General CUTEr documentation

N. I. M. Gould D. Orban Ph.L. Toint

March 24, 2005

CERFACS Technical Report TR/PA/02/13

Contents

Disclaimer

This software was written as a personal project and comes with NO WARRANTY of any kind, not even MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Please read the file LICENSE in the CUTEr *home directory prior to any other manipulation.*

The authors assume no responsibility for any use.

The authors, N. I. M. Gould, D. Orban and Ph.L. Toint

Contact

N. I. M. Gould, Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire OX11 0QX, England.
n.gould@rl.ac.uk
http://www.cse.clrc.ac.uk/Person/N.I.M.Gould

D. Orban, CERFACS, Parallel Algorithms Project, Toulouse, France.
Dominique.Orban@cerfacs.fr
http://www.cerfacs.fr/~orban

Ph.L. Toint, Facultés Universitaires Notre-Dame de la Paix, 61, rue de Bruxelles, B-5000 Namur, Belgium.
Philippe.Toint@fundp.ac.be
http://www.fundp.ac.be/~phtoint

Note

This documentation is in constant evolution, and so is the software. We advise the reader to consult the website http://cuter.rl.ac.uk/cuter-www for the latest information, bug fixes and patches concerning CUTEr.

CUTEr is a versatile testing environment for optimization and linear algebra solvers. The package contains a collection of test problems, along with Fortran 77, Fortran 90 and Matlab tools intended to help developers design, compare and improve new and existing solvers. This document describes installation and basic usage of the CUTEr environment, and is intended to be one of the main documentation sources available with the package; other sources include man pages, various README files and self-documented scripts.

The test problems provided are written in so-called Standard Input Format (SIF), and a decoder is provided to convert from this format into well-defined Fortran 77 and data files. Once translated, these files may be manipulated to provide tools suitable for testing optimization packages. Ready-to-use interfaces to existing packages, such as MINOS, SNOPT, filterSQP and KNITRO, are provided.

CUTEr is available on a variety of UNIX platforms, including LINUX and is designed to be accessible and easily manageable on heterogeneous networks.

"When all else fails, read the documentation." (fortune)

Chapter 1

Installation and usage

1.1 Installing and managing CUTEr

The current version of CUTEr comes in the form of a gzipped tarfile. To uncompress and extract the CUTEr distribution from it, move the file to a new directory of your choice—we shall refer to this directory as `$CUTER`—and issue the commands

```
prompt% gunzip cuter.tar.gz
prompt% tar xvf cuter.tar
```

or, more compactly,

```
prompt% gunzip -c cuter.tar.gz | tar xvf -
```

On GNU-based LINUX systems, this is also done by the single command

```
prompt% tar zxvf cuter.tar.gz
```

If you want the CUTEr files to be accessible to a number of users on a shared filesystem on your local network, you might need privileged access to your machines, or to have these steps performed by your system administrator.

The current installation is via a text-based interface, in which the user is prompted for choices pertaining to the desired installation. The main installation script is `install_cuter` and interacts with a number of auxiliary scripts. We examine these scripts in turn, using an example of a CUTEr installation on a shared-filesystem network. The scripts provided are:

1. `install_cuter`: installs a new instance of CUTEr on the system,
2. `update_cuter`: updates files in an installed instance of CUTEr,
3. `uninstall_cuter`: remove a particular instance of CUTEr installation.

In addition the the three above scripts, we will also examine a manner to re-generate parts of CUTEr, due to the modification of one or more files.

These scripts can be found in

```
$CUTER/build/scripts
```

Suppose, by way of example, that your local network contains the following machines (amongst others).

1. a SUN Ultra workstation running Solaris with an installed Sun Fortran 90 compiler, f90,
2. an intel-based personal computer running LINUX for which the Gnu Fortran 77 compiler, g77, is installed, and
3. a Compaq Alpha running Tru-64 for which the Compaq Fortran 77 compiler, f77, is available.

In the remainder of this documentation, we describe the role of the aforementioned scripts based on this example of network. Assume that you wish to install an instance of CUTEr for each of these machines, according to Table 1.1:

Machine	Compiler	Size	Precision
SUN	f90	large	double
Intel	g77	medium	single
Compaq	f90	large	double

Table 1.1: A possible installation of CUTEr on a shared-filesystem network. Size refers to the (maximum) dimension of the examples you wish to run, while Precision denotes the floating-point precision required.

1.1.1 `install_cuter`

This script serves the dual purposes of installing the initial instance of CUTEr on your system and of installing an additional instance, for a different architecture, where by *architecture*, we mean the combination machine–operating system–compiler–size–precision.

Assume, for the purpose of an illustration, that you wish to install all your different instances of CUTEr in the directory `$CUTER = /usr/share/cuter/`. Unpacking the CUTEr distribution in the `$CUTER` directory and launching the initial installation, say for the SUN Ultra machine, is done by simply typing

```
prompt% install_cuter
```

at the command prompt. However, before issuing the install command, we recommend that you check the files *system*`.cf`, where '*system*' represents your operating system, to make sure the commands there are correctly defined for your environment, and that the temporary directory is correctly set. The current directory '.' must *not* be used as temporary directory. Once you have issued the `install_cuter` command, you will be prompted for information regarding the instance of CUTEr you wish to install. The first question concerns your machine. In this case, select "Sun workstation" (7). Next, select the operating system your machine is running. Here, we select "Solaris" (1). You are then faced with a list of available Fortran compilers for your machine (without any guarantee that these compilers are actually *installed* on your machine, simply those we know are available for the machine–operating system combination you have selected)—we want to select "Sun f90" (4)—and a list of available C/C++ compilers for your machine—we select Sun Workshop6 C++ (2). Select next the precision of the CUTEr tools (single or double), and their size (small, medium, large or customized).

Once this information has been provided to the installation script, you are given a default directory name where the selected instance of CUTEr will be installed. This directory is a subdirectory of `$CUTER` that you chose earlier (in this case, `/usr/share/cuter/`). For the present instance, the default directory is

```
/usr/share/cuter/CUTEr.large.sun.sol.f90
```

reflecting the selections you made during the early installation phase. This directory name should be self-explanatory and should help you and other users determine where each installed instance of CUTEr is actually stored. Notice that the precision is not reflected in the directory name. The reason is that both single and double precision instances of CUTEr may be installed for the same machine-operating system-compiler-size combination; these will be stored in the `single/` and `double/` subdirectories of the above directory. If you wish, you may redefine the directory name and give it whatever name you like—it need not be a subdirectory of `$CUTER`. Note however that you should give the *full pathname* of the new directory that you choose, *e.g.*

```
/home/mjdpowell/software/yetAnotherCuter
```

even if this new directory is a subdirectory of the `$CUTER` directory:

```
/usr/share/cuter/aCustomCuter
```

It is probably good practice to be content with the default name or not to give it a cryptic or ambiguous name. After checks to see if a similar distribution has already been installed and for the existence of the specified directory, the installation itself begins.

The script `install_cuter` then creates the necessary directory structure, *Umakefiles* and configuration files. The final step of the installation is left to the user and is described below.

Once this phase is complete, `install_cuter` reminds you of what you should add to your `.cshrc`, `.bashrc`, or whichever UNIX configuration file corresponds to the shell you use. The directory structure after the initial installation is as described in the CUTEr paper provided in the CUTEr distribution and in §1.2 and Fig.1.1. In the case we are concerned with, the `CUTER` environment variable should be set to `/usr/share/cuter` and `MYCUTER` to `/usr/share/cuter/CUTEr.large.sun.sol.f90` (or the alternative directory you specified during the installation phase).

`install_cuter` also advises you to read the various `README` files scattered over the directory tree under `$MYCUTER`. We now describe the final step of the installation using *Umakefiles*. There is a *Umakefile* in each subdirectory of `$MYCUTER`. Each of these *Umakefiles* needs to appropriately use the configurations files stored in `$MYCUTER/config` so as to generate *Makefiles* suited to your local system. This process is usually referred to as *bootstrapping*[1]. This is done by changing to `$MYCUTER` and issuing the command

```
prompt% ./install_mycuter
```

at the command prompt. Please note that if both single and double precision were installed, the script `install_mycuter` requires a command-line argument, telling it for which precision it should bootstrap the *Umakefiles*. In an attempt to follow the main guidelines for the CPP — the C preprocessor, on which *Umakefiles* are based — the argument to `install_mycuter` takes the form of a symbol definition. More precisely, if the user wishes to remake the double precision version of CUTEr, the command is

```
prompt% ./install_mycuter -DDoublePrecision
```

[1] We have reused this terminology, used for Imakefiles—configuration files generally used to install the X-Window system. *Umakefiles* are in fact a much simplified version of *Imakefiles*

and similarly, for the single precision version,

```
prompt% ./install_mycuter -DSinglePrecision
```

Refer to the file `IMPORTANT` for the latest details. Do not let `make`'s output confuse you. On a LINUX system, and because `make` is usually accompanied by the `-w` command-line option, using the standard g77 compiler, the output of the above command looks like

```
umake -I./config -DIsg77 -DLargeSize -DDoublePrecision
+ /bin/rm -f Makefile.bak
+ /bin/mv Makefile Makefile.bak
umake -I./config  -DTOPDIR=. -DCURDIR=.  -DIsg77 -DLargeSize -DDoublePrecision
making Makefiles in bin...
make[1]: Entering directory `/home/do/Cuter4Linux/bin'
make[1]: Nothing to be done for `Makefiles'.
make[1]: Leaving directory `/home/do/Cuter4Linux/bin'
making Makefiles in double...
make[1]: Entering directory `/home/do/Cuter4Linux/double'
making Makefiles in double/bin...
make[2]: Entering directory `/home/do/Cuter4Linux/double/bin'
make[2]: Nothing to be done for `Makefiles'.
make[2]: Leaving directory `/home/do/Cuter4Linux/double/bin'
making Makefiles in double/config...
make[2]: Entering directory `/home/do/Cuter4Linux/double/config'
make[2]: Nothing to be done for `Makefiles'.
make[2]: Leaving directory `/home/do/Cuter4Linux/double/config'
making Makefiles in double/lib...
make[2]: Entering directory `/home/do/Cuter4Linux/double/lib'
make[2]: Nothing to be done for `Makefiles'.
make[2]: Leaving directory `/home/do/Cuter4Linux/double/lib'
making Makefiles in double/specs...
make[2]: Entering directory `/home/do/Cuter4Linux/double/specs'
make[2]: Nothing to be done for `Makefiles'.
make[2]: Leaving directory `/home/do/Cuter4Linux/double/specs'
make[1]: Leaving directory `/home/do/Cuter4Linux/double'
```

This is *normal* output and it indicates that everything worked out smoothly. `make` is simply echoing what it attempts to do in each subdirectory. A message like "Nothing to be done for 'Makefiles'." simply indicates that the subdirectory where `make` is currently working does not have further subdirectories. On most systems, `make` is less verbose.

