Comput Optim Appl (2015) 60:545–557
DOI 10.1007/s10589-014-9687-3

CUTEst: a Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization

Nicholas I. M. Gould · Dominique Orban · Philippe L. Toint

Received: 2 April 2014 / Published online: 24 August 2014

Abstract We describe the most recent evolution of our constrained and unconstrained testing environment and its accompanying SIF decoder. Code-named SIFDecode and CUTEst, these updated versions feature dynamic memory allocation, a modern thread-safe Fortran modular design, a new Matlab interface and a revised installation procedure integrated with GALAHAD.

Keywords CUTE · CUTEr · CUTEst · Optimization · Modeling · Benchmarking

1 Introduction

The Constrained and Unconstrained Testing Environment (CUTE) [3] and its associated set of optimization examples have been widely adopted and used since their introduction in 1993. The test set has grown over time and now numbers approximately 1,150 examples, many of them of variable dimension. In addition, since both the MPS linear programming format [14] and its quadratic programming extensions [13,16,17] are compatible with CUTE's Standard Input Format (SIF) [5, Chap. 7],

N. I. M. Gould (✉)
Scientific Computing Department, Rutherford Appleton Laboratory,
Chilton, Oxfordshire OX11 0QX, England, UK
e-mail: nick.gould@stfc.ac.uk

D. Orban
Department of Mathematics and Industrial Engineering, École Polytechnique,
and GERAD, Montréal, QC, Canada
e-mail: dominique.orban@gerad.ca

Ph. L. Toint
Department of Mathematics, University of Namur, Namur, Belgium
e-mail: philippe.toint@unamur.be

CUTE and its successor CUTEr [11] provide access to other test sets such as those from Netlib [9] and Maros and Meszaros [16].

To set the scene, recall that a SIF file provides a portable description of group-partially separable optimization problem [4]. Such a file is translated by the package SifDec to a number of Fortran subroutines that compute values and derivatives of constituent element and group functions, together with data that explains how the functions are glued together. Armed with these components, CUTE(r) reassembles them to allow users to compute values and derivatives of both the objective function and/or its constraints as required, and provides static information such as bounds on variables and constraints.

The core (Fortran 77) routines behind CUTE(r) have not changed significantly since their original release. A main limitation of standard Fortran 77 is that it offers no mechanism for dynamic memory management, something programmers of other languages (particularly C) take for granted. This defect had a significant implication for CUTE and CUTEr, namely that a one-size-fits-all set of array dimensions are set at compile time, and CUTEr and its relatives return to the user if this choice is insufficient, with a recommendation for recompilation with values that might be appropriate. Many CUTEr users have learned to detest this inflexibility, and it is certainly the main source of complaint we receive.

Of course, modern Fortran (90 and later) provide dynamic memory allocation, and it has mostly been the scale of the task of rewriting CUTEr to do this that has stopped us; CUTEr (and its dependent SifDec) number roughly 50,000 lines of code. Now, we have done so. The new, Fortran 2003 packages CUTEst and SIFDecode both request array storage as needed. The data for any array that is not currently large enough is written to temporary storage (or, if there is insufficient room in memory, to disk), the array de-allocated and re-allocated with some "elbow-room" and the existing data copied back. A more general concern about the lack of freely-available reliable modern Fortran compilers has also vanished with the arrival of both g95 and gfortran; there are of course many excellent commercial Fortran compilers available.

An additional limitation of CUTE(r) is that it relies on Fortran common blocks to share data between tools. Some of this data, such as that required to describe the structure of a problem, is fixed after calling "setup" routines, while the remainder is constantly rewritten as problem function (and derivative) evaluations take place. This use of common means that the packages are not thread safe, and cannot be used in a multi-threaded (parallel) environment (for example to test branch-and-bound methods for integer programming or global optimization). This deficiency has also been addressed in CUTEst, and indeed the data is now split so that a single copy of the fixed data is available to all threads, while dynamic data is stored on a per-thread basis.

We have also taken the opportunity to revise the way the packages are organised and installed. We have adopted the scheme we currently use within GALAHAD [12], leading to some isolation of common components that are distributed separately. Since GALAHAD is itself a major user of CUTEr, we have also updated components of GALAHAD to use CUTEst packages. In addition, we have upgraded all of CUTEr's interfaces to external optimization packages—written in a variety of languages —that are still distributed (there have been a few casualties since 2003), and have

also provided interfaces to a number of new ones. We have also considerably simplified the interface to Matlab.

