SQPpro - A solver of nonlinear optimization problems, using an SQP approach

Jean Charles Gilbert

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SQPpro – A solver of nonlinear optimization problems, using an SQP approach

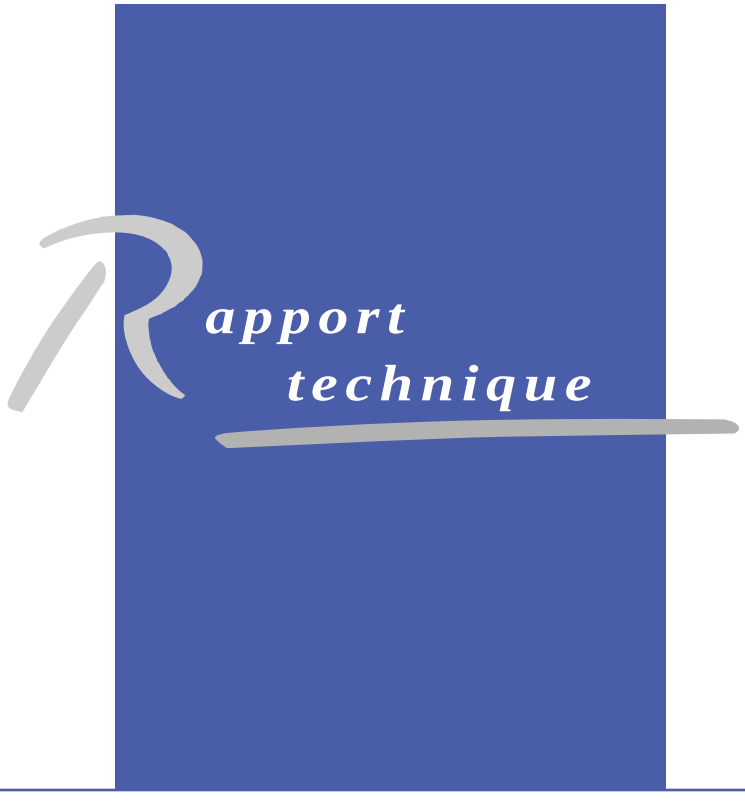
Version 0.5 (June 2009)

J. Charles GILBERT

N° 0378

19 décembre 2009

Thème NUM



ISSN 0249-0803 ISRN INRIA/RT--0378--FR+ENG

SQPpro – A solver of nonlinear optimization problems, using an SQP approach

Version 0.5 (June 2009)

J. Charles Gilbert[†]

Thème NUM — Systèmes numériques
Projet Estime

Rapport technique n° 0378 — 19 décembre 2009 — 24 pages

Abstract: SQPpro is a piece of software that aims at solving a nonlinear optimization problem with nonlinear equality and inequality constraints. The functions defining the problem must be at least once differentiable. The implemented algorithm uses an SQP approach, which is a workable version of the Newton and quasi-Newton methods. The quadratic optimization that has to be solved at each iteration uses the solver QPAL. The constraint Jacobian matrices can be strored in dense or sparse structures; in addition the Hessian of the Lagrangien can be approximated by the BFGS (dense) or ℓ-BFGS (sparse) formula. SQPpro is written in Fortran-2003.

Key-words: BFGS and ℓ-BFGS updates, Newton and quasi-Newton methods, nonlinear optimization, QPAL quadratic optimization solver, SQP algorithm.

[†] INRIA-Rocquencourt, team-project Estime, BP 105, F-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

SQPpro – Un solveur de problèmes d'optimisation non linéaire, fondé sur l'approche SQP

Version 0.5 (Juin 2009)

Résumé : SQPpro est un code destiné à minimiser une fonction non linéaire sous des contraintes non linéaires d'égalité et d'inégalité. Les fonctions définissant le problème doivent être au moins différentiables. L'algorithme implémenté utilise l'approche SQP, qui est une version réalisable des algorithmes de Newton et quasi-Newton. Le problème quadratique qui se pose à chaque itération est résolu par le solveur QPAL. Les matrices jacobiennes des contraintes peuvent être stockées dans des tructures denses ou creuses ; de plus le hessien du lagrangien peut être approché au moyen des formules de BFGS (pleine) ou ℓ-BFGS (creuse). SQPpro est écrit en Fortran-2003.