The above command should be able to properly generate the *Makefiles* in each subdirectory. These *Makefiles* should also only contain standard commands, as every effort has been made to avoid using exotic *Makefile* features and capabilities, such as the `$@@` construct. A `README` file accompanies every *Makefile* to describe what it does and which targets it recognizes. Users are advised to take a look at these files. The documentation files and a basic knowledge of `make` should be enough for you to feel comfortable with the (re)generation of the various parts of CUTEr. Once the *Makefiles* are generated, the only thing that remains to be done is the usual `make all`. However, as users who have some experience with `make` know, `make` outputs a lot of information—it basically echoes to the standard output every action it takes. The `-s` command-line option to `make` lowers its verbosity level and basic information on how the build progresses only is printed. Thus, users should build CUTEr using the command

```
prompt% make -s all
```

This command completes the installation of CUTEr, using *Umakefiles*. On my Linux system, the installation takes a couple of minutes and `make`'s output looks like

```
 Getting UNIX commands right             [Ok]
 Casting      script.sed                 [Ok]
 Casting      cast.sed                   [Ok]
 Casting      local.f                    [Ok]
 Adding       timer                      [Ok]
 Building     local.o                    [Ok]
making all in ./bin...
 Casting      cob                        [Ok]
 Casting      fil                        [Ok]
 Casting      gen                        [Ok]
   ...          ...                       ...
 Building     uofg.o                     [Ok]
 Building     uprod.o                    [Ok]
 Building     ureprt.o                   [Ok]
 Building     usetup.o                   [Ok]
 Building     ush.o                      [Ok]
 Building     uvarty.o                   [Ok]
 Archiving    libcuter.a                 [Ok]
making all in double/specs...
```

On workstations, the installation should be expected to take longer. During this phase, keep an eye on the screen and look for the `[Ok]` indicators. Should `make` come across some difficulty, this sequence of indicators should be interrupted by an error message. To know more about the problem, read the README file in the directory where the problem occured to try to indentify the target which `make` was attempting to build, and re-run `make` on that target without the `-s` option.

You may then install a new instance of CUTEr, which may be for a different architecture, or one corresponding to an already-installed instance, with a different precision or size. In all cases, the environment variable `MYCUTER` should point to the current, working, instance of CUTEr.

The `install_cuter` script keeps track of all installed instances of CUTEr on your system in the logfile `$CUTER/log/install.log`. This file may be used, for instance, to have `MYCUTER` point to the right distribution. For the purpose of illustrating the above, assume the three distributions given in Table 1.1 are installed in their default directory. Besides date information, the following will be found in `$CUTER/log/install.log`, where the exclamation mark (`!`) is a separator.

```
double large Sun-workstation sol f90   !   $CUTER/CUTEr.large.sun.sol.f90
double medium Intel-like-PC lnx g77    !   $CUTER/CUTEr.medium.pc.lnx.g77
double large Compaq-Alpha t64 f90      !   $CUTER/CUTEr.large.alp.t64.f90
```

1.1.2 update_cuter

As it is our intention to upgrade over time (or fix if necessary) the tools supplied in the CUTEr package, a mechanism for keeping an installed system up to date, and to install newer instances of the tools, is required. This is the role of the update_cuter script. If all goes well, you should not have to use update_cuter immediately. Announcements of bug-fixes and enhancements will be posted and indicated on the website. There are two forms of the command.

In its first form, update_cuter takes two command-line options, as follows

```
prompt% update_cuter filename
```

where *filename* is the name of the file to upgrade, possibly specified with a path. Suppose, for example, that the file ufn.f has been improved so as to perform its task faster, upgrading your current instance of CUTEr is achieved by typing

```
prompt% update_cuter ufn.f
```

at the command prompt. This command first copies the new source file to proper location, which is in this case $CUTER/common/src/tools. If there are currently both single and double precision instances, you will be asked to choose which you would like to update; if there is only one instance under $MYCUTER, the precision will be chosen accordingly. The script then casts and compiles the incoming file, and finally updates the CUTEr library ($MYCUTER/double/lib/libcuter.a). Of course, corresponding actions are performed depending on the type of *filename*: if it is a script, it is only cast, and stored in its proper place, and if it is a documentation file, it is simply moved to $CUTER/common/doc.

In its second form, update_cuter takes three command-line options, described as follows

```
prompt% update_cuter -a filename
```

where *filename* is the name of a file describing a list of CUTEr files to be upgraded. The file *filename* should contain

1. on its first line, the directory where the new (upgraded) files can be found, and
2. on subsequent lines, the names of those upgraded files, possibly preceded by their destination directory. A single file per line should be given.

Note that preceding the file names by their destination directory is not compulsory; in fact, the path is ignored and update_cuter tries to determine the correct path for itself. As an example, suppose that the tools ufn.f, install_cuter, compiler.cry.unc.f90 and sdknit.pro have been upgraded, and are temporarily stored in /home/upgrade. A corresponding input file might be

```
/home/upgrade
$CUTER/common/src/tools/ufn.f
$CUTER/build/scripts/install_cuter
compiler.cry.unc.f90
sdknit.pro
```

but exactly the same result would be produced by the simpler file

```
/home/upgrade
ufn.f
install_cuter
compiler.cry.unc.f90
sdknit.pro
```

or by the deliberately confusing file

```
/home/upgrade
/usr/share/junk/ufn.f
/home/upgrade/install_cuter
/home/downgrade/compiler.cry.unc.f90
/opt/degrade/sdknit.pro
```

As above, CUTEr copies these files from `/home/upgrade` to their proper location, prompts for the precision required (if necessary), casts and, where necessary, compiles the incoming files, and updates the specified instance stored under `$MYCUTER`.

The additional command-line option `-m` forces `update_cuter` to simply move the files to their proper location and to skip compilation. Help may be obtained from `update_cuter` through either of the `-h`, `-help` or `--help` flags.

To summarize, the complete synopsis of `update_cuter` is as follows

`update_cuter [-h | -help | --help] [-m] [-a` *listFile* | *newFile*`]`

In the situation where CUTEr has been unpacked but no further installation steps were performed, or all current instances were deleted, `update_cuter` still can move the updated source files to their proper location, skipping the compilation phase. The same syntax as above can be used.

Caution: attention should be paid to the fact that `update_cuter` works by `source`-ing the UNIX commands from the file `$MYCUTER/`*precision*`/config/cmds` (where *precision* is the required precision) and that these commands define the temporary directory used during compilation phase. In most cases, this temporary directory is simply `/tmp`. This temporary directory *must not* be the same as that specified in the first line of `update_cuter`'s input file (`/home/upgrade` in the examples above).

1.1.3 `uninstall_cuter`

The script `uninstall_cuter` is used to remove a previously installed instance of CUTEr from your system. If called with no argument, the user will be asked to choose which distribution to remove from a list of the instances found on the system. Otherwise, the only argument is the name of the directory containing the distribution to be removed. We illustrate the second case. Referring again to Table 1.1, assume we wish to remove the Compaq-Alpha distribution. This is done by issuing the command

```
prompt% uninstall_cuter $CUTER/CUTEr.large.alp.t64.f90
```

at the command prompt. If this directory contains both the single and double precision instances, you will be prompted for which should be removed. There is no possibility, at the moment, to remove both instances at once. If single or double precision instance only is present, the whole directory will be deleted as will the corresponding entry in `$CUTER/log/install.log`. Note that un-installing should be done from the same machine from which the installation command was issued, as the corresponding directory might not be recognized on other machines. Issuing the command

```
prompt% uninstall_cuter --help
```

will display a short help message. The script is itself self-documented and the user may consult it for more information.

1.1.4 Rebuilding CUTEr

A rebuild of CUTEr may turn out to be necessary whenever CUTEr informs the user that the workspace dimensions need to be increased—a rebuild may also turn out to be necessary whenever prototype files are modified, or in general, whenever *any* basic file is modified. CUTEr itself usually issues warning messages whenever the workspace is insufficient, urging the user to increase a particular (set of) parameters. These parameters may be tuned in `tools.siz` which can be found in

$MYCUTER/*precision*/config

where *precision* is either 'single' or 'double', according to your installation. For the change to take effect, the CUTEr tools need to be cast and compiled again. Assume the Solaris installation is modified. All the user needs to do to make sure he or she rebuilds everything that needs to be rebuilt is change to the directory `$MYCUTER` and issue a

```
prompt% make -s all
```

`make` then takes care of everything and rebuilds whichever targets depend on the updated files.

1.2 The CUTEr tree

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 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 impossible. Furthermore, 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 also impossible. The reason for these difficulties is that the structure of the CUTE files, as described in [**?**], 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 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 become 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 but 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 1.1, reflects these preoccupations. We now briefly described its components.

Starting from the top of the figure, the first subtree under the main `$CUTER` directory (the main 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 supported by CUTEr, allowing the user to install new architecture dependent subtrees in an evolving manner, depending on the testing needs, the evolution of the platforms, systems and compilers. The `prototypes` subdirectory contains the parts of the environment which have to be specialized to one architecture before it 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 last 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 its purpose, the testing of optimization packages, but that 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, the description of procedure to follow for interfacing the supported optimization packages, the complete SIF reference document, ...), 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. The `pkg` subdirectory of `src` is used to stored the information related to the various optimization packages for which CUTEr provides an interface. There is one subdirectory for each such package (we have represented that for the COBYLA and `VE12` packages), typically including an algorithmic specification file or the source code of the package if available. The subdirectory `include` of `common` contains the necessary header files for the interfaces between CUTEr and C codes. 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 *umake* when the latter is used to *bootstrap* the various *Imakfiles* 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 `$CUTER` are all architecture dependent: each of them corresponds to the installation of CUTEr on a specific machine, for a given operating system and compiler and for a given tool size. The figure only represents one, but the continuation dots at the bottom of the leftmost vertical line indicate that there might be more than one. The name of these directories are (by default) automatically chosen at installation, but a user of one of these subtrees would typically give it a symbolic name, like `$MYCUTER`, to refer to the instance of CUTEr currently in use. Each architecture-dependent subtree is divided into its single 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 optimization packages driving programs and, if relevant, of the package codes. The second, `lib`, contains library of cute tools and, if relevant, 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,