This short paper is arranged as follows. We first describe how CUTEst is now organised, and how it may be used to test external packages. We then give a few details of the new features provided. A few words about the SIF decoder are followed by a description of the installation procedure.

2 New package organisation

CUTEst is provided as a series of directories and files, all lying beneath a root directory that we shall refer to as `$CUTEST`. The directory structure is illustrated in Fig. 1.

Before installation begins, the subdirectories `objects`, `modules`, `makefiles`, `versions` and `bin/sys` will all be empty. The script `install_cutest` prompts the user for the answers to a series of questions aimed at determining what machine type, operating system and compiler should be used to build CUTEst—we call this combination of a machine, operating system and compiler an *architecture*. The architecture is selected from a large list of predefined possibilities encoded in a separate package, archdefs, that is shared between SIFDecode, CUTEst and GALAHAD. Each architecture is assigned a simple (mnemonic) architecture *code* name, say `architecture`—for example a version for the NAG Fortran 95 compiler on a PC runing Linux is coded `pc.lnx.n95`, while another for the IBM Fortran 95 compiler on a PowerPC system running AIX is `ppc.aix.x95`. Having determined the architecture, the installation script builds subdirectories of `objects`, `modules` and `packages` named `architecture`. Each of these subdirectories contains further subdirectories `double` and `single` to hold architecture-dependent compiled libraries, module file information and external package linking information if required. In addition, architecture-dependent makefile information and environment variables for execution scripts are placed in files named `architecture` in the `makefiles` and `bin/sys` subdirectories, and a file recording how the code is related to the architecture is put in `versions`.

The source code for each CUTEst package interface is located in a separate subdirectory of the `src` directory. The main CUTEst evaluation tools all lie in the subdirectory `tools`, while a set of comprehensive test programs are available in `test`. The remaining subdirectories contain interface programs between the tools and each of the external packages supported. Each subdirectory contains the interface, a default options specification file, an example program that tests the interface without requiring the external package, a README that explains what a user needs to do to make the external package work with CUTEst, and a makefile. Since the order of compilation of Fortran modules is important, and as we have seen there is a strong interdependency between the CUTEst packages, the makefiles have to be carefully crafted. For this reason, we have chosen not to use variants of tools such as `imake` to build and maintain the makefiles. A set of configuration files that provides default link details between CUTEst and each external package is provided in the directory `packages/defaults`. Users may copy and modify these to architecture and dimension specific subdirectories of `packages` to override the default choices. Man pages for CUTEst as a whole and for each individual tool are provided in sub-

