INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 2.1 –*

J. Charles GILBERT

**N° 0329 (revised)**

6 janvier 2009

Thème NUM


ISSN 0249-0803 ISRN INRIA/RT--0329 (revised)--FR+ENG

INRIA
ROCQUENCOURT

# Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 2.1 –

J. Charles Gilbert*

Thème NUM — Systèmes numériques
Projet Estime

Rapport technique n° 0329 (revised) — 6 janvier 2009 — 26 pages

**Abstract:** This note describes how the optimization problems of the Modulopt collection are organized within the Libopt environment. It is aimed at being a guide for using and enriching this collection in this environment.

**Key-words:** benchmarking – collection of problems – Libopt – Modulopt – optimization – testing environment

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

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

# Organisation de la collection de problèmes d'optimisation Modulopt dans l'environnement Libopt – Version 2.1 –

**Résumé :** Cette note décrit comment les problèmes d'optimisation de la collection Modulopt sont organisés dans l'environnement Libopt. Elle a pour but de servir de guide pour utiliser et enrichir cette collection dans cet environnement.

**Mots-clés :** collection de problèmes – environnement de test – évaluation de performance – Libopt – Modulopt – optimisation

## Contents



# 1 The problems of the collection

In the Libopt terminology [3, 4], a *collection* refers to a set of problems sharing some common features, such as their scientific domain, mathematical structure (if any), coding language, audience, etc. In this note, we describe the Modulopt collection [6] and its installation in the Libopt environment. The features of the Modulopt problems, from the Libopt viewpoint, are the following:

- they have an optimization nature and can be written in the form (1) below;
- they can be smooth or nonsmooth;
- they are written in Fortran 90/95;
- they are issued from various application areas in scientific or industrial computing;
- they can be freely distributed.

The collection has two companion ones, named

`modulopttoys` and
`moduloptmatlab`,

which have the same features, except that their problems have an academic nature and that `moduloptmatlab` has its problems written in Matlab [7]. In Libopt, these collections rub shoulders with the *CUTEr collection* [1, 5].

The Modulopt collection contains nonlinear optimization problems coming from various application areas. The optimization problems are supposed to be written in the following form

$$(P) \quad \left\{ \begin{array}{l} \min\ f(x) \\ l_B \le x \le u_B \\ l_I \le c_I(x) \le u_I \\ c_E(x) = 0, \end{array} \right. \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $c_I : \mathbb{R}^n \to \mathbb{R}^{m_I}$, $c_E : \mathbb{R}^n \to \mathbb{R}^{m_E}$, $l_B$, $u_B \in \overline{\mathbb{R}}^n$, and $l_I$, $u_I \in \overline{\mathbb{R}}^{m_I}$ ($\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$). Actually $B$ is the set of indices $\{1, \ldots, n\}$ and $I$ is another set of indices with $m_I$ elements. We write $l := (l_B, l_I) \in \overline{\mathbb{R}}^n \times \overline{\mathbb{R}}^{m_I}$ and $u := (u_B, u_I) \in \overline{\mathbb{R}}^n \times \overline{\mathbb{R}}^{m_I}$. It is assumed that $l < u$, meaning that $l_i < u_i$, for all $i \in B \cup I$. For making the notation compact, we note

$$c_B(x) := x, \quad c(x) := (c_B(x), c_I(x), c_E(x)), \quad \text{and} \quad m := n + m_I + m_E.$$

The Jacobian matrices of $c_I$ and $c_E$ at $x \in \mathbb{R}^n$ are also denoted by

$$A_I(x) := c_I'(x) \quad \text{and} \quad A_E(x) := c_E'(x).$$

We also introduce the nondifferentiable operator $(\cdot)^\# : \mathbb{R}^m \to \mathbb{R}^m$ defined by

$$v^\# = \begin{pmatrix} \max(0, l_B - v_B, v_B - u_B) \\ \max(0, l_I - v_I, v_I - u_I) \\ v_E \end{pmatrix},$$

so that $x$ is feasible for $(P)$ if and only if $c(x)^\# = 0$.

The *Lagrangian* of problem $(P)$ is the function $\ell : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ defined at $(x, \lambda)$ by

$$\ell(x, \lambda) = f(x) + \lambda^\top c(x). \tag{2}$$

Note that we take a single multiplier for two constraints present in the bound constraints $l_i \le c_i(x) \le u_i$, knowing that $l_i < u_i$ implies that at least one of the multipliers associated with $l_i \le c_i(x)$ and $c_i(x) \le u_i$ is zero. The *optimality conditions* at $\bar{x}$ read for some optimal multiplier $\bar{\lambda}$:

$$\left\{ \begin{array}{lll} \nabla f(\bar{x}) + c'(\bar{x})^\top \bar{\lambda} = 0 & & \\ c(\bar{x})^\# = 0 & & \\ i \in B \cup I,\ l_i < c_i(\bar{x}) < u_i & \Longrightarrow & \bar{\lambda}_i = 0 \\ i \in B \cup I,\ l_i = c_i(\bar{x}) & \Longrightarrow & \bar{\lambda}_i \le 0 \\ i \in B \cup I, \qquad c_i(\bar{x}) = u_i & \Longrightarrow & \bar{\lambda}_i \ge 0. \end{array} \right. \tag{3}$$

# 2 Solving a problem

## 2.1 Notation and relevant directories

We use the following typographic conventions. The `typewriter font` is used for a text that has to be typed literally and for the name of files and directories that exist as such (without making substitutions). In the same circumstances, a generic word, which has to be substituted by an actual word depending on the context, is written in *`italic typewriter font`*. Optional (part of) arguments in a Unix/Linux command line are surrounded by the brackets '`[`' and '`]`'.

Here are some directories of the Libopt hierarchy that will intervene continually in this note. Other important directories and files introduced in this note are listed in section 5.

- `$LIBOPT_DIR`
  is the *head directory* of the Libopt hierarchy (`LIBOPT_DIR` is the Unix/Linux environment variable specifying that directory),
- `$LIBOPT_DIR/collections/modulopt`
  is the *head directory of the Modulopt collection* in the Libopt environment,
- `$LIBOPT_DIR/collections/modulopt/probs`
  is the directory that has a sub-directory for each of the problems of the Modulopt collection installed in the Libopt environment,
- `$LIBOPT_DIR/solvers`
  is the *head directory of the solvers* installed in the Libopt environment.

