INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

LIBOPT – An environment for testing solvers on heterogeneous collections of problems – The manual, version 2.1 –

J. Charles GILBERT — Xavier JONSSON

N° 0331 (revised)

6 janvier 2009

Thème NUM

ISSN 0249-0803 ISRN INRIA/RT--0331 (revised)--FR+ENG

LIBOPT – An environment for testing solvers on heterogeneous collections of problems – The manual, version 2.1 –

J. Charles Gilbert[*] , Xavier Jonsson[†]

Thème NUM — Systèmes numériques
Projet Estime

Rapport technique n° 0331 (revised) — 6 janvier 2009 — 46 pages

Abstract: 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.

Key-words: benchmarking – collection of problems – CUTEr – Libopt – Modulopt – optimization – performance profile – scientific computing – solver comparison – testing environment.

[*] INRIA Rocquencourt, projet Estime, BP 105, 78153 Le Chesnay Cedex, France; e-mail: Jean-Charles.Gilbert@inria.fr.

[†] Mentor Graphics (Ireland) Ltd. - French Branch; 180, Avenue de l'Europe - Zirst Montbonnot; F-38334 Saint Ismier Cedex; e-mail: Xavier_Jonsson@mentorg.com.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

LIBOPT – Un environnement pour évaluer des solveurs sur des collections hétérogènes de problèmes – Le manuel, version 2.1 –

Résumé : L'environnement Libopt est à la fois une méthodologie et un ensemble d'outils qui peuvent être utilisés pour tester, comparer et profiler des solveurs sur des problèmes de diverses collections. Ces dernières peuvent être hétérogènes dans le sens où leurs problèmes peuvent avoir des caractéristiques communes qui diffèrent d'une collection à l'autre. Libopt apporte un point de vue unificateur sur ce monde composite en offrant, par exemple, la possibilité de lancer n'importe quel solveur sur n'importe quel problème compatible avec lui, en utilisant la même commande Unix/Linux. L'environnement fournit également des outils pour comparer les résultats obtenus par divers solveurs sur un jeu spécifié de problèmes. La plupart des scripts qui accompagnent l'environnement Libopt ont été écrits en Perl.

Mots-clés : calcul scientifique – collection de problèmes – comparaison de solveurs – CUTEr – environnement de test – évaluation de performance – Libopt – Modulopt – optimisation – profil de performance

Contents

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* [1, 12]. This one supplies ready-to-use interfaces between some known solvers and a collection of problems encoded with the SIF language [3]. 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 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, Libopt considers CUTEr 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* [14] 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 tedious 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 modeling language, the possibility to define their own collection of problems, using their preferred general purpose language, and to run these problems by using the same set of commands as those that can now run the CUTEr and Modulopt 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 this manual 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 manual 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 describes the

Libopt package and its directory structure; it enumerates the installation instructions. A deeper understanding of the Libopt mechanisms is necessary if one wishes to enrich the environment with new collections of problems and solvers; these "squalid details" are presented in section 4. 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. We gather in section 5, the subjects related to the administration of the environment: introduction of a new platform (section 5.1), installation of a new collection of problems (section 5.2) or a new solver (section 5.3), and activation a solver-collection cell (section 5.4). The manual ends with a short perspective section that presents possible improvements.

A shorter introduction to Libopt is given in [10].

Notation. We use the following typographical 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, for which you are supposed to substitute a value depending on the context, is written in *`italic typewriter font`*.

Throughout this manual, 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 (part of) arguments in a Unix/Linux command line are surrounded by the brackets '`[`' and '`]`'.

We use the following abbreviations of regular expressions:

`\u` for `[␣\t]` (the *blank* character),
`\s` for `[␣\f\n\r\t]` (the *space* character),
`\w` for `[a-zA-Z0-9_]` (the character for writing words),

where `\f` stands for a formfeed, `\n` for a newline, `\r` for a carriage return, and `\t` for a tab.

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 the many other subcommands described in this manual), and the dots stand for the subcommand arguments. This section offers an introduction to these subcommands.

It is worth mentioning that a manual page gives the details on the use of each of these subcommands. To get it, enter

```
% man libopt
```

A probably more convenient way of having a description of a subcommand is by using its '`-h`' option. For example

```
% libopt run -h
```

provides a short description of the `run` subcommand.

Before describing the three subcommands `run`, `base`, and `profile`, we start by defining terms that are continually used in this manual.

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 triggered 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. Well, not sure this is very explicit, but we shall not try to be more precise; in the same vein, we shall take the notion of *problem* as a concept, not requiring any definition. The *name* of a solver is a string that must be matched by `\w+` (see the notation above for the meaning of the character '`\w`'; the multiplier '`+`' means that one or more of the immediately previous character or character class, here '`\w`', must be present).

A *collection* is a set of problems in an arbitrary scientific computing area. The *name* of a collection is also a string that must be matched by `\w+`. Problems in the same collection must differ by their *name* (again a string that must be matched by `\w+`), 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 [15], Scilab [19], Gams [2], Ampl [7], SIF [3], 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. Its *name* must also be matched by `\w+`. 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 section 5.3. 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). This subject is discussed in section 5.2. 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 [9], 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*. If the suffix "*.subc*" is not present in the command string above, the `all` or `default` subcollection is assumed, depending on the presence or absence of problems in the command string. Subcollections are described by lists of problems (see section 4.2), which are searched in various directories by the `libopt run` command (see section 4.3.2). 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. We shall come back on this feature in section 4.6.

It may happen that a `libopt run` command has been launched by mistake and that it is desirable to kill it. Note, however, that a `CTRL-C` is not the appropriate action to kill the `libopt run` command. Indeed, this command may trigger many children processes, one after the other. Since a `CTRL-C` only kills the current child process, this one is usually immediately replaced by a new one, while the root process is left alive. A better solution is to enter

```
% pkill -f $LIBOPT_DIR
```

This Unix/Linux command kills all the processes whose command line contains the characters in the string "`$LIBOPT_DIR`", which is the case of all the scripts in the Libopt environment. This is usually radical enough, without being damaging.

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 behavior can be prevented by setting the option `-k` of the `libopt run` command, see the introduction of section 4 and section 4.3.2). 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). Therefore this string must appear only once in the standard output and the Libopt line cannot be split on several lines on the standard output.
- 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* (a string matched by \w+) 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 [9].

The Libopt line can be written with some flexibility: blanks (matched by \u+) surrounding the various fields and elements of fields are ignored, and a comment can be introduced (it starts with the sharp character '#' and goes up to the end of the line).

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 [16], 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. It is good practice to save the file *solv_coll_subc*.lbt preciously, since the previous command may have required much time to run, since it is not a large file, and since its ascii encoding makes it very stable with respect to the possible evolutions of the languages (Libopt, Perl, or Unix/Linux).

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. Possible file names are libopt_database.db or the pair libopt_database.dir/ libopt_database.pag. This variety of storages makes the database not very portable, which is also a reason to save the ascii files that have been used to generate it. The database stores a collection of key-value pairs: the key is the first part of the Libopt line, more exactly the string "*solv%coll%prob*", without useless blanks (there is no reason to store the invariant string "libopt%"); the value is the second part of the Libopt line, namely what we have denoted by the *sequence-of-token-number-pairs* above. 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: -l for listing its contents, -r for replacing entries and -d for deleting results. A full description of the libopt base command line is given in section 4.4 and in the manual page of libopt.

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 (see section 4.1 for a complete description of this 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 [5] 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 10, `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` (see section 4.1 for more precision on this file).

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. To see how to prescribe this, read about the "problem directive" in the description of the `libopt_profile.spc` specification file in the manual page of `libopt`.

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 section 4.5. 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 criterion $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 `self-portraiture` 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

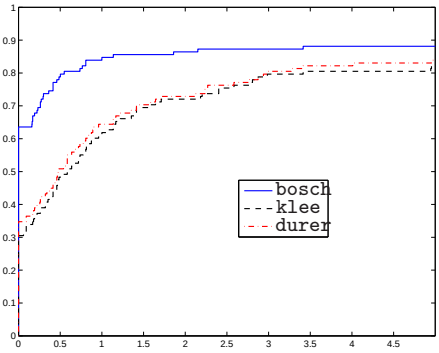

Figure 1: Typical performance profiles for three solvers; relative performance (abscissa) in $\log_2$ scale

are necessary to get precisely that graph).