Mots-clés : algorithme SQP, méthodes de Newton et de quasi-Newton, mises à jour de BFGS et ℓ-BFGS, optimisation non linéaire, solveur de problème quadratique QPAL.

1 Presentation

1.1 Scope of the program

SQPpro (pronounce S-Q-P-pro) has been designed to solve a general nonlinear optimization problem in $x \in \mathbb{R}^n$ of the form

$$(P_{EI}) \quad \begin{cases} \min\ f(x) \\ l \leqslant (x, c_I(x)) \leqslant u \\ c_E(x) = 0, \end{cases} \tag{1.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the *objective* of the problem, $c_I : \mathbb{R}^n \to \mathbb{R}^{m_I}$ is the *inequality constraint* function, and $c_E : \mathbb{R}^n \to \mathbb{R}^{m_E}$ is the *equality constraint* function. These functions are supposed smooth, possibly nonlinear and nonconvex. Smoothness means here that the functions must be once differentiable; first derivatives are indeed used by the solver.

The notation $l \leqslant (x, c_I(x)) \leqslant u$ expresses in compact form *bound constraints* on x and on $c_I(x)$. The bound vectors l and $u \in \bar{\mathbb{R}}^{n+m_I}$ can have components with infinite values $\pm\infty$ (the corresponding bounds are ineffective in that case), but must satisfy $l < u$, meaning that $l_i < u_i$ for all indices i. As a result, the inequality constraints cannot be used to model equality constraints by taking identical lower and upper bounds; the constraint function c_E must be used instead to introduce equality constraints.

SQPpro is flexible with respect to the memory representation of the constraint Jacobian matrices $c_I'(x)$ and $c_E'(x)$. Indeed, these may be *dense* or *sparse*. Sparse matrices are represented by the row-column indices of the nonzero elements, as well as the values of these elements.

To be concise, it is convenient to denote by $B := \{1, \ldots, n\}$ the index set of the variables x and by $c_B(x) \equiv x$ the identity on $\mathbb{R}^n$. We also use the function $c : \mathbb{R}^n \to \mathbb{R}^m$, $m := n + m_I + m_E$, defined at $x \in \mathbb{R}^n$ by

$$c(x) := (c_B(x), c_I(x), c_E(x)) \in \mathbb{R}^n \times \mathbb{R}^{m_I} \times \mathbb{R}^{m_E}.$$

The *feasible set* of problem (P_{EI}) is denoted

$$X := \{x \in \mathbb{R}^n : l \leqslant c_{B\cup I}(x) \leqslant u,\ c_E(x) = 0\}.$$

A point x belonging to X, hence satisfying the constraints of problem (P_{EI}) is said *feasible*.

SQPpro is written in ANSI Fortran 2003 (F03 for short), using double precision. The code uses this feature of F03 that makes possible calling a subroutine with a structure argument having components that are not yet allocated when the subroutine is called. SQPpro compiles with `gfortran` [5], version "`4.4.0 20090321 (experimental)`" or higher.

1.2 Detecting optimality

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

$$\ell(x, \lambda) = f(x) + \lambda^\top c(x).$$

This one is useful to write the KKT *optimality conditions* of problem (P_{EI}) (see [2] for instance). If x is a *solution* to (P_{EI}) and if the constraints are qualified at x, there exists a vector $\lambda \in \mathbb{R}^m$ such that:

$$\left\{\begin{array}{ll} \text{(a)} & \nabla_x \ell(x, \lambda) = 0 \\ \text{(b)} & l \leqslant (x, c_I(x)) \leqslant u \quad \text{and} \quad c_E(x) = 0 \\ \text{(c)} & \forall i \in B \cup I : \quad \lambda_i^-(l_i - c_i(x)) = \lambda_i^+(c_i(x) - u_i) = 0, \end{array}\right. \tag{1.2}$$

where $t^+ := \max(t, 0)$ and $t^- := \max(-t, 0)$ for $t \in \mathbb{R}$ and $()^+$ and $()^-$ act componentwise for vectors. In (c), infinite bounds are replaced by large numbers of the same sign, so that λ_i must be nonnegative when $l_i = -\infty$ and u_i is finite. The first condition refers to the *proper optimality*, the second one to the *feasibility*, and the third one is known as the *complementarity conditions*. The components of the vector λ in this equation are called the *optimal KKT multipliers* or the *dual solutions* or the *marginal costs*.