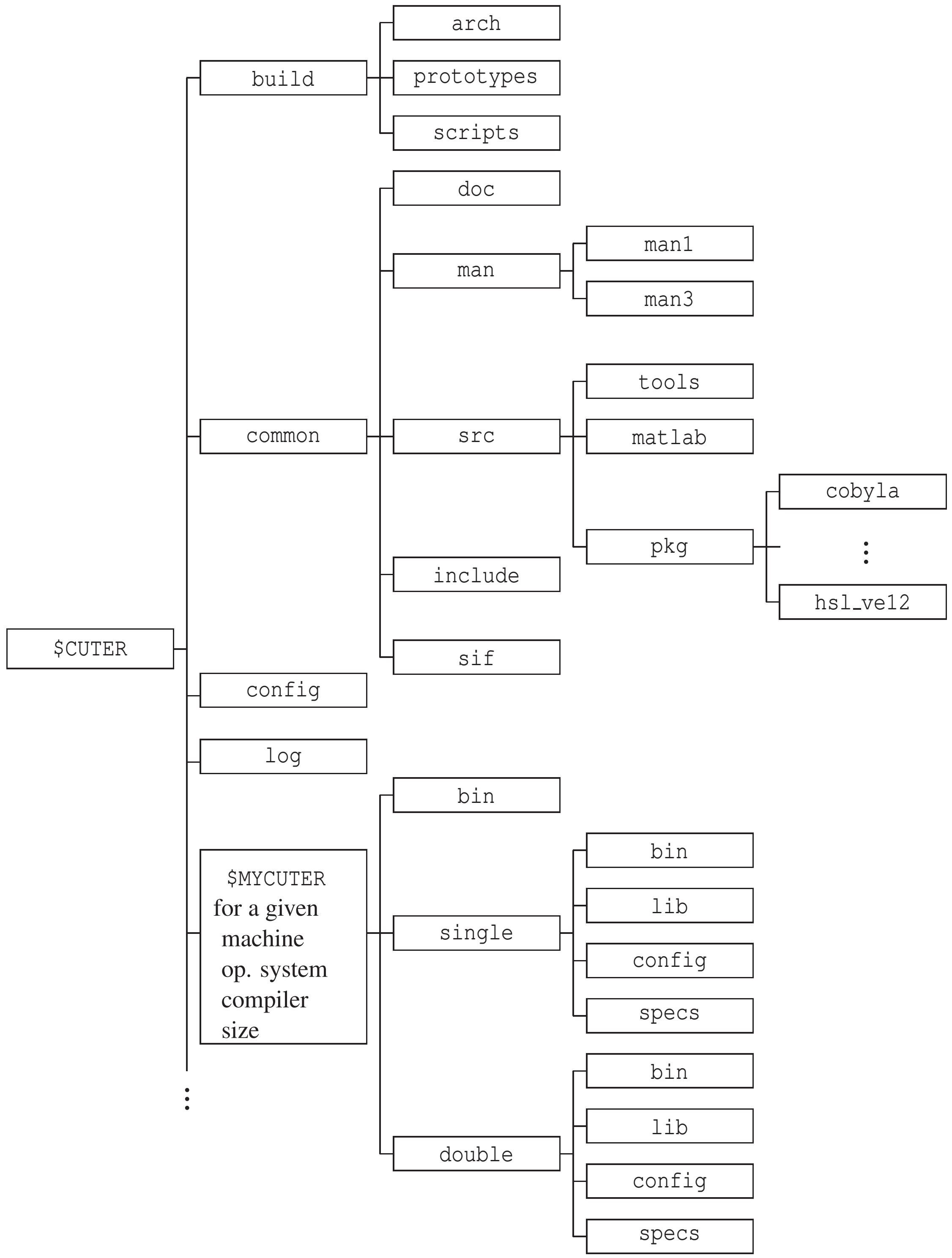

Figure 1.1: Structure of the CUTEr directories

if any. Finally, `$MYCUTER/bin` contains those scripts which are architecture-dependent, 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 need to specify the exact list of tools which have 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 both in acsii and pdf formats for the rest. It is hoped that this will make access to the relevant information more convenient for users.

1.3 Interfacing CUTEr and Matlab(R)

We describe in this section how CUTEr is interfaced with Matlab, remind basic concepts about MEX-Files and describe how new interfaces should be added.

1.3.1 MEX-Files basics

Besides being a self-sufficient environment, Matlab provides an Application Program Interface (API) to support external user-defined subroutines. This interfacing is realized through dynamically-linked subroutines, compiled by Matlab from C or Fortran source code, referred to as MEX-Files. For a thorough exploration of MEX-Files accompanied by numerous examples and details, refer to your local Matlab documentation. Recall also that all API-related documentation is available online using the Matlab Help Desk. We now briefly review the main features of MEX-Files and what the user should provide for Matlab to be able to compile the interface.

Any user-provided C or Fortran computational routine may be interfaced with the Matlab environment using the `mex` script. Within Matlab, the `mex` command takes the name of the routine to be interfaced as an argument plus a number of options and possibly other files. Refer to the `mex` script documentation for a complete list of all supported options. For every routine the user wishes to interface with Matlab, a *gateway* routine must be provided in order to inform Matlab about the number of arguments that the computational routine takes and what their type is. This gateway routine calls the user-defined computational routine as a subroutine. The file resulting of the compilation and linking of these two routines is called a MEX-File. Assume your own Fortran routine `qrFactor.f` is to be interfaced with Matlab and that a gateway is provided in the file `qrFactorg.f`, Matlab compiles and links `qrFactor.f` once it is given the command

```
matlab% mex qrFactor.f qrFactorg.f
```

Possibly, if several user-defined routines are to be compiled within Matlab, the gateway routine may interface them all at once. For details regarding the construction of a gateway, the reader is referred to the Matlab documentation.

1.3.2 CUTEr and MEX-Files

With CUTEr, gateway interfaces to the CUTEr tools are provided in three files. The file `utools.f` interfaces the unconstrained optimization tools, `ctools.f` interfaces the constrained optimization tools and gtools.f contains general tools used by utools.f and ctools.f. These gateway files can be found in

```
$CUTER/common/src/tools/
```

while shortcut files to call the corresponding routines may be found in

```
$CUTER/common/src/matlab/
```

For example, $CUTER/common/src/matlab/ contains the file usetup.m which allows the user to simply call the usetup tools by typing

```
matlab% [x,bl,bu]=usetup;
```

at the Matlab prompt instead of

```
matlab% [x,bl,bu]=utools('usetup');
```

Note that the calling sequence from within Matlab may to some extent differ from the "usual" Fortran calling sequence.

Help is available from within Matlab the usual way, by typing

matlab% help *toolname*.

For example,

```
matlab% help usetup
```

briefly documents the usetup tool. Note that MEX-Files created from Fortran source code may only handle double precision data. As a general rule, a C or Fortran routine or function compiled and linked into a MEX-File is called using

matlab% [$o_1, o_2, \ldots, o_n$] = function_name($i_1, i_2, \ldots, i_m$);

where $o_1, o_2, \ldots, o_n$ is the output arguments list (specified within square brackets) and $i_1, i_2, \ldots, i_m$ is the input arguments list.

The gateway interfaces use the mxCopy__ construct by default. If your C or Fortran compiler supports the %val construct, which implements calls by address instead of calls by value, it should be used so as to free memory used by (no longer necessary) temporary variables and to speed up execution. Besides being more intuitive, the %val construct also considerably eases the programming effort, shortens the code and makes better use of available memory.

1.3.3 Using CUTEr from within Matlab

Compiling and linking the CUTEr tools with a solver written in C or Fortran, and to be used in Matlab is done by creating the corresponding MEX-File. This is the purpose of the two scripts sdmx and mx, found under $MYCUTER/bin. The purpose of these two scripts shadows that of the other scripts; sdmx first SIF-decodes the problem it is given as an argument and leaves the creation of the MEX-Files to mx. mx invokes the C or Fortran compiler distributed with Matlab, links all the object files and libraries together and finally creates the MEX-File, which can be called as a regular Matlab function.

Versions of Matlab older than 6.0 used to have separate C and Fortran compilers, often called respectively cmex and fmex. Recent versions have merged the two compilers into one, mex, making the other two obsolete. You should update your file $MYCUTER/*precision*/config/cmds to reflect your local instance of Matlab. Please also update `$CUTER/build/arch/system.`*your_system* accordingly, to make your modifications system-wide. For the above reason, the CUTEr interfaces to Matlab are no longer called `sdmex` and `mex`, but `sdmx` and `mx`.

For instance, to compile and link all the unconstrained CUTEr tools with the problem `MSQRTALS.SIF`, use

```
prompt% sdmx -u MSQRTALS
```

Note the use of the `-u` command-line option to `sdmx` in order to use the unconstrained tools. To use the constrained tools, simply omit the `-u` option. Also note that Matlab MEX-Files always use double precision and therefore, a double-precision SIF decoder should have been installed on the system.

Place the MEX-File thus generated in the same directory as the files produced by the SIF decoder, and in particular `OUTSDIF.d`, and the interfaces described in §1.3.2 are ready to be called from Matlab.

If your problem has already been decoded, the MEX-File can be regenerated using the command

```
prompt% mx -u
```

1.3.4 Adding a new tool

Should the user add new C or Fortran tools to CUTEr that are to be interfaced with Matlab, `utools.f` and/or `ctools.f` should be updated accordingly and the corresponding `.m` file should be created and stored in the directory `$CUTER/common/src/matlab/`. For information on MEX-File debugging, refer to your local MATLAB documentation.

The user should also pay attention to their local implementation of pointers. If the `mxCopy__` construct is used, pointers should be declared as `integer*8` on DEC Alpha and 64-bit SGI machines and as `integer*4` on all other platforms. If the local Fortran compiler supports this option, a clear multi-platform code may be obtained by having the C preprocessor map the pointers to the correct declarations at the pre-processing stage.

Note finally that after compilation and linking, the name of the resulting MEX-File will have an extension that depends on the platform on which compilation was performed. For example, this extension is `mexsol` on Sun machines running Solaris, and `mexsg64` on 64-bit SGI machines.

1.4 User-modifiable parts

Nearly all the Fortran source files, stored in `$CUTER/common/src/tools`, have user-modifiable parts. These parts are not directly included in the Fortran source code, but cast prior to compilation. The files containing the user-modifiable data are `tools.siz` and `sifdec.siz`. After the initial installation, these files will be found in the CUTEr directory `$MYCUTER/[single|double]/config`. If modified, the CUTEr distribution may be rebuilt using the new parameter values by the `rebuild` script, located in `$CUTER/build/scripts`.