## 2.2 The `libopt run` script

The simplest way of running a single Modulopt problem in the Libopt environment is by typing ('`%`' is the Unix/Linux prompt)

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

where, here and below,

- *`solv`* stands for the name of a solver installed in the Libopt environment, one of those listed by the command

  ```
  % libopt solvers -x
  ```

  Actually, the solver *`solv`* must also have been prepared to run Modulopt problems, otherwise the `libopt run` command will not be understood by the Libopt environment; this will be the case if an '`x`' appears at the intersection of the '*`solv`*' row and '`modulopt`' column in the output of the `libopt solvers -x` command. See section 4 to know how to make *`solv`* able to solve Modulopt problems.

- *`prob`* stands for the name of a Modulopt problem currently available in the Libopt environment, one in the list

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

The name *prob* of some problems can be composed, formed of two strings separated by a dot like in

*pnam.pdat*

In this case, Libopt only sees the composed name *prob* = *pnam.pdat*, but for Modulopt, the *radical name* *pnam* of the problem is a string used to identify the problem directory and the *data name* *pdat* is a string used to identify one of its data sets. Hence, a problem may have several data sets. In the Modulopt collection, the problem *pnam* is stored in the directory (*pnam* and *prob* are identical if there is no dot in the string, i.e., when there is a single data set)

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

called the *problem directory*. How the data sets are built from the string *pdat* depends on each problem.

By the `libopt run` command above, the optimization solver *solv* is used to solve the Modulopt optimization problem *prob*. Of course *solv* has to be able to solve a problem with the features of *prob* (for example, a solver for unconstrained optimization problems is unable to solve problems with constraints). The solver *solv* keeps in the file

`$LIBOPT_DIR/solvers/`*solv*`/modulopt/all.lst`

the list of the Modulopt problems that it can structurally solve.

See the Libopt manual [4] or the manual page of `libopt` to learn how to run a group of problems with a given solver, using a single command line or a file describing what has to be done.

The directory where the `libopt run` command given above is typed is called the *working directory*. When this is important, the Libopt commands take care that this directory is not in the Libopt hierarchy. If this were the case, there could be a danger of incurable destruction. Indeed, a command like `libopt run` generally removes several files from the working directory after a problem has been solved.

## 2.3 The *solv*`_modulopt` script

By decoding the directive "*solv* `modulopt` *prob*", where *prob* is the string

*pnam*`[.`*pdat*`]`,

the `libopt run` command above knows that it has to launch the following script:

`$LIBOPT_DIR/solvers/`*solv*`/modulopt/`*solv*`_modulopt`

with *prob* in argument. In the standard distribution, *solv*`_modulopt` is a Perl script, but nothing imposes that such a language be used. Such a script has to be written for each solver that wants and is able to solve Modulopt problems. Luckily, this script can be generated from a template (see section 4 for the details). For the while, it is enough to know that it contains the following main steps.

- The environment variables given on the left in the table below are set the value given on the right:

| | |
|---|---|
| MODULOPT_PROB | *prob* |
| MODULOPT_PNAM | *pnam* |
| MODULOPT_PDAT | *pdat* |
| WORKING_DIR | *working directory.* |

These variables can then be used in the scripts and makefiles mentioned below. Actually, the environment variable WORKING_DIR is probably useless since all the Unix/Linux commands in the scripts or makefiles are executed from the working directory (there is no change of directory made in them).

- Then the following Perl script is launched with the argument *pdat*

  $LIBOPT_DIR/collections/modulopt/probs/*pnam*/Makebin.

  The aim of this script is to take care of the data selection/construction and to make symbolic links in the working directory to the source and data files in the problem directory, to produce an archive named *pnam*.a in the working directory, which contains the problem object files allowing the execution of the problem, and finally to remove, from the working directory, the now useless just created symbolic links. This is further explained in section 3.3.

- Next, the Perl script executes the target *solv*_modulopt_main of the following makefile

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

  Its aim is to make, in the working directory, a symbolic link to the source file

  $LIBOPT_DIR/solvers/*solv*/modulopt/*solv*_modulopt_main.f90

  of the main program, to compile it and to link it with the archive *pnam*.a of the Modulopt problem previously generated. This produces the executable file

  *solv*_modulopt_main

  in the working directory. Then the target removes from the working directory the now useless symbolic link *solv*_modulopt_main.f90 and file *solv*_modulopt_main.o.

- The program *solv*_modulopt_main is then executed in the working directory. This one solves the problem *prob* with the solver *solv*.

- Some cleaning is then done in the working directory: *solv*_modulopt_main is removed (probably with other files, depending on the solver) and the following Perl script is launched:

  $LIBOPT_DIR/collections/modulopt/probs/*pnam*/Makeclean

  Its aim is to remove from the working directory, the files related to the problem just solved. See section 3.3 for the details.

# 3 Introducing/removing a problem in/from the collection

## 3.1 The `libopt addproblem` command

Suppose we want to add a new problem named

*prob* or *pnam*`[.`*pdat*`]`

into the Modulopt collection. Libopt has the following command to partly help us to do this (it is recommended to use the option `-v` to have the details on what this command does):

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

Because the Libopt commands are designed to work independently of any collection of problems and any solver, after having verified that `modulopt` is a valid collection, the `libopt addproblem` command hands over to a script that is provided by the Modulopt collection, namely

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

The script is launched with the name of the problem in argument (and the option `-v` if it is present in the `libopt addproblem` subcommand). To be more specific, we now summarize what is realized by this last script.

Of course, this `libopt_addproblem_modulopt` script cannot invent a new problem, but it can help us to do the routine tasks at the Libopt level. This includes the following steps.

- The problem name *prob* = *pnam*`[.`*pdat*`]` (and not *pnam*, if there is a dot in the problem name *prob*) is added to the list

  `$LIBOPT_DIR/collections/modulopt/all.lst`

  of all the Modulopt problems. Therefore, a problem name *pnam* will be present in this file with all its possible data set names *pdat*.

- We have said that the Modulopt collection stores each of its problems in a separate directory. Therefore, the script creates the problem directory

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

  Let us insists on the fact that the directory name is *pnam*, not *pnam.pdat* (if there is a dot in the problem name), since all the data sets of the problem are supposed to be stored in the problem directory.

- Next, the files

  `$LIBOPT_DIR/collections/modulopt/probs/`*pnam*`/Makebin`
  `$LIBOPT_DIR/collections/modulopt/probs/`*pnam*`/Makeclean`
  `$LIBOPT_DIR/collections/modulopt/probs/`*pnam*`/Makefile`

  are generated from the templates with the same names in

  `$LIBOPT_DIR/collections/modulopt/templates`.

The role of these files is explained in section 3.3. To generate them from the templates, `libopt_addproblem_modulopt` uses the file

`$LIBOPT_DIR/collections/modulopt/probs/`*`pnam`*`/links.lst`,
`$LIBOPT_DIR/collections/modulopt/probs/`*`pnam`*`/unlinks.lst`.