Performance profiles have been introduced by Dolan and Moré [5]. To be comprehensive, we feel it necessary to give a few words on the meaning of these curves. More can be found in the original paper. The main objective is to replace tables of numbers by curves (one per solver), which, with some reading-keys, provide a rapid understanding of the *relative* performance of various solvers on a given set of problems. More specifically, these curves are used to compare the efficiency of a set $\mathcal{S}$ of solvers on a set $\mathcal{P}$ of test problems. Let

$$\tau_{p,s} := \text{performance of the solver } s \text{ on the problem } p,$$

where the *performance* refers to the number of a token-number performance pair. The *relative performance* of a solver s (with respect to the other solvers in $\mathcal{S}$) on a problem p is the ratio

$$\rho_{p,s} = \frac{\tau_{p,s}}{\min\{\tau_{p,s'} : s' \in \mathcal{S}\}}.$$

Of course $\rho_{p,s} \geq 1$. On the other hand, it is assumed that $\rho_{p,s} \leq \bar{\rho}$ for all problems p and solvers s, which can be ensured only by setting $\rho_{p,s}$ to the large number $\bar{\rho}$ if the solver s cannot solve the problem p (`info` is nonzero in the Libopt line). Actually, we shall consider that s fails to solve p if and only if $\rho_{p,s} = \bar{\rho}$. The *performance profile* of the solver s (relative to the other solvers) is then the function

$$t \in [1, \bar{\rho}] \mapsto \wp_s(t) := \frac{|\{p \in \mathcal{P} : \rho_{p,s} \leq t\}|}{|\mathcal{P}|} \in [0, 1],$$

where $|\cdot|$ is used to denote the number of elements of a set (its cardinality).

Only three facts need to be kept in mind to have a good interpretation of these upper-semi-continuous piecewise-constant nondecreasing functions:

- $\wp_s(1)$ gives the fraction of problems on which the solver s is the best; note that two solvers may have an even score and that all the solvers may fail to solve a given problem, so that it is not guaranteed to have $\sum_{s\in\mathcal{S}} \wp_s(1) = 1$;
- by definition of $\bar{\rho}$, $\wp_s(\bar{\rho}) = 1$; on the other hand, for small $\varepsilon > 0$, $\wp_s(\bar{\rho} - \varepsilon)$ gives the fraction of problems that the solver s can solve; this value is independent of the performance token chosen for the comparison;
- the value $\wp_s(t)$ may be given an interpretation by inverting the function $t \mapsto \wp_s(t)$: for the fraction $\wp_s(t)$ of problems in $\mathcal{P}$, the performance of the solver s is never worse than t times that of the best solver (this one usually depends on the considered problem); in this respect the argument at which $\wp_s$ reaches its "almost maximal" value $\wp_s(\bar{\rho}-\varepsilon)$ is meaningful.

With performance profiles, the relative efficiency of each solver appears at a glance: the higher is the graph of $\wp_s$, the better is the solver s.

When developing a solver, one is often led to make harmless modifications, which should have no effect on the performance of the solver. To be more confident on the fact that these innocuous modifications have indeed no side effect, one can see whether the solver provides the same results on a more or less large collection of problems. Libopt provides a convenient way of realizing this. Suppose that the file `res_before.lbt` (resp. `res_after.lbt`) contains the Libopt lines obtained on this collection of problems before (resp. after) having made the modifications. Then the command

```
% libopt diff -p perf res_before.lbt res_after.lbt
```

will tell whether the performance *perf* has been affected by the modification and, in case the answer is affirmative, for which problems.

3 The Libopt package

This section contains a brief explanation on how to unpack and install the Libopt package, and a rather detailed discussion of its directory/file structure.

3.1 Downloading Libopt

Libopt has been designed to be used by more than one person at the same time, so that it is recommended to put the Libopt hierarchy in a location that is accessible to all the potential users. The Libopt hierarchy can be retrieved in one of the following two manners.

- The first possibility is to get the package from the Inria GForge `http://gforge.inria.fr`, with an anonymous access, using the `svn` command

```
% svn checkout \
?     svn://scm.gforge.inria.fr/svn/libopt-public libopt
```

This procedure requires to have `svn` installed on the user system. This is the method of choice for those who are desired to update their local copy with the future versions of the software, just by entering in the local `libopt` directory:

```
% svn update
```

- The other possibility is to retrieve the tarball

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

  and to unpack it, in an appropriate directory, using successively the commands `gunzip` and `tar`. The update of the software can be more difficult with this technique.

The current address of the Libopt site is

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

3.2 Structure of the Libopt hierarchy

The retrieving procedure above creates a tree of subdirectories, whose root name is `libopt`. This top-level directory contains the directories `.libopt` (Libopt's secret directory whose contents should not be modified by the users), `bin` (Perl scripts of the Libopt commands), `collections` (problem collections), `doc` (some documentation), `man` (manual page), `platforms` (platform descriptions, see section 5.1), and `solvers` (solver descriptions and interfaces), as well as other plain files; see figure 2. You are supposed to have accepted the terms of the file `LICENSE` to use the software.

```
Level 0      Level 1      Level 2      Description
.libopt ............................. Secret directory
bin ................................. Libopt scripts
collections
        +--- cuter .................. CUTEr collection
        +--- modulopt ............... Modulopt collection
doc ................................. Libopt documentation
man     +--- man1 ................... Manual page
platforms ........................... Platform descriptions
solvers +--- m1qn3   +--- bin ....... M1qn3 binary directory
        |            +--- cuter ..... Interface M1qn3/CUTEr
        |            +--- doc ....... M1qn3 documentation directory
        |            +--- modulopt ... Interface M1qn3/Modulopt
        +--- sqplab  +--- cuter ..... Interface Sqplab/CUTEr
                     +--- doc ....... Sqplab documentation directory
```

Figure 2: Part ot the Libopt hierarchy in the standard distribution

The `collections` directory gathers a set of subdirectories, each of them corresponding to an installed collection of problems. The standard distribution includes the collections CUTEr [12] in `cuter` and Modulopt [14, 9] 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. More is said about this directory in section 5.2.

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`" (see also section 4.2). 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 some scripts when no list is specified as one of their arguments (see section 4.3.2); it is sometimes a symbolic link to the list `all.lst`.

It is also natural (but not mandatory) to have in `collections/coll` a subdirectory, usually called `probs`, which contains all the problems of the collection. For example, 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. Of course `probs` can be a symbolic link to the actual problem directory.

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. 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 3):

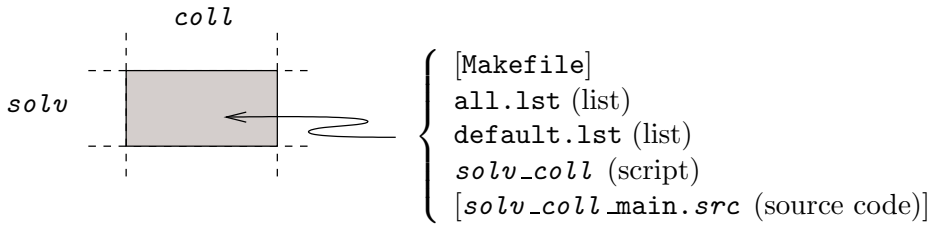

Figure 3: A single cell of the {solvers} × {collections} Cartesian product

- `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 4.3.2);
- *`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 4.3.3); 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 4.3.4).

3.3 Installation instructions

Once the Libopt has been unpacked, a few things need to be done before being able to use the environment. We suggest the following steps.

