar to `sdpak` except it assumes the input problem has already been decoded.

Generic interfacing scripts—`sdgen` and `gen`—may also be found in the directory `$MYCUTER/bin`, and these serve the purpose of helping users to design an interface to a new, or currently-unsupported, package. The corresponding prototype files may be found under the directory `$CUTER/build/prototypes`.

Besides the general CUTEr documentation, man pages describing all supplied interfaces are included in the distribution. Documentation on the package *pak* and on how to compile the related sources may be found under the directory `$CUTER/common/src/pkg/`*pak*. Note however that the supported packages are *not* supplied in CUTEr. Object files resulting from the compilation of these sources should be placed somewhere where CUTEr can find and link them, for instance in `$MYCUTER/`*precision*`/bin`, where *precision* is the working precision. The precision-independent specification file for the package *pak* are found under the directory `$CUTER/common/src/pkg/`*pak*, whereas if the options specification files depend on the working precision, they are found under `$MYCUTER/`*precision*`/specs`.

5. An isolated SIF decoder

In this section, we examine the new design of the SIF decoder. In contrast with earlier version of CUTE [2], the SIF decoder is no longer embedded in CUTEr. We believe that this may be justified for a number of reasons. Firstly, while the decoder is used intensively by the CUTEr testing environment, there is no *a priori* reason why it shouldn't also be useful in other contexts. As a prime example, the SIF decoder plays a vital role in the upcoming package LANCELOT B (an updated version of the LANCELOT package [9]), from the GALAHAD [28] 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 which refer to it are then guaranteed to use the same version. Finally, the SIF decoder may evolve in its own right and develop separately. An illustration of this fact is that it has recently been extended so as to generate routines for function evaluation suited for input to the HSL automatic differentiation packages `HSL_AD01` and its threadsafe counterpart `HSL_AD02` [29]. The resulting package containing the isolated decoder has been named SifDec.

5.1. A new design

The design and contents of the SifDec directory tree is very similar to the new design of CUTEr, described in section 2, reflects similar concerns, and is depicted in Fig. 5.1. Corresponding environment variables play the 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 of the SIF

5.2.1. Quadratic programs A long source of irritation for CUTE users was that the SIF representation of test problems did not explicitly allow for quadratic objective functions (although it was obviously possible to represent such function via the definition of suitable nonlinear element functions). Since this situation arises frequently (most especially in quadratic programming), and as a number of similar extensions to the MPS Linear Programming format from which SIF evolved are in use [30,33,34], we have chosen to extend the original definition of the 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 definition of the SIF format as specified in [9]. The terminology we used is adopted from there.

In [9], 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, in addition, one of the groups to be specified as a quadratic objective group, that is a group whose type of nonlinearity is immediately specified by its definition, without the need to define additional nonlinear group or element functions. More precisely, the objective function is now assumed to have

