LIBOPT – An environment for testing solvers on heterogeneous collections of problems

J. Charles GILBERT
INRIA Rocquencourt
and
Xavier JONSSON
Mentor Graphics

The Libopt environment is both a methodology and a set of tools that can be used for testing, comparing, and profiling solvers on problems belonging to various collections. These collections can be heterogeneous in the sense that their problems can have common features that differ from one collection to the other. Libopt brings a unified view on this composite world by offering, for example, the possibility to run any solver on any problem compatible with it, using the same Unix/Linux command. The environment also provides tools for comparing the results obtained by solvers on a specified set of problems. Most of the scripts going with the Libopt environment have been written in Perl.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; G.1.m [**Numerical Analysis**]: Miscellaneous; G.4 [**Mathematical Software**]: Efficiency

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: benchmarking, collection of problems, CUTEr, Libopt, Modulopt, optimization, performance profile, scientific computing, solver comparison, testing environment

1. INTRODUCTION

Two of the issues that come up with software development in scientific computing have to do with benchmarking and profiling solvers on some (possibly large) collections of problems. For example, in the optimization community, these issues are frequently dealt with the *CUTEr testing environment* [Bongartz et al. 1995; Gould et al. 2003]. This one supplies ready-to-use interfaces between some known solvers and a collection of problems encoded with the SIF language [Conn et al. 2003]. It is intended to help developers to test and improve their optimization solver. The Libopt environment has been designed for a similar purpose but, unlike CUTEr, it

Author's address: J. Charles Gilbert, INRIA-Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France; e-mail: `Jean-Charles.Gilbert@inria.fr`.
Xavier Jonsson, Mentor Graphics (Ireland) Ltd. - French Branch; 180, Avenue de l'Europe - Zirst Montbonnot; F-38334 Saint Ismier Cedex; e-mail: `Xavier_Jonsson@mentorg.com`.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

provides neither collections of problems written in a specific format, nor decoders for converting problems written in some language into FORTRAN or C; on the other hand, it is not restricted to optimization. Rather, Libopt has been thought up for organizing and using problems coming from heterogeneous sources. Heterogeneity refers, in particular, to the variety of languages used to write solvers and problems. As a result, CUTEr is considered by Libopt as a particular collection of problems having its own features and solver-collection interface.

Originally, we had in mind to make the problems of the *Modulopt collection* [Lemaréchal 1980] as easily available as the CUTEr problems, despite the diversity of their encoding. The Modulopt collection is formed of problems coming from industrial or scientific computing sources. Because of their complexity, these codes are often not modeled on the academic standard of idealized problems, so that it would have been difficult and boring to rewrite them in SIF. This state of affairs motivated the development of the Libopt environment, which resulted in a layer covering various collections of problems (including CUTEr and Modulopt) and solvers, organizing and normalizing the communication between them. As a consequence, the Libopt environment also offers to the solver developers who are not familiar with a particular language, the possibility to define their own collection of problems, using their preferred (modeling) language, and to run these problems by using the same set of commands as those that can now run the CUTEr problems.

In addition to its solver-collection setting, Libopt also provides a number of programs, mainly Perl scripts, that perform repetitive tasks, such as collecting the results of solvers on problems and comparing these. The features of the solvers and collections are actually encoded in these programs, some of them having to be written for each solver-collection pair. Once these are available, all the results are obtained and compared using the same Unix/Linux commands. This is one of the advantages of the proposed approach. Now, Libopt is an open structure, which can grow, depending on the needs of its users, by incorporating more solvers, problems, or even collections of problems. A large part of a *companion manual* [Gilbert and Jonsson 2007] describes this aspect of the software.

The suffix "opt" of the environment name reflects the fact that Libopt was first introduced to deal with optimization solvers and problems. However, the concepts implemented in this software are sufficiently general for being suitable to other scientific computing fields or even to a larger domain in which the comparison of codes can be based on quantifiable measurements. The tools have been designed with this idea of generality in mind.

The paper is organized as follows. The guided tour of section 2 invites the reader to discover the main aspects of the software. This view of Libopt, from 10,000 feet, essentially presents the tools that are intended to be routinely employed. Section 3 goes a little deeper in the Libopt machinery. Its goal is to shed some light on the techniques used to implement the Libopt commands. As a representative example, the `libopt run` command, used to run solvers on problems, is decorticated to show how a high level solver-problem-independent command may be decomposed in sub-scripts more and more attached to solvers and collections. The reasons for preceding like this are also discussed. This paper is rather short since we already give some words of conclusion in section 4. Now, many more details are given in the companion manual [Gilbert and Jonsson 2007], in particular those that are too technical to be presented here or that might change with the evolution of the software.

The current address of the Libopt site is

```
http://www-rocq.inria.fr/estime/modulopt/libopt/
```

Notation. We use the following typographic conventions. The `typewriter font` is used for a text that has to be typed literally and for the name of files and directories that exist as such (without making substitutions). In the same circumstances, a generic word, which has to be substituted by an actual word depending on the context, is written in *`italic typewriter font`*.

Throughout this paper, we assume that the operating system is Unix or Linux. The *system prompts* are denoted by the characters '`%`' and '`?`' (the latter for multiple lines) and are used to distinguish command lines from other indications. Optional arguments in a command line are surrounded by the brackets '`[`' and '`]`'.

2. A GUIDED TOUR

The Libopt environment has been designed to make easier and faster the repetitive tasks linked to testing, comparing, and profiling solvers on problems coming from heterogeneous collections of problems. These tasks can be divided into three stages: running solvers on problems, gathering the results, and comparing them. Libopt has three *subcommands* associated with these stages: `run`, `base`, and `profile`. The corresponding Unix/Linux command line has the form

```
% libopt subcommand ...
```

where *`subcommand`* can be `run`, `base`, or `profile` (or many other subcommands described in the companion manual), and the dots stand for the subcommand arguments. This section offers an introduction to these subcommands. But before this, let us start by defining terms that are continually used in this paper.