1. First, add the following environment variable definitions to the shell startup file `~/.tcshrc` (the `tcsh` shell is assumed; adapt the format and use another startup file if the shell is different; echoing your `SHELL` environment variable might be useful here):

   ```
   setenv LIBOPT_DIR    dir
   setenv LIBOPT_PLAT   plat
   setenv PATH          ${PATH}:${LIBOPT_DIR}/bin
   ```

   The variable `LIBOPT_DIR` sets the head directory of the Libopt hierarchy; hence change "*`dir`*" to the location of this head directory. The variable `LIBOPT_PLAT` provides the *platform* on which you work, see section 5.1 for valid values for the string "*`plat`*". The third setting adds to `$PATH` the directory of the Libopt commands. In some Linux system, this setting automatically adds `$LIBOPT_DIR/man` to the possible search paths for the manual page. If this is not the case with your system, you might have to set

   ```
   setenv MANPATH       ${MANPATH}:${LIBOPT_DIR}/man
   ```

   With these settings and a resourcing of the startup file, you should have an appropriate answer to the following commands:

   ```
   % libopt -h
   % man libopt
   ```

2. You can now activate the installation procedure (from any directory, since the `libopt` command is in `$LIBOPT_DIR/bin`):

   ```
   % libopt install -v
   ```

   The option `-v` specifies the verbose mode of the command. This command defines the list of collections existing in the standard distribution

   `$LIBOPT_DIR/collections/.collections.lst`,

   the list of solvers existing in the standard distribution

`$LIBOPT_DIR/solvers/.solvers.lst`,

and for each solver *solv* in the previous list, it defines the list of collections that the solver *solv* has been prepared to consider

`$LIBOPT_DIR/solvers/`*solv*`/.collections.lst`.

These lists cannot be given in the standard distribution, since they depend on each installation. They will not have to be managed by the user, which motivates the fact their name starts with a dot. The `libopt install` command also verifies the consistency of the Libopt hierarchy, by checking whether expected files are present.

3. The third step is optional but recommended. It consists in introducing the startup file `~/.liboptrc` (hence in your home directory). This file provides additional information to some Libopt commands, for example for helping `libopt base` to detect typos in Libopt lines. See section 4.1 for a detailed description of the directives that can be put in this startup file.

   In optimization, one can start by copying a file given in the `doc` directory

   ```
   % cp $LIBOPT_DIR/doc/liboptrc_optim ~/.liboptrc
   ```

   and modify it afterwards if the tokens used in the file do not suit one's needs.

4. The fourth step deals with the installation of a collection. Some collections are already installed in the standard distribution. They are listed by the command

   ```
   % libopt collections
   ```

   You will recognize the CUTEr and Modulopt collections, respectively denoted by `cuter` and `modulopt`. For the listed collections, nothing needs to be done for being able to use them. For the installation of another collection, follow the procedure described in section 5.2.

5. The fifth step deals with the installation of a solver in the Libopt environment. Indeed, Libopt does not provide solvers. If you are lucky, you might have one of the solvers of the standard distribution, those listed by the command

   ```
   % libopt solvers
   ```

   In this case, the installation is usually rapid; it is described in the file `$LIBOPT_DIR/solvers/`*solv*`/README_install` if *solv* is the considered solver. In particular, there is nothing to do for the installation of the Matlab solver `fmincon` (of course you will need to have Matlab to be able to use it). The installation of a new solver is actually quite simple. The procedure is described in section 5.3.

6. The sixth step is the most complex. It deals with the connection between a solver *solv* and a collection *coll*. This connection is necessary to be able to run *solv* on the problems of *coll*. To establish it, one has to fill in the cell (*solv*, *coll*) of the Cartesian product (3.1). To see the connections that have been set up in the standard distribution, you can enter one of the following commands:

   ```
   % libopt solvers -x
   % libopt collections -x
   ```

A cross 'x' in the Cartesian product indicates that the connection is established. How to fill in the cells of the Cartesian product (3.1) is explained in section 5.4.

You should now be able to run a solver (say *solv*) on a problem (say *prob*) of some collection (say *coll*, to which *solv* is connected), by entering

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

The option `-v` specifies the verbose mode of the command. The next section gives more details on the possibilities of the Libopt environment and explains how the commands are dealt with.

4 Libopt in depth

Section 2 has presented the tools the most frequently used of the Libopt environment. Sometimes, however, it is necessary to do other operations like introducing a new collection of problems or a new solver in the environment. A deeper understanding of the Libopt mechanisms is required for realizing these operations safely. The goal of this section is to provide a comprehensive description of the principles governing the software, as well as the available commands.

The Libopt commands are located in the `bin` directory of the hierarchy. They are written in Perl. Actually, when one enters `libopt` *subcommand*, the `libopt` command launches the `libopt_`*subcommand* script. The latter could be launched directly, but it is better to let the `libopt` command do this. Many of these commands accept several of the following options, whose meaning will not be repeated:

- `-h` help mode; a short help message describing the command usage is printed;
- `-k` keep mode; the files generated by the command and its children are not removed on exit;
- `-t` test mode; same as `-v`, but the commands are not executed;
- `-v` verbose mode; the commands are executed and are also printed on the standard output, with possible additional information.

When a command launches other scripts and some of the options `-k`, `-t`, and/or `-v` are present, these options are transmitted to these scripts.

4.1 The `~/.liboptrc` startup file

The file `~/.liboptrc` is used to provide some additional information to Libopt, in connection with the activity of the user. Here are the *directives* that are meaningful:

```
tokens = list-of-tokens
performance_tokens = list-of-performance-tokens
data_base = DBFile
```

The first two directives are related to the Libopt line (see page 9). The *list-of-tokens* is a blank-separated list of the tokens that are considered to be valid in the Libopt line. If this directive is present, `libopt base` uses the tokens of this list to determine whether the tokens encountered in the submitted Libopt lines are valid (otherwise there

is no verification). The *list-of-performance-tokens* is a blank-separated list of tokens that can be considered as performance tokens. If the directive `tokens` is present, it is verified that the performance tokens are among the tokens in the *list-of-tokens*. The performance tokens are used by `libopt base` and `libopt profile` to determine whether the token proposed to make the comparison of solvers is a performance token.

The third directive, `data_base`, specifies the database name *DBFile* that has to be used when this one is not specified in the command line of some commands that need such a database (`libopt base`, `libopt profile`, ...). Hence the name given on the command line has priority over the name given in the startup file. If neither the command line nor the startup file provide a file name, the commands assume that the database is the file `libopt_database` in the working directory. The actual name of the database on disk may have the extension "`.db`" or may be represented by two files with names having the extensions "`.dir`" and "`.pag`"; this depends on the operating system.

An example of `liboptrc` file can be found in the directory `doc` (under the name `liboptrc_optim`). It may be useful in optimization.

4.2 Lists

The behavior of the Libopt scripts depends on various lists, which are used to control the consistency of the commands with respect to the contents of the Libopt hierarchy. These lists are located in various directories and, for clarity, they have the suffix "`.lst`". One finds lists of solvers, of collections, and of problems (*subc*`.lst`):

```
$LIBOPT_DIR/collections/.collections.lst
$LIBOPT_DIR/collections/coll/subc.lst
$LIBOPT_DIR/solvers/.solvers.lst
$LIBOPT_DIR/solvers/solv/.collections.lst
$LIBOPT_DIR/solvers/solv/coll/subc.lst
```

A list is simply a sequence of strings separated by spaces (`\s+`). *Comments* are possible; they start from the character '`#`' up to the end of the line. Comments are used to describe the list and its elements.

4.3 The `libopt run` command

4.3.1 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 the script

  `$LIBOPT_DIR/bin/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*. (4.1)

  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 4.3.2.

- The operating-system level.

  The second level is the one that takes in charge the elementary operation (4.1). 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 4.3.3.

- 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 4.3.4.

4.3.2 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 need not be adapted when the Libopt environment is enriched with new solvers or new collections of problems.

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 is discussed in section 4.6), *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 (matched by `\u`), 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(comments - starting with the sharp character '`#`' and going up to the end of the line - and empty lines are allowed). 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 actually tried to be solved by *solv* is established by `libopt run` in the following manner.

1. If there is no subcollection specified in the command line (no "*.subc*"), `libopt run` assumes either the "`all`" subcollection if the *list-of-problems* is nonempty or the "`default`" subcollection otherwise. The logic behind this rule is to be non-restrictive when some problems are mentioned in the command line (the "`all`" subcollection is supposed to give all the problems of the collection), and to limit the number of problems

to consider when no problem is specified (the "`default`" subcollection is supposed to give a not too large subset of typical problems of a huge collection).