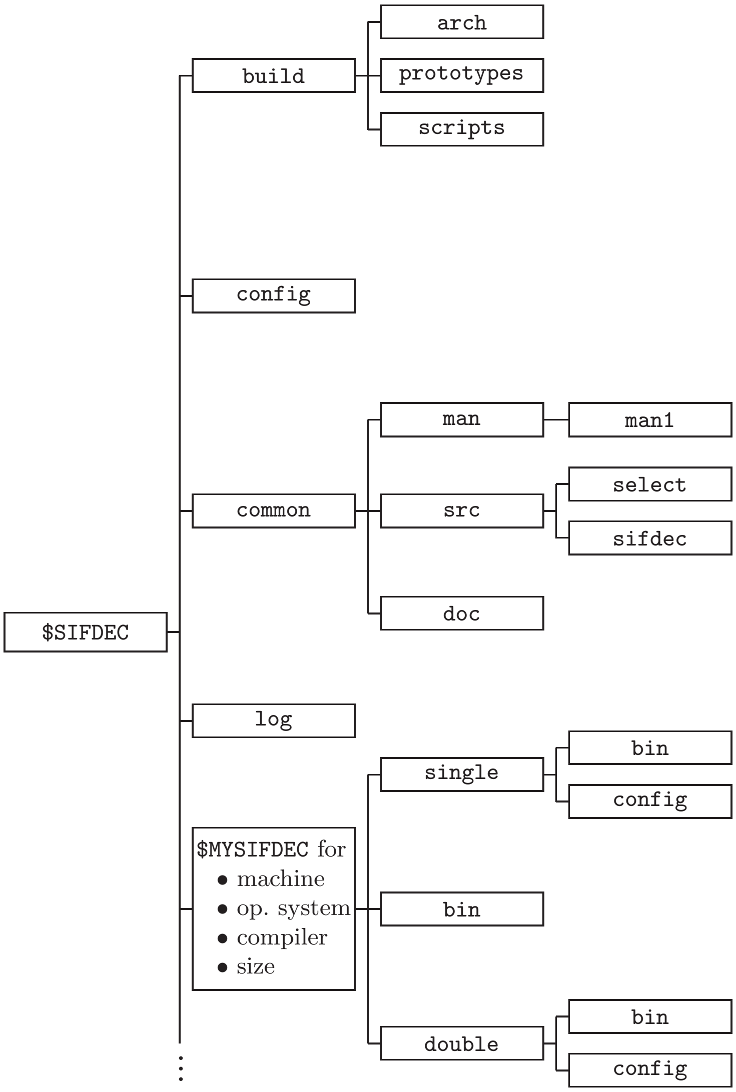

Fig. 5.1. Structure of the SifDec directories

the form

$$f(x) = \sum_{i \in I_O} g_i \left(\sum_{j \in J_i} w_{i,j} f_j(\bar{x}_j) + a_i^T x - b_i \right) + \tfrac{1}{2} \sum_{j=1}^{n} \sum_{k=1}^{n} h_{j,k} x_j x_k,$$

where $x = (x_1, x_2, \ldots, x_n)$. The additional term $\frac{1}{2}\sum_{j=1}^{n}\sum_{k=1}^{n} h_{j,k}x_j x_k$ in the objective function is the *quadratic objective group* and constitutes an extension to the format proposed in [9]; the leading $\frac{1}{2}$ is present by convention.

In order to fix the ideas, let us consider the optimization 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

$$\tfrac{1}{2}(h_{1,1}x_1x_1 + h_{1,2}x_1x_2 + h_{2,1}x_2x_1 + h_{2,2}x_2x_2),$$

where $h_{1,1} = 0$, $h_{1,2} = h_{2,1} = 4$ and $h_{2,2} = 2$.

The quadratic objective group is specified in the `SIF` file by using an additional section starting with the keyword (or indicator *card*) `QUADRATIC` (the cards `HESSIAN`, `QUADS`[1], `QUADOBJ`[2] and `QSECTION`[3] are treated as synonyms of `QUADRATIC`); this section must appear between the `START POINT` and `ELEMENT TYPE` sections (see [9, §7.2.1]).

Within this new section, each line is used to specify at most two values of $h_{i,j}$ that share a common value of i or j; any $h_{i,j}$ not recorded is assumed to have the value zero, only one of the pair $(h_{i,j}, h_{j,i}), i \neq j$, should be given, and any repeated values will be summed. The syntax for data following these indicator cards is given in Table 5.1.

<> F.1	<—10—> Field 2	<—10—> Field 3	<—-12—-> Field 4	<—10—> Field 5	<—-12—-> Field 6
QUADRATIC or HESSIAN or QUADS or QUADOBJ or QSECTION					
	varbl-name	varbl-name	numerical-vl	varbl-name	numerical-vl
X	varbl-name	varbl-name	numerical-vl	varbl-name	numerical-vl
Z	varbl-name	varbl-name		r-p-a-name	
2 3	5 14	15 24	25 36	40 49	50 61

Table 5.1. Possible data cards for `HESSIAN`, `QUADS`, `QUADOBJ` or `QSECTION`

The strings `varbl-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 $h_{j,k}$ 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 a `V` 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 `numerical-vl` in data fields 4 and (optionally) 6 contain the associated

[1] For compatibility with Ponceleón's proposal [34].

