# Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 1.0

Jean Charles Gilbert

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

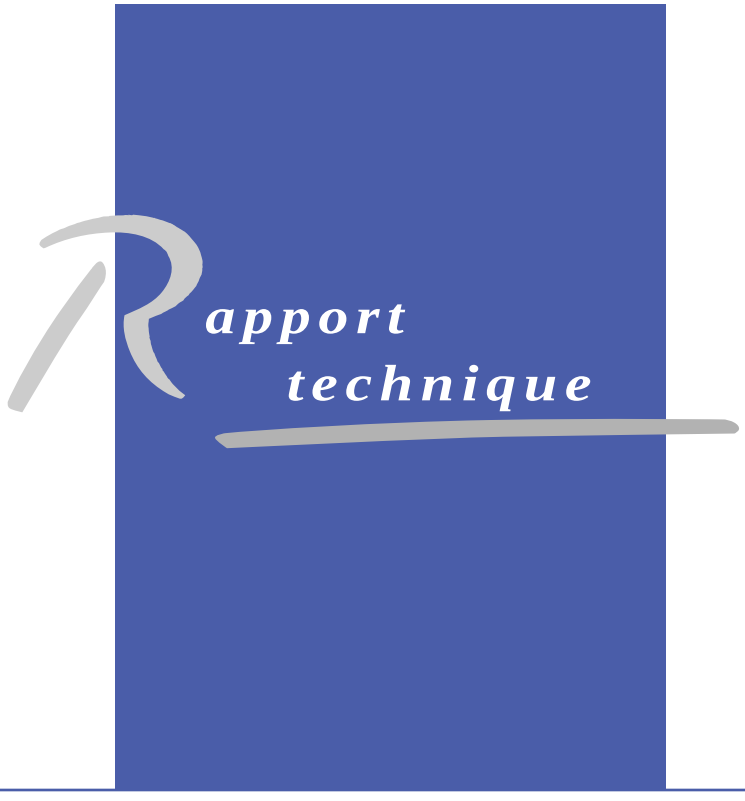
# *Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 1.0 –*

J. Charles GILBERT

**N° ????**

21 février 2007

Thème NUM




ISSN 0249-0803 ISRN INRIA/RT--????--FR+ENG

# Organization of the Modulopt collection of optimization problems in the Libopt environment – Version 1.0 –

J. Charles GILBERT*

Thème NUM — Systèmes numériques
Projet Estime

Rapport technique n° ???? — 21 février 2007 — ?? 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 1.0 –

**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

# 1 The problems of the Modulopt collection

In the Libopt terminology [3], a *collection* refers to a set of problems sharing some common features, such as their mathematical structure, coding language, audience, etc. In this note, we describe the installation of the Modulopt collection [5] 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 a companion one, named "`moduloptttoys`", which has the same features, except that the problems have an academic nature. In Libopt, these collections rub shoulders with the *CUTEr collection* [1, 4].

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

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

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 $l_i \leq c_i(x) \leq u_i$, knowing that $l_i < u_i$ implies that at least one of the multipliers associated with $l_i \leq c_i(x)$ and $c_i(x) \leq u_i$ is zero. The *optimality conditions* at $\bar{x}$ read for some optimal multiplier $\bar{\lambda}$:

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

# 2 Running a Modulopt 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`*.

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. The main directories are

- `$LIBOPT_DIR`
  is the environment variable that specifies the *root directory* of the Libopt hierarchy,
- `$LIBOPT_DIR/collections/modulopt`
  is the *root 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 *root directory of the solvers* installed in the Libopt environment.

## 2.2 The `runopt` script

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

`% echo "`*`solv`* `modulopt` *`prob`*`" | runopt`,

where, here and below,

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

`$LIBOPT_DIR/solvers/solvers.lst`;

actually, the solver *solv* must have been prepared to run Modulopt problems, otherwise this command will not be understood by the Libopt environment; this subject is discussed in section 4;

- *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`.

By this command the optimization code *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 code *solv* keeps in the file

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

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

See [3] or the manual page of `runopt` 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 `runopt` script given above is typed is called the *working directory*. The Libopt scripts 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, a script like `runopt` generally removes several files from the working directory after a problem has been solved.

### 2.3 The *solv*`_modulopt` script

By decoding its standard input "*solv* `modulopt` *prob*", the `runopt` script above knows that it has to launch the following command:

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

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 (actually for each solver-collection pair). Section 4 explains how to do this. For the while, it is enough to know that it is decomposed in the following main steps.