The SQPpro solver cannot guarantee to find a local minimum of problem (P_{EI}) (even less a global minimum), but is designed to find a *stationary point*, which is a pair $(x, \lambda) \in \mathbb{R}^n \times \mathbb{R}^m$ satisfying (1.2). Often, a stationary point is a solution to (P_{EI}).

1.3 Brief description of the method

SQPpro implements a sequential quadratic programming algorithm, using an augmented Lagrangian approach for solving the osculating quadratic problems and linesearch for its globalization. It is therefore a *primal-dual algorithm*, meaning that it generates the primal variables x and the dual variables λ, independently.

One iteration of the SQP algorithm consists in solving the *osculating quadratic problem* (QP), which at the current iterate $(x, \lambda) \in \mathbb{R}^n \times \mathbb{R}^m$ reads (we drop the dependence of the functions in (x, λ) for more concision):

$$\left\{\begin{array}{l} \min\limits_{d\in\mathbb{R}^n} \ g^\top d + \dfrac{1}{2} d^\top M d \\ \tilde{l} \leqslant (d, A_I d) \leqslant \tilde{u} \\ c_E + A_E d = 0, \end{array}\right. \tag{1.3}$$

where g stands fo the gradient $\nabla f(x)$, M is an approximation of the Hessian of the Lagrangian $L := \nabla^2_{xx}\ell(x,\lambda)$, $A_I := c'_I(x)$, $A_E := c'_E(x)$, $\tilde{l} = l - c_{B\cup I}(x)$, and $\tilde{u} := u - c_{B\cup I}(x)$. It is classical to impose the positive semi-definiteness of M (even though L does not have that property), in order to avoid a QP that, otherwise, would be NP-hard. The osculating QP is then convex. In SQPpro, the solution to this QP is obtained by the solver QPAL (version 0.6.1) [3, 6]. If $(d, \lambda^{\rm QP})$ is a primal-dual solution to (1.3), the new iterate (x_+, λ_+) is obtained by

$$x_+ = x + \alpha\, d \quad \text{and} \quad \lambda_+ = \lambda + \alpha\,(\lambda^{\rm QP} - \lambda),$$

where $\alpha \in]0, 1]$ is a stepsize determined by linesearch.

The approximation $M \in \mathbb{R}^{n\times n}$ of the Hessian of the Lagrangian or its inverse (for unconstrained problems) is a matrix updated by the BFGS or ℓ-BFGS formula. The Jacobian matrices A_I and A_E can be stored in dense or sparse data structures.

The SQP algorithm is described in part III of [2], where other references can be found. The combination of the SQP algorithm and the augmented Lagrangian approach for solving the osculating quadratic problems is described in [4], where it is also applied to a seismic reflection tomography problem, and studied in [3],

1.4 The package

1.4.1 Description

The SQPpro package is formed of the files and directories described below. In this description, `$PLAT` is the value of the environment variable that designates the platform for which the software has to be compiled; see the description of the file `make.a_platform` below and stage 2 of the installation procedure in section 1.4.2.

- The files `COPYRIGHT.*` give the conditions of use of the software. You are supposed to agree with these conditions to be authorized to use it. If this has not been done yet, send a filled in and signed copy of the commitment letter `COPYRIGHT.pdf` to the author of the software.
- The directory `bin` is originally empty and will contain the binaries gathered in libraries, named `lib*.a`, after compilation of SQPpro:
  - `libblas.$PLAT.a` contains the BLAS routines used in SQPpro, those in the directory `blas` of the QPAL solver; this archive can be used if the compiler does not provide a BLAS library;
  - `liblapack.$PLAT.a` contains the LAPACK routines used in SQPpro, those in the directory `lapack` of the QPAL solver; this archive can be used if the compiler does not provide a LAPACK library;
  - `libqpal.$PLAT.a` contains the routines peculiar to the QP solver QPAL [6], which is used in SQPpro;
  - `libsqppro.$PLAT.a` contains the routines peculiar to SQPpro, those in the directory `src` (see below).
- The directory `cuter` contains all the files that are useful to install SQPpro in CUTEr [1, 8]; see how to proceed in the `README` file of the `cuter` directory.
- The directory `doc` contains several interesting files:
  - `VERSIONS.txt` briefly describes the successive versions of the software;
  - `doc.pdf` is this documentation in PDF;
  - `sqppro.spc` is an example of specification file for the solver, containing explanations on how to build such a file.
- The directory `example` gives elementary examples of nonlinear optimization problem, that are solved by SQPpro. The goal of these examples is to make concrete the way of encoding the problem and to call the SQP solver.