2.1 A few definitions

The *Libopt hierarchy* is the set of directories (and files) that form the skeleton (and material aspect) of the Libopt environment. The directory from which the Libopt commands are launched is called below the *working directory*. When this is important, the commands take care that this directory is not in the Libopt hierarchy. If this were the case, there would be a danger of incurable destruction. Indeed, the scripts launched by the `libopt run` command generally removes several files from the working directory after a problem has been solved.

A *solver* is a computer program that can find the solution to some classes of problems. Hem! Not sure this is very explicit, but we shall not try to be more precise; in particular, we shall take the notion of *problem* as a concept, not requiring any definition.

A *collection* is a set of problems in an arbitrary scientific computing area. Problems in the same collection must differ by their *name*, but two problems belonging to two different collections may have the same name (luckily, since usually collection designers do not communicate and may well choose the same name for two different problems). There are good reasons for gathering problems having common features in the same collection. For instance, problems in a collection are often written in the same language (Fortran, C, Matlab [Mathworks], Scilab [Scilab 2007], Gams [Brooke et al. 1988], Ampl [Fourer et al. 2003], SIF [Conn et al. 2003], to mention a few). The motive is that, in the Libopt environment, all the problems of a collection are dealt with the same scripts and that these scripts are easier to conceive if the problems are written in the same language. For the same reason, problems in a collection usually belong to the same scientific computing domain (optimization, linear algebra, differential equations, etc). Another property that is usually shared by all the problems of a collection is the extent to which they are protected against dissemination; this is useful for determining the public to which the collection can be distributed.

A *subcollection* is a subset of problems belonging to the same collection. A collection may contain several subcollections, and a problem may belong to more than one subcollection. The reason why the notion of subcollection is introduced is clear: some solvers can sometimes only solve a part of the problems of a given collection and it is useful to be able to designate them. For example, in optimization, there is some advantages in distinguishing the subcollections of unconstrained problems, of linear problems, of quadratic problems, of bound constrained problems, etc, since there are solvers that are dedicated to these classes of problems.

2.2 Running solvers with `libopt run`

Libopt can only deal with solvers that are registered in its environment. The procedure to do this is described in the companion manual [Gilbert and Jonsson 2007]. Below, we denote by

solv

the generic name of such a solver. Similarly, Libopt can only consider collections of problems that are installed in its hierarchy. This simply means that some directory (or a symbolic link to it) has to contain the problems of the collection (these can be in an arbitrary format) and some files and scripts have to tell how to use the collection (this is why the format can be arbitrary). Below, we denote by

coll, *subc*, and *prob*

the generic names of a collection, subcollection, and problem, respectively.

The simplest way of running the solver *solv* on the problem *prob* of the collection *coll* in the Libopt environment is by entering

```
% libopt run "solv coll prob"
```

We quote a few advantages to have at our disposal such a command to run solvers on problems.

- First, the form of the command for running any solver on any problem of any collection is invariant: it does not depend on the solver, the collection, or the problem. In our experience, this property saves much memorization effort. In particular, it gives to a collection, which is not used often and is coded in a manner that is difficult to remember, more chance to be tested, even after it has been abandoned for several years.

- Second, it defines a standard, to which solver developers can contribute by providing the interfaces between their solver and various collections.
- Also, the possibility to consider a large diversity of collections should allow the environment to accept problems coming from various sources.

Libopt can deal with collections whose problems may have several data sets. If *prob* in the `libopt run` command above is made of two strings linked by a dot, like in

pnam.pdat,

Libopt considers that the *radical* of the problem name is *pnam* and that this problem has to be solved with a data set identified by the string *pdat*. How the data set is built from this string depends on the collection and its installation. The Modulopt collection [Gilbert 2007a], which stores each of its problems in a different directory, takes advantage of this feature to avoid duplicating a problem directory and its contents for each of its data sets.

A slightly more powerful use of the `libopt run` command is

```
% libopt run "solv coll.subc"
```

where *subc* is a subcollection of the collection *coll*. By this command the solver *solv* is run on all the problems of the subcollection *subc*. The `libopt run` command can also take into account more than a single directive "*solv coll prob*" or even a file listing such directives.

Another interesting way of running a solver on a collection of problems is by entering

```
% libopt run "solv.tag coll"
```

where *tag* is some string tagging the solver name. This feature can be useful to mark the results obtained with a particular version of the solver *solv* or the one corresponding to a particular set of options (see the companion manual for the details and section 2.5 for an example of use).

2.3 Gathering results with `libopt base`

By the `libopt run` command a solver-problem pair writes its output on its usual files, which probably include the Unix/Linux standard output. In order to compare solvers, which is one of the main goals of the Libopt environment, there is no reason to save all these files, which mainly interest the developer of the problem code. In fact, the standard Libopt scripts normally remove these files after having run a solver on a problem. This is because, Libopt has been designed to *compare* the results of various solvers and does not provide any tool to *analyze* them. For this reason, Libopt is interested in the value of various counters that reflect the performance of a solver on a particular problem. In optimization, these counters are often the number of function or derivative evaluations, the CPU time, the precision of the obtained solution, and many others.

In order to be able to make comparisons, Libopt imposes that the results relevant to a comparison be condensed on the standard output in a string of the form

```
libopt%solv%coll%prob%sequence-of-token-number-pairs
```

This one is called the *Libopt line.* It is formed of a sequence of fields separated by the character '`%`'.