The file `links.lst` specifies the names of the files in the problem directory that must be symbolically linked to files in the working directory when the problem is executed; while the file `unlinks.lst` specifies the files related to the problem *`prob`* that must be deleted from the working directory after the problem *`prob`* as been solved. If the file `links.lst` (resp. `unlinks.lst`) does not exist, `Makebin` (resp. `Makeclean`) is not generated. This is necessary the case the first time `libopt addproblem` is run to introduce the problem; hence rerun the command after having introduced the possibly empty files `links.lst` and `unlinks.lst`. This second run will complete the installation of the problem, without destroying what has already been done by the first run.

The `libopt addproblem` command also lists what has to be done manually to complete the installation of the *`prob`* problem into Modulopt. This includes one or more of the following items.

- If this is appropriate, add the name *`prob`* to other lists of problems

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

  such as the one related to unconstrained problems `unc.lst`, quadratic problems `quad.lst`, etc, as well as the list of typical problems of the Modulopt collection `default.lst`. These are `ascii` files. An alpha-numeric order has been adopted, but this feature is not taken into account by the Libopt scripts. *Comments* are possible; they start from the character '`#`' up to the end of the line.

- If a solver called *`solv`* is able to solve a problem like *`prob`*, it may be appropriate to add the name *`prob`* in one or more files among

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

  This assumes that the directory `$LIBOPT_DIR/solvers/`*`solv`*`/modulopt` exists and that the solver has been prepared to solve problems from the Modulopt collection (see section 4 to know how to do this).

- Put in the directory

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

  all the files that define the problem *`prob`*: source files, header files (if appropriate), and data files (if appropriate). This is further described in section 3.2 below.

- Make it clear in

  `$LIBOPT_DIR/collections/modulopt/probs/`*`prob`*`/Makebin`,

  how to generate the data set from the string *`pdat`* and how the main program *`solv`*`_modulopt_main` can have access to this data set. Examples are given in some problem directories. See also section 3.3.

## 3.2 The subroutines defining a Modulopt problem

In principle, the problem can be described in any compiled language, provided the binary files can be gathered into an archive. Below, we assume that the problem is written in `Fortran 95`.

The *problem-independent* makefile

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

assumes that the problem to execute is in the archive `prob.a` in the working directory. On the other hand, the *problem-independent* main program `solv_modulopt_main` assumes that the archive `prob.a` contains seven subroutines: `dimopt`, `initopt`, `simulopt`, `postopt`, `inprodopt`, `ctonbopt`, and `ctcabopt`, which are described below.

In the description of the subroutine arguments, an argument tagged with (I) means that it is an *input* variable, which has to be initialized before calling the subroutine; an argument tagged with (O) means that it is an *output* variable, which only has a meaning on return from the subroutine; and an argument tagged with (IO) is an *input-output* argument, which has to be initialized and which has a meaning after the call to the subroutine. Arguments of the type (O) and (IO) are generally modified by the subroutine and therefore *should not be Fortran constants*!

### The subroutine `dimopt`

The subroutine `dimopt` is called by the main program `solv_modulopt_main` to get the dimensions of the problem. In `Fortran 95`, it has the following calling structure:

```
subroutine dimopt (n, nb, mi, me, nizs, nrzs, ndzs)
```

`n` (O): positive `integer` variable. This is the number $n$ of variables to optimize in the problem, those denoted $x = (x_1, \ldots, x_n)$ in (1).

`nb` (O): nonnegative `integer` variable. This is the number of variables $x_i$ with a lower and/or an upper bound.

`mi` (O): nonnegative `integer` variable. This is the number $m_I$ of nonlinear inequality constraints, of the form $l_i \leq c_i(x) \leq u_i$ $(i = 1, \ldots, m_I)$, for some nonlinear functions $c_i : \mathbb{R}^n \to \mathbb{R}$.

`me` (O): nonnegative `integer` variable. This is the number $m_E$ of nonlinear equality constraints, of the form $c_i(x) = 0$ $(i = 1, \ldots, m_E)$, for some nonlinear functions $c_i : \mathbb{R}^n \to \mathbb{R}$.

`nizs` (O), `nrzs` (O), `ndzs` (O): positive `integer` variables. These are the dimensions of the variables `izs`, `rzs`, and `dzs` (respectively), which are `integer`, `real`, and `double precision` working zones for the Modulopt problem. The solvers must not affect their content. The main program `solv_modulopt_main` associated with the solver `solv` must allocate memory for the variables `izs`, `rzs`, and `dzs` just after having called `dimopt`, see section 4, point 4.1 on page 21. This implies that using `Fortran`

77 is not an appropriate language for writing the main program *solv*_modulopt_main. Note that the value of nizs, nrzs, and ndzs can be zero.

### The subroutine initopt

The subroutine initopt is called to initialize the problem. In Fortran 95, it has the following calling structure:

```
subroutine initopt (pname, n, mi, me, x, lx, ux, dxmin, li, ui,
                    dcimin, infb, tolopt, simcap, info, izs,
                    rzs, dzs)
```

pname (O): character string of length 132, giving the name of the problem.

n (I), mi (I), me (I): dimensions of the problem. Their meaning is given in the description of dimopt.

x (O): double precision array of dimension $n$, providing a starting point for the optimization solver.

lx (O), ux (O): double precision array of dimension $n$, providing the bounds on the variable $x$, if any. If the variables are not subject to bounds (nb is zero on return from dimopt), these variables will not be set and can be ddeclared in the calling program as scalars; if some variables are subject to bounds their must be declared in the calling program with the dimension $n$.

If the variables are subject to bounds, their values $x_i$ are required to satisfy lx($i$) $\leq x_i \leq$ ux($i$), for $i = 1, \ldots, n$. The lower (resp. upper) bound lx($i$) (resp. ux($i$)) is set to -infb (resp. infb) is the bound does not exist; see below for the meaning of infb.

dxmin (O): double precision variable, providing the resolution in $x$ for the $l_\infty$ norm: two points whose distance in $\mathbb{R}^n$ for the *sup*-norm is less than dxmin can be considered as indistinguishable. This data can be used in line-search or trust-region. It is also useful to detect bounds that are active up to that precision.

li (O), ui (O): double precision array of dimension mi $:= m_I$, providing the bounds on the constraint values $c_I(x)$. In other words, $c_i(x)$ is required to satisfy li($i$) $\leq c_i(x) \leq$ ui($i$), for $i = 1, \ldots, m_I$.

dcimin (O): double precision variable, providing the resolution in $c_I$ for the $l_\infty$ norm: two inequality constraint values whose distance in $\mathbb{R}^{m_I}$ for the *sup*-norm is less than dcimin can be considered as indistinguishable. This data can be useful to detect inequality constraints that are active up to that precision.

infb (O): double precision variable, specifying what is the infinite value for the bounds on $x$ and $c_I(x)$. In other words, when lx($i$) $\leq -$infb (resp. li($i$) $\leq -$infb), there is no lower bound on $x_i$ (resp. $c_i(x)$). A similar convention is adopted for the upper bounds.