- The directory `libopt` contains all the files that are useful to install SQPpro in the Libopt environment [7]; see how to proceed in the `README` file of the `libopt` directory.
- The file `make.a_platform` is an example of `make.$PLAT` file that is used by the `Makefile`'s of the package to compile various programs. You will probably need to adapt this file to your own platform, by redefining some of its variables. Stage 2 in section 1.4.2 gives the details.
- The directory `mod` is originally empty and will contain, after compilation of SQPpro, the following module descriptors:
  - `sqppro_mod.$PLAT.mod` for the `sqppro_mod` module, which provides, in particular, the description of the public derived types discussed in section 2.1.
- The directory `src` contains the peculiar routines of the SQPpro solver and a Makefile to compile them.

1.4.2 Installation

The QP solver used by SQPpro is QPAL (version 0.6.1) [6]. The QPAL solver must therefore be installed somewhere in your hierarchy before you can use SQPpro. Once this has been completed, the installation of SQPpro can be done by following the stages given below.

1. Normally, the SQPpro package is distributed as a tarball named

   ```
   SQPPRO-xxx-distrib.tar.gz
   ```

   where `xxx` stands for a version number. Place this tarball in a directory where you want to keep SQPpro. Decompress and untar it using

   ```
   tar -zxvf SQPPRO-xxx-distrib.tar.gz
   ```

   This creates the directory `SQPPRO-xxx-distrib`, which can be renamed. Below, we call it the `sqppro` *directory* and denote it by

   `sqppro_directory`

2. The second stage deals with platform matters. To know the chosen compiler, linker and their options, the Makefile's of the SQPpro package include a file named `make.$PLAT`, located in *`sqppro_directory`*. Here, `$PLAT` is the value of an environment variable defined by a Unix/Linux command similar to

   ```
   setenv PLAT mach.os.comp
   ```

   The string "`mach.os.comp`" above is arbitrary; the proposed form allows you to identify the machine type "`mach`", its operating system "`os`", and the chosen compiler suite "`comp`"; examples might be

   ```
   setenv PLAT pc.linux.pgf90
   setenv PLAT mac.osx.gcc
   ```

   The file "`make.a_platform`" in *`sqppro_directory`* is an example of such a `make.$PLAT` file and, now that the environment variable `PLAT` has been defined, you may want to adapt it to your platform by first copying it

   ```
   cp make.a_platform make.$PLAT
   ```

   and next adapting the following variables, which are the only ones used in the SQPpro `Makefile`'s:

   ```
   F03        = # name of the Fortran 2003 compiler
   F03DBGFLAG = # option requiring generating code for debugging
   F03FLAGS   = # usual options to use with $(F03)
   ```

```
F03MDIR    = # option introducing a module directory
F03NOLD    = # option preventing from making a load object
```

3. The third stage consists in specifying the location of the QPAL solver. This is done by making in *sqppro_directory* a symbolic link to the `qpal` *directory*, named *qpal_directory* below:

   ```
   cd sqppro_directory
   ln -s qpal_directory qpal
   ```

4. You are now ready to compile the software. Go in the directory `src` and type

   ```
   make
   ```

   As said above, this command places object libraries in the directory `bin` and module descriptors in the directory `mod`. These can now be linked to a program that uses SQPpro as a nonlinear optimization solver. A good starting point could be to run the examples in the directory `exemple`.