- The first field is the string "`libopt`". It is present to make it easy to locate the Libopt line in the standard output (using the command `grep` for example).
- The next three fields give in order the name of the solver (*`solv`*), the name of the collection (*`coll`*), and the name of the problem (*`prob`*), whose relevant results are given in the following fields.

These first four fields are positional, i.e., their order is imposed. This is not the case for the following ones.

- The *`sequence-of-token-number-pairs`* is a string formed of a sequence of token-number pairs, again separated by the character '`%`'. A *token-number pair* is a string of the form

  `token=number`

  The character '`=`' must separate the *token* (an arbitrary string) from the *number* (a string representing a real number). There must be at least two token-number pairs, one of which is used to compare the results (it must be a *performance* token-number pair actually, see section 2.4) and another one must have the form

  `info=`*`number`*

  where the string "`info`" is imposed and a *`number`* equal to `0` means that the solver *`solv`* has successfully solved the problem *`prob`*.

  Note that, since the Libopt line is written by some program provided by the developer of a solver, it is that program that decides whether the solver has successfully solved a given problem; Libopt has no means to take such a decision. In the Modulopt collection, each problem helps the solver to take that decision by specifying criteria that must be satisfied for deciding whether the problem has been solved [Gilbert 2007a].

An example of Libopt line in optimization could be

```
libopt%m1qn3%modulopt%u1mt1%n=1875%nfc=587%nga=587%info=0
```

to mean that the solver `m1qn3` has been run on the problem `u1mt1` of the collection `modulopt`, that this problem has 1875 variables (`n=1875`) and that a solution has been found (`info=0`) using 587 function and gradient evaluations (`nfc=587` and `nga=587`).

The concept of Libopt line, close to the notion of *trace files* used by the PAVER server [Mittelmann and Pruessner 2003], introduces some standardization that could be viewed as an unnecessary constraint. As we shall see below, it plays however a crucial role in avoiding the duplication of results, which would bias the information contained in performance profiles.

Now, the command

```
% libopt run -l "solv coll.subc" > solv_coll_subc.lbt
```

gathers in the file *`solv_coll_subc`*`.lbt` a sequence of Libopt lines describing the behavior of the solver `solv` on the problems of the subcollection `subc` of the collection `coll`. The option `-l` greps indeed the Libopt lines in the standard output.

There may be many files of results like `solv_coll_subc.lbt` and there is some advantages in gathering all the results they contain in a single file. This gathering operation is also a good opportunity to verify that the Libopt lines in the result files are consistent and that there is at most one result for each (solver, collection, problem) triple. This is exactly what the `libopt base` command with the option `-a` does (more generally, the `libopt base` command deals with the operations in connection with the result databases). If one enters

```
% libopt base -a solv_coll_subc.lbt
```

the Libopt lines in the result file are decoded, checked (see below), and stored in a database, whose default name is `libopt_database` in the working directory. This database is no longer an ascii file, but a binary file or a pair of binary files, depending on the operating system. By entering the `libopt base -a` command for all the relevant result files, one can obtain a database containing all the results of interest, without duplicated or contradictory data. The database can be managed, using the options of the `libopt base` command.

The amount of verifications done on the Libopt lines by the `libopt base -a` command depends on the presence and contents of the `~/.liboptrc` startup file. If there is no such file, `libopt base -a` just verifies the conventions mentioned in the beginning of this section. It cannot do more. In particular, it makes no assumption on what the tokens are and on the quantity of token-number pairs. These pieces of information are actually very domain dependent and certainly not identical in optimization, in linear algebra, or in the differential equation domain. However, to avoid typos, you can list the valid tokens in the startup file `~/.liboptrc`. If this file is present, `libopt base -a` will read it, and if it contains the directive `tokens`, the command will verify that the tokens in the Libopt line are among those listed by this directive.

2.4 Comparing results with `libopt profile`

The `libopt profile` command has been designed to make comparisons between solvers on a selected set of problems. The comparison is based on the results stored in a database, one generated by the `libopt base -a` command. A single criterion can be used for making this comparison, and it must be one of the tokens present in the Libopt lines summarizing the results to compare. This comparison made by `libopt profile` produces files that can be subsequently dealt with Matlab or Gnuplot. These files describe the *performance profiles* à la Dolan and Moré [Dolan and Moré 2002] of the solvers.

In the Libopt line, one finds descriptive token-number pairs (or descriptive tokens) and performance ones. The semantics behind this distinction is rather intuitive: a *performance token* is a token that can be used to compare solvers, while a *descriptive token* is a non-performance token. A performance token-number pair must have the following properties:

- the token-number pair must, obviously, depend on the solver (not only on the problem, like the one specifying the number of unknowns),

- the number in the token-number pair must be positive (> 0),
- the number in the token-number pair is *better* (i.e., indicates a better performance) when it is *smaller*.

In the example given on page 6, `n=1875` must be considered as a descriptive token-number pair (the dimension of the problem is not a property of the solver), while `nga=587` is normally a performance one, since an optimization solver can be considered to be better when it requires less derivative evaluations. Libopt cannot verify that these rules have been observed (except for the positivity of the number), but the performance profiles that `libopt profile` generates assume that they hold; they would have no meaning otherwise. On the other hand, if you want Libopt to make some verifications on the correct use of performance tokens, you can list them in the startup file `~/.liboptrc`.

Descriptive tokens are not useless. In addition to giving information on the problems, they can be used for selecting the problems on which a comparison is made. For example, one can be interested in making a comparison of solvers on problems with more (or less) than 1000 variables, and/or those with (or without) inequality constraints, etc. Read the companion manual or the `libopt` manual page to learn how to specify a selection of problems.