Some Fortran source files, like `clsf.f` and `slct.f`, have hardcoded user-modifiable parts. These are usually located at the top of the file, between banners, such as

```
C--------- THE FOLLOWING SPECIFICATIONS MAY BE MODIFIED BY THE USER ----------
```

and

```
C--------------- END OF THE USER MODIFIABLE SPECIFICATION ------------------
```

1.5 CUTEr tools

Problems are fully described in a Standard Input Format (SIF) file. This file may subsequently be decoded to provide data and Fortran subroutines for input to a nonlinear programming package.

Here we describe auxiliary subroutines which are available for users to manipulate the decoded data. The Fortran source of these programs, along with the subroutines obtained when decoding the SIF file, should be compiled with the user's optimization package.

The CUTEr tools are described in the man pages, category 3, which may be viewed using the `man` command, its X interface `xman`, or, on LINUX systems, by issuing `less` *manpage* where *manpage* is the man page to be viewed. The man pages are stored in `$CUTER/common/man`, and this directory should appear in the user's `MANPATH`. Table 1.2 contains those tools related to unconstrained or bound-constrained minimization as of March 24, 2005, along with a brief description, and Table 1.3 contains a list of CUTEr tools for constrained minimization. The `ureprt` and `creprt` tools produce statistics about a particular run. Users of the previous versions of CUTE will notice the strong similarity in the tools names.

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. Explicit mentions appear whenever this matrix is stored in finite-elements format.

Note on the `creprt` tool

As CUTEr features tools allowing users to evaluate a *single* constraint and as reporting the number of evaluations of *each* constraint in the final statistics is not practical, the statistics report a measure of the this quantity, defined as

$$\#\mathrm{eval}(c) = \frac{\sum_i \#\mathrm{eval}(c_i)}{m},$$

where $\sum_i \#\mathrm{eval}(c_i)$ is the sum of the total number of times each individual constraint is evaluated and m is the number of constraints in the problem. Note that if the algorithm always evaluate *all* constraints at once, this measure is an integer. Otherwise, it may be a real number. The purpose of this ratio is to provide a measure of the number of constraint function evaluations (as compared to its maximum : m) in the course of the iterations.

1.6 CUTEr sizes

The CUTEr package is distributed with three default "sizes": large, medium, and small. In addition, there is a *custom* size, which, as the term indicates, may be suitable for situations demanding a specialized configuration. These sizes refer to the size of the memory available for problem decoding and solution, and hence are directly related to the size (the amount of data) of the problems that CUTEr can tackle. It may happen that the predetermined sizes do not fit your favorite problem or your machine, and that you wish to specify your own. Typically, when running too large a problem, CUTEr will complain that one of the size parameters is too small and stop. You then have to increase this parameter (if this is possible on your machine) in order to handle the problem. This modification of the CUTEr array sizes is explained below.

We first note that the dependency on problem size occurs in both the decoding of the problem `SIF` file into data structures and subroutines and in the computation of the required problem values by the provided evaluation tools. Indeed, the problem dependent data is fully specified by its associated `SIF` file and must be taken into account in all stages up to the numerical solution process. Therefore, the size of both the `SIF` decoder and the tools (and indeed, the interface optimizers) must be adequate for the problem.

The actual choice of one of the predetermined sizes is made when running the `install_cuter` command, which prompts the user for the desired size. In fact, `install_cuter`, or the *Makefiles*, depending on which CUTEr you are using, cast the source code against a "size mask" corresponding to the selected size, and thereby determines the dimensions of the various arrays used in the code. The assignment statements are differentiated by their first four characters:

`CBIG`	specifies the large size
`CMED`	the medium size
`CTOY`	the small size
`CCUS`	the custom size

Note that the custom size is first thought of as larger than the large size, but nothing prevents the user from building an intermediary size or a smaller size than the toy size.

Changing the size of the CUTEr distribution in the sense just described may call for a partial re-installation. If most (or all) parameters must be, say, increased, it might be worth considering simply re-installing CUTEr using a larger size (*e.g.* large if medium turns out to be insufficient for your purposes). To that end, execute `install_cuter` again and select the correct size. In case very few parameters need to be changed, the procedure described below might be considered. We now examine this procedure in more detail.

1.6.1 tools sizes

The tools sizes are gathered in the file `$MYCUTER/`*precision*`/config/tools.siz`, which contains the following parameters.

Parameter	Brief description
`LIWK`	the size of the integer workspace array used by the algorithms
`LWK`	the size of the single or double precision workspace array used
	the algorithms, according to the precision of the instance installed
`LLOGIC`	the size of the logical workspace array used by the algorithms
`LCHARA`	the number of ten character strings used as workspace by the algorithms
`LFUVAL`	the size of the array used to store the problem's function and derivative values.

These parameters are assigned a value upon initial installation of CUTEr on the system. These values should be changed according to CUTEr messages issued at run-time, and CUTEr should be rebuilt.

Note that the main drivers, whose names match the regular expressions `*ma.f` and `*ma.f90`, declare parameters such as `MMAX`—the maximal number of constraint functions in the problem—which are not found in `tools.siz`. It may happen that CUTEr aborts the solving of a problem because one of these parameters has not been set, in the main driver source file, to an appropriate value. Since these parameters are package-dependent rather than architecture-, or size-dependent, they should be set to an appropriate value *in the source file* and the latter should be recast and recompiled.

1.6.2 Sizes for the MATLAB interface tools

If the size of the MATLAB interface tools is to be modified, the parameters `NMAX` (the maximum number of variables in a problem) and `MMAX` (the maximum number of constraints in a problem) should be altered at the beginning of the files `ctools.f` and `utools.f`.

Changing compiler flags

In some circumstances, it might be useful to alter the predefined compiler flags. An example might be when some new level of code optimization becomes available on your machine. Note that care should be exercised with code optimizers: we know of cases where the optimizers introduce real bugs into the code. As a consequence, it might be a good idea to turn optimization off before deciding that some strange behaviour of the package is anomalous and worth reporting. This is another reason why modifying compiler flags might be useful. Some operating system revisions might also require that you change machine dependent constants or procedures (such as the timer).

If compiler flags should be changed prior to a rebuild, the user should do so by altering the `CompileCmd` and `LoadCmd` variables corresponding to their compiler in the file `$MYCUTER/config/`*system*`.cf` or in `$MYCUTER/config/all.cf`.

If compiler flags should be changed to affect *all* subsequent installations of CUTEr, the user should do so in some or all the files `$CUTER/config/`*system*`.cf`, including `$MYCUTER/config/all.cf`.

For instance, if the compiler in question is only found on SUN machines, the file `sun.cf` should be modified. If it may be found on any machine, the file `all.cf` should be modified.

System dependent constants and functions

All the system dependent constants and functions are specified in the `$MYCUTER/config/`*system*`.cf` or in `$MYCUTER/config/all.cf` files, and also in the Fortran file `$MYCUTER/`*precision*`/config/local.f` after the initial installation. If these need to be changed, this latter file is the one on which to operate before rebuilding CUTEr. Keep in mind that altering some or all the `$CUTER/config/`*system*`.cf` or in `$CUTER/config/all.cf` files will affect *all* subsequent installations of CUTEr.

A set of hashing routines

The routines `HASHA`, `HASHB`, `HASHC` and `HASHE` provide a Fortran hashing tool. They are system dependent in that they rely on the number of bytes used to represent an integer within the particular Fortran dialect used. This number of bytes is set in the parameter `NBYTES` in `$MYCUTER/config/`*system*`.cf` or in `$MYCUTER/config/all.cf`. If your Fortran compiler uses an “unorthodox” number of bytes for its integers, you will have to change the value of `NBYTES`.

A definition of the arithmetic constants

The supplied functions `SMACHR` and `DMACHR` return values for various machine dependent constants, for single and double precision arithmetic, respectively. These machine constants are denoted R*n* in single precision and D*n* in double precision. We recapitulate them in the following table

Parameter	Brief description
R1, D1	the smallest positive number ε_1 such that $1+\varepsilon_1 > 1$
R2, D2	the smallest positive number ε_2 such that $1-\varepsilon_2 < 1$
R3, D3	the smallest nonzero positive number
R4, D4	the smallest full precision positive number;
R5, D5	the largest finite positive number

Each of these numbers should be modified, either in the configuration files `all.cf` and/or <your_system>.cf, found under `$MYCUTER/config`.

A CPU timer

This is a real function `CPUTIM`, that returns the current CPU-time used by the package, expressed in seconds. This timer is, unfortunately, highly system dependent. The specific code for `CPUTIM` is originally located in `$CUTER/build/arch/compiler.*.*.*` and concatenated with `local.f` at cast time, during installation.

1.6.3 Rebuilding CUTEr

Rebuilding CUTEr is done as described in §1.1.4. Simply change to `$MYCUTER` and issue a `make -s all` to make sure that everything that needs to be rebuilt is rebuilt.

1.7 Driver programs

Driver programs are Fortran source main programs that call relevant user-provided subroutines from a particular optimization or linear algebra package, and which obtain function, derivative and other problem information directly from CUTEr subroutine tools. A driver is compiled and run by the *interface* to that package.

For example, the CUTEr distribution includes an interface to the PRAXIS package. This interface is provided by two UNIX scripts, `sdprx.pro` and `prx.pro`, stored in `$CUTER/build/prototypes`, which are cast into `sdprx` and `prx` and subsequently stored in `$MYCUTER/bin`. Both scripts make use of the auxiliary script `runpackage`, cast from prototype `runpackage.pro`, which is common to all interfaces. More information on the usage of interfaces is given in §1.9. The role of these three scripts is to decode the input problem into the proper Fortran subroutines, gather the necessary libraries and object files, link and compile them together and finally launch the PRAXIS driver, whose source `prxma.f` is cast and linked into `prxma.o`, stored under `$MYCUTER/`*precision*`/bin`. The driver sets up all the necessary data structures and environment required by PRAXIS and calls the PRAXIS subroutines to solve the input problem.

All supported packages are represented by an abbreviated name. For the purpose of an illustration, assume this abbreviation is *pak*. The interfaces scripts are called `sd`*pak*`.pro` and *pak*`.pro`, and the Fortran driver program *pak*`ma.f`.

The packages themselves are **not** supplied in the CUTEr distribution as we only aim to provide a useful and efficient testing and developing environment. It is the responsibility of the user to get the package source or object files and properly link them.

More details regarding a specific supported package *pak* may be found in

`$CUTER/common/src/pkg/`*pak*`/README.`*pak*.