2. Once a subcollection, say *subc2*, has been determined, either from the one given in the command line or from the rule just mentioned, `libopt run` looks for a file describing the subcollection *subc2*. It searches in the following order:

   coll`.`*subc2*`.lst` in the working directory
   `$LIBOPT_DIR/solvers/`*solv*`/`*coll*`/`*subc2*`.lst`
   `$LIBOPT_DIR/collections/`*coll*`/`*subc2*`.lst`

   The logic is the following. Since `libopt run` first looks in the working directory, the user is allowed to build a temporary list of problems for a special purpose. Next, priority is given to the lists the solver *solv* has declared in connection with the collection *coll*, since it is the solver that knows the type of problems it can solve. If no file *subc2*`.lst` is found in the above locations, the command line is ignored.

3. `Libopt run` takes the additional precaution of eliminating from *subc2* the problems that are not solvable by *solv* (i.e., those that are not in `$LIBOPT_DIR/solvers/`*solv*`/`*coll*`/all.lst`). Let *subc3* be the resulting list.

4. The final list is then, either *subc3* if there is no *list-of-problems* in the command line, or the intersection of *subc3* and the *list-of-problems* if the latter is nonempty. Let *subc4* be the final list.

Then `libopt run` triggers the script `$LIBOPT_DIR/solvers/`*solv*`/`*coll*`/`*solv*`_`*coll* for each problem *prob* of the list *subc4*.

4.3.3 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 4.3.2. 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 4.3.2. It is the `libopt run` command that determines the problem *prob* to solve.

As discussed in section 4.3.1, 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 [9]), 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 sd*solv* (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 sd*solv* 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 sd*solv* 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

(see sections 5.2.1 and 5.4.1). 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.

4.3.4 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 4.3.3.

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*.

4.4 The `libopt base` command

We have discussed in section 2.3, the role of the `libopt base -a` command in the basic work cycle of the Libopt environment. In particular, all the details on the Libopt line have been presented in that section. In the present section, we give the precise form of the `libopt base` command.

The accepted forms of the `libopt base` command are the following

```
% libopt base -h
% libopt base [-t] [-v] [-b DBFile] [-a|-r] ResFile
% libopt base [-t] [-v] [-b DBFile] -d selection
% libopt base      [-v] [-b DBFile] -l
```

If the option `-h` is present, `libopt base` prints a short help message describing the command usage and exits. The second command is used for adding/replacing results in a database, the third one for deleting results, and the fourth one for listing the contents of the database.

The database *DBFile* is specified with the `-b` option. If this option is not present, `libopt base` looks for the database specified by the directive `data_base` in the startup file

`~/.liboptrc`. Finally, if this one does not exist, `libopt base` assumes that the database is the file `libopt_database` in the working directory (see section 4.1 for the details). The file `ResFile` used in the second form of the command is supposed to contain Libopt lines, typically generated by the libopt run command. A result corresponding to a triple (solver, collection, problem), already existing in the database is not replaced, unless the option `-r` is present.

The string `selection` used in the third form is interpreted as a file name if it is not terminated by the character '`%`' or as a way of selecting triples (solver, collection, problem) otherwise.

- If `selection` is view as a file name, this file is supposed to be formed of Libopt lines and the results with the same triples (solver, collection, problem) as in the Libopt lines are deleted from the database. This form of the command is typically used when Libopt lines in a file *`ResFile`* have been introduced by mistake in the database and that it is desired to delete them from the database. Be careful, however, since when the sets A and B have a nonempty intersection, $(A \cup B) \backslash B \neq A$.
- If `selection` is the string "*`solv%coll%prob%`*", the result corresponding to the triple (solver, collection, problem) = (*`solv`*, *`coll`*, *`prob`*) is deleted from the database (if present). An empty solver (resp. collection, problem) field in the string `selection` matches any solver (resp. collection, problem). For instance, if `selection` is the string "*`solv%%%`*" (can be simplified in "*`solv%`*"), all the results corresponding to the solver *`solv`* are deleted from the database. As another example, if `selection` is the string "*`%coll%prob%`*", all the results corresponding to the problem *`prob`* of the collection *`coll`* are deleted from the database. The option `-t` can be used to see the effect that a deleting command would have before really doing it.

The last form of the subcommand `base`, the one using the flag `-l`, is used to list the contents of the database. For example

```
% libopt base -l | grep "^solv" --color
```

list (coloring *`solv`*, the solver name) all the problems for which results of the solver *`solv`* have been stored in the default database.

4.5 The `libopt profile` command

The `libopt profile` command was introduced in section 2.4, where its role in the basic work cycle of the Libopt environment was shown. In this section, we give more details on the command. It has the following forms:

```
% libopt profile -h
% libopt profile [-v] [-f file]
```

This command generates performance profiles a la Dolan and Moré [5]. To achieve this goal, the command uses a result database, typically generated by the commands `libopt run` (section 4.3) and `libopt base -a` (section 4.4). Because of their complexity, performance profiles cannot be shortly described in the command line. Instead, the description is

done through a specification file, whose default name is `libopt_profile.spc` in the working directory. Another `file` can be specified by the option `-f`. When starting, `libopt profile` tries to read this file and terminates if it cannot find and read it. Therefore, *this file is mandatory*. The file is described in the manual page of `libopt`. To make the discussion that follows understandable, some of the directives of the specification file have to be clarified. First, at least two solvers have to be selected for the comparison, say *`solv1`* and *`solv2`*, which is done by one or more directives of the form

`solver` *`solv1 solv2 ...`*

This is because `libopt profile` cannot choose default solvers and it makes no sense to compare a solver with itself. Optionally, some problems may have been selected by one or more directives of the form

`collection` *`coll`*`[.`*`subc`*`] ...`

The list of problems specified by this directive is searched in order in the following files

`coll.subc``.lst` in the working directory
`$LIBOPT_DIR/collections/`*`coll/subc`*`.lst`

where *`subc`* is set to the string '`all`' if it is not specified by the directive. Finally, a performance token must have been specified, say *`ptok`*, either in the command line of `libopt profile` or by a directive of the form

`performance` *`ptok`*

Here is how `libopt profile` selects the actual problems that will be used for the comparison of the solvers specified by the '`solver`' directives. These problems are those that satisfy the following conditions:

- they are among those problems that have been specified by a '`collection`' directive, if any such directive as been used in the specification file `profile.spc`,
- they are present in the database *`DBFile`* with a *`ptok`* performance token,
- they have not been discarded by the '`problem`' directive (see the manual page of `libopt`),
- they have been solved by all the solvers specified by the '`solver`' directives.

4.6 Getting results from a modified version of an installed solver

Suppose that some parameters of some solver, say `goya`, are modified and that it is desirable to see the effect of these modifications on the performances of the solver. This situation also occurs during the development of a solver, when it is desirable to see whether a newly developed technique has a good effect, by comparing the results of the new version with the previous ones. We see three ways of realizing this and consider them from the hardest to the easiest one.

1. A first possibility would be to define a full new hierarchy under the `$LIBOPT_DIR/solvers` directory with the new version of the solver. This results in defining a new row in the {solvers} × {collections} Cartesian product. This is certainly safe, but it is not an easy and rapid task. In addition, the solver with the new technique may not exist for a long time. Therefore, this option is probably not the best idea.
2. A better way of doing is to modify temporarily the code that generates the Libopt line, so that instead of generating a line of the form

   ```
   libopt%goya.quatro-de-junio% ...
   ```

   where `goya` has been changed to `goya.quatro-de-junio`. The modified Libopt lines can also be obtained by editing the file containing the Libopt lines of the new version of the solver. The `libopt base` command will believe that this line has been generated by the solver `goya.quatro-de-junio`, even though judiciously this one does not exist in the Libopt hierarchy. Therefore, the comparison will be made without difficulty with other solvers whose results have been stored in some database (using the `libopt base` command), possibly with `goya.tres-de-mayo`, the presumably best version of `goya` obtained so far.
3. Alternatively, instead of modifying the code generating the Libopt lines, one could modify these lines after they have been written in a file, using a text editor. Actually, Libopt offers an automatic way of getting the same effect. It is indeed possible to add a *tag* to the solver name in the Libopt line. This is obtained by entering

   ```
   % libopt run "goya.quatro-de-junio coll prob"
   ```

   Then, `libopt run` concatenates the string "`.quatro-de-junio`" to the solver name in the Libopt line of the standard output, which then becomes

   ```
   libopt%goya.quatro-de-junio%coll%prob%...
   ```

   Of course, `libopt run` considers that the solver is `goya` not `goya.quatro-de-junio` (because of the dot '`.`', the latter does not exist in the Libopt hierarchy). This effect is obtained by filtering the standard output of `libopt run` with the stream editor `sed`, which only modifies the Libopt lines.