- The environment variables

  `$MODULOPT_PROB` and `$WORKING_DIR`

  are respectively set to *prob* and to the *working directory*, so that these variables can be used in the makefiles mentioned below. Actually, `$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 target *prob* of the following makefile is executed

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

  The aim of this target is to make symbolic links in the working directory to the source and data files in the *prob* directory

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

  to produce an archive named *prob*`.a` in the working directory, which contains the problem object files allowing the execution of the problem, and finally to remove the now useless symbolic links from the working directory.
- 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 a symbolic link in the working directory to the source file

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

  of the main program, to compile it and link it with the archive *prob*`.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 target *prob*`_clean` of the following makefile is executed:

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

  Its aim is to remove from the working directory, the files related to the problem just solved, typically the archive *prob*`.a` and the *prob* data files.

# 3 Introducing a new problem in the Modulopt collection

## 3.1 Overview

Suppose that we want to introduce a new problem in the Modulopt collection and that this one is called

*prob*.

The description of the `runopt` script above shows that, one has to proceed as follows.

- Insert the name *prob* in the list of Modulopt problems

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

  (and possibly in other lists in the same directory, 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`, if this is appropriate). This is an `ascii` file. 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 insert 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).

- Create the directory

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

  and put in that directory, 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.

- Create the makefile

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

  with the following two targets:

  - *prob*, which specifies how to obtain in the working directory an archive with all the object files defining *prob* and which makes symbolic links in the working directory to the data files required to solve the problem;
  - *prob*`_clean`, which specifies which files has to be removed from the working directory after having solved *prob*.

  This is further discussed in section 3.3 below.

## 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, 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).

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 17. 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, inf, 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 $x$. In other words, $x_i$ is required to satisfy $\texttt{lx}(i) \leq x_i \leq \texttt{ux}(i)$, for $i = 1, \ldots, n$. The lower (resp. upper) bound $\texttt{lx}(i)$ (resp. $\texttt{ux}(i)$) is set to `-inf` (resp. `inf`) is the bound does not exist; see below for the meaning of `inf`. Therefore, the arrays `lx` and `ux` must have been declared in the calling program with dimension $n$, even if a solver not dealing with bound constraints is intended to be used.

`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 $\texttt{mi} := m_I$, providing the bounds on the constraint values $c_I(x)$. In other words, $c_i(x)$ is required to satisfy $\texttt{li}(i) \leq c_i(x) \leq \texttt{ui}(i)$, 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.

`inf` (O): `double precision` variable, specifying what is the infinite value for the bounds on $x$ and $c_I(x)$. In other words, when $\texttt{lx}(i) \leq -\texttt{inf}$ (resp. $\texttt{li}(i) \leq -\texttt{inf}$), 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 $\mathrm{sgn}_x(\lambda) \in \mathbb{R}^m$ is defined as follows

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

The tolerance `tolopt(4)` is aimed at being used by minimization software for *nonsmooth functions* and provides a tolerance of the duality gap. This way of checking optimality will probably be improved in a future version of the collection, in the light of [2].

`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 `f` $= f(x) \in \mathbb{R}$ (cost function), `ci` $= c_I(x) \in \mathbb{R}^{m_I}$ (inequality constraints), and `ce` $= c_E(x) \in \mathbb{R}^{m_E}$ (equality constraints) at a given point $x$.

= 3: The simulator is asked to compute `g` $= \nabla f(x) \in \mathbb{R}^n$ (gradient of $f$ at $x$ for the Euclidean scalar product), `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 `ae` $= c_E'(x)$ ($m_E \times n$ Jacobian matrix of $c_E$ at $x$).

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

= 5: The simulator is asked to prepare for subsequent computations of products $Hv$, where $H := \nabla^2_{xx}\ell(x,\lambda)$ is the Hessian of the Lagrangian at $(x,\lambda)$ (see (2)) and $v$ will be an arbitrary vector (see `indic` = 6 below). In some cases, it is convenient to compute the full Hessian of the Lagrangian $H$ when `indic` = 5. In other cases, computing $H$ is too expensive and nothing has to be done when `simul` is called with `indic` = 5.