The easiest way of using the command `libopt profile` is by entering

```
% libopt profile
```

Then, the script scrutinizes the database (its default name is `libopt_database` in the working directory) and reads the specification file `libopt_profile.spc` in the working directory to know what it has to do. This file is described in detail in the companion manual. It can contain many *directives*. Three of the most important ones are those that specify the solvers that have to be compared, the problems on which they have to be compared, and the performance token chosen for the comparison. For example

```
solver bosch klee durer
collection painting.selfportraiture
performance time
```

is understood by `libopt profile` as a requirement to compare the execution time for the three solvers `bosch`, `klee`, and `durer`, on the problems from the subcollection `selfportraiture` of the collection `painting` (the solver names do not refer to the painters with the same names: painters never work on the same “problems” and comparing their works on their execution time would not be a good idea). Then `libopt profile` generates performance profiles like those in figure 1 (a few additional tunings in the specification file are necessary to get precisely this graph). We refer the reader to the paper by Dolan and Moré [Dolan and Moré 2002] (or to the companion manual) for the meaning of the performance profiles in the figure.

2.5 An example of use

To make the guided tour more concrete, we present an example in which the Libopt environment is used to compare two options of an optimization piece of software. The chosen solver is `m1qn3`, an unconstrained optimization code using the ℓ-BFGS

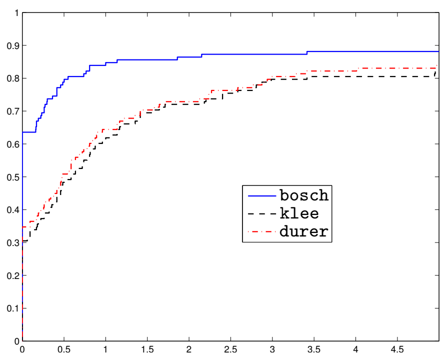

Fig. 1. Typical performance profiles for three solvers; relative performance (abscissa) in $\log_2$ scale

formula [Gilbert and Lemaréchal 1989]. This example will show in the same time the usefulness of tagging a solver name, with the optional string "`.`*tag*" in the `libopt run` command (see the end of section 2.2).

We want to compare the scalar and diagonal running modes of `m1qn3` on 143 unconstrained CUTEr problems (identified by the list `unc.lst` in the `cuter` directory). For this, we start by running `m1qn3` with its default setting (diagonal running mode):

```
% libopt run -l "m1qn3.diag cuter.unc" | libopt base -a
```

Note that the name of the solver has been tagged with the string "`.diag`" to remember the `m1qn3` option used in the run. The effect of this tagging is to generate Libopt lines in which the name "`m1qn3`" of the solver is suffixed with the string "`.diag`". Note also that the option `-l` has been used to grep the libopt lines from the standard output of the `libopt run` command. Then, these lines are piped to the standard input of the `libopt base` command, which stores (option `-a`) their contents in the default database. We assume here that the default database is empty at the beginning of our experiment; otherwise the option `-r` of the `libopt base` command could have been more appropriate to replace possibly already existing and obsolete results. The previous command takes some time to be completed and it does not store the Libopt lines in an intermediate file, which is not particularly cautious. An alternative way of proceeding, in which the Libopt lines are saved in the file `m1qn3diag-cuter-unc.lbt`, could be

```
% libopt run -l "m1qn3.diag cuter.unc" > m1qn3diag-cuter-unc.lbt
% libopt base -a -v m1qn3diag-cuter-unc.lbt
```

The option `-v` has been added in the second command to get more information on the results inserted in the database.

To get the results of the `m1qn3` solver in scalar running mode, one has to modify the specification file

```
$LIBOPT_DIR/solvers/m1qn3/cuter/m1qn3.spc,
```

by replacing the key-value pair "`imode1 0`" by "`imode1 1`". The results with this running mode are then inserted in the database by the commands

```
% libopt run -l "m1qn3.scal cuter.unc" | libopt base -a -v
```

Observe that, this time, the solver name has been tagged with the string "`.scal`" to distinguish the presently generated Libopt lines from those generated by the previous commands. Actually, `libopt run` considers that the solver is still `m1qn3`, while `libopt base` considers that the libopt lines with `m1qn3.diag` are different from those with `m1qn3.scal`, so that the latter will indeed be added to the database.

We are now ready to produce performance profiles, comparing the two running modes of `m1qn3`. Choosing to compare them on the number of gradient evaluations, identified by the token `nga` in the Libopt lines, we adapt the `solver` and `performance` directives of the specification file `libopt_profile.spc` as follows:

```
solver m1qn3.diag m1qn3.scal
performance criterion $nga
gnuplot filetype epsc
```

The `performance criterion` directive expects to find a Perl expression; this is why, above, `nga` has been prefixed by the character '`$`' to make it a Perl variable. The next `gnuplot` directive asks to generate an encapsulated postscript file for color printers, using the Gnuplot plotting software (while Matlab was used to draw the picture in figure 1). Now the command

```
% libopt profile
```

generates the performance profiles shown in figure 2. A reader used to this kind of

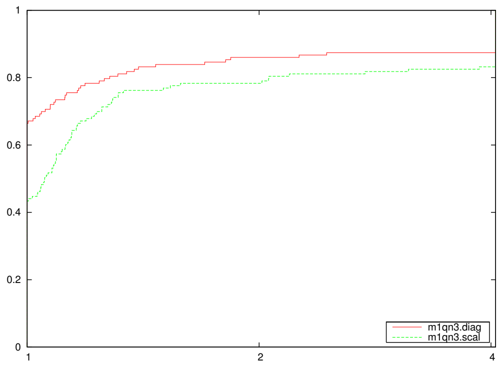

Fig. 2. Performance profiles of the diagonal (`m1qn3.diag`, red top curve) and scalar (`m1qn3.scal`, green bottom curve) running modes of `m1qn3` on 143 unconstrained problems of the CUTEr collection, comparing the number of function evaluations; relative performance (abscissa) in $\log_2$ scale

curves can deduce that the diagonal mode of `m1qn3` can be considered as slightly more efficient than the scalar mode (its profile is higher). This is an average estimation, since the plot with its 2-logarithmic abscissa reveals that, if the scalar mode can require about 4 times more function evaluations than the diagonal mode on some problems, the latter can also require sometimes about 2.5 times more function evaluations than the former on other problems.