2 Usage

We start in section 2.1 by specifying the data structures of the SQPpro solver, the Fortran derived types used to define the data exchanged with the solver. In section 2.2, we describe the problem simulator that is expected by SQPpro, which is the part of the program that evaluates the value of the functions defining problem (P_{EI}) and their derivatives. The role and the arguments of the subroutines `sqppro_allocate`, `sqppro_default_options`, and `sqppro_solve`, which are the three entry points into the solver, are described in section 2.3. In section 2.3.4, we give the typical sequence of statements that must precede a call to the solver.

2.1 Data structures

The SQPpro solver is structured as a Fortran module, named `sqppro_mod`, with its private/public derived type definitions, data, functions, and subroutines. The entry point to the solver is `sqppro_solve`. The associated subroutine `sqppro_solve` gathers its arguments in structures (having a given Fortran derived type), in order to make clearer the links between them and to make easier passing them from one procedure to the other. The public derived types used to define the arguments are described below.

Theses public derived types have allocatable components that are allocated by the subroutine `sqppro_allocate`; see section 2.3.1. You will be able to use variables defined by these types only after having called that subroutine. Useless components are not allocated.

We remind the reader that the value of the component `c` of a variable `v` of a given derived type is given by `v%c`.

2.1.1 Data types

SQPpro can store the Jacobians of the inequality and equality constraints in dense or sparse matrices. This section describes the Fortran public derived types that have been defined to store these various data structures.

We start with the public derived type for storing sparse matrices.

```
type, public :: sqppro_sparse_type
```

```
  sequence
  integer                          :: nnz
  integer, allocatable             :: i(:), j(:)
  double precision, allocatable :: v(:)
end type sqppro_sparse_type
```

With the keyword `sequence`, the components are stored in the specified order (otherwise, no storage sequence is implied by the order of the component definitions). As a result, a structure with a type identical to `sqppro_sparse_type` (also with the keyword `sequence`) but with a different type name will be correctly identified with a structure of type `sqppro_sparse_type`.

Here is a description of the components.

- `nnz`: this `integer` specifies the number of nonzero elements of the matrix.
- `i(:)` and `j(:)`: the `integers` `i(k)` and `j(k)`, for `k` $= 1, \ldots,$ `nnz`, are the row and column indices of the `k`th nonzero element of the matrix.
- `v(:)`: the `double precision` value `v(k)`, for `k` $= 1, \ldots,$ `nnz`, gives the value of the `k`th nonzero element of the matrix.

We now introduce the public derived type named `sqppro_hessian_type`, which describes the memory representation of the Hessian of the Lagrangian, knowing that this one can be a dense BFGS matrix, a direct ℓ-BFGS matrix, or an inverse ℓ-BFGS matrix.

```
type, public :: sqppro_hessian_type
  integer                          :: id
  double precision, allocatable :: dense(:,:)
  type(lbfgs_dir_hessian_type)  :: lbfgs_dir
  type(lbfgs_inv_hessian_type)  :: lbfgs_inv
end type sqppro_hessian_type
```

Description of the components.

- `id`: this `integer` component specifies the *Hessian identificator*, which describes the type of storage SQPpro has to provide for the Hessian of the Lagrangian. The following 3 values can be used, named by public parameters:
  - `dense` informs SQPpro that the Hessian of the Lagrangian must be stored in the component `dense`;
  - `lbfgs_dir` informs SQPpro that the Hessian of the Lagrangian is a *direct* ℓ-BFGS matrix that must be stored in the data structure `lbfgs_dir`;
  - `lbfgs_inv` informs SQPpro that the Hessian of the Lagrangian is an *inverse* ℓ-BFGS matrix that must be stored in the data structure `lbfgs_inv`.
- `dense`: this `allocatable double precision` array of dimension `(n,n)` is used to store the Hessian when `id` = `dense`.
- `lbfgs_dir`: this memory structure of type `lbfgs_dir_hessian_type` is used to store a direct ℓ-BFGS matrix, when `id` = `lbfgs_dir`. The memory structure is defined in the module `modulopt_lbfgs_mod`.
- `lbfgs_inv`: this memory structure of type `lbfgs_inv_hessian_type` is used to store an inverse ℓ-BFGS matrix, when `id` = `lbfgs_inv`. The memory structure is defined in the module `modulopt_lbfgs_mod`.