`tolopt` (O): `double precision` array of dimension 4, providing the tolerances on optimality that a pair $(x,\lambda)$ must satisfied in order to be considered as a solution to the problem. More specifically, the pair $(x,\lambda)$ can be considered as a satisfiable KKT point if

$$\begin{aligned} \|\nabla_x \ell(x,\lambda)\|_\infty &\leq \texttt{tolopt(1)} \\ \|c(x)^\#\|_\infty &\leq \texttt{tolopt(2)} \\ \|\operatorname{sgn}_x(\lambda)\|_\infty &\leq \texttt{tolopt(3)}, \end{aligned}$$

where $\operatorname{sgn}_x(\lambda) \in \mathbb{R}^m$ is defined as follows

$$(\operatorname{sgn}_x(\lambda))_i = \begin{cases} \lambda_i^+ & \text{if } i \in B \cup I \text{ and } x_i \notin [l_i + \texttt{tolopt(2)}, +\infty[ \\ \lambda_i & \text{if } i \in B \cup I \text{ and } x_i \in [l_i + \texttt{tolopt(2)}, u_i - \texttt{tolopt(2)}] \\ \lambda_i^- & \text{if } i \in B \cup I \text{ and } x_i \notin \quad\quad\quad\quad\quad\quad ]-\infty, u_i - \texttt{tolopt(2)}] \\ 0 & \text{if } i \in E. \end{cases}$$

This way of checking optimality will probably be improved in a future version of the collection, in the light of [2].

In (unconstrained) *nonsmooth convex optimization*, convergence is considered to be reached when an $\varepsilon$-subgradient of $f$ with a Euclidean norm less than $\eta$ is obtained, with $\varepsilon = \texttt{tolopt(4)} > 0$ and $\eta = \texttt{tolopt(1)} > 0$. In other words, an $x$ must be found satisfying

$$\forall\, y, \quad f(y) \geq f(x) + \langle g, y - x\rangle - \varepsilon, \quad \text{for some } \|g\| \leq \eta.$$

`simcap` (O): `integer` array of dimension 4. It specifies the simulator capabilities. A negative values means that the related function is not present or that the capability is not considered by the simulator.

- `simcap(1)` $< 0$ the simulator cannot evaluate the cost-function $f$; it may be assumed then that this one is constant (or zero), so that the problem is a feasibility one;
  - $= 0$ the simulator can evaluate the cost-function $f$;
  - $= 1$ the cost-function $f$ is *nonsmooth* (this is the only place where this property of the problem can be detected) and the simulator can evaluate $f$ and a subgradient $g$;
  - $= 2$ the simulator can evaluate the cost-function $f$ and its gradient $g$;
- `simcap(2)` $< 0$ the simulator cannot evaluate the inequality constraint function $c_I$; this is normally because there is no inequality constraints;
  - $= 0$ the simulator can evaluate $c_I$;
  - $= 1$ the simulator can evaluate $c_I$ and its Jacobian $c_I'$;
- `simcap(3)` $< 0$ the simulator cannot evaluate the equality constraint function $c_E$; this is normally because there is no equality constraints;
  - $= 0$ the simulator can evaluate $c_E$;
  - $= 1$ the simulator can evaluate $c_E$ and its Jacobian $c_E'$;
- `simcap(4)` $< 0$ the simulator cannot evaluate $Hv$, the product of the Hessian of the Lagrangian $H := \nabla^2_{xx}\ell(x,\lambda)$ times a vector $v$;

$= 1$ the simulator can evaluate a product $Hv$;
$= 2$ the simulator can evaluate the $H$.

`info` (O): `integer` variable. If negative ($< 0$), *`solv`*`_modulopt_main` should consider that the initialization of the problem by `initopt` has failed and should stop.

`izs`, `rzs`, `dzs` (O): `integer`, `real`, and `double precision` arrays that `initopt` should initialize. These variables are made available to the Modulopt problem. Their dimensions have been provided on return from `dimopt` and they should have been allocated by the main program *`solv`*`_modulopt_main` associated with some code *`solv`*.

### The subroutine `simulopt`

The subroutine `simulopt` is the simulator of the problem. It can be called by *`solv`*`_modulopt_main`, before calling *`solv`*. It is also called by the latter to have information (function and their derivatives) on the problem to solve. In `Fortran 95`, it has the following calling structure:

```
subroutine simulopt (indic, n, mi, me, x, lm, f, ci, ce, g, ai,
                     ae, v, hlv, hl, izs, rzs, dzs)
```

`indic` (IO): `integer` variable monitoring the communication between the solver and the simulator. The simulator `simulopt` recognizes the following values of `indic`.

$= 1$: The simulator can do anything except changing the value of the arguments of `simulopt`. Typically it prints some information on the screen, in a file, or on a plotter. Some solver calls the simulator with this value of `indic` at each iteration.

$= 2$: The simulator is asked to compute the value of the functions $\texttt{f} = f(x) \in \mathbb{R}$ (cost function), $\texttt{ci} = c_I(x) \in \mathbb{R}^{m_I}$ (inequality constraints), and $\texttt{ce} = c_E(x) \in \mathbb{R}^{m_E}$ (equality constraints) at a given point $x$.

$= 3$: The simulator is asked to compute $\texttt{g} = \nabla f(x) \in \mathbb{R}^n$ (gradient of $f$ at $x$ for the Euclidean scalar product), $\texttt{ai} = c_I'(x)$ ($m_I \times n$ Jacobian matrix of $c_I$ at $x$, hence the $(i, j)$ entry of `ai` must be the partial derivative $\partial c_i / \partial x_j$ evaluated at $x$), and $\texttt{ae} = c_E'(x)$ ($m_E \times n$ Jacobian matrix of $c_E$ at $x$).

$= 4$: The simulator is asked to compute $\texttt{f} = f(x)$, $\texttt{ci} = c_I(x)$, and $\texttt{ce} = c_E(x)$ at a given point $x$, as well as the gradient $\texttt{g} = \nabla f(x) \in \mathbb{R}^n$, $\texttt{ai} = c_I'(x)$, and $\texttt{ae} = c_E'(x)$.