= 6: The simulator is asked to compute a product $Hv$, where $H := \nabla^2_{xx}\ell(x,\lambda)$ is the Hessian of the Lagrangian at the point $(x,\lambda)$ and $v \in \mathbb{R}^n$ is an arbitrary vector. Note that it is not necessary to evaluate the whole matrix $H$ to compute a product $Hv$; indeed, $Hv$ is also the directional derivative of the gradient of the Lagrangian in the direction $v$:

$$Hv = \frac{d}{dt}\Big(\nabla_x \ell(x+tv,\lambda)\Big)\Big|_{t=0}.$$

= 7: 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 unconstrained optimization solvers can deal with gradients that are associated with an inner product, say $(x, y) \mapsto \langle x, y\rangle$, different from the Euclidean inner product $(x, y) \mapsto x^\top y$. 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 subropuitine `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 makefile describing how to run the Modulopt problems

In the framework introduced in section 3.1, the target ***prob*** of the makefile

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

has to specify how to get in the working directory the archive ***prob***`.a` containing the code of `dimopt`, `initopt`, `simulopt`, `postopt`, `inprodopt`, `ctonbopt`, and `ctcabopt`. It must also create in the working directory symbolic links to the data files (if any) useful for the execution of the problem. Note that this makefile is launched from the working directory, so that '`.`' in the makefile refers to that directory.

The target *prob*_clean of the same makefile, should also clean up the working directory from the files related to the problem just solved. This makefile is also launched from the working directory.

One must keep in mind that the makefile must be organized in such a way that no file is generated in the problem directory $LIBOPT_DIR/collections/modulopt/probs/*prob*. Only the working directory can be modified, so that several users can use the Libopt environment at the same time.

# 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. If the solver does not already exist in the Libopt environment, the name *solv* has to be added to the file

$LIBOPT_DIR/solvers/solvers.lst,

which contains the list of solvers of the Libopt environment, and the following directories must be created

$LIBOPT_DIR/solvers/*solv*
$LIBOPT_DIR/solvers/*solv*/bin,

as well as an empty file

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

which contains the list of collections that the solver *solv* can consider (none if this is the first time *solv* is introduced in the Libopt environment).

We now give the next steps to follow, together with some explanations.

1. Create the directory

   $LIBOPT_DIR/solvers/*solv*/modulopt,

   which will contain the programs/scripts to run *solv* on the Modulopt problems, and add the name modulopt to the list

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

   which indicates that *solv* can deal with the Modulopt collection.

2. Create 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 given in the files

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

  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 `runopt` script. 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`.

3. Create the executable file

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

   This is the script launched by `runopt` to run *`solv`* on a single Modulopt problem. More precisely, this script takes care of the tasks that can be described using Unix/Linux commands, such as making symbolic links, executing makefiles, removing files, etc. This is not an easy program to write from scratch (see [3]) but adapting a script used by another solver is rather straightforward. For example, one can copy, rename, and modify the Perl script

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

   The only changes to make in the copied and renamed script (hence now called *`solv`*`_modulopt`) consists in substituting the two occurrences of the solver name `sqppro` by *`solv`* (one in a comment, another in the definition of the variable `$solvname`). That's all!

4. 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 collection selected by the `runopt` script, those in the archive *prob*`.a` (if the selected problem is *prob*) created by the makefile `$LIBOPT_DIR/collections/modulopt/probs/`*prob*`/Makefile` (see section 3.3). The language used to write this main program is arbitrary, provided it (or its object form generated by some compilor) can be linked with the object files issued from the compilation of the Fortran files describing the Modulopt problems (`dimopt`, `initopt`, `simulopt`, `postopt`, `inprodopt`, `ctonbopt`, and `ctcabopt`).

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.

4.1. 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 `runopt` script. These dimensions are then used to allocate dimension dependent variables, including `izs`, `rzs`, and `dzs`.

4.2. 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`.

4.3. 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`.

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

4.5. 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 [3] or the `libopt` man page.

4.6. 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`.

5. 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 `runopt` script. 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

`echo "`*solv* `modulopt` *prob*`" | runopt -v`

where the flag `-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 root 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.
- `$LIBOPT_DIR/collections/modulopt`:
  root directory of the Modulopt collection in the Libopt environment.
- `$LIBOPT_DIR/collections/modulopt/all.lst`:
  list of all the problems of the Modulopt collection.
- `$LIBOPT_DIR/collections/modulopt/probs`:
  directory containing one sub-directory for each problem of the Modulopt collection.
- `$LIBOPT_DIR/solvers`:
  root 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 `runopt` script, say *`prob`*. This program is linked with the object files (gathered in the archive *`prob`*`.a` in the working directory) describing *`prob`*.

## 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.

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

[3] J.Ch. Gilbert, X. Jonsson (2007). LIBOPT – An environment for testing solvers on heterogeneous collections of problems. Technical report, INRIA, BP 105, 78153 Le Chesnay, France. (to appear).

[4] 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.

[5] C. Lemaréchal (1980). Using a Modulopt minimization code. Unpublished technical Note.

## Index