Now that the experiment has revealed that the diagonal running mode is usually better than the scalar running mode, it is probably a good idea to switch back the option `imode1` to `0` in the specification file `m1qn3.spc`.

3. BEHIND THE SCENERY

The goal of this section is to shed some light on the techniques used to implement the commands introduced in the guided tour of section 2. We restrict ourselves to the aspects of the software that are unlikely to change with its evolution, letting the companion manual [Gilbert and Jonsson 2007] give an account of the details and the last improvements.

Libopt can be viewed as a rigid empty shell (the core of its directory structure sketched in section 3.1) and a flexible methodology (embodied in its scripts). Some scripts are permanent, independent of the considered problem collection and solver. These are called the Libopt *commands* and cannot be modified by the user. Other scripts, launched by the Libopt commands, depend on the considered problem collection and solver, on the way they are installed in the Libopt hierarchy, on their features. Some of them must be installed at definite places in the Libopt hierarchy. Because of this decomposition of the commands in levels, Libopt can be very permissive on the kinds of problem collection and solver installation, and should be easily adaptable to various situations. This organization in execution levels is encountered in various Libopt subcommands, such as `run`, `addcell`, and `addproblem`. We examine the case of the `run` subcommand in section 3.2; the other ones are considered in the companion manual.

3.1 Structure of the Libopt hierarchy

The Libopt hierarchy is formed of a tree of subdirectories, whose root name is `libopt`, which is identified by the Unix/Linux environment variable

`LIBOPT_DIR`.

This top-level directory contains the directories `bin` (Perl scripts of the Libopt commands), `collections` (problem collections), and `solvers` (solver descriptions and interfaces), as well as other directories and plain files; see figure 3.

```
Level 0      Level 1      Level 2      Description

bin ................................. Libopt scripts
collections
       +--- cuter ................... CUTEr collection
       +--- modulopt ................ Modulopt collection
solvers +--- m1qn3   +--- bin ........ M1qn3 binary directory
       |            +--- cuter ...... Interface M1qn3/CUTEr
       |            +--- modulopt ... Interface M1qn3/Modulopt
       +--- sqplab  +--- cuter ...... Interface Sqplab/CUTEr
```

Fig. 3. Part ot the Libopt hierarchy

The `collections` directory gathers a set of subdirectories, each of them corresponding to an installed collection of problems. In the standard distribution, one finds the collections CUTEr [Gould et al. 2003] in `cuter` and Modulopt [Lemaréchal 1980; Gilbert 2007a] in `modulopt`. More generically, the directory `collections/`*`coll`* contains the description of the collection *`coll`*. What this actually means is reflected in the scripts that use this collection, so that these directories can be organized with a great freedom, as far as Libopt is concerned.

It is natural to put in these collection directories (the subdirectories of `collections`) some *lists of problems*. These are files whose name must have the suffix "`.lst`". Two of these lists are mandatory:

- `all.lst` is the list of all the problems of the collection;
- `default.lst` is a list of a subset of the problems of the collection; this list is chosen by the `run` subcommand when no list is specified as one of its arguments (see section 3.2.2); it is sometimes a symbolic link to the list `all.lst`.

It is also natural to have in `collections/`*`coll`* a subdirectory, usually called `probs`, which contains all the problems of the collection. Of course `probs` can be a symbolic link to the actual problem directory. For example, in our Libopt hierarchy, `collections/cuter/probs` is a symbolic link to a directory containing the gzipped SIF encoding of the CUTEr problems. On the other hand, in the standard distribution, `collections/modulopt/probs` contains a subdirectory for each of the problems of the Modulopt collection, which describes this problem by programs, data, Perl scripts, and a makefile.

The directory `collections` provides no information on the solvers. This information is given by the directory `solvers`, which has a subdirectory for each solver known to the Libopt environment: `m1qn3` and `sqplab` in the standard distribution. We have already encountered the solver `m1qn3` in section 2.5, while `sqplab` is a still-evolving nonlinear optimization solver, written in Matlab, using an SQP approach [Gilbert 2007b].

The directory `solvers` is actually very rich, since, in some sense, it reflects the structure of the Cartesian product

$$\{\texttt{solv\_1}, \ldots, \texttt{solv\_m}\} \times \{\texttt{coll\_1}, \ldots, \texttt{coll\_n}\} \tag{3.1}$$

corresponding to the interfaces between the solvers and collections. An element (*`solv`*, *`coll`*) of this Cartesian product, where *`solv`* is some solver `solv_i` and *`coll`* is some collection `coll_j`, is called a *cell* below. For each cell (*`solv`*, *`coll`*), the directory `solvers/`*`solv`*`/`*`coll`* contains information describing how to run the solver *`solv`* on the problems of the collection *`coll`*. This directory includes the following mandatory files/scripts (see figure 4):