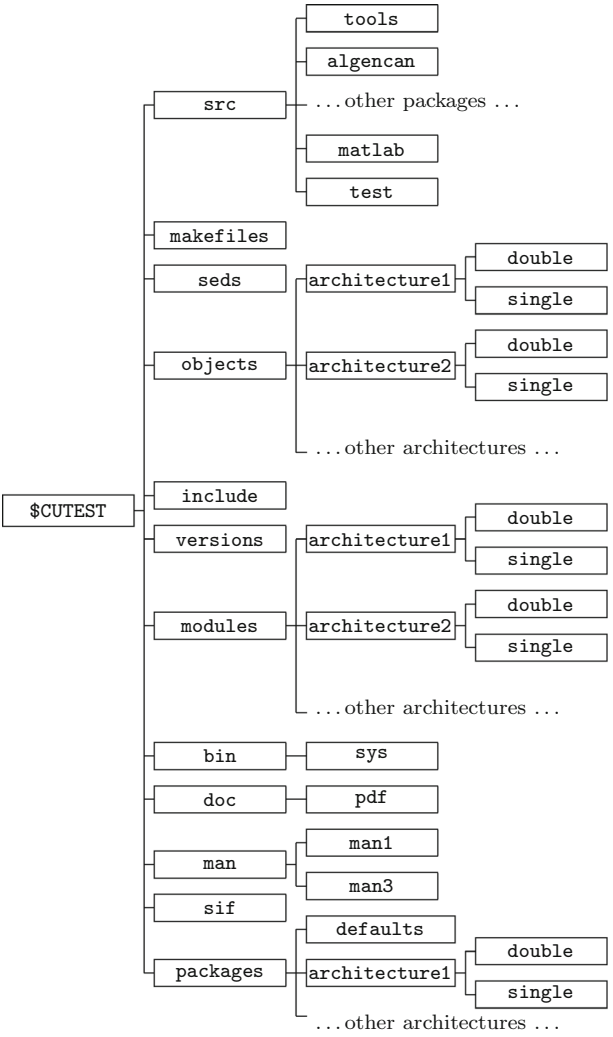

Fig. 1 Structure of the CUTEst directories

directories of the `man` directory, and versions in PDF format are also available in the `pdf` subdirectory of the documentation directory `doc`.

Once the correct directory structure is in place, the installation script builds a random-access library of the required precision by visiting each of the subdirectories of `src` and calling the Unix utility `make`. CUTEst package interfaces are all written in double precision, but if a user prefers to use single precision, the makefiles call suitable Unix `sed` scripts (stored in `seds`) to transform the source prior to compilation. A user may choose to install all of CUTEst with or without Matlab support. The tools for unconstrained and constrained optimization may also be installed separately. Recompilation following updates is easily performed by issuing the com-

mand `make` from the `src` directory, while `make test` from the same directory runs comprehensive tests of all compiled components.

3 Interfaces to the CUTEst test set

To run one of the supported packages on an example stored in `EXAMPLE.SIF`, say, a user needs to issue the command

```
runcutest -A architecture -p package -D EXAMPLE[.SIF]
```

where `architecture` is the architecture code discussed in §2, `package` defines the package to be used—the manual page for `runcutest` gives a list of current possibilities—and the suffix [`.SIF`] is optional. This command translates the SIF file into Fortran subroutines and related data using the decoder provided in SIFDecode, and then calls the required optimization package to solve the problem. A default architecture may be defined by setting the environment variable `$MYARCH`, and if so the `-A` flag may be avoided. Once a problem has been decoded, it may be re-used (perhaps with different options) using the auxiliary command

```
runcutest -A architecture -p package
```

For Matlab use, the command

```
cutest2matlab EXAMPLE[.SIF]
```

may be used instead; since Matlab is very specific about the Fortran compilers it supports, requests for a CUTEst-Matlab installation will adjust compiler options accordingly.

A few SIF examples are given in the `sif` directory, while the `runcutest` and `cutest2matlab` commands are in the `bin` subdirectory, and have man-page descriptions in the `man/man1` subdirectory.

4 Improvements

4.1 New features

As we mentioned in the introduction, CUTEst and SIFDecode both use the Fortran 2003 allocate/deallocate features to create and modify array storage. Each CUTEst tool uses a module `CUTEST` that provides access to two derived types `CUTEST_data_type` and `CUTEST_work_type` used to store workspace arrays. The former collects data that describes problem structure and is set prior to any problem function evaluation, and unchanged thereafter, while the latter is used to hold data that may change at every evaluation. Data in these arrays is made available to the tools through a scalar `CUTEST_data_global` of type `CUTEST_data_type` and an allocatable array `CUTEST_work_global` of type `CUTEST_work_type`.

All evaluation tools are available as both simple (unthreaded) and threaded versions. For the latter, which may be distinguished by the suffix `_threaded`, `CUTEST_work_global` is allocated to be large enough to hold all the threads

that will be used by the setup subroutines. Each evaluation call specifies access to `CUTEST_work_global(i)` for the particular thread `i` required.

We give a list of all CUTEst tools and their functionality in Appendix 1. Most are rewritten versions of their CUTEr counterparts, with argument-list changes to remove redundant size parameters and occasional order changes to handle inconsistencies. An additional `status` argument has been added to each to report any fatal memory errors (such as array allocation/deallocation failures) and inabilities to evaluate functions at specified values; CUTEr deals with such eventualities by terminating execution. We have added termination tools to allow users to deallocate all storage created by the setup procedure when they have no longer need for it. In addition, constraints may now be ordered so that equalities precede or follow inequalities, or so that linear constraints precede or follow nonlinear ones, and variables so that those that only appear linearly in the problem precede or follow those that appear nonlinearly. New tools have also been introduced to describe the sparsity patterns of the Hessian of the objective and Lagrangian functions. Finally, a new tool has been added to compute the sparse gradient of the objective function for constrained problems; this corrects an oversight since we already provide similar functionality for the gradients of individual constraints.