$= 5$: The simulator is asked to compute the Hessian of the Lagrangian $H := \nabla^2_{xx} \ell(x, \lambda)$ at the point $(x, \lambda)$.

On the other hand, the simulator `simulopt` can also send a message to the solver, by giving to `indic` one of the following values.

$\geq$ 0: normal call; the required computation has been done.

$= -1$: by this value, the simulator tells the solver that it is impossible or undesirable to do the calculation at the point $x$ given by the solver. The reaction of the solver will vary from one solver to the other.

$= -2$: the simulator asks the solver to stop, for example because some events that the solver cannot understand (not in the field of optimization) has occurred.

`n` (I), `mi` (I), `me` (I): dimensions of the problem. Their meaning is given in the description of `dimopt`.

`x` (I): `double precision` array of dimension $n$, providing the point at which the simulator has to evaluate functions and derivatives.

`lm` (I): `double precision` array of dimension $m$, providing the current value of the dual variable $\lambda$. This one determines, with $x$, the primal-dual variables at which the simulator has to evaluate the Hessian of the Lagrangian or the product of this Hessian with a vector (this depends on the value of `indic`).

`f` (O): `double precision` variable, providing the cost function value $f(x)$ if `indic` $= 2$ or 4 on entry.

`ci` (O): `double precision` array of dimension $m_I$, providing the inequality constraint value $c_I(x)$ if `indic` $= 2$ or 4 on entry.

`ce` (O): `double precision` array of dimension $m_E$, providing the equality constraint value $c_E(x)$ if `indic` $= 2$ or 4 on entry.

`g` (O): `double precision` array of dimension $n$, providing the gradient of the cost function $\nabla f(x)$ if `indic` $= 3$ or 4 on entry.

`ai` (O): `double precision` array of dimension $m_I \times n$, providing the Jacobian matrix of the inequality constraint function $c_I'(x)$ if `indic` $= 3$ or 4 on entry.

`ae` (O): `double precision` array of dimension $m_I \times n$, providing the Jacobian matrix of the equality constraint function $c_E'(x)$ if `indic` $= 3$ or 4 on entry.

`v` (I): `double precision` array of dimension $n$, providing the vector $v$ that multiplies the Hessian of the Lagrangian if `indic` $= 6$ on entry.

`hlv` (O): `double precision` array of dimension $n$, providing the product $Hv$ of the Hessian of the Lagrangian $H$ with a vector $v$ if `indic` $= 6$ on entry.

`hl` (O): `double precision` array of dimension $(n, n)$, providing the Hessian of the Lagrangian $H$ if `indic` $= 7$ on entry.

`izs`, `rzs`, `dzs` (IO): `integer`, `real`, and `double precision` arrays that `simulopt` can use and modify. These variables are made available to the Modulopt problem. Their dimensions have been provided on return from `dimopt` and they should have been allocated by the main program *`solv`*`_modulopt_main` associated with some code *`solv`*.

### The subroutine `postopt`

The subroutine `postopt` is normally called by the main program *`solv`*`_modulopt_main` to allow the problem to provide a post-optimal analysis. Some problems will take advantage

of this opportunity, but most of them won't (they will provide a subroutine with en empty body). The most trivial operation that can be done in this subroutine is to print the solution on the screen. Another possibility is to check second order optimality. The flexibility offered by this subroutine will allow the user of `libopt` to make other job than comparing the effect of using various solvers on his/her problem.

In `Fortran 95`, `postopt` has the following calling structure:

```
subroutine postopt (n, mi, me, x, lm, f, ci, ce, g, ai, ae, hl,
                    izs, rzs, dzs)
```

`n` (I), `mi` (I), `me` (I): dimensions of the problem. Their meaning is given in the description of `dimopt`.

`x` (IO), `lm` (IO): `double precision` arrays of dimension $n$ and $m$ respectively. They provide the primal (`x` $= x$) and dual (`lm` $= \lambda$) variables determined by the solver. They may be modified, since `libopt` will no longer use them.

`f` (IO), `ci` (IO), `ce` (IO), `g` (IO), `ai` (IO), `ae` (IO), `hl` (IO): variables providing the value of $f(x)$, $c_I(x)$, $c_E(x)$, $g(x)$, $A_I(x)$, $A_E(x)$, and $\nabla^2_{xx}\ell(x,\lambda)$ found by the last call to `simulopt` (hence the actual values depend on the capabilities of the simulator and the design of the solver). See the description of `simulopt` for the type and dimension of these variables. These may be modified, since `libopt` will no longer use them.

`izs`, `rzs`, `dzs` (IO): `integer`, `real`, and `double precision` arrays that `postopt` can use and modify. These variables are made available to the Modulopt problem. Their dimensions have been provided on return from `dimopt` and they should have been allocated by the main program *`solv`*`_modulopt_main` associated with some code *`solv`*.

### The subroutines `inprodopt`, `ctonbopt`, and `ctcabopt`

Some optimization solvers can deal with inner product in the variable space $\mathbb{R}^n$ that is different from the *Euclidean inner product*

$$(x,y) \in \mathbb{R}^n \times \mathbb{R}^n \mapsto x^\top y = \sum_{i=1}^{n} x_i y_i.$$

An *inner product* is a map

$$(x,y) \in \mathbb{R}^n \times \mathbb{R}^n \mapsto \langle x,y \rangle \in \mathbb{R}$$

that is symmetric (i.e., $\langle x,y \rangle = \langle y,x \rangle$ for all $x$ and $y \in \mathbb{R}^n$) and positive definite (i.e., $\langle x,x \rangle > 0$ for all nonzero $x \in \mathbb{R}^n$). Such an inner product is a way of rescaling the problem. These solvers must be informed of this inner product and this is the role of the subroutine `inprodopt`. We describe the structure of the subroutine in `Fortran 95`.

```
subroutine inprodopt (n, v1, v2, ip, izs, rzs, dzs)
```

`n` (I): dimension of the vectors whose inner product is going to be taken.

`v1` (I), `v2` (I): `double precision` arrays of dimension $n$. These are the vectors whose inner product is desired.

`ip` (O): `double precision` variable representing the inner product of `v1` and `v2`.

`izs`, `rzs`, `dzs` (IO): `integer`, `real`, and `double precision` arrays that `postopt` can use and modify. These variables are made available to the Modulopt problem. Their dimensions have been provided on return from `dimopt` and they should have been allocated by the main program *`solv`*`_modulopt_main` associated with some code *`solv`*.