1.8 The `SIF` decoder

1.8.1 Where is the `SIF` decoder?

At this point, it is crucial to mention that, in contrast with earlier version of CUTE [?], the `SIF` decoder is no longer embedded in CUTEr. This choice was made for several reasons, some of which are now briefly explained. First, it seemed important to us to have a consistent set of tools within CUTEr which all depend, in an indentical manner, on the `SIF` decoder. The decoder, however, could serve other purposes than that of being a gear of the testing environment. As a prime example, the `SIF` decoder is a vital part of the forthcoming second release of the LANCELOT package [?], LANCELOT-B. It thus appeared more consistent to isolate the decoder and simply have the other packages—CUTEr, LANCELOT-B, but there could also be others—refer to it whenever necessary. Another reason is ease of maintenance, and consistency when upgrading the decoder. All the packages which refer to it are using the same version. Finally, the `SIF` decoder in its own right may evolve and develop separately. An illustration of this fact is its recent ability to generate routines for function evaluation suited for input to the HSL automatic differentiation packages HSL_AD01 and its threadsafe counterpart HSL_AD02 [?].

1.8.2 `SIF` decoder sizes

Warning or error messages issued by the `SIF` decoder should be interpreted as related to the SifDec package, and the adjustments which they suggest should be made in the file `sifdec.siz` found under `$MYSIFDEC/`*precision*`/config/`. For more information, see [?].

1.8.3 CUTEr and automatic differentiation

The Harwell Subroutine Library [?] contains two packages supplying automatic differentiation facilities. If either of these packages, HSL_AD01 or its threadsafe counterpart HSL_AD02, is available to the user, automatic differentiation may be used within CUTEr. Please note however that HSL_AD01 is a Fortran 90 package while HSL_AD02 is a Fortran 95 package. Suitable compilers must therefore be available. Forward and backward modes both provide first and second-order derivatives, while higher-order derivatives are available in forward mode only. We refer the user to [?] for more detailed information. As will be explained in § 1.9, command-line options to the interfaces allow users to select forward or backward mode, and which package they wish to use. The automatic differentiation packages should be used like any optimization package, *i.e.* they should be compiled but not linked. The object files should then be placed, or linked to from, the directory `$MYCUTER/`*precision*`/bin`.

1.9 Interfaces

This section describes existing CUTEr interfaces with optimization and linear algebra packages and how to create a new interface.

Information and usage of the different interfaces to existing optimization and linear algebra packages may be found in the man pages `$CUTER/common/man/man1`, and users should ensure that the directory `$CUTER/common/man` appears on their `MANPATH`. The man page for the generic script *script* may be viewed by issuing the command `man` *script*, its X interface `xman`, or, on LINUX systems, by executing the command `less` *script.1*. Table 1.4 shows the interfaces provided and the packages to which they correspond.

For the purpose of an illustration, let us now consider the `unc` and `sdunc` interfaces to the UNCMIN unconstrained minimization package. Their calling sequences are as follows

```
sdunc  [-s] [-h] [-k] [-o j] [-l secs] [-f] [-b] [-a j] [-show]
       [-param name=value[,name=value...]]  [-debug] probname[.SIF],
```

```
unc  [-n] [-h] [-s] [-k] [-r] [-o j] [-l secs] [-debug].
```

The purpose of `sdunc` is to `SIF`–decode `probname.SIF`, set environment variables defining object and specification files necessary to compile the main UNCMIN executable, and launch `runpackage`. The script `unc` is simliar, except that it assumes that the problem has already been decoded by the `SIF` decoder. The main executable is linked, compiled and run by `runpackage`. An important difference with previous versions of CUTE is that `runpackage` is independent of any interface; only those scripts given in Table 1.4 depend on the optimization package which must be interfaced. The arguments of `sdunc` and `unc` are thoroughly described in the man pages. We briefly review them here.

-s link, compile and run the single precision instance. Double precision is the default;

-h prints a help message;

-k keep the load module after use;

-r (`unc` only) discourage recompilation of the test problem;

-o *j* verbosity level: -o 0 is silent mode and -o 1 is verbose mode. The default is -o 0;

-l *secs* limits the CPU running time to *secs* seconds;

-f (`sdunc` only) generate the relevant subroutines for automatic differentiation in *forward* mode;

-b (`sdunc` only) generate the relevant subroutines for automatic differentiation in *backward* mode;

-a *j* (`sdunc` only) when used in conjunction with **-f** or **-b**, **-a 1** uses the older HSL automatic differentiation package AD01, which **-a 2** uses the newer, threadsafe, automatic differentiation package AD02;

-show (`sdunc` only) displays possible parameter settings for probname[.SIF]. Other options are ignored;

-param (`sdunc` only) cast probname[.SIF] against explicit parameter settings;

-debug links the libraries and compile with -g option so as to allow debugging;

-n (`unc` only) use the load module if it exists. The default is to recompile.

The main object files for the supported packages (*i.e.* in this case, `uncmin.o`) should be placed in (or symlinked to from the directory) `$MYCUTER/`*precision*`/bin`, while the required specification files should be placed in (or symlinked to from) the current directory.

1.10 Creating a new interface for an optimization package

1.10.1 General procedure for Fortran and C interfaces

The purpose of this section is to explain how one can build interface tools for another optimization package, similar to those interfaces `sdunc` and `unc` provided for UNCMIN. We provide generic scripts `sdgen` and `gen` to make this process easier. These scripts can be found in the `$CUTER/build/prototypes` directory.

For illustrative purposes, we assume the package for which one wishes to provide an interface is called `pack`. We suppose that both single and double precision instances of the package are available and that interfaces in both precisions are required. An interface for just one of the precisions can be obtained by ignoring any of the comments relating to the other. A number of additional comments regarding the interfacing of packages written in C are given in §1.10.2. We suggest the following steps.

1. Construct a driver program calling the new package and using the Fortran tools provided (for evaluating the objective function, its gradient, *etc.*) The existing driver programs (`uncma`, `mnsma`, `ve09ma`, `stnma`, *etc.*) might help you get started in writing this new driver.

   Now compile this program into an object file called `packma.o`. The double precision object should reside in the directory

   `$MYCUTER/double/bin`

   and the single precision object should reside in the directory

   `$MYCUTER/single/bin`,

2. Compile the double precision instance of the complete set of programs contained in the `pack` package into one object file[2] (for example, `packd.o`). Repeat this for the single precision instance of the package, producing a second object file (for example, `packs.o`),

3. Connect to `$CUTER/build/prototypes` and copy `sdgen.pro` and `gen.pro` to `sdpack.pro` and `pack.pro` respectively. This is done by typing

   ```
   prompt% cd $CUTER/build/prototypes
   prompt% cp sdgen.pro sdpack.pro
   prompt% cp gen.pro pack.pro
   ```

   at the command prompt,

4. Create the new directory `$CUTER/common/src/pkg/pack`,

[2] If users prefer, they may instead create random or static libraries `libpack.a` or `libpack.so`.

5. Edit `sdpack.pro` and `pack.pro` and modify them as follows:

   (a) change the name of the package to be interfaced from `gen` to `pack`. This is done in the assignment of the shell variables `PAC` and `PACKAGE` at the beginning of the script, so that it reads
   ```
   setenv PAC = pack
   setenv PACKAGE = pack
   ```

   (b) if applicable, add or remove command-line options to the interfaces `sdpack.pro` and `pack.pro`, paying special attention to those options that are passed over to the `SIF` decoder and to `runpackage.pro`,

   (c) properly set the `PACKOBJ` and `SPECS` environment variables to contain the object file(s)/libraries for the package and the specification file (if any) respectively. The object files[3] should lie in `$MYCUTER/`*precision*`/bin`, where *precision* is either single or double, and the specification file should lie in the directory `$CUTER/common/src/pkg/$PACKAGE`, *i.e.* in this case, `$CUTER/common/src/pkg/pack`.

6. Possibly, alter `runpackage.pro` to link BLAS libraries, or other relevant libraries.

7. The scripts now need to be cast against your machine-dependent specifications. This may be done by issuing the commands
   ```
   prompt% sed -f $MYCUTER/precision/config/script.sed file.pro > $MYCUTER/bin/file
   prompt% chmod a+x $MYCUTER/bin/file
   ```
   at the command prompt, where *file* is successively `sdpack` and `pack`. If you altered `runpackage.pro`, the same should be done for *file*=`runpackage`.

Note: In case your package is not going to be available on all the platforms for which you have a CUTEr installation, modify the scripts `sdgen`, `gen` and `runpackage` found in `$MYCUTER/bin` instead of the prototype scripts. Otherwise, it is recommended that you write prototypes and store them in `$CUTER/build/prototypes`.

We would be very pleased if you could send your interface and driver program to us, so that we can redistribute it with future versions of CUTEr, with proper ackowledgments. Thank you in advance and good luck!

1.10.2 Interfacing packages written in C: `cuter.h`

We comment in this section the possibility of interfacing packages originally written in C with CUTEr. As of yet, the LOQO interface is the only one written in C, but it is hoped that given the growing interest in C/C++ optimization packages, this number is meant to increase.

The new subdirectory `include` of `$CUTER/common` hosts the C header file `cuter.h` containing various declarations related to the coexistence of object files originating from Fortran and C source files, and simplifying calling sequences to the CUTEr tools from C. All the definition in this header file may be accessed by specifying

[3]If the user has created random or static libraries, these should appear in `$MYCUTER/`*precision*`/lib`

```
#include{cuter.h}
```

at the top of your C driver.

We now briefly describe the header file `cuter.h` and the apparent prototypes of the CUTEr tools, as seen from the C language.

Partly inspired by `f2c.h`, `cuter.h` defines the types

```
typedef long int integer;
typedef float    real;
typedef double   doublereal;
typedef long int logical;
```

meant to imitate the corresponding Fortran data types. Users may then define variables of type `doublereal` when having in mind a corresponding Fortran 77 variable or argument of type `double precision`, or a Fortran 90 variable or argument of type `real` whose `kind` is that of `1.0D+0`.

It also defines the two macros

```
#define FALSE_ (0)
#define TRUE_  (1)
```

simulating the two possible values for Fortran variables of type `logical`. Note the trailing underscores.

In CUTEr, positive or negative infinite values are achieved by any number larger in modulus than 10^{20}, hence the definition

```
#define CUTE_INF 1e20
```

in `cuter.h`.