[2] For compatibility with Maros and Meszaros' test set [33].

[3] For compatibility with OSL [30].

numerical values of the coefficients $h_{j,k}$. On cards for which field 1 contains the character `Z`, the string `r-p-a-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.

Returning to our example above, and assuming that the variables x_1 and x_2 are named `X1` and `X2`, the `QUADRATIC` section for this problem then takes the form given in Table 5.2.

<>	<—10—>	<—10—>	<—-12—->	<—10—>	<—-12—->
F.1	Field 2	Field 3	Field 4	Field 5	Field 6
QUADRATIC					
	X1	X1	2.0	X2	4.0

Table 5.2. The SIF file specification for the example

This extension to the SIF format has resulted in our including the Maros and Meszaros collection of quadratic programming test problems [33] as an annex to the main CUTEr collection. The complete 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 recognise, nor to alter, instance-dependent variable parameters such as problem dimensions or critical coefficients. Many real models, particularly those that arise 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—these files were usually provided with a number of suggested values, all but one of which were "commented out". Since a number of users found this to be very inconvenient, SifDec makes provisions both for the definition and for the altering of variable parameters from the problem-decoding scripts.

Any real or integer parameter definition containing the comment `$-PARAMETER` in field 5 (i.e., in columns 40-50) in a SIF file defines that parameter to be a *variable* parameter—this is consistent with old-style SIF-encoded problems since strings starting with `$` in this field were previously treated as comments. Any characters after `$-PARAMETER` will be 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 variables 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, or possible, 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 current state-of-the art solvers.

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

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

where `SAMPLE.SIF` does not contain the parameter setting `N=1000`. 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 value 1000 for `N`. Note that nothing guarantees that this value is valid in that context, and that the `-force` command-line option should be used carefully.

6. The new installation procedures

We now describe the procedure to follow to install the CUTEr package. The complete procedure applies equally for the SifDec package, the only difference being the names of the procedures invoked, as we mention at the end of this section.

CUTEr comes in two similar yet fundamentally different flavours. In the first, the installation, un-installation and update procedures are entirely operated by shell scripts. The second implements the obvious solution offered by *Makefiles*. We now describe each in turn, starting with the script-based version of CUTEr.

Installation is performed by the script `install_script_cuter` which prompts for information on the local architecture and environment and the desired compiler. Basic system commands and definition of a temporary directory are stored in files called `system.`*os* in the directory `$CUTER/build/arch`, where '*os*' stands for the local operating system. If necessary, some or all of these files may need to be properly modified, although suitable settings are given for systems we have access to. The installation script searches the `$CUTER/build/arch` directory for files named `compiler.*` from which to choose a compiler. This does not imply that the corresponding compilers are actually installed on the local system but these files are meant to represent the most common compilers on that system. The creation of a suitable `compiler.*` file, if none is available, is left to the user, but can normally be achieved by modifying one "similar" to those provided. These two sets of files should be checked before the installation procedure is initiated. 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 re-issuing 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 linking and possibly compiling the result. 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 may be updated by the script update_cuter, as updates and bug fixes become available. A fourth script called rebuild serves the purpose of rebuilding or upgrading an installed version of CUTEr when the size parameters, compiler flags or system commands need to be changed, possibly as the result of warning or error messages issued by the SIF decoder or the CUTEr tools.

The second means of installing CUTEr is driven by portability concerns. The user is prompted for information by the install_cuter script, which creates the appropriate directory structure but leaves the local installation to *Imakefiles*. *Imakefiles* 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 fully documented within CUTEr and in [15]. This option greatly eases 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. This option also makes the script rebuild redundant. The *Imakefiles* 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, namely Imake.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-created and CUTEr needs to be rebuilt using usual make commands.