   If the string *`solv.tag`* contains several *dots*, the first dot is used to separate the solver name from the tag. Hence a solver name cannot contain a dot, while a tag can contain several dots.

For example, if it is desired to compare the scalar and diagonal running modes of the ℓ-BFGS code M1qn3 [11] on 143 unconstrained CUTEr problems, one can proceed as follows. Run M1qn3 with its default setting (diagonal running mode), tagging the solver with the string "`diag`" to remember the option used in the run, and store the Libopt lines

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

Set the option `imode(1) = 1` in the main program `m1qn3ma.f` used by the CUTEr hierarchy to put M1qn3 in scalar running mode, generate the object file `m1qn3ma.o`, put it in the appropriate place in the CUTEr hierarchy, and run-store again:

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

Adapt the `solver` directive of the specification file `libopt_profile.spc`:

```
solver m1qn3.diag m1qn3.scal
```

Now enter

```
% libopt profile
```

This last command with appropriate directives in the specification file `libopt_profile.spc` generates the performance profiles, those that are given in figure 4. An accustomed

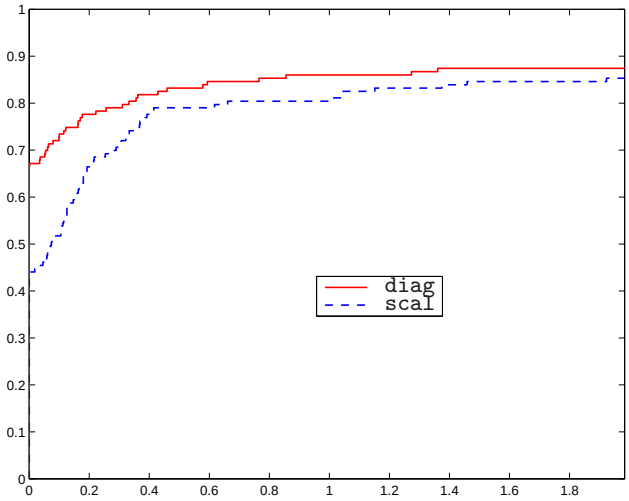

Figure 4: Performance profiles of the diagonal (`diag`, red solid curve) and scalar (`scal`, blue dashed curve) running modes of `M1qn3` on 143 unconstrained problems of the CUTEr collection, comparing the number of function evaluations

reader can deduced from these curves that the diagonal mode of `M1qn3` can be considered as slightly more efficient than the scalar mode (its profile is higher, see section 2.4). This is an average estimation, since the plot with its 2-logarithmic abscissa reveals that, if the scalar mode can require $2^{1.9} \simeq 3.7$ times more function evaluations than the diagonal mode, the latter can also require sometimes $2^{1.4} \simeq 2.6$ times more evaluations than the former.

4.7 Other subcommands

We gather in this section, other Libopt subcommands that can be useful.

4.7.1 The `libopt problems` command

The accepted forms of the `libopt problems` command are the following

```
% libopt problems -h
% libopt problems -c coll.[subc] [-s solv]
% libopt problems "solv coll.[subc]" ...
```

It has been designed to list problems of the specified subcollection *coll*[.*subc*]. If *subc* is omitted, the `all` subcollection is assumed, which normally lists all the problems of the specified collection *coll*.

As usual, the first form of the command print a short message recalling the syntax of the subcommand.

If the option `-c` is used, the subcollection *subc* refers to the one defined by the collection *coll*. If the options `-c` and `-s` are used, the subcollection is related to the collection *coll* considered by the solver *solv* and should list problems of that collection that the solver *solv* can structurally solve.

If no option is used, the subcollections are those that would be selected by the Libopt command:

```
libopt run "solv coll[.subc]" ...
```

5 Administrating the environment

We gather in this section the operations that are less often executed, since they change the contents of the Libopt environment or adapt the environment to a new platform.

5.1 Managing platforms

The environment variable `LIBOPT_PLAT` provides the *platform* on which the user works. This is a string of the form "*mach.os.comp*", where *mach* designates the machine, *os* is its operating system, and *comp* is the compiler suite. The current standard distribution of Libopt includes the following platforms:

```
mac.osx.gcc
pc.lnx.pg
```

where

`mac`	Macintosh computer (Apple),
`pc`	PC-like computer,
`lnx`	Linux operating system,
`osx`	OSX operating system (Macintosh),
`gcc`	GNU Compiler Collection,
`pg`	Portland Group compilers.

The platform name is used by some scripts and makefiles of the Libopt environment. The Libopt hierarchy of the standard distribution contains indeed no object/compiled files. When such files are required they are generated in the working directory and the value `$LIBOPT_PLAT` is used to generate them. This description is given by files of the form

```
make.plat
```

where "*plat*" is one of the strings above. These files are located in the directory

`$LIBOPT_DIR/platforms`

A new platform is therefore obtained by adding a new file `make.`*`plat`* in that directory, adapting the various variables defined in the file. It is used by resetting the environment variable `LIBOPT_PLAT`.

5.2 Managing collections of problems

There are certainly many ways of hooking a collection of problems to the Libopt environment. This installation depends in great part on the features of the considered collection. For example, the installation of the CUTEr and Modulopt collections are quite different. Almost everything is possible provided the scripts using the collection reflect the choices taken during the installation. This is why Libopt can just be viewed as an empty shell (the directory structure) and a methodology (embodied in its scripts).

We start by presenting the part of the installation procedure that is common to any collection and the command that is available for realizing this (section 5.2.1). Next, we consider the commands that are available for removing a collection from Libopt (section 5.2.2) and for adding/removing a problem to a collection (section 5.2.3). Finally, the operations just described are discussed in the case of the CUTEr and Modulopt collections (sections 5.2.4 and 5.2.5 respectively), which are already hooked to Libopt in the standard distribution.

5.2.1 Hooking a collection of problems

Hooking a new collection, say *`coll`*, to the Libopt environment starts by entering the Unix/Linux command (using the option `-v` to be informed of what this command does is recommended)

`% libopt addcollection [-v]` *`coll`*`.`

In addition to doing some verifications, in order to check and maintain the consistency of the environment, this command takes care of the operations at the Libopt level. More specifically,

- it adds the name *`coll`* of the collection to the list

  `$LIBOPT_DIR/collections/.collections.lst`,

- it makes the new directory

  `$LIBOPT_DIR/collections/`*`coll`*,

- it creates two empty files (see section 3.2 and below for a description of the expected contents of these files)

  `$LIBOPT_DIR/collections/`*`coll`*`/all.lst`,
  `$LIBOPT_DIR/collections/`*`coll`*`/default.lst`,

- and it makes the two new directories

  `$LIBOPT_DIR/collections/`*`coll`*`/bin`,
  `$LIBOPT_DIR/collections/`*`coll`*`/templates`;

the `bin` directory is intended to contain scripts associated with the collection and, not surprisingly, the directory `templates` is intended to contain templates (see below).

What is performed by `libopt addcollection` is not much, but it discharges the user from taking care of these tedious operations that are common to the installation of any collection.

The rest of the installation is very collection dependent. As far as Libopt is concerned, the installation can be done in an arbitrary manner by introducing files in the directory `$LIBOPT_DIR/collections/`*`coll`*. The choices made during the installation will have a direct impact on the scripts

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

which will have to be written for each solver *`solv`* desiring to solve the problems of the collection *`coll`*. The role of these *`solv_coll`* scripts has been described in section 4.3.3. We have observed that, for a given collection, these scripts hardly differ from one solver to the other (they can depend on the language used to write the solver). Therefore, they can often be automatically generated from a collection-dependent template

`$LIBOPT_DIR/collections/`*`coll`*`/templates/SOLV_`*`coll`*.