The next derived type is used to describe the memory representation of the inequality and equality constraint Jacobians, $c'_I(x)$ and $c'_E(x)$ respectively. These Jacobians can be dense or sparse.

```
type, public :: sqppro_constraint_type
  integer                          :: id
  double precision, allocatable :: dense(:,:)
  type(sqppro_sparse_type)         :: sparse
end type sqppro_constraint_type
```

Description of the components.

- `id`: this `integer` component specifies the *constraint Jacobian identificator*. The following 2 values can be used, named by public parameters:
  - `dense` informs SQPpro that the jacobian is a dense matrix that must be stored in the component `dense`;
  - `sparse` informs SQPpro that the jacobian is a sparse matrix that must be stored in the component `sparse`.
- `dense`: this `allocatable double precision` array of appropriate dimension is used to store the constraint Jacobian when `id` = `dense`.
- `sparse`: this memory structure of type `sqppro_sparse_type` is used to store a sparse constraint Jacobian when `id` = `sparse`.

2.1.2 Problem data

The public derived type `sqppro_data_type` must be used to define a structure whose aim is to give data on problem (P_{EI}) on entry in `sqppro_solve` (section 2.3.3) and during the run at the generated iterates. Its allocatable parts are allocated by `sqppro_allocate` (section 2.3.1).

```
type, public :: sqppro_data_type
  integer                          :: n, nb, mi, me
  double precision                 :: f
  double precision, pointer        :: g(:)
  type(sqppro_hessian_type)        :: h
  double precision, allocatable :: lb(:), ub(:)
  double precision, allocatable :: li(:), ui(:), ci(:)
  type(sqppro_constraint_type)  :: ai
  double precision, allocatable :: ce(:)
  type(sqppro_constraint_type)  :: ae
end type sqppro_data_type
```

Here are the components of this type.

- `n`, `nb`, `mi`, `me`: these `integer`s give respectively the number n of variables, the number of bounded variables (those with $l_i > -$`inf` and/or $u_i <$ `inf`, see the description of the `sqppro_options_type` type in section 2.1.3 for the meaning of `inf`), the number m_I of nonlinear inequality constraints (those modelled by c_I), and the number m_E of equality constraints (those modelled by c_E).

- `f`: this `double precision` variable aims at receiving the value $f(x) \in \mathbb{R}$ of the objective f at a given primal iterate x.
- `g(:)`: this variable points to a `double precision` array used to store the gradient $g(x) = \nabla f(x) \in \mathbb{R}^n$ of the objective of (P_{EI}) at a given primal iterate x.
- `h`: this variable of `sqppro_hessian_type` type is aimed at receiving the approximation of the Hessian of the Lagrangian at a given primal-dual iterate (x, λ).
- `lb(:)`, `ub(:)`: these allocatable `double precision` arrays are used to store the vectors l_B and $u_B \in \mathbb{R}^n$ giving respectively the lower and upper bounds on the variables x.
- `li(:)`, `ui(:)`, `ci(:)`, `ai`: these `double precision` arrays and `sqppro_constraint_type` variable are related to the nonlinear inequality constraints. The lower and upper bounds $l_I \in \mathbb{R}^{m_I}$ and $u_I \in \mathbb{R}^{m_I}$ are stored in `li` and `ui`, respectively. The other variables are aimed at receiving the value at a given primal iterate x of the nonlinear inequality constraint function c_I (in `ci`) and of the Jacobian $c_I'(x)$ of c_I (in `ai`). More precisely, assuming that $I = \{1, \ldots, m_I\}$, for $\mathtt{i} \in I$ and $\mathtt{j} \in B$, there hold

$$\mathtt{ci(i)} = c_{\mathtt{i}}(x), \quad \mathtt{li(i)} = l_{n+\mathtt{i}}, \quad \mathtt{ui(i)} = u_{n+\mathtt{i}},$$

  while a dense inequality constraint Jacobian is stored in

$$\mathtt{ai\%dense(i,j)} = \frac{\partial c_{\mathtt{i}}(x)}{\partial x_{\mathtt{j}}}$$

  and, if $(c_I'(x))_{ij}$ is the `k`th element of a sparse inequality constraint Jacobian, it is stored by

$$\mathtt{ai\%sparse\%i(k)} = \mathtt{i}, \quad \mathtt{ai\%sparse\%j(k)} = \mathtt{j}, \quad \mathtt{ai\%sparse\%v(k)} = \frac{\partial c_{\mathtt{i}}(x)}{\partial x_{\mathtt{j}}}.$$