Some unconstrained optimization solvers not only need the inner product subroutine `inprodopt` but also subroutines that make a change of coordinates from the *canonical orthogonal basis* of $\mathbb{R}^n$ to some *orthogonal basis* for the inner product $\langle\cdot,\cdot\rangle$. The *canonical orthogonal basis* of $\mathbb{R}^n$ is the set of vectors $\{\hat{e}^i\}_{i=1}^n$, where the $j$th component of $\hat{e}^i$ is equal to $\delta_{ij}$ (the *Kronecker symbol*, which is one when $i=j$ and zero otherwise). If a vector is written $\sum_i x_i\hat{e}^i$ in the canonical basis and $\sum_i y_i e^i$ in the considered orthogonal basis, the subroutine `ctonbopt` gives the coordinates $y := (y_1,\ldots,y_n)$ from $x := (x_1,\ldots,x_n)$ and the subroutine `ctcabopt` gives the coordinates $x$ from $y$.

For example, suppose that

$$\langle u, v\rangle = u^\top M^\top M v,$$

where $M$ is a nonsingular $n\times n$ matrix, such that a linear system with the matrix $M$ is easy to solve (for example $M$ could be triangular). One can take $e^i = M^{-1}\hat{e}^i$, for $1\le i\le n$, since then $\langle e^i, e^j\rangle = (e^i)^\top M^\top M e^j = (\hat{e}^i)^\top\hat{e}^j = \delta_{ij}$. Knowing the coordinates $x := (x_1,\ldots,x_n)$ of a vector in the canonical basis, its coordinates $y := (y_1,\ldots,y_n)$ in the basis $\{e^i\}_{i=1}^n$ can be computed by

$$y_j = \langle\sum_i x_i\hat{e}^i, e^j\rangle = \sum_i x_i\langle\hat{e}^i, M^{-1}\hat{e}^j\rangle = \sum_i x_i(\hat{e}^i)^\top M^\top\hat{e}^j = (Mx)_j.$$

We have shown that $y = Mx$. In that example, the subroutine `ctonbopt` will compute $y = Mx$ knowing $x$, while the subroutine `ctcabopt` will compute $x = M^{-1}y$ knowing $y$.

Here is the description of the subroutines `ctonbopt` and `ctcabopt` in `Fortran 95`. The variables `x` $= x$ and `y` $= y$ have the same meaning as in the discussion above. The parameters `izs`, `rzs`, and `dzs` have the same meaning as in the subroutine `inprodopt`.

```
subroutine ctonbopt (n, x, y, izs, rzs, dzs)
```

```
subroutine ctcabopt (n, y, x, izs, rzs, dzs)
```

Of course, if the inner product implemented in `inprodopt` is the Euclidean inner product, `ctonbopt` will just copy `y` into `x`, while `ctcabopt` will just copy `x` into `y`.

## 3.3 The files describing how to run a Modulopt problem

A problem, whose name is *pnam*`[.`*pdat*`]`, can be stored in an arbitrary manner in its directory

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

Of course, the Libopt environment must be told how to make the information contained in that directory available to the solvers that want to solve the problem. As far as Libopt is concerned, three files (two scripts and a makefile) located in the problem directory suffice: `Makebin`, `Makeclean`, and `Makefile`. These files have been encountered in sections 2.3 and 3.1 and their role and contents is fully described in this section. Recall from section 3.1 that these three files can be largely automatically generated, using the templates with the same name in

`$LIBOPT_DIR/collections/modulopt/templates`.

This file generation is done by the script `libopt_addproblem_modulopt`, itself called by the `libopt addproblem` command. Sometimes, however, these files must be customized, so that understanding what they do is certainly useful.

The script `Makebin` has two goals: it takes care of the data selection or construction and, thanks to `Makefile`, it produces in the working directory an archive, named *prob*`.a`, which contains all the binaries related to the problem. On the other hand, the script `Makeclean` removes from the working directory the files that have been generated before, during, and/or after the problem *prob* is solved.

One must keep in mind that `Makebin`, `Makeclean`, and `Makefile` must be designed in such a way that no file is generated in the problem directory. This is to make sure that several users can use the Libopt environment at the same time.

### The script `Makebin` and the `Makefile`

A *solv*`_modulopt` script can invoke `Makebin` through the command

`% Makebin [-g] [-k] [-t] [-v] [`*pdat*`]`

where the option `-g` asks to put in *prob*`.a` binaries with symbolic debug information, the option `-k` asks to keep any generated file in the working directory (files that become useless at a certain stage of the operations are not removed), the option `-t` asks for a test running mode, meaning that commands are displayed but not executed, the option `-v` asks for a verbose execution of the script. The script contains the following steps.

- The argument *pdat* (if any) is normally used to select or construct the data files corresponding to the problem *pnam*`[.`*pdat*`]`. If it is a selection of data files, symbolic links to the relevant files will be created in the working directory. If it is a construction of data files, these will be placed in the working directory, not in the problem directory in order to preserve the integrity of the Libopt directory contents.
- Next, symbolic links are defined in the working directory towards files that are useful for running the problem. As already said in section 3.1, to generate `Makebin` from a template, the script `libopt_addproblem_modulopt` reads the list

$LIBOPT_DIR/collections/modulopt/probs/*pnam*/links.lst

to get the files that need to be linked.

- In the last step, `Makebin` runs the makefile

  $LIBOPT_DIR/collections/modulopt/probs/*pnam*/Makefile

  with *prob* in argument (the target) and with appropriate flags inherited from those of `Makebin`. The role of this makefile is to build the archive *prob*`.a` with all the binaries related to the problem, including the object files corresponding to the subroutines described in section 3.2. `Makefile` uses the environment variable `LIBOPT_PLAT` to tune the binaries to the correct platform.

#### The script `Makeclean`

After having solved *prob* with some solver, the `libopt run` command removes the files that have been generated in the working directory (unless the option `-k` asks to keep them). Part of these files are directly linked to the problem (the data files, for example). The script `Makeclean` is there to tell the *solv*`_modulopt` script which files to remove. It can be called by

```
% Makeclean [-t] [-v]
```

where the options `-t` and `-v` have the same meaning as for the script `Makebin`.

As already said in section 3.1, to generate `Makeclean` from a template, the script `libopt_addproblem_modulopt` reads the list

$LIBOPT_DIR/collections/modulopt/probs/*pnam*/unlinks.lst

to get the files that need to be removed.

### 3.4 The `libopt rmproblem` command

In Libopt, the counterpart of the command that can add a problem to a collection (`libopt addproblem`, see section 3.1) is the `libopt rmproblem` command, which can be used to remove a problem from a collection. For the Modulopt collection, it reads

```
% libopt rmproblem [-v] -c modulopt -p prob,
```

where *prob* is the name of the problem that has to be removed (the form *pnam*`[.`*pdat*`]` of the problem name can be used instead).