We have adopted this technique for the CUTEr and Modulopt collections in the standard distribution. The collection-dependent script that takes care of generating a *`solv_coll`* script from the `SOLV_`*`coll`* template is

`$LIBOPT_DIR/collections/`*`coll`*`/bin/libopt_addcell_`*`coll`*,

which is launched by the collection-independent Libopt command `libopt addcell` (see section 5.4.1). The two scripts

```
$LIBOPT_DIR/collections/cuter/bin/libopt_addcell_cuter
$LIBOPT_DIR/collections/modulopt/bin/libopt_addcell_modulopt
```

and their associated templates

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

can be used as examples to derive other (script, template) pairs adapted to other collections.

We can now list the points that have to be done to complete the installation of the collection *`coll`*. These points will also be listed by the command `libopt addcollection`.

- Decide how the problems must be installed in the collection directory and do this installation.
- Fill in the list `all.lst`, which provides all the problems of the collection *`coll`*.
- Fill in the list `default.lst`, which provides a subset of the problems of the collection representing a typical (and not too large) sample. This list is chosen by some scripts when no list is specified as one of their arguments. Therefore, it can be a symbolic link to the list `all.lst`, obtained using the Unix/Linux command '`ln -s all.lst default.lst`'. However, if the collection contains many problems, it is probably better the put in `default.lst` a not too long list of problems reflecting the collection.

- Other lists of problems can also be defined in the directory *coll*. They will be considered as *subcollections* of problems.
- When this is possible, it is nice to write a template

  $LIBOPT_DIR/collections/*coll*/templates/SOLV_*coll*

  and the script

  $LIBOPT_DIR/collections/*coll*/bin/libopt_addcell_*coll*

  that uses this template to generate the *solv_coll* scripts (see the comments above).
- If this is appropriate, write the scripts

  $LIBOPT_DIR/collections/*coll*/bin/libopt_addproblem_*coll*
  $LIBOPT_DIR/collections/*coll*/bin/libopt_rmproblem_*coll*

  that are respectively used by the Libopt command `libopt addproblem` and `libopt rmproblem` to add/remove a problem to/from the collection *coll* (see section 5.2.3).

5.2.2 Removing a collection

A collection can contain precious scripts and a huge amount of data. By removing a collection we do not mean removing this information, but making it concealed from Libopt. By the command

% libopt rmcollection [-v] *coll*,

you simply ask Libopt to remove the name "*coll*" of the collection from the list

$LIBOPT_DIR/collections/.collections.lst.

Then Libopt becomes unaware of the existence of the collection *coll*. If you really want to remove all the pieces of information describing the collection *coll* from the disk, you have to remove yourself the directory $LIBOPT_DIR/collections/*coll* and its contents.

5.2.3 Adding/removing a problem to/from a collection

Suppose that one wants to add a new problem, say *prob*, to an installed collection *coll*. Libopt has the following command to partly help its users to do this (it is recommended to use the option -v to be informed of what this command does):

% libopt addproblem [-v] -c *coll* -p *prob*.

The Libopt commands are designed to work independently of any collection of problems and any solver. Clearly, adding a problem to a collection is very dependent on the organization of the considered collection. For this reason, after having verified that *coll* is a valid collection, the `libopt addproblem` command hands over to a script that must be provided by the specified collection, namely

$LIBOPT_DIR/collections/*coll*/bin/libopt_addproblem_*coll*.

If this script exists, it is launched with the name of the problem in argument (and the option `-v` if it is present in the `libopt addproblem` subcommand). Otherwise, nothing is done. This last script, when *coll* is the Modulopt collection, is described in [9].

A command also exits for removing a problem from a collection. It reads

```
% libopt rmproblem [-v] -c coll -p prob.
```

Like for the `addproblem` subcommand, because the Libopt commands are designed to work independently of any collection of problems and any solver, the `rmproblem` subcommand essentially hands over to a script that must be provided by the specified collection, namely

```
$LIBOPT_DIR/collections/coll/bin/libopt_rmproblem_coll.
```

If this script exists, it is launched with the name of the problem in argument (and the option `-v` if it is present in the `libopt rmproblem` subcommand). Otherwise, nothing is done. This last script, when *coll* is the Modulopt collection, is described in [9].

5.2.4 The CUTEr collection

Libopt is a layer covering the CUTEr collection [1, 12] and other collections of problems. Therefore, CUTEr must already be installed before being hooked to Libopt; if this is not the case, see [12] to know how to proceed. Make sure that the following environment variables have been set:

- `CUTER`: it specifies the head directory of the CUTEr hierarchy,
- `MASTSIF`: it specifies the directory containing the CUTEr problems,
- `MYCUTER`: it specifies the name of the CUTEr sub-directory, containing material adapted to your platform.

Then the use of the CUTEr collection should be straightforward, since there is nothing else to do. Indeed, the standard distribution of Libopt provides the directory `$LIBOPT_DIR/collections/cuter`, which contains some lists of problems `*.lst` and the directories `bin` and `templates`. Note that the scripts present in the standard distribution of Libopt offer the possibility to store the problem files (those in `$MASTSIF`) in uncompressed or `gzip`-compressed format.

If the installation of the CUTEr collection is quite simple, this does not mean that it can readily be used. Indeed, the solvers installed in Libopt have now to be prepared to run CUTEr problems. This has already been achieved for the solvers in the standard distribution, but some adjustments may be necessary for a new solver; this is discussed in section 5.4.2.

Let us conclude this section by a few notes on how we have installed CUTEr in Libopt. We have not introduced a script

```
$LIBOPT_DIR/collections/cuter/bin/libopt_addproblem_cuter,
```

since adding a problem to CUTEr has little interaction with the Libopt environment. Libopt refers to the CUTEr problems, only through the `MASTSIF` environment variable and the lists of problems `*.lst`. If CUTEr was modified, these elements would have to be updated and a script for doing this looks rather difficult to conceive. On the other hand, we have introduced the (script, template) pair

```
$LIBOPT_DIR/collections/cuter/bin/libopt_addcell_cuter
$LIBOPT_DIR/collections/cuter/templates/SOLV_cuter,
```

which helps activating a cell (*solv*, `cuter`), as discussed in section 5.4.2.

5.2.5 The Modulopt collection

The Modulopt collection is distributed with Libopt, but its full description goes out of the scope of this manual; the paper [9] is dedicated ot this task. Two companion collections, named *Modulopttoys* and *Moduloptmatlab*, have also been hooked to Libopt. They have the same features as Modulopt, except that the problems have an academic nature (for Modulopttoys) and are written in Matlab (for Moduloptmatlab).

The Modulopt collection directory

```
$LIBOPT_DIR/collections/modulopt
```

contains the following items:

- lists of problems `*.lst`,
- the directory `doc`, which contains some documentation,
- the directories `bin` and `templates`, as announced in section 5.2.1,
- the directory `probs`, which is aimed at receiving the problems of the collection; each of which, say *pnam*`[.`*pdat*`]`, has a subdirectory named *pnam*.

The (script, template) pair

```
$LIBOPT_DIR/collections/modulopt/bin/libopt_addcell_modulopt
$LIBOPT_DIR/collections/modulopt/templates/SOLV_modulopt,
```

helps the `libopt addcell` command to activate a cell (*solv*, `modulopt`). This feature is discussed in section 5.4.3. The script

```
$LIBOPT_DIR/collections/modulopt/bin/libopt_addproblem_modulopt
```

is launched when one enters the `libopt addproblem` command discussed in section 5.2.3. This one uses the templates

```
$LIBOPT_DIR/collections/modulopt/templates/Makebin
$LIBOPT_DIR/collections/modulopt/templates/Makeclean
$LIBOPT_DIR/collections/modulopt/templates/Makefile.
```

The script

```
$LIBOPT_DIR/collections/modulopt/bin/libopt_rmproblem_modulopt
```

is launched by the `libopt rmproblem` command discussed in section 5.2.3.

5.3 Managing solvers

5.3.1 Installing a new solver

A new solver *solv* is introduced in the Libopt environment by entering the Unix/Linux command (add the option `-v` to be informed of what this command does)

```
% libopt addsolver solv
```

In addition to doing some verifications, in order to check and maintain the consistency of the environment, this command adds the name *solv* of the solver to the list

`$LIBOPT_DIR/solvers/.solvers.lst`,

makes the directories

`$LIBOPT_DIR/solvers/`*solv*,
`$LIBOPT_DIR/solvers/`*solv*`/doc`,

and creates the files

`$LIBOPT_DIR/solvers/`*solv*`/.collections.lst`,
`$LIBOPT_DIR/solvers/`*solv*`/doc/`*solv*`_features`.