- `ce(:)`, `ae`: these `double precision` array and `sqppro_constraint_type` variable are aimed at receiving the value at a given primal iterate x of the nonlinear equality constraint function c_E (in `ce`) and of the Jacobian $c_E'(x)$ of c_E (in `ae`). More precisely, assuming that $E = \{1, \ldots, m_E\}$, for $\mathtt{i} \in E$ and $\mathtt{j} \in B$, there hold

$$\mathtt{ce(i)} = c_{\mathtt{i}}(x),$$

  while a dense equality constraint Jacobian is stored in

$$\mathtt{ae\%dense(i,j)} = \frac{\partial c_{\mathtt{i}}(x)}{\partial x_{\mathtt{j}}}$$

  and, if $(c_E'(x))_{ij}$ is the `k`th element of a sparse equality constraint Jacobian, it is stored by

$$\mathtt{ae\%sparse\%i(k)} = \mathtt{i}, \quad \mathtt{ae\%sparse\%j(k)} = \mathtt{j}, \quad \mathtt{ae\%sparse\%v(k)} = \frac{\partial c_{\mathtt{i}}(x)}{\partial x_{\mathtt{j}}}.$$

The dimensions `n`, `mi`, and `me` are set by `sqppro_allocate`, the constant components (independent of the current iterates x) `lb`, `ub`, `li`, and `ui` are set by the user after having called `sqppro_allocate` (see section 2.3.4). The other components are filled in by the *simulator* (see section 2.2).

As Fortran-2003 pointers, `g`, `ai`, and `ae` are just other names for variables used to describe the corresponding data of the osculating quadratic problem (1.3); these variables are inherited from variables allocated by the QP solver QPAL.

2.1.3 Solver options

The public derived type `sqppro_options_type` is used to describe the options of the SQPpro solver, i.e., those parameters that can be used to tune the behavior of the solver. A variable of that type can receive the default options of the solver by calling `sqppro_default_options` (section 2.3.2).

```
type, public ::  sqppro_options_type
  integer          :: fout, plevel
  integer          :: max_iter
  integer          :: qp_max_alit, qp_max_cgit, qp_max_hvpd, qp_max_avpd
  character(len=3) :: qp_norm
  integer          :: qp_precision_id
  double precision :: qp_tol_glan, qp_tol_feas
  double precision :: qp_forcing_factor
  double precision :: qp_feas_decr_factor
  double precision :: kkt_tol(3)
  double precision :: inf, dxmin, dcmin
end type sqppro_options_type
```

The components of this type are gathered above by Fortran type and nature. For making their localization faster, they are presented below in alphabetic order. Valid and default values are indicated.

- `dcmin`: positive `double precision` variable that is used to detect active bounds on $c_I(x)$. To this respect, there must also hold

  $$\forall i \in I : \quad u_i - l_i > 2\,(\texttt{dcmin}).$$

  This value also intervenes in the complementarity conditions.
  Valid values: > 0. *Default value:* 10^{-10}.

- `dxmin`: positive `double precision` variable that specifies the precision to which the primal variables must be determined. If `sqppro_solve` needs to make a step smaller than `dxmin` in the infinity-norm to progress to optimality, it will stop. To this respect, a too small value for `dxmin` will force the solver to work for nothing at the very end when rounding errors prevent making any progress.
  The value of `dxmin` is also used to detect active bounds on x. To this respect, there must also hold

  $$\forall i \in B : \quad u_i - l_i > 2\,(\texttt{dxmin}).$$

  This value also intervenes in the complementarity conditions.
  Valid values: > 0. *Default value:* 10^{-10}.

- `fout`: `integer` variable that is taken as the channel number for the outputs, i.e., these are written by:

  ```
  write (fout,...) ...
  ```

  Valid values: > 0. *Default value:* 6 (screen).