4.2 New Matlab calls

The Matlab interface has been substantially revised and simplified. It is now entirely written in C instead of Fortran to facilitate interaction with Matlab's own API and usage of its index types. The interface merges the constrained and unconstrained tools together so users may use familiar and consistent function calls such as `cutest_obj()` to obtain the objective function value and possibly its gradient regardless of the presence of constraints. In this regard, the interface exploits Matlab's ability to determine how many output arguments are required by the user. This allows both `f = cutest_obj(x)` and `[f,g] = cutest_obj(x)`. Another example is the `cuter_cons()` function, which evaluates all or individual constraints and/or constraint gradients.

The current interface makes it easier to decode problems and build the corresponding MEX files from inside Matlab. For instance, the commands

```
probname = 'LUBRIFC'; unix(['cutest2matlab ', probname])
```

generate the MEX file corresponding to problem `LUBRIFC` in the current directory. A problem is "loaded" into Matlab by calling a simple function with no arguments: `prob = cutest_setup()`. The single output argument of the setup function is a Matlab structure containing problem data such as the number of variables, number of constraints, number of nonzeros in the Jacobian and Hessian, initial guess, bounds, initial multipliers, and arrays indicating which constraints are linear and which are equality constraints. Those fields may be accessed using the familiar dot notation, e.g., `prob.n`, `prob.x`, `prob.cl`, etc.

Help is included with all CUTEst tools, available to Matlab users by way of the familiar `help` call. In particular, `help cutest` gives an overview of the tools available.

See Appendix 2 for a complete description of the Matlab CUTEst tools and their functionality.

4.3 New interfaces

In addition to the still-current packages supported by CUTEr, CUTEst provides new interfaces to `ALGENCAN` [2], `BOBYQA` [18], `Direct Search` [7], `filterSD` [8], `LINUOA` [19], `NEWUOA` [20], `NLPQLP` [22], `NOMAD` [6], `PENNLP` [15], `QL` [23], `SPG` [1] and `SQIC` [10], as well as various new packages within GALAHAD [12]. An interface is also provided to translate SIF examples into a format used by the QPLIB2014 [21] quadratic programming collection.

Interfaces to the obsolete packages `hsl_ve12`, `osl`, `va15`, `ve09` and `ve14` supported in CUTEr have been withdrawn.

4.4 New test examples

Almost 200 new examples have been added to the test-problem collection since the release of CUTEr. These include large collections of problems arising from linear complementarity, and of real-life quadratic programming problems.

All test problems are now under version control in the same way as the source code and users may update their local repository easily when new problems are added or changes are made to existing problems.

5 A revised SIF decoder

As we mentioned in the introduction, the lack of dynamic memory allocation affects the SIF decoding package SifDec just as severely as it does CUTEr. A new stand-alone Fortran package SIFDecode has been written to address this issue. All of the functionality of the subroutines previously in SifDec have been combined into a single Fortran 2003 module `SIFDECODE`. Since required array sizes are not known beforehand, default initial values are increased as required as the package makes a single pass through the SIF file under consideration; default initial values may be changed to make the processing more efficient, but this is not crucial.

The distributed package is organised in the same way as CUTEst (see §2), although now the `src` directory simply contains two source subdirectories: `decode` that holds the decoder and its main program, and `select` containing the test-problem database interrogation tools from CUTE [3, §2.3].

Once the package has been installed, the decoder is called by issuing the command

```
sifdecoder -A architecture EXAMPLE[.SIF]
```

where `architecture` and `EXAMPLE.SIF` are as before; the `-A` option may be omitted when using the default architecture.

6 New installation procedures

The installation procedure has been updated to recognise that most users will need to install both SIFDecode and CUTEst, and may also wish to integrate these with GALAHAD. A single script, `install_optsuite`, prompts the user to describe what features are needed and which architecture is desired. An opportunity to modify default compilation flags is provided, after which the script will automatically download and install the software.