Because the Libopt commands are designed to work independently of any collection of problems and any solver, after having verified that `modulopt` is a valid collection, the `libopt rmproblem` command hands over to a script that is provided by the Modulopt collection, namely

$LIBOPT_DIR/collections/modulopt/bin/libopt_rmproblem_modulopt.

The script is launched with the name of the problem in argument (and the option `-v` if it is present in the `libopt rmproblem` command).

A problem can contain a large amount of programs and data. By removing a problem, the script `libopt_rmproblem_modulopt` will *not* remove that information, but will make it concealed from Libopt. It is the responsibility of the designer of the collection to decide whether the directory containing the problem data really needs to be removed. Actually, `libopt_rmproblem_modulopt` essentially modifies lists of problems and, to be friendly, enumerates the possible modifications that must be made by hand.

Let us summarize what is realized by this script `libopt_rmproblem_modulopt`.

- The problem name *pnam*`[.`*pdat*`]` (and not *pnam*, if there is a dot in the problem name *prob*) is removed from the list

  `$LIBOPT_DIR/collections/modulopt/all.lst`

  of all the Modulopt problems.

- It is then asked whether *prob* has to be removed from all, some, or none of the other lists (those files with the suffix `.lst`) in the directory

  `$LIBOPT_DIR/collections/modulopt`.

  The script acts according to the answer given by the user.

- Next, it is asked whether *prob* has to be removed from all, some, or none of the lists in the existing directories

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

  where *solv* is any of the solvers installed in the environment. The script acts according to the answer given by the user.

The script concludes by enumerating what has to be done manually to complete the removal of the *prob* problem from the Modulopt collection.

# 4 Making a solver able to solve Modulopt problems

In this section, we consider the case when it is desirable to make a solver of optimization problems, installed in the Libopt environment, able to solve problems from the Modulopt collection. Let

*solv*

be the name of the considered solver. We refer the reader to the Libopt manual [4] to learn how to install the solver *solv* in Libopt if this one is not already present in the environment.

In the Libopt terminology, building the interface between *solv* and Modulopt is called *activating* the (*solv*, `modulopt`) cell; the word *cell* refers to an element of the {solvers} × {collections} Cartesian product. The interface is the set of scripts and programs that allows *solv* to solve Modulopt problems. These pieces of software are placed in the *interface directory*

$LIBOPT_DIR/solvers/*solv*/modulopt.

Luckily, there is a command that takes in charge part of the job:

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

where the flag `-s` prefixes the solver name *solv*, the flag `-c` prefixes the collection name `modulopt`, and the option `-v` asks for a verbose running mode (recommended). In addition to doing some verifications, in order to check and maintain the consistency of the Libopt environment, the `libopt addcell` command completes various lists (hidden to the user) and fills in the interface directory.

An important goal of the `libopt addcell` command is to generate the *solv*`_modulopt` script in the interface directory. Since this script is linked to the Modulopt collection, it cannot be generated at the Libopt level. Instead, `libopt addcell` hands over to the generating script

$LIBOPT_DIR/collections/modulopt/bin/libopt_addcell_modulopt

which is provided with the Modulopt collection. This one generates *solv*`_modulopt` by transforming the template

$LIBOPT_DIR/collections/modulopt/templates/SOLV_modulopt,

essentially replacing the occurrences of `<SOLV>` (resp. `<COLL>`) by *solv* (resp. `modulopt`) and adding 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 the *solv*`_modulopt` script to make it work as desired. It is wise to check it, however, recalling that its required contents has been given in section 2.3.

The `libopt addcell` command also specifies what needs to be done by hand. These includes the following points.

1. Fill in the files

   $LIBOPT_DIR/solvers/*solv*/modulopt/all.lst
   $LIBOPT_DIR/solvers/*solv*/modulopt/default.lst.

   - The first file (`all.lst`) must list the problems from the Modulopt collection that *solv* is able to solve or, more precisely, those for which it has been conceived. It can contain *comments*, which start with the '`#`' character and go up to the end of the line. The easiest way of doing this is to start with a copy of the file

     $LIBOPT_DIR/collections/modulopt/all.lst,

     which lists all the Modulopt 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. The features of the Modulopt problems are often specified by comments in the file

`$LIBOPT_DIR/collections/modulopt/all.lst`.

Note that other lists exist in the directory `$LIBOPT_DIR/collections/modulopt`, which might be more appropriate to start with than the list `all.lst`.

- The second file above (`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/collections/modulopt`.

2. Create the main program

   `$LIBOPT_DIR/solvers/`*`solv`*`/modulopt/`*`solv`*`_modulopt_main.f90`.

   This program is very solver dependent and is, with the next step to which it is linked, the most difficult task to realize. It is the main program that will be linked with the subroutines describing the problem from the Modulopt problem selected by the `libopt run` command, those in the archive *`prob`*`.a` (if the selected problem is *`prob`*, see section 3.3). The language used to write this main program is arbitrary, provided it (or its object form generated by some compiler) can be linked with the object files in *`prob`*`.a`.

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

   `$LIBOPT_DIR/solvers/sqppro/modulopt/sqppro_modulopt_main.f90`

   into the file

   `$LIBOPT_DIR/solvers/`*`solv`*`/modulopt/`*`solv`*`_modulopt_main.f90`.

   Since this main program is very solver dependent, its part dealing with the solver will have to be thoroughly modified. Let us describe the structure of the program.

   2.1. After the declaration of variables, the program calls the subroutine `dimopt` to get the dimensions of the Modulopt problem that will be selected by the `libopt run` command. These dimensions are then used to allocate dimension dependent variables, including `izs`, `rzs`, and `dzs`.

   2.2. The problem data are then obtained by calling the subroutine `initopt`. This is the good spot to verify that the features of the problem are compatible with the solver capabilities, using the variable `simcap`.

   2.3. Some optimization solver requires that the simulator be called before launching the optimization. In this case, this is the good spot for doing so, by calling `simulopt`.

   2.4. Next, the program calls the optimization solver *`solv`*, after having initialized its arguments and opened relevant files.

   2.5. Once the optimization has been completed, it is important to write the `libopt` line, which summarizes the performance of the solver *`solv`* on the currently solved Modulopt problem. See the Libopt manual [4] or the `libopt` man page.

2.6. It is nice to let the problem do its post-optimal analysis (if any) by finally calling `postopt`.

Note that a particular solver usually requires a simulator with another structure than the one of `simulopt`. Therefore an interface between `simulopt` and the simulator required by *solv* should be written and placed in the file *solv*`_modulopt_main.f90`.