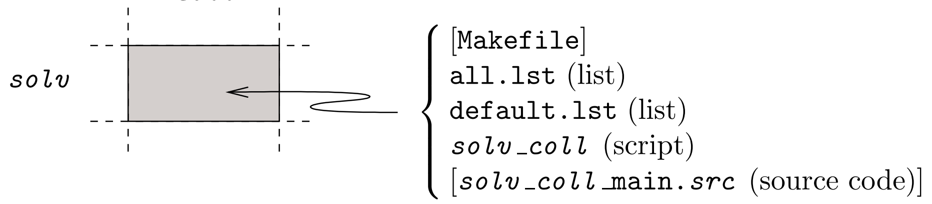

Fig. 4. A single cell of the {solvers} × {collections} Cartesian product

- `all.lst` is the list of all the problems of the collection *coll* that the solver *solv* is structurally able to solve (note that this meaning is quite different from the one of the file `collections/`*coll*`/all.lst` described above); for instance, the file `solvers/m1qn3/cuter/all.lst` is a symbolic link to the list of unconstrained problems of the CUTEr collection;
- `default.lst` lists a subset of the problems in `all.lst`; this list is used by some scripts when no list is specified as one of its arguments (see section 3.2.2);
- *solv*_*coll* is a script that tells how to run the solver *solv* on a problem of the collection *coll*; it essentially takes care of the operating system commands that are required to launch *solv* (see section 3.2.3); note that the name of the script depends on the directory where it is placed.

The directory may also contain other files, like the source code of a main program *solv*_*coll*`_main.`*src* able to run the solver *solv* on a single problem of the collection *coll*, which takes care of the instructions that cannot be written in a Unix/Linux script, and a makefile `Makefile` (see section 3.2.4).

3.2 The `libopt run` command

3.2.1 *Overview.* The `libopt run` command, with its dependent scripts and programs, is probably one of the most complex commands offered by the Libopt environment. It is used to run solvers on problems (see section 2.2 for an introduction). Because of its genericity, this command cannot perform by itself all the details of this operation. Indeed, it cannot know all the features of any possible solver and any possible collection of problems, especially those that are still not installed in the hierarchy! For this reason, the `libopt run` command must delegate part of the operations to other scripts/programs, some of them being written by the persons introducing new solvers and collections into the Libopt environment.

The operation of running solvers on problems is actually decomposed in three levels of scripts/programs.

- The Libopt level.

  The first level is formed of the operations implemented in a script named

  `libopt_run`.

  These operations are those that are independent of any solver and any problem collection. The aim of this level is to analyze the `libopt run` command line and to decompose it in a sequence of elementary operations of the form

  run the solver *solv* on the problem *prob* of the collection *coll*. (3.2)

  There are as many elementary operations as there are combinations of solvers, collections, and problems expressed by the `libopt run` command line. To execute each elementary operation, the `libopt_run` script calls the *solv*_*coll* script of the second level.

  The `libopt_run` script is further described in section 3.2.2.

- The operating-system level.

  The second level is the one that takes in charge the elementary operation (3.2).

It is performed by the programs

`$LIBOPT_DIR/solvers/`*`solv`*`/`*`coll`*`/`*`solv_coll`*

These are actually Perl scripts in the standard distribution. There is a third level since, by choice and for more flexibility and readability, the *`solv_coll`* scripts are restricted to the necessary operations *at the operating system level*. However, unlike the `libopt_run` script, these operations depend here on the solver *`solv`* and the problem collection *`coll`*. Typical operations consist in copying data files in the working directory, running makefiles to get executable programs, launching the main program, and removing data and result files from the working directory on exit from the main program. For some collections of problems (like CUTEr), a part of these operations may also have been delegated to other programs intimately incorporated into the collection.

More is said about the *`solv_coll`* scripts in section 3.2.3.

- The main program level.

  Running a program on a problem does not only depend on operations done at the operating system level. A main program has to be written, which reads the data, calls the solver, and delivers diagnostics. It is at this level that the *Libopt line* is generally written to standard output. This is the part of the Libopt environment that is the most interlinked with the solver and collection structures. As a result, such a program has to be written for each new solver-collection pair.

  This level is further described in section 3.2.4.

3.2.2 *The `libopt_run` script.* The `libopt run` command calls the `libopt_run` script to execute the required operations (both `libopt` and `libopt_run` are located in the `bin` directory of the Libopt hierarchy). We have said that the latter is independent of any solver and problem, so that it does not have to be adapted when the Libopt environment is enriched with new solvers or new collections of problems.

This paper might not the right place to describe in large the `libopt_run` script. However, a rather detailed description of the command should give a good idea of the possibilities of the Libopt environment.

The general form of the `libopt_run` script is the following

```
% libopt_run [-g] [-h] [-k] [-l] [-t] [-v] \
?            [directives] [-f DFile]
```

It is here equivalent to enter '`libopt run`' instead of '`libopt_run`', since the former is transformed into the latter by the `libopt` command.

Let us start by describing the options. If the option `-h` (help) is present, the other arguments of the command are ignored and a short help message is given. This is probably the best way of getting a reminder of any `libopt` subcommand. The option `-v` (verbose) sets the command in verbose mode, which is interesting when one has to debug the installation of a new solver or collection, or the activation of a cell. The option `-t` (test) is similar to the verbose option, except that nothing is executed; the command just tells what it would do without doing it. This is also interesting to check new installations, with an additional degree of safety.

We have already encountered the option `-l` (libopt line), which filters the libopt lines from the standard output. The option `-k` (keep) can be used to prevent the command from removing files once they are no longer useful. In that case, all the files generated during the run remain in the working directory. With the option `-g` (debug), the final executable program that can solve a given problem with a given solver is not executed but saved in the working directory with symbolic debug information to allow debugging.