7 Obtaining the packages

All of the required and optional packages archdefs, SIFDecode, CUTEst and GALAHAD are available from the CCPForge project, funded by The Joint Information Systems Committee (http://www.jisc.ac.uk) and maintained by the Scientific Computing Department of the Science and Technology Facilities Council (http://www.stfc.ac.uk/SCD/default.aspx) under the departmental Service Level Agreement with the Engineering and Physical Sciences Research Council (EPSRC). See

`http://ccpforge.cse.rl.ac.uk/gf/project/cutest/wiki`

for download details.

Both SIFDecode and CUTEst are distributed and made available under the terms of the GNU Lesser General Public License. See

`http://www.gnu.org/licenses/lgpl-3.0.txt`

for details.

8 Conclusions and perspectives

We believe CUTEst is a considerable improvement over past versions because of its improved modular and thread-safe design exploiting recent additions to the Fortran standard, dynamic allocation, simplified and unified tool calling sequences, improved Matlab interface and, last but not least, the more user-friendly installation process. Despite the age of the standard input format and the advent of more modern modeling languages, CUTE and CUTEr remain widely-used tools in the optimization community and beyond, as illustrated by the large number of user comments and request that we receive, and the associated problem collection remains a staple of optimization software testing and benchmarking.

Since all packages and test problems are maintained and distributed via a source code revision system, bug fixes, improvements and additions are easily available.

Acknowledgments We are extremely grateful to Roger Fletcher, Philip Gill, Bill Hager, Michal Kočvara, Michael Powell, Klaus Schittkowski and Elizabeth Wong for making their latest codes available to us so that we could build and test interfaces, and to two anonymous referees whose enthusiastic comments lead to a better paper. The work of the first author was supported by the EPSRC Grant EP/I013067/1. The work of the second author was supported by an NSERC Discovery Grant.

Appendix 1: available tools

Separate evaluation tools are provided for unconstrained and constrained problems. Both unthreaded and threaded versions are available when this is relevant. See the appropriate man page for full details.

Unconstrained problems:

cutest_udimen (both threaded and unthreaded)
determine the number of variables.

cutest_usetup (unthreaded) and **cutest_usetup_threaded** (threaded)
setup internal data structures and determine variable bounds.

cutest_unames (both threaded and unthreaded)
determine the names of the problem and the variables.

cutest_uvartype (both threaded and unthreaded)
determine whether the variables are continuous or discrete.

cutest_udimsh (both threaded and unthreaded)
determine the number of nonzeros in the sparse Hessian.

cutest_udimse (both threaded and unthreaded)
determine the number of nonzeros in the finite-element Hessian.

cutest_ufn (unthreaded) and **cutest_ufn_threaded** (threaded)
evaluate the objective function value.

cutest_ugr (unthreaded) and **cutest_ugr_threaded** (threaded)
evaluate the gradient of the objective function.

cutest_uofg (unthreaded) and **cutest_uofg_threaded** (threaded)
evaluate both the values and gradients of the objective function.

cutest_udh (unthreaded) and **cutest_udh_threaded** (threaded)
evaluate the Hessian of the objective function as a dense matrix.

cutest_ugrdh (unthreaded) and **cutest_ugrdh_threaded** (threaded)
evaluate the objective gradient and dense Hessian.

cutest_ushp (both threaded and unthreaded)
evaluate the sparsity pattern of the Hessian of the objective function.

cutest_ush (unthreaded) and **cutest_ush_threaded** (threaded)
evaluate the Hessian of the objective function as a sparse matrix.

cutest_ugrsh (unthreaded) and **cutest_ugrsh_threaded** (threaded)
evaluate the objective gradient and sparse Hessian.

cutest_ueh (unthreaded) and **cutest_ueh_threaded** (threaded)
evaluate the Hessian of the objective function as a finite-element matrix.

cutest_ugreh (unthreaded) and **cutest_ugreh_threaded** (threaded)
evaluate the objective gradient and finite-element Hessian.

cutest_uhprod (unthreaded) and **cutest_uhprod_threaded** (threaded)
evaluate the product of the Hessian of the objective function with a vector.

cutest_ubandh (unthreaded) and **cutest_ubandh_threaded** (threaded)
obtain the part of the Hessian of the objective that lies within a specified band.

cutest_ureport (unthreaded) and **cutest_ureport_threaded** (threaded)
discover how many evaluations have occured and how long this has taken.

cutest_uterminate (both unthreaded and threaded)

remove internal data structures when they are no longer needed.