For convenience, and to account for the differences between the plethora of Fortran and C compilers out there, common *apparent* prototypes for the CUTEr tools have been defined in `cuter.h`. These apparent prototypes follow the general pattern

```
TOOLNAME( arg1, arg2, ..., argn )
```

where the tool name `TOOLNAME` must be specified in uppercase letters. Caution should be exercised when specifying the arguments of a routine, given the fact that to interface Fortran and C, *all* the arguments appearing in the argument list should be *pointers*. In practice, this has the consequence that `integer` variables appearing in an argument list in the driver should be declared as

```
integer *variable_name;
```

instead of

```
integer variable_name;
```

As is always the case in C, the exception to this is arrays, since they are always treated as pointers. Thus, for instance, a reference to the CUTEr tool `CSH` with the help of `cuter.h` appears as

```
CSH( &n, &m, x, &m, v, &nnzh, &lh, h, irnh, icnh );
```

with the declarations

```
integer *n, *m, *nnzh, *lh, *irnh, *icnh;
doublereal *x, *v, *h;
```

Note that in these declarations, `irnh`, `icnh`, `x`, `v` and `h` are arrays, while the other variables simulate Fortran `integer` variables. This is the reason why the addresses of the latter variables appear explicitly in the calling sequence to `CSH`, while the arrays appear as if they were the "real" Fortran arguments.

The calling sequences of most tools are exactly as in Fortran—they can be seen by typing

```
prompt% man toolname
```

at the command prompt. The prototypes say that the CUTEr tools all return a `void` ouput.

One notable difference between Fortran and C is the way external files are handled. C uses *streams* while Fortran requires that a *unit number* be associated to each open file. To account for this difference, and as the unit number is required by several CUTEr tools, the two functions `FORTRAN_OPEN` and `FORTRAN_CLOSE` have been defined, with apparent prototypes

```
void FORTRAN_OPEN( integer *funit, char *fname, integer *ierr );
void FORTRAN_CLOSE( integer *funit, integer *ierr );
```

`funit` being the unit number associated to the file, and `ierr` being the error code returned by these functions, a value of zero indicating a successful operation. These two functions may be called to open and close the `OUTSDIF.d` file generated by the `SIF` decoder.

1.11 Checking the integrity of a `SIF` file

All the interfaces given in Table 1.4 follow the same pattern. If the `SIF` problem has not yet been decoded, the interface first calls the `SIF` decoder by means of the `sifdecode` script. Please pay attention to the fact that, as mentioned in § 1.8, the `SIF` decoder is now distributed as a separate package and *must* be installed prior to using any of the CUTEr interfaces. The main executable for the decoder must be found in `$MYSIFDEC/`*precision*`/bin/sifdec`, where `$MYSIFDEC` is an environment variable pointing to the current instance of SifDec. Failing to do so will result in an abort.

Once the problem has been decoded, the interface calls a common script called `runpackage` which links the relocatables together, creates an executable file and finally executes it. It may be useful in some cases to decode a `SIF`-encoded problem without running an optimization package afterwards, or to simply check the syntax of the `SIF` file. In that respect, the `sifdecode` script may be called independently, from the command line. Its syntax is similar to that of the interfaces:

```
sifdecode   [-s] [-h] [-k] [-o j] [-l secs] [-f] [-b] [-a j] [-show]
            [-param name=value[,name=value...]]  [-force] [-debug] probname[.SIF]
```

Note that some of the command-line options only make sense when an optimization package is called after the problem has been decoded. For more information on `sifdecode`, we refer the reader to the documentation of SifDec, [?].

1.12 Attempting installation on an unsupported architecture

As far as UNIX-like platforms are concerned, it should not be too difficult to port CUTEr. This might require, however, a number of changes in several files. We suggest in this section where some of these modifications could take place. Additional modifications may be necessary, depending on your local system.

First, the installation scripts themselves may need to be altered, for compatibility reasons: the local C shell, if there is one, may be different, or require different command-line options. For example, the very first line of `install_cuter` may be `#!/bin/csh` under Solaris, but has to be `#!/bin/csh -f` on LINUX machines. All the scripts included in the CUTEr distribution are thoroughly self-documented and should be rather quickly understood by anyone familiar with the UNIX environment and the C shell. Similarly, as all the CUTEr scripts use the C shell, they may all need corresponding modifications.

You may need to alter a few *Umake* configuration files stored under `$CUTER/config`, such as `all.cf` and/or `<your_system>.cf`. Also make sure that new compilers that you define there appear in

`$CUTER/build/arch/f.arch` or `$CUTER/build/arch/c.arch`

for Fortran and CC++ compilers respectively, with matching symbols. More specifically, if your compiler name is *abc*, then the symbol which represents it in the configuration file must be “Isabc” and the block defining your compiler must look like

```
#ifdef Isabc
#define CompilerTagId           abc
#define umakeCompilerFlag       -DIsabc
#define CompileCmd              abc77 -c
#define LoadCmd                 abc77
#define CompilerIsF9095         yes
#define Compile9095Cmd          abc90 -c
#define Load9095Cmd             abc90
#define FortranFlags            -O
#define NumberOfBytes           8
#endif
```

where `abc77` and `abc90` represent the true compiler command for Fortran 77 and Fortran 90/95 source files respectively; these need not match the `abc` pattern. If the compiler `abc` does not support Fortran 90/95, then `CompilerIsF9095` should be set to `no` in the above block, and the two symbols `Compile9095Cmd` and `Load9095Cmd` should be defined to the empty string, *i.e.* :

```
#ifdef Isabc
#define CompilerTagId           abc
#define umakeCompilerFlag       -DIsabc
#define CompileCmd              abc77 -c
#define LoadCmd                 abc77
```

```
#define CompilerIsF9095        no
#define Compile9095Cmd
#define Load9095Cmd
#define FortranFlags           -O
#define NumberOfBytes          8
#endif
```

If you wish to support a compiler for your platform which is already defined in `all.cf`, but the compiler options are different on your platform, you need to make sure that your settings are not overwritten by those in `all.cf`. You might, for this, define a flag in `you_platform.cf` as part of the compiler definition and modify `all.cf` so as to skip the corresponding compiler definition. For instance, compiler `n95` is defined in both `mac.cf` and `all.cf`. Since `all.cf` will be sourced in all cases, the compiler definition in `mac.cf` contains the line

```
#define n95Defined
```

In `all.cf`, the definition of `n95` will be skipped if the symbol `n95Defined` has already been defined:

```
#ifdef Isn95
#ifndef n95Defined
...
#endif
#endif
```

The file `$CUTER/build/scripts/makefile.cmds` will need to be altered so as to include your new `your_system.cf`. This modification is however trivial.

If your system does not support man pages, these will be provided in pdf and other formats on the CUTEr website, as will updates to this general documentation and other information.

Fortran 77 files should be standard and compatible for the most part. Check your local compiler documentation for possible incompatibilities. If there is no available Fortran 90 compiler on your platform, you will not be able to use those tools (unless you write one).

If your new installation procedure is a success, we will be pleased to include it in the next releases of CUTEr, with proper credits. In this case, please send detailed information on your changes and on your local system. On the other hand, please feel free to contact us if you think we may be of some help.

Many thanks and again, good luck!

Tool name	Brief description
ubandh	extract a banded matrix out of the Hessian matrix,
udh	evaluate the Hessian matrix,
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 coordinate 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,
ugreh	evaluate the gradient and Hessian matrix in finite element format,
ugrsh	evaluate the gradient and Hessian matrix in coordinate 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 minimization,
ush	evaluate the sparse Hessian matrix,
uvarty	determine the type of each variable.
ureprt	obtain statistics concerning function evaluation and CPU time used,

Table 1.2: The unconstrained minimization CUTEr tools as of March 24, 2005.

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,
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,
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 minimization,
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 1.3: The constrained minimization CUTEr tools as of March 24, 2005.

Interface	Package
cgd/sdcgd	CG_Descent (Hager and Zhang)
cgp/sdcgp	CG+ (Liu, Nocedal and Waltz)
cob/sdcob	COBYLA (Powell)
fil/sdfil	FilterSQP (Fletcher and Leyffer)
gen/sdgen	Generic Fortran 77 interface
gen90/sdgen90	Generic Fortran 90 interface
genc/sdgenc	Generic C/C++ interface
hrb/sdhrb	SIF–Harwell- or Rutherford-Boeing sparse matrix format converter (Gould)
ipopt/sdipopt	IPOPT (Wächter)
knit/sdknit	KNITRO (Byrd, Nocedal and Waltz)
la04/sdla04	LA04 (Reid)
lbb/sdlbb	L-BFGS-B (updated) (Nocedal)
lbs/sdlbs	L-BFGS (Nodedal)
lmb/sdlmb	L-BFGS-B (Nocedal)
lqo/sdlqo	LOQO (Benson, Shanno and Vanderbei)
mns/sdmns	MINOS (Murtagh and Saunders)
nits/sdnits	NITSOL (Pernice and Walker)
nps/sdnps	NPSOL (Gill, Murray, Saunders and Wright)
osl/sdosl	OSL (IBM)
pds/sdpds	PDS (Torczon)
prx/sdprx	PRAXIS (Brent and Chandler)
snp/sdsnp	SNOPT (Gill, Murray and Saunders)
stn/sdstn	Stenmin (Bouaricha)
tao/sdtao	TAO (Benson, Curfman McInnes, Moré and Sarich)
ten/sdten	Tenmin (Schnabel and Chow)
trn/sdtrn	TRON (Lin and Moré)
unc/sdunc	Uncmin (Koontz, Schnable and Weiss)
va15/sdva15	VA15 (Nocedal)
ve09/sdve09	VE09 (Gould)
ve12/sdve12	HSL_VE12 (Gould)
ve14/sdve14	VE14 (Gould)
vf13/sdvf13	VF13 (Powell)

Table 1.4: Interfaces between the CUTEr tools and existing optimization and linear algebra packages as of March 24, 2005.

Chapter 2

CUTE log

In this chapter, we kept track of all the changes that CUTE has undergone since the first release.

2.1 CUTE 1.0

This is the first version of CUTE, made available in March 1993.

2.1.1 Updates since March 93

Additional interfaces

COBYLA This package is a direct search method for inequality constrained problems, that models the objective and constraint function by linear interpolation and does not use derivatives. It is available from Professor M.J.D. Powell, DAMTP, Cambridge University, Cambridge, UK (e-mail address: mjdp@damtp.cambridge.ac.uk).

TENMIN This package is intended for problems where the cost of storing one n by n matrix (where n is the number of variables), and factoring it at each iteration, is acceptable. The software allows the user to choose between a tensor method for unconstrained optimization, and an analogous standard method based upon a quadratic model. The tensor method bases each iteration upon a specially constructed fourth-order model of the objective function that is not significantly more expensive to form, store, or solve than the standard quadratic model. TENMIN is available via anonymous ftp from ftp.cs.colorado.edu, in the directory pub/cs/distribs/tensor. Any questions about this software should be addressed to: eskow@cs.colorado.edu

The interface includes the scripts sdten.* and ten.*, the driver tenma.f, and the file README.tenmin. The driver was originally written by Ali Bouaricha, of CERFACS, Toulouse, France.