3. Create the makefile

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

   The aim of this makefile is to tell the Libopt environment how to link the solver binary with the object files describing the Modulopt problem selected by the `libopt run` command. If the latter is *prob*, the corresponding object files will be at link time in the working directory in the archive *prob*`.a` (see section 3.2). The easiest way of doing this is to start with an existing makefile, like

   `$LIBOPT_DIR/solvers/sqppro/modulopt/Makefile`.

   This one will be copied and renamed into the file

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

   and then modified.

   You can now try the command

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

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

## 5 Directories and files

In this section, we list some important directories and files encountered in this note. Recall that `LIBOPT_DIR` is the environment variable that specifies the head directory of the Libopt hierarchy. Below, *solv* is the generic name of a particular solver known to the Libopt environment.

- `$LIBOPT_DIR/collections`:
  directory of the collections of problems the Libopt environment can deal with.
- `$LIBOPT_DIR/collections/.collections.lst`:
  list of collections known to and installed into Libopt.
- `$LIBOPT_DIR/collections/modulopt`:
  head directory of the Modulopt collection in the Libopt environment.
- `$LIBOPT_DIR/collections/modulopt/all.lst`:
  list of all the problems of the Modulopt collection.
- `$LIBOPT_DIR/collections/modulopt/bin`;
  contains scripts that help some libopt commands to add a (*solv*, `modulopt`) cell and to add/remove a problem to/from the Modulopt collection.

- `$LIBOPT_DIR/collections/modulopt/probs`:
  directory containing one sub-directory for each problem of the Modulopt collection.
- `$LIBOPT_DIR/collections/modulopt/templates`;
  contains scripts and makefiles that help some libopt commands to add a (*solv*, `modulopt`) cell and to add a problem to the Modulopt collection.
- `$LIBOPT_DIR/solvers`:
  head directory of the solvers the Libopt environment can deal with.
- `$LIBOPT_DIR/solvers/.solvers.lst`:
  list of solvers known to and installed into Libopt.
- `$LIBOPT_DIR/solvers/`*solv*`/.collections.lst`:
  list of collections the code *solv* has been prepared to deal with.
- `$LIBOPT_DIR/solvers/`*solv*`/modulopt`:
  directory containing the scripts and programs specifying how to run the code *solv* on problems from the Modulopt collection.
- `$LIBOPT_DIR/solvers/`*solv*`/modulopt/all.lst`:
  list of problems from the Modulopt collection, for which the solver *solv* is designed.
- `$LIBOPT_DIR/solvers/`*solv*`/modulopt/`*solv*`_modulopt`:
  Perl script specifying the Unix/Linux commands useful to run the solver *solv* on a single problem of the Modulopt collection.
- `$LIBOPT_DIR/solvers/`*solv*`/modulopt/`*solv*`_modulopt_main.f90`:
  Fortran 90/95 main program that is used to run *solv* on a Modulopt problem selected by a `libopt run` command, say *prob*. This program is linked with the object files (gathered in the archive *prob*`.a` in the working directory) describing *prob*.

# 6 Two companion collections

The Modulopt collection has two companion collections, more or less organized in the same way. They are named

`modulopttoys`
`moduloptmatlab`.

This section describes the differences between them and their mother collection `modulopt`.

## 6.1 The Modulopttoys collection

The Modulopttoys collection is formed of problems having the same features as those given in section 1 for the Modulopt collection, except that the problems have an *academic nature*. We mean by this imprecise term that the problems are motivated by "theoretical" facts (in optimization for instance), rather than their usefulness in some scientific computing or industrial field. Often, they are easier to write, their simulator is faster, but they can raise serious difficulties to be solved.

The Modulopttoys collection is organized in the same way as the Modulopt collection. To have a detailed description of it, just read sections 1 to 5, with `modulopt` substituted by `modulopttoys`. In particular, the collection is located in the Libopt environment in the directory

`$LIBOPT_DIR/collections/modulopttoys`.

## 6.2 The Moduloptmatlab collection

The Moduloptmatlab collection differs from the Modulopt and Modulopttoys collections by the fact that its problems are written in Matlab [7]. The organization of the collection in the Libopt environment is similar to the one of the Modulopt and Modulopttoys collections, in particular, it is located in the directory

`$LIBOPT_DIR/collections/moduloptmatlab`.

The main differences are due to the fact that Matlab is used to describe the problems. It is no longer *subroutines* that describe the problems but *functions*. The various functions described below can get the name of the problem to solver (this should be useless) and the name of its data file from the following environment variables:

| | |
|---|---|
| `MODULOPTMATLAB_PROB` | full problem name |
| `MODULOPTMATLAB_PNAM` | problem directory name |
| `MODULOPTMATLAB_PDAT` | data name. |

The functions below have the same role as the subroutines with the same name described in section 3.2, but must of course be called in a different manner. We refer the reder to section 3.2 for a description of the input/output arguments.

```
[n,nb,mi,me] = dimopt ()
```

```
[x,lx,ux,dxmin,li,ui,dcimin,infb,tolopt,simcap,info] = ...
    initopt ()
```

The simulator can be called by one of the following manners, depending on the value of indic (given in the right side of the box). The output parameter `outdic` has the values that `indic` has on ouput with the Fortran version of `simulopt`.

```
[outdic] = simulopt (indic,x)                          % indic = 1
[outdic,f,ci,ce,g,ai,ae] = simulopt (indic,x)    % indic = 2 : 4
[outdic,hl] = simulopt (indic,x,lm)                    % indic = 5
```

```
postopt (x,lm,f,ci,ce,cs,g,ai,ae,hl)
```

In this version of the collection, we have not assumed that the functions `inprodopt`, `ctonbopt`, and `ctcabopt` are implemented.

## 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. 4

[2] E.D. Dolan, J.J. Moré, T.S. Munson (2006). Optimality measures for performance profiles. *SIAM Journal on Optimization*, 16, 891–909. 12

[3] J.Ch. Gilbert, X. Jonsson (2008). LIBOPT – An environment for testing solvers on heterogeneous collections of problems. Submitted to *ACM Transactions on Mathematical Software*. 3

[4] J.Ch. Gilbert, X. Jonsson (2009). LIBOPT – An environment for testing solvers on heterogeneous collections of problems – The manual, version 2.1. Technical Report 331 (revised), INRIA, BP 105, 78153 Le Chesnay, France. 3, 6, 19, 21

[5] 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. 4

[6] C. Lemaréchal (1980). Using a Modulopt minimization code. Unpublished Technical Note. 3

[7] MATHWORKS (2008). The Matlab distributed computing engine.
http://www.mathworks.com. 4, 24

## 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