The installation procedure for the SifDec package is identical, with the sole proviso that in this context the names install_script_cuter, install_cuter, update_cuter, uninstall_cuter, CUTER and MYCUTER, in the above description 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 is standard ISO Fortran 77, but additionally CUTEr provides some support for Fortran 90/95. 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 their main webpages, at the locations

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

respectively.

Information on updates and how to obtain both packages will be available on the websites.

8. Conclusion and perspectives

This paper described improvements and new features of CUTEr, the latest release of the CUTE testing environment, 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 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 which may require the decoder, such as CUTEr and the forthcoming LANCELOT-B,
- ease its maintenance, upgrading and ease the 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, since machine dependencies have been carefully isolated. A number of previously-unsupported optimization and linear algebra packages are now interfaced by CUTEr, and corresponding driver programs are supplied. New tools for both constrained and unconstrained programming 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 not-yet supported optimization package, are relatively easy, as has been the case with CUTE.

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 both packages new features, bug fixes, new documentation and more.

Acknowledgements

Thanks to the following people for providing interfaces: Phil 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.

References

1. R. H. Bielschowsky and F. A. M. Gomes. Dynamical control of infeasibility in nonlinearly constrained optimization. Presentation at the Optimization 98 Conference, Coimbra, 1998.
2. I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.
3. M. G. Breitfeld and D. F. Shanno. Preliminary computational experience with modified log-barrier functions for large-scale nonlinear programming. In W. W. Hager, D. W. Hearn, and P. M. Pardalos, editors, *Large Scale Optimization: State of the Art*, pages 45–66, Dordrecht, The Netherlands, 1994. Kluwer Academic Publishers.
4. M. G. Breitfeld and D. F. Shanno. Computational experience with penalty-barrier methods for nonlinear programming. *Annals of Operations Research*, 62:439–463, 1996.
5. R. H. Byrd, J. Ch. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.
6. R. H. Byrd, J. Nocedal, and R. A. Waltz. Feasible interior methods using slacks for nonlinear optimization. Technical Report 11, Optimization Technology Center, Argonnne National Laboratory, Argonne, Illinois, USA, 2000.
7. T. F. Coleman and W. Yuan. A new trust region algorithm for equality constrained optimization. Technical Report TR95-1477, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1995.
8. A. R. Conn, N. I. M. Gould, D. Orban, and Ph. L. Toint. A primal-dual trust-region algorithm for non-convex nonlinear programming. *Mathematical Programming*, 87(2):215–249, 2000.
9. A. R. Conn, N. I. M. Gould, and Ph. L. Toint. LANCELOT*: a Fortran package for Large-scale Nonlinear Optimization (Release A)*. Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.
10. A. R. Conn, N. I. M. Gould, and Ph. L. Toint. A primal-dual algorithm for minimizing a nonconvex function subject to bound and linear equality constraints. In G. Di Pillo and F. Giannessi, editors, *Nonlinear Optimization and Related Topics*, pages 15–50, Dordrecht, The Netherlands, 1999. Kluwer Academic Publishers.
11. A. R. Conn, L. N. Vicente, and C. Visweswariah. Two-step algorithms for nonlinear optimization with structured applications. *SIAM Journal on Optimization*, 9(4):924–947, 1999.
12. J. E. Dennis, M. El-Alem, and K. A. Williamson. A trust-region approach to nonlinear systems of equalities and inequalities. *SIAM Journal on Optimization*, 9(2):291–315, 1999.
13. M. A. Diniz-Ehrhardt, M. A. Gomes-Ruggiero, and S. A. Santos. Comparing the numerical performance of two trust-region algorithms for large-scale bound-constrained minimization. *Revista Latino Americana de Investigación Operativa*, 7:23–54, 1997.
14. M. A. Diniz-Ehrhardt, M. A. Gomes-Ruggiero, and S. A. Santos. Numerical analysis of leaving-face parameters in bound-constrained quadratic minimization. Technical Report 52/98, Department of Applied Mathematics, IMECC-UNICAMP, Campinas, Brasil, 1998.
15. P. Dubois. Software Portability with imake. O'Reilly & Associates, Inc, 1993.