NPSOL This package is designed to minimize smooth functions subject to constraints, which may include simple bounds, linear constraints, and smooth nonlinear constraints. The software uses a sequential quadratic programming algorithm, where bounds, linear constraints and nonlinear constraints are treated separately. Unlike MINOS, NPSOL stores all matrices in dense format, and is therefore not intended for large sparse problems. NPSOL is available from the Office of Technology Licensing at Stanford University.

The interface includes the scripts sdnps.* and nps.* the driver npsma.f, an options file NPSOL.SPC, and the file README.npsol. The driver is based on one written by Peihuang Chen of Northwestern University, Chicago, U.S.A.

VA15 This package solves general nonlinear unconstrained problems using a limited memory BFGS method. It is intended for large-scale problems. VA15 is part of the Harwell Subroutine Library, 1993. It is distributed United Kingdom Atomic Energy Authority, Harwell, subject to certain license agreements. It is copyrighted jointly by the UKAEA and SERC (Science and Engineering Research Council).

The interface includes the scripts sdlmq.* and lmq.*, and the driver va15ma.f.

MINOS 5.5 The interface written for MINOS 5.4 works without change for MINOS 5.5.

Changes to interfaces

MINOS There are now different default MINOS specifications for each size of CUTE installation (small, medium, and large). In the CUTE distribution, these different specifications files are named MINOS.sml, MINOS.med, and MINOS.lrg. The unwrap procedure copies all three of these files to the $CUTEDIR/minos directory. The install procedure then copies the specifications file of the appropriate size to MINOS.SPC.

MATLAB/CUTE In addition we have also included an interface which allows the CUTE evaluation tools to be called from MATLAB

Additional platforms

DEC OSF/1 Note that OSL is not available on this platform. (All other optimization packages and CUTE programs are available.)

DEC VMS (using g-floating double precision)

Note that OSL is not available on this platform. (All other optimization packages and CUTE programs are available.)

DOS using WATCOM Fortran compiler

Note that OSL is not available on this platform. (All other optimization packages and CUTE programs are available.)

HP-UX All optimization packages are available on this platform.

Additional tools Two new constrained tools were added to CUTE in October 1994. These tools compute the function value and possibly the gradient of a single constraint. One tool, ccifg, stores the constraint gradient in dense format, while the other tool, cscifg, stores it in sparse format.

In December 1994, we added second derivatives available as a sparse matrix stored in "finite-element" format This added five new corresponding tools asmbe.f, ceh.f, csgreh.f, ueh.f and ugreh.f .

Changes to tools In October 1994, the constrained tools were updated to make them more efficient for unconstrained problems. On most unconstrained problems, these changes will make a small (not dramatic) difference in solution time.

Changes to scripts Linking compiled, library versions of BLAS (Basic Linear Algebra Subprograms)

The linking of compiled, library versions of the BLAS is now permitted by all scripts which use the BLAS (bqp.*, cns.*, lmq.*, mns.*, nps.*, osl.*, qp.* ten.*, and unc.*), and the generic script gen.*, EXCEPT for the .vax scripts.

If there are library versions of the level-1 BLAS available, the variable BLAS in these scripts should be set to a list of names of the object library suffix -lx, where the object library libx.a contains the relevant BLAS. For example, if the BLAS are shared between object libraries libblas1.a and libblas2.a, the variable BLAS should be set to ”-lblas1 -lblas2”, noting that those subprograms in libblas1.a will take precedence over those in libblas2.a.

If compiled BLAS are not available, the variable BLAS should be set to ””. (This is the default setting.) In this case, the link statement includes linpac.o, which is the compiled object for the Fortran source file linpac.f provided in the CUTE distribution.

Linking compiled, library versions of HSL (Harwell Subroutine Library) The linking of compiled, library versions of HSL is now permitted by all scripts which link subroutines from HSL (bqp.*, cns.*, lmq.*, and qp.*), EXCEPT for the .vax scripts.

If there is a library version of the HSL available, the variable HSL in these scripts should be set to -lx, where the object library libx.a contains the relevant HSL. For example, if HSL is contained in the object library libhsl.a, the variable HSL should be set to ”-lhsl”.

If a compiled version of HSL is not available, the variable HSL should be set to ””. (This is the default setting.) In this case, the link statement includes the name of the appropriate object file for the optimization package in question. For example, if the variable HSL is set to ”” in bqp.*, the link statement includes $CUTEDIR/qp/ve14s.o for single precision, or $CUTEDIR/qp/ve14d.o for double precision. (The names of the appropriate object files are given in the section entitled ”Running the scripts available within CUTE” in the file $CUTEDIR/READ.ME. Before installation, this READ.ME is entitled README.mcn, where mcn is the three-letter extension for your platform.)

2.1.2 Bug fixes since November 93

30/Nov/93: gps.f — Correction 1. 3 lines interchanged.

03/Dec/93: mi54ma.f — Correction 10. increased NWCORE for small and medium installations.

03/Dec/93: MINOS.sml, MINOS.med, MINOS.lrg Removed line setting Crash Option to 0. Now Crash Option defaults to 3.

03/Dec/93: README.minos — Added a few lines to indicate that DEC VAX/VMS users should use mi10vms.f, not mi10unix.f, to create the MINOS 5.4 object module.

13/Jan/94: makefn.f, makegr.f — Two lines modified for correcting a format problem in conditional expressions (ELEMENTS and GROUPS sections).

13/Jan/94: classify.osf classall.osf — The scripts are updated to avoid problems with echo.

20/Jan/94: genma.f, gend.f, gens.f — Converted these files to upper case and added a dummy argument in the call to gen.

21/Jan/94: cns.* (formerly con.*), sdcns.* (formerly sdcon.*), paper.tex, README.cry, README.dec, README.install, README.osf, README.rs6, README.sun, README.vax — Renamed con.* to cns.* to allow CUTE to run under DOS. Also renamed sdcon.* to sdcns.* for consistency.

21/Jan/94: select.* — Removed extraneous basic system commands.

21/Jan/94: slct.f — Removed translation of file name FILEN to upper case. This change allows the user to specify a full path name for the .DB file, and thus the .DB file need not necessarily reside in $MASTSIF.

21/Feb/94: ccfg.f, cdh.f, cfn.f, cgr.f, cgrdh.f, cnames.f, cofg.f, cprod.f, cscfg.f, csetup.f, csgr.f, csgrsh.f, csh.f, ubandh.f, udh.f, ufn.f, ugr.f, ugrdh.f, ugrsh.f, unames.f, uofg.f, uprod.f, usetup.f, ush.f, README.tools — Replaced sized array declarations with declarations using parameters. This change means that these arrays can be resized by changing only the parameter value, without changing the array declaration itself.

25/Apr/94: ubandh.f — Declared previously undefined variable NNZH as integer.

25/Apr/94: va15ma.f — Declared previously undefined variables MAXIT, LP, MP, INFO as integer.

02/May/94: README.cry, README.dec, README.install, README.osf, README.rs6, README.sun, README.vax — Removed HS25.SIF from documentation, since it is no longer included in CUTE as a test problem.

04/May/94: _specs, README.tools, README.depend — Renamed former README.tools to README.depend and renamed former _specs to README.tools, to have the names better reflect the contents of these documentation files.

04/May/94: unfold.* — Added line to move README.depend to $CUTEDIR/doc/depend.rdm.

10/May/94: csetup.f — Added OUTPUT common block to SAVE statement.

10/May/94: sd*.*, bqp.*, cns.*, gen.*, lmq.*, mns.*, nps.*, osl.*, qp.*, ten.*, unc.*, except *.vax scripts — Added check for installation of requested precision (single or double), to make the failure more graceful when the user tries to run a precision which has not been installed. If the requested precision is not installed, each script writes an error message and terminates.

11/May/94: select.*, slct.f — select.* scripts now create SLCT.DAT file containing the setting of $MASTSIF, in order that the slct program can give the full path name for the default classification file.

12/May/94: select.* — Removed cd to $MASTSIF since creation of SLCT.DAT means it is no longer necessary to initiate slct from $MASTSIF.

12/May/94: unfold.* — Added line to remove sysdp*.* files. These are system dependent files required to install CUTE on some platforms.

12/May/94: *ma.f, clsf.f, local.f, runsd.f, slct.f — Added machine-dependent lines for WATCOM Fortran installations. All these lines begin with CWFC.

12/May/94: *.wfc —

13/May/94: Added batch files to run CUTE under DOS with WATCOM Fortran compiler.

25/May/94: asmbl.f — Fixed error in the calculation of the Hessian which arose when the same problem variable was assigned to two or more elemental variables.

27/May/94: sd*.vax, bqp.vax, cns.vax, gen.vax, lmq.vax, mns.vax, nps.vax, qp.vax, ten.vax, unc.vax — Added check for installation of requested precision (single or double), to make the failure more graceful when the user tries to run a precision which has not been installed. If the requested precision is not installed, each script writes an error message and terminates.

31/May/94: mi53ma.f, mi54ma.f, npsma.f, oslma.f, vf13ma.f — Replaced UNIX machine-dependent OPEN statements for OUTSDIF.d with generic UNIX OPEN statements.

31/May/94: *.hp — Added scripts to run CUTE on HP9000 workstations under HP-UX.

02/Jun/94: unfold.* — Changed (for the sake of DOS) to handle postfixes in filenames limited to 3 chars.

03/Jun/94: local.f — Added machine-dependent lines for HP installations. All these lines begin with CHP.

12/Jul/94: local.f — Added machine-dependent lines for Silicon Graphics installations. All these lines begin with CSGI.

18/Jul/94: initw.f —

25/Jul/94: gen.*, mns.*, nps.*, ten.*, unc.* — Added check for existence of required object file.

25/Jul/94: bqp.*, cns.*, lmq.*, qp.*, unfold.*, paper.tex, README.cry, README.dec, README.hp, README.install, README.osf, README.rs6, README.sun, README.vax, README.wfc — Reorganized directories for Harwell subroutine executables. Now each Harwell optimization subroutine included in CUTE has its own directory, with the same name as the subroutine (i.e., va15, ve09, ve14, vf13). Users linking compiled objects corresponding to the Harwell subroutines should place these objects in the corresponding directories. Users linking the Harwell subroutine library are unaffected by this change.

26/Jul/94: instll.* — Added check that unwrap has taken place before execution of instll procedure.

26/Jul/94: gen.vax — Removed erroneous blank in line setting ctools.