Constrained problems:

cutest_cdimen (both threaded and unthreaded)
determine the number of variables and constraints.

cutest_csetup (unthreaded) and **cutest_csetup_threaded** (threaded)
setup internal data structures and determine variable and constraint bounds.

cutest_cnames (both threaded and unthreaded)
determine the names of the problem, the variables and the constraints.

cutest_connames (both threaded and unthreaded)
determine the names of the constraints.

cutest_cvartype (both threaded and unthreaded)
determine whether the variables are continuous or discrete.

cutest_cdimsj (both threaded and unthreaded)
determine the number of nonzeros in sparse constraint Jacobian.

cutest_cdimsh (both threaded and unthreaded)
determine the number of nonzeros in the sparse Hessian.

cutest_cdimse (both threaded and unthreaded)
determine the number of nonzeros in the finite-element Hessian.

cutest_cfn (unthreaded) and **cutest_cfn_threaded** (threaded)
evaluate the objective function and constraint values.

cutest_cgr (unthreaded) and **cutest_cgr_threaded** (threaded)
evaluate the gradients of the objective function and constraints.

cutest_cofg (unthreaded) and **cutest_cofg_threaded** (threaded)
evaluate both the value and gradient of the objective function.

cutest_cofsg (unthreaded) and **cutest_cofsg_threaded** (threaded)
evaluate both the value and sparse gradient of the objective function.

cutest_csgr (unthreaded) and **cutest_csgr_threaded** (threaded)
evaluate the sparse gradients of the objective function and constraints.

cutest_ccfg (unthreaded) and **cutest_ccfg_threaded** (threaded)
evaluate the values and gradients of the constraints.

cutest_ccfsg (unthreaded) and **cutest_ccfsg_threaded** (threaded)
evaluate the values and sparse gradients of the constraints.

cutest_ccifg (unthreaded) and **cutest_ccifg_threaded** (threaded)
evaluate the value and gradient of an individual constraint.

cutest_clfg (unthreaded) and **cutest_clfg_threaded** (threaded)
evaluate both the value and gradient of the Lagrangian function.

cutest_ccifsg (unthreaded) and **cutest_ccifsg_threaded** (threaded)
evaluate the value and sparse gradient of an individual constraint.

cutest_cdh (unthreaded) and **cutest_cdh_threaded** (threaded)
evaluate the Hessian of the Lagrangian function as a dense matrix.

cutest_cidh (unthreaded) and **cutest_cidh_threaded** (threaded)
evaluate the Hessian of the objective function or an individual constraint as a dense matrix.

cutest_cgrdh (unthreaded) and **cutest_cgrdh_threaded** (threaded)
evaluate the constraint Jacobian and Hessian of the Lagrangian function as dense matrices.

cutest_cshp (both threaded and unthreaded)
evaluate the sparsity pattern of the Hessian of the Lagrangian function.
cutest_csh (unthreaded) and **cutest_csh_threaded** (threaded)
evaluate the Hessian of the Lagrangian function as a sparse matrix.
cutest_cshc (unthreaded) and **cutest_cshc_threaded** (threaded)
evaluate the Hessian of the Lagrangian function not including the objective as a sparse matrix.
cutest_cish (unthreaded) and **cutest_cish_threaded** (threaded)
evaluate the Hessian of the objective function or an individual constraint as a sparse matrix.
cutest_csgrsh (unthreaded) and **cutest_csgrsh_threaded** (threaded)
evaluate the constraint Jacobian and Hessian of the Lagrangian function as sparse matrices.
cutest_ceh (unthreaded) and **cutest_ceh_threaded** (threaded)
evaluate the Hessian of the Lagrangian function as a finite-element matrix.
cutest_csgreh (unthreaded) and **cutest_csgreh_threaded** (threaded)
evaluate the constraint Jacobian as a sparse matrix and the Hessian of the Lagrangian function as a finite-element matrix.
cutest_chprod (unthreaded) and **cutest_chprod_threaded** (threaded)
evaluate the product of the Hessian of the Lagrangian function with a vector.
cutest_chcprod (unthreaded) and **cutest_chcprod_threaded** (threaded)
evaluate the product of the Hessian of the Lagrangian function not including the objective with a vector.
cutest_cjprod (unthreaded) and **cutest_cjprod_threaded** (threaded)
evaluate the product of the constraint Jacobian or its transpose with a vector.
cutest_creport (unthreaded) and **cutest_creport_threaded** (threaded)
discover how many evaluations have occured and how long this has taken.
cutest_cterminate (both unthreaded and threaded)
remove internal data structures when they are no longer needed.
Both unconstrained problems and constrained problems:
cutest_probname (both threaded and unthreaded)
determine the name of the problem.
cutest_varnames (both threaded and unthreaded)
determine the names of the variables.