The first file, `.collections.lst`, is empty and will be completed by the `libopt addcell` command; the user does not have to take care of it. On the other hand, the second file, *solv*`_features`, must be tuned to give information on the solver *solv* that will help `libopt addcell` to generate other scripts automatically. The comments in that file provide help to set the directives correctly.

With this installation alone, the solver *solv* cannot access any collection. To be able to run *solv* on problems of the collection *coll*, the cell (*solv*, *coll*) of the Cartesian product (3.1) must be completed. This is the subject of the section 5.4.

5.3.2 Removing a solver

A solver directory can contain precious scripts and pieces of software. By removing a solver we do not mean removing that directory and its contents, but making it concealed from Libopt. By the command

```
% libopt rmsolver [-v] solv,
```

you simply ask Libopt to remove the name "*solv*" of the solver from the list

`$LIBOPT_DIR/solvers/.solvers.lst`.

Then Libopt becomes unaware of the existence of the solver *solv*, even though the solver directory is still in the Libopt hierarchy. If you really want to remove all the pieces of information describing the solver *solv* from the disk, as well as its interfaces with the installed collections, you have to remove yourself the directory `$LIBOPT_DIR/solvers/`*solv* and its contents.

5.4 Activating a (solver, collection) cell

We have seen in section 5.2 that hooking a collection of problems to Libopt is quite simple and, in section 5.3, that installing a solver can often be completely automated. However, these installations do not allow a newly installed solver *solv* to run a problem of an installed collection *coll*. This is because the directory associated with the *cell* (*solv*, *coll*) of the {solvers} × {collections} Cartesian product (3.1), namely $LIBOPT_DIR/solvers/*solv*/*coll*, has not been filled in. Figure 3 on page 16 shows what this directory must contain. This section describes how to fill in this directory. We also say that the cell (*solv*, *coll*) is *activated*.

The Libopt environment has a command to help you activating a cell: `libopt addcell`. It is discussed in section 5.4.1. An important goal of the `libopt addcell` command is, when this is possible, to install the script *solv_coll* in the appropriate directory. For the collections installed in the standard distribution, this script is generated by transforming the template SOLV_*coll* located in the directory $LIBOPT_DIR/collections/*coll*/templates. For other collections, this script has to be written by mimicking one of the templates SOLV_*coll*. We describe in section 5.4.2 and 5.4.3 respectively, the structure of these scripts when the collection is CUTEr or Modulopt.

5.4.1 The `libopt addcell` command

The cell (*solv*, *coll*) is activated by running the Unix/Linux command

```
% libopt addcell -s solv -c coll [-v]
```

The option `-v` is recommended to be informed on the details of what this command does. In addition to doing some verifications, in order to check and maintain the consistency of the environment, this command adds the name *coll* of the collection to the list

$LIBOPT_DIR/solvers/*solv*/.collections.lst,

makes the directory

$LIBOPT_DIR/solvers/*solv*/*coll*,

creates empty lists

$LIBOPT_DIR/solvers/*solv*/*coll*/all.lst
$LIBOPT_DIR/solvers/*solv*/*coll*/default.lst,

and tries to generate the important *solv_coll* script

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

If any if the elements above is already present in the environment, the `libopt addcell` command will not replace it. However, if the *solv_coll* script is already present, the command will ask whether this one should be regenerated; if the answer is positive, the previous version of the script will be saved under a suitable and specified name. Therefore, the `libopt addcell` command can be used several times, for example each time the features of the solver *solv* given in the file

`$LIBOPT_DIR/solvers/`*`solv`*`/doc/`*`solv`*`_features`

are modified (the contents of this feature file is used by `libopt addcell`).

The role of the lists of problems `all.lst` and `default.lst` has been described in section 3.2. Let us be more specific.

- The file `all.lst` must list the problems from the collection *`coll`* that *`solv`* can *potentially* solve or, more precisely, those for which it has been conceived. The easiest way of doing this is to start with a copy of the file

  `$LIBOPT_DIR/collections/cuter/all.lst`,

  which lists all the *`coll`* problems, and to remove from the copied file those problems that do not have the structure expected by *`solv`*. For example, if *`solv`* is a solver of unconstrained optimization problems, remove from the copied file `all.lst`, all the problems with constraints. Note that other lists might exist in the directory `$LIBOPT_DIR/collections/`*`coll`*, which might be more appropriate to start with than the list `all.lst`.

- The list `default.lst` can contain any subset of the problems listed in the first file (`all.lst`). This file is used as the default subcollection when no list is specified in the `libopt run` command. Therefore, it is often a symbolic link to the first file `all.lst`, obtained using the Unix/Linux command

  ```
  ln -s all.lst default.lst
  ```

  in the directory `$LIBOPT_DIR/solvers/`*`solv`*`/`*`coll`*.

Specifying other problem lists in the same directory can be appropriate.

The role of the *`solv`*`_`*`coll`* script has been described in section 4.3.3. Generating it is a rather complex operation, since it is a priori linked to a particular solver and collection. Actually, we have observed that it is its dependence on the collection that is the most important, at such a point that it can often be generated by a command that is provided by the collection, whose location/name is supposed to be

`$LIBOPT_DIR/collections/`*`coll`*`/bin/libopt_addcell_`*`coll`*.

When this command exists, `libopt addcell` runs it, otherwise the task of writing the *`solv`*`_`*`coll`* script is left to the user. We further discuss the `libopt_addcell_`*`coll`* commands for the CUTEr and Modulopt collections in sections 5.4.2 and 5.4.3 respectively. These ones use the templates

`$LIBOPT_DIR/collections/`*`coll`*`/templates/SOLV_`*`coll`*.

Depending on the collection *`coll`*, it might be necessary to install other files in the directory

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

When the Modulopt collection is involved, the directory must also contain the source code of the main program and a makefile that gathers all the pieces, associated with the solver and the problem, to form an executable program. For the CUTEr collection, this main program and the way of running it are parts of the CUTEr environment.

5.4.2 Interfacing a solver with the CUTEr collection

We suppose in this section that

- the CUTEr collection has been installed in the Libopt environment, which assumes that CUTEr is also installed on your platform (see section 5.2.4),
- the solver *solv* has been introduced in the Libopt environment (see section 5.3),
- the cell (*solv*, `cuter`) has been activated, using the `libopt addcell` command (section 5.4.1).

We now want to be more specific about the generation of the *solv*`_cuter` scripts by the command

`$LIBOPT_DIR/collections/cuter/bin/libopt_addcell_cuter`,

which uses the template

`$LIBOPT_DIR/collections/cuter/templates/SOLV_cuter`.

The `libopt_addcell_cuter` script considers two cases, depending on the fact that solver is written in Matlab or not. This fact is detected by reading the `language` directive in the file

`$LIBOPT_DIR/solvers/`*solv*`/doc/`*solv*`_features`.

If the solver is not written in Matlab, its associated `sdname`, say `sd`*solva* (it must start with the characters `sd`), is also search in this file. This is the name of the script that decodes a SIF file and then runs the solver in the CUTEr environment.

If the solver is written in Matlab, the lines of the `SOLV_cuter` template starting with `#-MATLAB` are deleted and those starting with the directive `#+MATLAB` are inserted (without the directive). The opposite is true if the solver is not written in Matlab: the lines starting with `#+MATLAB` are deleted and those starting with the directive `#-MATLAB` are inserted (without the directive).

The case of a solver written in Matlab

This case is somewhat simpler, since the solver need not be installed in the CUTEr environment before its installation in Libopt. The activation of the (`fmincon`, `cuter`) cell (`fmincon` is the SQP solver of Matlab) in the standard distribution can be used as an example for other (*solv*, `cuter`) cell activations.

The *solv*`_cuter` script generated by `libopt_addcell_cuter` achieves the following tasks. We assume that the CUTEr problem name is *prob*.

1. It is checked that the problem exists in the `$MASTSIF` directory in uncompressed (*prob*`.SIF`) or compressed (*prob*`.SIF.gz`) form and an uncompressed version of the problem is made available in the working directory (it will be a symbolic link to the problem in the `$MASTSIF` directory if this one exists in uncompressed form there).
2. The Unix/Linux command `sdmx`, located in the directory `$MYCUTER/bin`, is used to decode the SIF file *prob*`.SIF` and then to create the Matlab MEX-file for the CUTEr tools.