If *`directives`* is present, it must be formed of one or more *directives*, separated by blanks. Each directive must be surrounded by quotes (`"`) and must be a string of the form

```
solv[.tag] coll[.subc] [list-of-problems]
```

where *`solv`* is the name of a solver installed in the Libopt hierarchy, *`tag`* is an optional string that will tag the solver name in the Libopt line (the usefulness of this option was illustrated in section 2.5 and is futher discussed in the companion manual), *`coll`* is the name of a collection installed in the Libopt hierarchy, *`subc`* is the name of a subcollection of the collection *`coll`*, and *`list-of-problems`* is a list of strings separated by blank characters, which selects some problems from the considered (sub)collection. If the option '`-f` *`DFile`*' is present, each line of the file *`DFile`* must be formed of at most one directive of the form above. Then the `libopt run` command considers the directives in *`directives`* and those in the file *`DFile`* in sequence. For each of them, it runs the solver *`solv`* on each of the problems of the collection *`coll`* from the *`list-of-problems`*.

For each directive, the list of problems that are passed over to *`solv`* depends on the presence of the strings *`subc`* and *`list-of-problems`* in the command line. In addition, the subcollection *`subc`* may refer to lists of problems with the same name in one of the directories

```
$LIBOPT_DIR/collections/coll
$LIBOPT_DIR/solvers/solv/coll,
```

or in the working directory. To determine the final list of problems to solve, the `run` subcommand follows a number of rules with a certain logic, in which the `all.lst` and `default.lst` lists, with their directory-dependent meaning (see section 3.1), play a key role. These rules are explained in the companion manual.

3.2.3 *The `solv_coll` scripts.* A *`solv_coll`* script must exist for each solver *`solv`* that has been installed in the Libopt environment to run problems from the collection *`coll`*. It must be placed in the directory

```
$LIBOPT_DIR/solvers/solv/coll
```

and is launched by the `libopt_run` script discussed in section 3.2.2. It must accept the following command line structure

```
% solv_coll [-g] [-k] [-l] [-t] [-v] prob
```

The command options, inherited from the `libopt run` command that launches *`solv_coll`*, have been described in section 3.2.2. It is the `libopt run` command that determines the problem *`prob`* to solve.

As discussed in section 3.2.1, the `solv_coll` script is in charge of the operations, at the operating system level, for running the solver `solv` on the problem `prob` of the collection `coll`. Let us describe the tasks that the script has to perform.

1. It must build in the working directory the executable program, say `solv_coll_main` (a binary code that will solve the problem `prob` using the solver `solv`), and must construct into the working directory, from information stored somewhere in the directory `$LIBOPT_DIR/collections/coll`, the data files that are useful for running `solv_coll_main`.

   These operations strongly depend on the installation of the collection `coll`. We consider two cases.

   - For the Modulopt collection (see [Gilbert 2007a]), a problem name `prob` may have the form

     `pnam[.pdat]`,

     where `pnam` is the *radical* of the problem name, referencing to a directory name, and `pdat` is an optional string, referencing to a data set. For this collection, the operations mentioned above are encoded first in the Perl script

     `$LIBOPT_DIR/collections/modulopt/probs/pnam/Makebin`,

     which takes care of the data selection or construction, and next as appropriate targets in the makefiles

     ```
     $LIBOPT_DIR/collections/modulopt/probs/pnam/Makefile
     $LIBOPT_DIR/solvers/solv/modulopt/Makefile.
     ```

     The target `pnam` in the first makefile builds an archive with the procedures describing the problem and the target `solv_modulopt_main` in the second one takes care of the solver dependent procedures and of the main program.

   - For the CUTEr collection, this first task could have been skipped, since the CUTEr command `sdsolv` (or something similar) takes it in charge. However, for allowing to store the problems in a compressed format, a symbolic link to the file `prob.SIF[.gz]` is created in the working directory and decompressed there if necessary.

2. It must run the executable program `solv_coll_main` (or the script `sdsolv` for the CUTEr collection) in the working directory.

3. It must remove from the working directory the now useless executable program `solv_coll_main` (not necessary for the CUTEr collection), the data files, and the output files that have been generated during the execution of the program. This task also depends on the collection `coll` and the solver `solv`. For the CUTEr collection, part of the task is taken in charge by the command `sdsolv` itself.

Writing a `solv_coll` script is not an easy task. Luckily, this one can often be adapted from a similar script existing in the Libopt hierarchy. For example, all the `solv_modulopt` scripts only differ by a few character strings, at such a point that it can actually be generated by the command activating the cell (`solv`, `modulopt`), namely

```
% libopt addcell -s solv -c modulopt
```

using the template

```
$LIBOPT_DIR/collections/modulopt/templates/SOLV_modulopt
```

and the generating script

```
$LIBOPT_DIR/collections/modulopt/bin/libopt_addcell_modulopt.
```

This quasi-independence of the *solv_coll* scripts from the solver *solv* looks like a general rule. It is therefore the charge of the designer of the collection *coll* to provide a `SOLV_`*coll* template and a generating script `libopt_addcell_`*coll*. If these are not found in the appropriate directories, the *solv_coll* script is not generated by the `addcell` subcommand and must be written by hand.

3.2.4 *The main programs.* This is the main program that would have to be written if one had to run the solver *solv* on a particular problem *prob* of the collection *coll*. The required genericity of this main program (it has to be able to consider *any* problem *prob* of the collection) can be obtained in the following way. All the problems of the collection are described by procedures with a name that depends on its function but *is identical for all the problems.* Typically, one finds

- *initialization procedures*, which specify the dimensions of the problem, possibly its name, initialize the variables, read the data, etc,
- a *simulator*, which gives the state of the system to solve at a given iterate when the problem is nonlinear and the solver has an iterative nature,
- *auxiliary procedures*, which precise the way some objects are computed (for example, the procedure that performs the inner product used for computing a gradient),
- and possibly *post-solution procedures*, which can do some post-solution computations.