A call to cutest_u/csetup[_threaded] must precede calls to any other evaluation tool with the exception of cutest_u/cdimen. Once cutest_u/cterminate[_threaded] has been called, no further calls should be made without first recalling cutest_u/csetup[_threaded].

Appendix 2: Matlab interfaces

The updated Matlab tools are described in Table 1.

Table 1 Available Matlab tools

Matlab tool	CUTEst tool (s)	Purpose
`cutest_dims`	`cdimen`	Obtain problem dimensions
`cutest_setup`	`usetup/csetup`	Setup problem data structure
`cutest_obj`	`uofg/cofg`	Evaluate objective function value and its gradient if requested
`cutest_grad`	`ugr/cgr`	Evaluate objective function gradient
`cutest_sobj`	`cofsg`	Evaluate objective function value and its gradient as a sparse vector if requested
`cutest_objcons`	`cfn`	Evaluate objective and constraints
`cutest_cons`	`ccifg`	Evaluate constraint bodies and their gradients if requested. Evaluate a single constraint value and its gradient if requested
`cutest_scons`	`ccifsg`	Evaluate constraint bodies and Jacobian in sparse format. Evaluate a single constraint value and its gradient as a sparse vector
`cutest_lag`	`clfg`	Evaluate value and gradient of the Lagrangian
`cutest_lagjac`	`cgr`	Evaluate Jacobian and gradient of either objective or Lagrangian
`cutest_slagjac`	`csgr`	Evaluate Jacobian in sparse format and gradient of either objective or Lagrangian as a sparse vector
`cutest_Jprod`	`cjprod`	Evaluate the matrix-vector product between the Jacobian and a vector
`cutest_Jtprod`	`cjprod`	Evaluate the matrix-vector product between the transpose Jacobian and a vector
`cutest_hess`	`udh/cdh`	Evaluate the Hessian matrix of the Lagrangian, or of the objective if the problem is unconstrained
`cutest_ihess`	`udh/cidh`	Evaluate the Hessian matrix of the i-th problem function (i = 0 is the objective function), or of the objective if problem is unconstrained
`cutest_hprod`	`uhprod/chprod`	Evaluate the matrix-vector product between the Hessian of the Lagrangian (or the objective if unconstrained) and a vector
`cutest_gradhess`	`ugrdh/cgrdh`	Evaluate the gradient of either the objective or the Lagrangian, the Jacobian (or its transpose) and the Hessian of the Lagrangian in dense format
`cutest_sphess`	`ush/csh`	Evaluate the Hessian matrix of the Lagrangian, or of the objective if the problem is unconstrained, in sparse format
`cutest_isphess`	`ush/cish`	Evaluate the Hessian matrix of the i-th problem function (i = 0 is the objective function), or of the objective if problem is unconstrained, in sparse format
`cutest_varnames`	`varnames`	Obtain variable names as a list of strings
`cutest_connames`	`cnames`	Obtain constraint names as a list of strings
`cutest_terminate`	`uterminate/cterminate`	Remove existing internal workspace