3. A symbolic link to the main program *solv*_cuter_main.m (see below) is made in the working directory.
4. The problem *prob* is solved, using Matlab in batch mode.
5. The files generated during execution in the working directory are removed.

   Be sure that the Matlab command mex called in

```
$MYCUTER/bin/sdmx
$MYCUTER/bin/mx
```

is well defined in these files by the variables MEXFORTRAN and MEXFFLAGS.

The *solv*_cuter script assumes that a main program, named *solv*_cuter_main.m, is available in the directory

```
$LIBOPT_DIR/solvers/solv/cuter
```

and that it contains nested functions calling the CUTEr simulators in a form adapted to the solver.

The case of a solver not written in Matlab

If the solver *solv* is not written in Matlab, it must be first installed in the CUTEr environment (see the CUTEr manual [12] for the procedure). This implies, in particular, that a driver or main program *solv*ma (or a similar name) has been inserted in the CUTEr environment and that the command sd*solva* can be used, as in

```
% sdsolva prob,
```

where sd*solva* is the sdname associated with the solver *solv* (it depends on the developer choice) and *prob* is the name of a CUTEr problem.

The *solv*_cuter script generated by libopt_addcell_cuter performs the following tasks.

1. It gets the problem file *prob*.SIF in the working directory, either by making a symbolic link to the file $MASTSIF/*prob*.SIF or by uncompressing the file $MASTSIF/*prob*.SIF.gz, depending on the way it is stored.
2. It solves the problem using the Unix/Linux command

   ```
   % sdsolva prob,
   ```

   with possible additional options.
3. It removes from the working directory, the files that have been generated during the run.

You should be able to launch your solver *solv* on a CUTEr problem, using the Libopt command

```
% libopt run "solv cuter prob".
```

This one should give the same result as when one enters "sd*solva prob*".

5.4.3 Interfacing a solver with the Modulopt collection

We suppose in this section that

- the Modulopt collection has been installed in the Libopt environment, which is presently the case in the standard distribution of the environment,
- the solver *solv* has been introduced in the Libopt environment (see section 5.3),
- the (*solv*, `modulopt`) cell has been activated, using the `libopt addcell` command (section 5.4.1).

As announced in section 5.4.1, the `libopt addcell` command hands over to the script

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

that is provided by the Modulopt collection. The latter uses the template

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

to generate the *solv*`_modulopt` script. Very few modifications are brought to the template: replacement of `<SOLV>`/`<COLL>` occurrences by the actual name of the solver/collection and addition of Perl lines to remove the files listed by the `outfiles` directive mentioned in the file

`$LIBOPT_DIR/solvers/`*solv*`/doc/`*solv*`_features`.

It is likely that nothing will have to be modified in this script to make it work as desired.

The *solv*`_modulopt` script assumes that the following tasks are performed. They cannot be automated. More details are given in [9].

1. It is necessary to design the main program (*scr* is the suffix identifying the program language)

   `$LIBOPT_DIR/solvers/`*solv*`/modulopt/`*solv*`_modulopt_main.`*src*.

   This main program is very solver dependent. It has to get information from the Modulopt problem *prob* selected by the `libopt run` command (dimensions, data, initialization, *etc*) and to launch the solver *solv*. This program will be linked with the archive describing the Modulopt problem, named *prob*`.a` in the working directory. This one is created by the script `Makebin` and the `Makefile` in the problem directory

   `$LIBOPT_DIR/collections/modulopt/probs/`*prob*.

   If Fortran 90/95 is the adopted language, the easiest way to proceed is to copy and rename one of the files

   `$LIBOPT_DIR/solvers/SOLV/modulopt/SOLV_modulopt_main.f90`,

   where `SOLV` is some solver connected to Modulopt. Since this main program is very solver dependent, its part dealing with the solver will have to be thoroughly modified.

2. It is also necessary to create the makefile

   `$LIBOPT_DIR/solvers/`*solv*`/modulopt/Makefile`.

Its goal is to tell the Libopt environment how to link the solver binary with the archive *prob*.a describing the selected Modulopt problem. Again, the easiest way of doing this is to start with an existing makefile, like

$LIBOPT_DIR/solvers/SOLV/modulopt/Makefile.

These tasks achieve the activation of the cell (*solv*, modulopt).

You should now be able to launch the command

```
% libopt run -v "solv modulopt prob"
```

where the option -v (verbose) is used to get detailed comments from the Libopt scripts, which then tell what they actually do. The option -t (test mode) can be used instead, if you want to see what the scripts would do without asking them to do it.

6 Discussion and perspective

In this manual, we have presented the version 2.1 of the Libopt environment and have motivated its main features. The software has been designed to make more pleasant the tedious, 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 user-friendly 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 (which is presently rather embryonic, see section 5.1), to the efficiency of the *solv_coll* script generation from templates (see section 4.3.3), 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 [16] or for downloading part of it), to the implementation of other measures of performance [16, 17], to the uniformization of the solver stopping criteria (see [6] and the Examiner tool in [8]), 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 [13] are present in the CUTEr and Schittkowski [18] 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 [4], 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 {solvers} $\times$ {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.

References

[1] I. Bongartz, A.R. Conn, N.I.M. Gould, Ph.L. Toint (1995). CUTE: Constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software*, 21, 123–160. 5, 34

[2] A. Brooke, D. Kendrick, A. Meeraus (1988). *GAMS: A User's Guide*. The Scientific Press, San Francisco, CA, USA. 7

[3] A.R. Conn, N.I.M. Gould, D. Orban, Ph.L. Toint (2003). The SIF Reference Report (revised version). Online report. 5, 7

[4] COPS. http://www-unix.mcs.anl.gov/~more/cops/. 42

[5] E.D. Dolan, J.J. Moré (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91, 201–213. 12, 13, 26

[6] E.D. Dolan, J.J. Moré, T.S. Munson (2006). Optimality measures for performance profiles. *SIAM Journal on Optimization*, 16, 891–909. 42

[7] R. Fourer, D.M. Gay, B.W. Kernighan (2003). *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole-Thomson Publishing Company, Pacific Grove, CA, USA. 7

[8] GAMS (2003). *The Solver Manuals*. GAMS Development Corporation, Washington, DC, USA. 42

[9] J.Ch. Gilbert (2007). Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 2.1. Technical Report 329 (revised), INRIA, BP 105, 78153 Le Chesnay, France. 8, 10, 15, 24, 34, 35, 41

[10] J.Ch. Gilbert, X. Jonsson (2008). LIBOPT – An environment for testing solvers on heterogeneous collections of problems. Submitted to *ACM Transactions on Mathematical Software*. 6

[11] J.Ch. Gilbert, C. Lemaréchal (1989). Some numerical experiments with variable-storage quasi-Newton algorithms. *Mathematical Programming*, 45, 407–435. 28

[12] N. Gould, D. Orban, Ph.L. Toint (2003). CUTEr (and SifDec), a Constrained and Unconstrained Testing Environment, revisited. *ACM Transactions on Mathematical Software*, 29, 373–394.
http://hsl.rl.ac.uk/cuter-www/interfaces.html. 5, 15, 34, 40

[13] W. Hock, K. Schittkowski (1981). *Test Examples for Nonlinear Programming Codes*. Lecture Notes in Economics and Mathematical Systems 187. Springer-Verlag, Berlin. 42

[14] C. Lemaréchal (1980). Using a Modulopt minimization code. Unpublished Technical Note. 5, 15

[15] MATHWORKS (2008). The Matlab distributed computing engine. http://www.mathworks.com. 7

[16] H.D. Mittelmann, A. Pruessner (2003). A server for automated performance analysis of benchmarking data. Working paper. 10, 42

[17] J.J. Moré, S.M. Wild (2008). Benchmarking derivative-free optimization algorithms. Submitted to *SIAM Journal on Optimization*. 42

[18] K. Schittkowski (2002). Test problems for nonlinear programming – User's guide. Online Paper. 42

[19] SCILAB (2008). The open source platform for numerical computation. http://www.scilab.org. 7

Index

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803