The connection between the main program and the problem is then made by the linker, to which the object files associated with the main program and the procedures describing the selected problem are given by the *solv_coll* script described in section 3.2.3.

It is normally the main program that prints the *Libopt line* on the standard output. It can do this after the problem has been solved by collecting the various features of the problem and the various counters that depict the behavior of the solver *solv* on the problem *prob*.

4. DISCUSSION AND PROSPECTIVE

In this short paper, we have presented the Libopt environment and have decorticated one of its main commands. The software has been designed to make more pleasant the boring, but essential, task of solver developers that consists in routinely running solvers on problems and comparing their results using performance profiles. Now, adding solvers and collections to the environment and activating (solver, collection) pairs are still delicate operations, despite our effort to automate them. Even though these do not occur often, we expect to make them more userfriendly in the future, when we will have acquired more experience with a larger variety of solvers and collections.

The Libopt software would certainly deserve other improvements: we think, for instance, to the introduction of modularity in the Perl scripts, to the notion of platform (see the companion manual), to the efficiency of the *solv_coll* script generation from templates (see section 3.2.3), to the normalization of the tokens in the Libopt line for a specific scientific computing domain (see section 2.3), to the development of a Web interface with the environment (either for using it [Mittelmann and Pruessner 2003] or for downloading part of it), to the implementation of other measures of performance [Mittelmann and Pruessner 2003], to the uniformization of the solver stopping criteria (see [Dolan et al. 2006] and the Examiner tool in [GAMS 2003]), and to the many other subjects that today are out of our imagination. Another breach of the current version of Libopt is that it does not provide tools, or even concepts, for avoiding to consider as different a problem that would be present in more than one collection (for example the Hock and Schittkowski problems [Hock and Schittkowski 1981] are present in the CUTEr and Schittkowski [Schittkowski 2002] collections). Clearly, such a duplication biases the performance profiles. In the same vein, it would be interesting to be able to qualify what is a good selection of problems, since taking in such a selection many problems with similar features would also degrade the relevance of performance profiles based on it.

Only very few collections and solvers are incorporated in the standard distribution of the current version of the software. Installing new collections (like [COPS], to mention another parameterizable collection in nonlinear optimization) and solvers (they are numerous) in a consistent way, maintaining the compatibility with the possible evolution of Libopt, may require some expertise and obstinacy. Note, indeed, that a change in the Libopt software may affect all the cells of the $\{\text{solvers}\} \times \{\text{collections}\}$ Cartesian product. We intent to provide these new installations in the future versions of Libopt, at least in the nonlinear optimization domain, taking advantage of the possible contributions of generous users.

ACKNOWLEDGMENTS

We thank Jorge Nocedal for having offered the second author hospitality at Northwestern University (Evanston, USA) when this work was initiated and François Clément for his careful reading of the manuscript. Many thanks also to two referees, whose remarks lead us to propose a largely rethought version of the environment.

REFERENCES

Bongartz, I., Conn, A., Gould, N., and Toint, P. 1995. CUTE: Constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software 21*, 123–160.

Brooke, A., Kendrick, D., and Meeraus, A. 1988. *GAMS: A User's Guide*. The Scientific Press, San Francisco, CA, USA.

Conn, A., Gould, N., Orban, D., and Toint, P. 2003. The SIF Reference Report (revised version). Online report.

COPS. `http://www-unix.mcs.anl.gov/~more/cops/`.

Dolan, E. and Moré, J. 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming 91*, 201–213.

Dolan, E., Moré, J., and Munson, T. 2006. Optimality measures for performance profiles. *SIAM Journal on Optimization 16*, 891–909.

Fourer, R., Gay, D., and Kernighan, B. 2003. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, Brooks/Cole-Thomson Publishing Company, Pacific Grove, CA, USA.

GAMS. 2003. *The Solver Manuals.* GAMS Development Corporation, Washington, DC, USA.

Gilbert, J. 2007a. Organization of the Modulopt collection of optimization problems in the Libopt environment. Technical Report 329, INRIA, BP 105, 78153 Le Chesnay, France.

Gilbert, J. 2007b. SQPlab – A Matlab software for solving nonlinear optimization problems and optimal control problems – Version 0.4. Online Paper.

Gilbert, J. and Jonsson, X. 2007. LIBOPT – An environment for testing solvers on heterogeneous collections of problems – The manual, version 2.0. Technical Report 331 (revised), INRIA, BP 105, 78153 Le Chesnay, France.

Gilbert, J. and Lemaréchal, C. 1989. Some numerical experiments with variable-storage quasi-Newton algorithms. *Mathematical Programming 45*, 407–435.

Gould, N., Orban, D., and Toint, P. 2003. CUTEr (and SifDec), a Constrained and Unconstrained Testing Environment, revisited. *ACM Trans. Math. Softw. 29*, 373–394.

Hock, W. and Schittkowski, K. 1981. *Test Examples for Nonlinear Programming Codes.* Number 187 in Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin.

Lemaréchal, C. 1980. Using a Modulopt minimization code. Unpublished Technical Note.

Mathworks. The Matlab distributed computing engine. `http://www.mathworks.com/`.

Mittelmann, H. D. and Pruessner, A. 2003. A server for automated performance analysis of benchmarking data. Working paper.

Schittkowski, K. 2002. Test problems for nonlinear programming – User's guide. Online Paper.

Scilab. 2007. An open source platform for numerical computation. `http://www.scilab.org`.

Received March 2007; revised November 2007; accepted —