16. I. S. Duff, A. M. Erisman, and J. K. Reid. Direct Methods for Sparse Matrices. Oxford University Press, Oxford, England, 1986.
17. I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989.
18. I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 1992.
19. I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing sparse matrix collection. Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 1997.
20. R. Fletcher, N. I. M. Gould, S. Leyffer, and Ph. L. Toint. Global convergence of trust-region SQP-filter algorithms for nonlinear programming. Technical Report 99/03, Department of Mathematics, University of Namur, Namur, Belgium, 1999.
21. R. Fletcher and S. Leyffer. Nonlinear programming without a penalty function. *Mathematical Programming*, 91(2):239–269, 2002.
22. R. Fletcher and S. Leyffer. User manual for filterSQP. Numerical Analysis Report NA/181, Department of Mathematics, University of Dundee, Dundee, Scotland, 1998.
23. E. M. Gertz. *Combination Trust-Region Line-Search Methods for Unconstrained Optimization*. PhD thesis, Department of Mathematics, University of California, San Diego, California, USA, 1999.
24. P. E. Gill, W. Murray, and M. A. Saunders. User's guide for SNOPT 5.3: a Fortran package for large-scale nonlinear programming, 1998.
25. F. A. M. Gomes, M. C. Maciel, and J. M. Martínez. Nonlinear programming algorithms using trust regions and augmented Lagrangians with nonmonotone penalty parameters. *Mathematical Programming*, 84(1):161–200, 1999.
26. N. I. M. Gould, S. Lucidi, M. Roma, and Ph. L. Toint. Solving the trust-region subproblem using the Lanczos method. *SIAM Journal on Optimization*, 9(2):504–525, 1999.
27. N. I. M. Gould and J. Nocedal. The modified absolute-value factorization norm for trust-region minimization. In R. De Leone, A. Murli, P. M. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, pages 225–241, Dordrecht, The Netherlands, 1998. Kluwer Academic Publishers.
28. N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD—a library of thread-safe fortran 90 packages for large-scale nonlinear optimization. Technical Report (in preparation), Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2002.
29. HSL. A collection of Fortran codes for large scale scientific computation, 2002.
30. IBM Optimization Solutions and Library. *QP Solutions User Guide*. IBM Corportation, 1998.
31. M. Lalee, J. Nocedal, and T. D. Plantenga. On the implementation of an algorithm for large-scale equality constrained optimization. *SIAM Journal on Optimization*, 8(3):682–706, 1998.
32. M. Marazzi and J. Nocedal. Wedge trust region methods for derivative free optimization. Technical Report 2000/10, Optimization Technology Center, Northwestern University, Evanston, Illinois, USA, 2000.
33. I. Maros and C. Meszaros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11-12:671–681, 1999.

34. D. B. Ponceleón. *Barrier methods for large-scale quadratic programming*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, USA, 1990.

35. J. D. Pryce and J. K. Reid. `AD01`, a Fortran 90 codes automatic differentiation. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 1998.

36. R. W. H. Sargent and X. Zhang. An interior-point algorithm for solving general variational inequalities and nonlinear programs. Presentation at the Optimization 98 Conference, Coimbra, 1998.

37. A. Sartenaer. Automatic determination of an initial trust region in nonlinear programming. *SIAM Journal on Scientific Computing*, 18(6):1788–1803, 1997.

38. J. S. Shahabuddin. *Structured trust-region algorithms for the minimization of nonlinear functions*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1996.

39. S. Ulbrich and M. Ulbrich. 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, 1999.

40. Y. Xiao. *Non-monotone algorithms in optimization and their applications*. PhD thesis, Monash University, Clayton, Australia, 1996.