References

1. Birgin, E.G., Martinez, J.M., Raydan, M.: Algorithm 813: SPG–software for convex-constrained optimization. ACM Trans. Math. Softw. **27**, 340–349 (2001)
2. Birgin, E.G., Castillo, R., Martinez, J.M.: Numerical comparison of augmented Lagrangian algorithms for nonconvex problems. Comput. Optim. Appl. **31**(1), 31–56 (2005)
3. Bongartz, I., Conn, A.R., Gould, N.I.M., Toint, Ph.L.: CUTE: constrained and unconstrained testing environment. ACM Trans. Math. Softw. **21**(1), 123–160 (1995)
4. Conn, A.R., Gould, N.I.M., Toint, Ph.L.: An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. In: Glowinski, R., Lichnewsky, A. (eds.) Computing Methods in Applied Sciences and Engineering, pp. 42–51. SIAM, Philadelphia (1990)
5. Conn, A.R., Gould, N.I.M., Toint, Ph.L.: LANCELOT: A Fortran Package for Large-Scale Nonlinear Optimization (Release A). Springer Series in Computational Mathematics. Springer, Heidelberg, Berlin, New York (1992)
6. Le Digabel, S.: Algorithm 909: NOMAD: nonlinear optimization with the MADS algorithm. ACM Trans. Math. Softw. **37**(4), 1–15 (2011)
7. Dolan, E.D., Gurson, A.P., Shepherd, P.L., Siefert, C.M., Torczon, V.J., Yates, A.: C++ direct searches. http://www.cs.wm.edu/va/software/DirectSearch/direct_code/ (2001)
8. Fletcher, R.: A sequential linear constraint programming algorithm for NLP. SIAM J. Optim. **22**(3), 772–794 (2012)
9. Gay, D.M.: Electronic mail distribution of linear programming test problems. Mathematical Programming Society COAL Newsletter, December (1985). http://www.netlib.org/lp/data/
10. Gill, P.E., Wong, E.: Methods for convex and general quadratic programming. Technical Report NA 10–1, Department of Mathematics, University of California, San Diego, 2013. To appear Mathematical Programming Computation (2014)
11. Gould, N.I.M., Orban, D., Toint, Ph.L.: CUTEr (and SifDec), a constrained and unconstrained testing environment, revisited. ACM Trans. Math. Softw. **29**(4), 373–394 (2003)
12. Gould, N.I.M., Orban, D., Toint, Ph.L.: GALAHAD—a library of thread-safe fortran 90 packages for large-scale nonlinear optimization. ACM Trans. Math. Softw. **29**(4), 353–372 (2003)
13. IBM Optimization Solutions and Library: QP Solutions User Guide. IBM Corportation (1998)
14. International Business Machine Corporation: Mathematical programming system/360 version 2, linear and separable programming-user's manual. Technical Report H20–0476-2, IBM Corporation, 1969. MPS Standard
15. Kocvara, M., Stingl, M.: PENNON: a code for convex nonlinear and semidefinite programming. Optim. Methods Softw. **18**(3), 317–333 (2003)
16. Maros, I., Meszaros, C.: A repository of convex quadratic programming problems. Optim. Methods Softw. **11–12**, 671–681 (1999)
17. Ponceleón, D.B.: Barrier methods for large-scale quadratic programming. Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, USA (1990)
18. Powell, M.J.D.: The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report DAMTP NA2009/06, Department of Applied Mathematics and Theoretical Physics, Cambridge University, Cambridge, UK (2009)
19. Powell, M.J.D.: The LINUOA software for linearly unconstrained optimization without derivatives. http://www.netlib.org/na-digest-html/13/v13n42.html (2013)
20. Powell, M.J.D.: The NEWUOA software for unconstrained optimization without derivatives. In: Di Pillo, G., Roma, M. (eds.) Large-Scale Nonlinear Optimization. Nonconvex Optimization and Its Applications, vol. 83, pp. 255–297. Springer, Heidelberg, Berlin, New York (2006)
21. QPLIB2014: a Quadratic Programming Library. http://www.lamsade.dauphine.fr/QPlib2014/doku.php (2014)
22. Schittkowski, K.: NLPQLP: a Fortran implementation of a sequential quadratic programming algorithm with distributed and non-monotone line search. University of Bayreuth, Department of Computer Science, Technical report (2010)
23. Schittkowski, K.: QL: a Fortran code for convex quadratic programming–User's guide, Version 2.11. Technical report, University of Bayreuth, Department of Computer Science (2005)