26/Jul/94: classall.cry, classall.dec, classall.hp, classall.osf, classall.rs6, classall.sun, classall.vax — Added check for existence of CLASSF.DB before removing it.

26/Jul/94: classall.wfc — Added line to type final classf.db file.

26/Jul/94: classify.wfc — Replaced block of lines to prevent failure when the specified directory is the current one.

26/Jul/94: sysdp1.wfc (renamed by unfold.wfc to classone.wfc) — Replaced block of lines to prevent failure when classf.udb does not exist.

26/Jul/94: classify.cry, classify.dec, classify.hp, classify.osf, classify.rs6, classify.sun, classify.vax — Added check for existence of CLASSF.UDB before moving it to CLASSF.DB.

27/Jul/94: sdgen.vax — Replaced 'purge' with 'purge/nolog' on three lines.

27/Jul/94: slct.f — Changed matching for fixed number of variables or constraints. A variable number ('V' in the classification string) is no longer considered to match a fixed number. Also fixed the initialization of FILEN for non-Unix platforms.

27/Jul/94: tenma.f, uncma.f, va15ma.f, ve09ma.f, ve14ma.f — Deleted CIBM lines, since CUTE does not support installations under VM/CMS.

27/Jul/94: README.install — Added CWFC and CHP to keywords table. Also explained presence of CIBM in local.f and runsd.f.

02/Aug/94: slct.f — Changed matching for number of variables or constraints in an interval. A variable number ('V' in the classification string) is no longer considered to match a number in an interval.

05/Aug/94: osl.* — Added check for existence of executable after link and load statement. If the executable does not exist, the error message reminds the user to ensure that FLIBS points to the Optimization Subroutine Library.

05/Aug/94: bqp.*, cns.*, lmq.*, qp.* — Added check for existence of required object file. If the object file does not exist, the error message states that either the object file must be placed in the appropriate directory, or HSL must point to the user's Harwell Subroutine Library.

08/Aug/94: README.install — Made changes to reflect recent changes to CUTE package.

23/Aug/94: MINOS.lrg, MINOS.med, MINOS.sml Replaced line setting 'Superbasics Limit' with line setting 'Hessian Dimension'.

24/Aug/94: README.*, maketo.*, mns.*, sdmns.*, unfold.* — Changed mi54*.* to minos*.*, since the scripts and tools for MINOS 5.4 work without modification for MINOS 5.5, and these are now the standard versions of MINOS. Also explicitly added MINOS 5.5 to README.minos.

12/Sep/94: osl.cry, osl.hp, osl.rs6, osl.sun — Moved stanza setting FLIBS to follow stanza setting BLAS, and expanded comment in this stanza. Also deleted space between '-l' and '$FLIBS' in link commands.

14/Sep/94: ccfg.f — Fixed bug in Jacobian calcuation for groups with only linear elements.

15/Sep/94: cofg.f, ccfg.f, cscfg.f — Removed incorrect storage of nonzero entries in FUVALS(LGRJAC) and updating of indices in IWK(LSTAJC). This error did not affect the output of these routines, but would affect other routines using these arrays. Also removed setting of FIRSTG to .FALSE.

04/Oct/94: ccifg.f, cscifg.f — Added new tools to evaluate the function and possibly the gradient of a single constraint, in both dense and sparse formats.

04/Oct/94: README.*, gen.*, maketo.*, unfold.* — Added ccifg.f and cscifg.f as appropriate.

05/Oct/94: csetup.f, usetup.f — Rearranged variable declarations to separate common and local variables.

05/Oct/94: ccfg.f, cdh.f, cfn.f, cgr.f, cgrdh.f, cnames.f, cofg.f, cprod.f, cscfg.f, csetup.f, csgr.f, csgrsh.f, csh.f — Changed constrained tools to make them more efficient for unconstrained problems.

05/Jan/95: SAMPLE.SIF — renamed sifcmd.lst to avoid the confusion with SIF files describing actual problems. Suitable modifications in the README.* and in unfold.*.

2.2 CUTE version 2.0

This version of CUTE corresponds to the paper published in TOMS. It incorporates all changes, corrections and updates described above for CUTE 1.0 and updates.

2.2.1 Updates since January 1995

06/01/95: Output printing improved for vf13ma.f

22/01/95: Output printing improved for cobma.f

20/08/98: Additional tools cidh.f and cish.f, which compute the Hessians of individual problem functions (objective or constraints) in dense and sparse formats respectively, added.

17/05/99: Output printing improved for ush.f, ugrsh.f, csh.f, cish.f and csgrsh.f

25/08/99: redundant format statements removed from usetup.f, csetup.f, vf13ma.f, minosma.f, oslma.f, cobma.f and uncmai.f

25/08/99: tabs removed from cofg.f, ccfg.f and cscfg.f

2.2.2 Bug fixes since January 1995

15/08/95: Error message and output format improvements in makefn.f and makegr.f

24/10/95: Dummy array dimension corrected in call to SETVL in asmbe.f

20/03/96: Order of two statements changed in asmbl.f and asmbe.f

16/01/97: KA properly initialized in minosma.f and mi53ma.f

06/02/97: IOBJ properly initialized in minosma.f and mi53ma.f

25/04/97: checks for space allocation fro the Jacobian in csgr.f and minosma.f

19/08/97: The Hessian sparsity pattern no longer depends on the values of the problem unknowns and Lagrange multipliers, but just on the structure of the problem. This implies that, for certain argument values, zero entries will occur where previously there would have been no entry. The advantage of a fixed pattern is that this simplifies the job for users of sparse-matrix solvers which often presume that this is the case.

27/08/99: Length of arrays IPRNHI and IPRHI properly checked in asmbe.f, ueh.f, ugreh.f, ceh.f and csgreh.f

2.3 CUTE version 2.99999

This version adds a number of new tools in anticipation of CUTEr, which is due for release in 2001. It incorporates all changes, corrections and updates described above for CUTE 2.0 and updates.

2.3.1 Major additions

New routines

- UDIMEN, UDIMSH, UDIMSE, CDIMEN, CDIMSH, CDIMSE, CDIMSJ to determine appropriate array dimensions in advance
- UVARTY, CVARTY to detect integer/zero-one variables

SIF Extension to SIF format to allow users to specify explicit quadratic terms for the objective function; these extensions have been made by others to the MPS format to handle quadratic programs.

21/02/00: Now a non-fatal return when the SIF file is missing or incomplete. Also, added OSL-like alias QSECTION for QUADRATIC card.

06/03/00: Increased integer formats from I6 to I8. Increased filename format in slct.f from 39 to 256. Improved workspace partitions. Fixed bug introduced re: added OSL-like alias QSECTION on 21/02/00.

07/09/00: Checks added to ensure that the range transormation is ”useful”.

Chapter 3

Future versions of CUTEr

3.1 Future features

- GUI,
- Have all the memory allocated in one place at the beginning. This would require further versions (like CUTEst) to be written in Fortran95, Fortran2000, or similar,
- AMPL to SIF converter (maybe not)
- GAMS to SIF converter (even less likely)
- C interfaces (aaargh),
- Support for Windows (double aaargh)

Chapter 4

License

SOFTWARE LICENSE AGREEMENT NOTICE - THIS SOFTWARE IS BEING PROVIDED TO YOU BY CERFACS UNDER THE FOLLOWING LICENSE. BY DOWN-LOADING, INSTALLING AND/OR USING THE SOFTWARE YOU AGREE THAT YOU HAVE READ, UNDERSTOOD AND WILL COMPLY WITH THESE FOLLOWING TERMS AND CONDITIONS.

1. This software program provided in source code format (the ”Source Code”) and any associated documentation (the ”Documentation”) are licensed, not sold, to you.

2. CCLRC, CERFACS and FUNDP grant you a personal, non-exclusive, non-transferable and royalty-free right to use, copy or modify the Source Code and Documentation, provided that you agree to comply with the terms and restrictions of this agreement. You may modify the Source Code and Documentation to make source code derivative works, object code derivative works and/or documentation derivative works (called ”Derivative Works”). The Source Code, Documentation and Derivative Works (called ”Licensed Software”) may be used by you for personal and non-commercial use only. ”non-commercial use” means uses that are not or will not result in the sale, lease or rental of the Licensed Software and/or the use of the Licensed Software in any commercial product or service. CCLRC, CERFACS and FUNDP reserve all rights not expressly granted to you. No other licenses are granted or implied.

3. The Source Code and Documentation are and will remain the sole property of CCLRC, CERFACS and FUNDP. The Source Code and Documentation are copyrighted works. You agree to treat any modification or derivative work of the Licensed Software as if it were part of the Licensed Software itself. In return for this license, you grant CCLRC, CERFACS and FUNDP a non-exclusive perpetual paid-up royalty-free license to make, sell, have made, copy, distribute and make derivative works of any modification or derivative work you make of the Licensed Software.

4. The licensee shall acknowledge the contribution of the Source Code in any publication of material dependent upon the use of the Source Code. The licensee shall use reasonable endeavours to send to CCLRC, CERFACS and FUNDP a copy of each such publication.
   For CCLRC, contact n.gould@rl.ac.uk, for CERFACS, contact orban@cerfacs.fr and for FUNDP, contact Philippe.Toint@fundp.ac.be.

5. CCLRC, CERFACS and FUNDP have no obligation to support the Licensed Software it is providing under this license.

   THE LICENSED SOFTWARE IS PROVIDED ”AS IS” AND CCLRC, CERFACS and FUNDP MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CERFACS MAKE NO REPRESENTATIONS OR WARRANTIES OF MERCHANTIBILY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. CCLRC, CERFACS, FUNDP AND THE AUTHORS OF THE LICENSED SOFTWARE WILL NOT BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, OR SPECIAL DAMAGES, OR ANY OTHER RELIEF, OR FOR ANY CLAIM BY ANY THIRD PARTY, ARISING FROM YOUR USE OF THE LICENSED SOFTWARE.

6. This license is effective until terminated. You may terminate this license at any time by destroying the Licensed Software.

Acknowledgements

Phil Gill for Snopt, Jorge Nocedal and Richard Waltz for KNITRO. Sven Leyffer for the interface to FilterSQP.

Appendix

Environment variables

The environment variables described in Table 4.1 are vital to CUTEr. Refer to your local documentation or system administrator for more information on how to set these environment variables.

Name	Purpose
`CUTER`	Location of the source of the CUTEr package;
`MYCUTER`	Location of the local instance of CUTEr;
`SIFDEC`	Location of the source of the SifDec package;
`MYSIFDEC`	Location of the local instance of SifDec;
`MASTSIF`	Location of the local collection of SIF problems;

Table 4.1: Environment variables vital to CUTEr.