41. Y. Xiao and E. K. W. Chu. Nonmonotone trust region methods. Technical Report 95/17, Monash University, Clayton, Australia, 1995.

42. H. Yamashita, H. Yabe, and T. Tanabe. A globally and superlineraly convergent primal-dual point trust region method for large scale constrained optimization. Technical report, Mathematical Systems, Inc., Sinjuku-ku, Tokyo, Japan, 1997.

43. C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778. L-BFGS-B: Fortran subroutines for large-scale bound constrained optimization. *ACM Transactions on Mathematical Software*, 23(4):550–560, 1997.

A. Calling sequences for the new evaluation tools

Here we give the complete argument lists for those subroutines summarized in Tables 3.1 and 3.2 that are new to CUTEr; the remaining subroutines were fully documented in the appendix to [2]. 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.

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(NE^o, NZH^o, NZIRNH^o)

– Obtain the type of each variable:
`CALL UVARTY(` N^i, $IVARTY^o$ `)`
– Obtain statistics concerning function evaluation and CPU time use:
`CALL UREPRT (` $UCALLS^o$, $TIME^o$ `)`

A.2. Generally constrained problems

– Discover how many variables and constraints are involved in the problem:
`CALL CDIMEN (` $INPUT^i$, N^o, M^o `)`
– 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 (` $NNZJ^o$ `)`
– Determine how many nonzeros are required to store the Hessian matrix of the Lagrangian (when stored in a sparse format):
`CALL CDIMSH (` $NNZH^o$ `)`
– 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(` NE^o, NZH^o, $NZIRNH^o$ `)`
– Obtain the type of each variable:
`CALL CVARTY(` N^i, $IVARTY^o$ `)`
– Evaluate an individual constraint function and possibly its gradient (when this is stored in a sparse format):
`CALL CCIFSG (` N^i, I^i, X^i,CI^o, $NNZSGC^o$, $LSGCI^i$,$SGCI^o$, $IVSGCI^o$, $GRAD^i$ `)`
– Obtain statistics concerning function evaluation and CPU time use:
`CALL CREPRT (` $CCALLS^o$, $TIME^o$ `)`

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.`
`I` is the index of the general constraint function to be evaluated by `CCIFSG`.
`INPUT` is the unit number for the decoded data, i.e., from which `OUTSDIF.d` (see [2]) is read.
`IVARTY` 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).

`IVSGCI` is an array whose i-th component is the index of the variable with respect to which `SGCI`(i) is the derivative.

`LSGCI` is the actual declared dimension of `SGCI`.

`M` is the total number of general constraints.

`N` is the number of variables for the problem.

`NE` is the number of elements in a finite-element representation of the Hessian for the problem.

`NZH` is the dimension of the array needed to store the real values of the finite-element Hessian for the problem.

`NZIRNH` is the dimension of the array needed to store the integer values of the finite-element Hessian for the problem.

`NNZH` is the number of nonzeros in the Hessian for the problem.

`NNZJ` is the number of nonzeros in the constraint Jacobian for the problem.

`NNZSGC` is the number of nonzeros in `SGCI`.

`SGCI` is an array which gives the values of the nonzeros of the gradient of the general constraint function `I` evaluated at `X`. The i-th entry of `SGCI` gives the value of the derivative with respect to variable `IVSGCI`(i) of function `I`.

`TIME` is an array whose components give CPU times (in seconds) for various activities during the current execution of the tools. Components are:

- `TIME(1)` CPU time for call to `USETUP/CSETUP`.
- `TIME(2)` CPU time since last call to `USETUP/CSETUP`.

`UCALLS` is an array whose components give counts for various activities during the current execution of the unconstrained tools. Components are:

- `UCALLS(1)` number of objective function evaluations
- `UCALLS(2)` number of objective gradient evaluations
- `UCALLS(3)` number of objective Hessian evaluations
- `UCALLS(4)` number of Hessian-vector products

`X` is an array which gives the current estimate of the solution of the problem.