- `inf`: `double precision` variable specifying what is the infinite value for the bounds l_i and u_i. More precisely, if $l_i \leqslant -$`inf` (resp. $u_i \geqslant$ `inf`), the ith lower (resp. upper) bound is assumed to be absent.
  Valid values: > 0. *Default value:* `huge(1.d0)`.
- `kkt_tol`: `double precision` array of dimension 3, which provides the tolerances on the KKT conditions (1.2) for detecting a solution (more precisely a stationary point). An iterate (x, λ) is indeed considered to be a satisfactory approximate primal-dual solution to problem (P_{EI}) if

$$\begin{aligned} \texttt{info\%kkt\_glan}(x,\lambda) &\leqslant \texttt{kkt\_tol(1)} \\ \texttt{info\%kkk\_feas}(x) &\leqslant \texttt{kkt\_tol(2)} \\ \texttt{info\%kkk\_cmpl}(x,\lambda) &\leqslant \texttt{kkt\_tol(3)}, \end{aligned}$$

  where `info%kkt_glan`(x, λ), `info%kkk_feas`(x), and `info%kkk_cmpl`(x, λ) have been defined by (2.1), (2.2), and (2.3), respectively. Hence `kkt_tol(1)` controls the proper optimality (1.2)-(a), `kkt_tol(2)` controls the feasibility (1.2)-(b), and `kkt_tol(3)` controls the complementarity (1.2)-(c).
  Valid values: > 0. *Default value:* 10^{-6}.
- `max_iter`: `integer` variable specifying the maximal number of iterations allowed.
  Valid values: > 0. *Default value:* ∞.
- `plevel`: `integer` variable that specifies the printing level of the solver, i.e., the amount of information that is written on channel `options%fout`. The following values are meaningful:
  $= 0$: nothing is printed; the only manner to be informed of the behavior of `sqppro_solve` is to look at the the argument `info` of the solver (see section 2.1.4 for the description of its components);
  $\geqslant 1$: error and warning messages (default);
  $\geqslant 2$: initial setting and final status;
  $\geqslant 3$: one line per iteration;
  $\geqslant 4$: details on the iterations;
  $\geqslant 5$: details on the step computation are written in a file named `sqppro-qp.txt` in the working directory.
  Valid values: $\geqslant 0$. *Default value:* 1.
- `qp_feas_decr_factor`: `double precision` variable giving the desired decrease factor for the constraint norm in the QP solver QPAL. A small value looks better but it forces QPAL to take a large augmented Lagrangian parameter, inducing ill-conditioning. For a small easy problem, `qp_feas_decr_factor` can be chosen rather small. For a large scale ill-conditioned problem, choose `qp_feas_decr_factor` closer to 1.
  Valid values: $]0, 1[$. *Default value:* 10^{-1}.
- `qp_max_alit`: `integer` variable specifying the maximal number of augmented Lagrangian iterations in any QPAL run.
  Valid values: > 0. *Default value:* 50.
- `qp_max_cgit`: `integer` variable specifying the maximal number of conjugate gradient iterations in any QPAL run.
  Valid values: > 0. *Default value:* 10000.
- `qp_max_hvpd`: `integer` variable specifying the maximal number of Hessian-vector products in any QPAL run.
  Valid values: > 0. *Default value:* 10000.

- `qp_max_avpd`: `integer` variable specifying the maximal number of Jacobian-vector products in any QPAL run.
  Valid values: > 0. *Default value:* 10000.
- `qp_norm`: this string, made of at most 3 characters, specifies the type of vector norm $\|\cdot\|_{\text{QP}}$ that must be used to check optimality of the QP (1.3):
  - `'2'` or `'euc'` for the ℓ_2 or Euclidean norm $\|v\|_2 := (\sum_i v_i^2)^{1/2}$,
  - `'inf'` for the infinity or sup norm $\|v\|_\infty := \max_i |v_i|$.

  Of course, for any vector v, $\|v\|_\infty \leqslant \|v\|_2$, so that for identical tolerances, the QP solver stops more rapidly with the sup norm than with the Euclidean norm.
  Valid values: `'2'`, `'euc'`, `'inf'`. *Default value:* `'inf'`.
- `qp_forcing_factor`: when `qp_precision_id` is set to `variable`, this `double precision` component is used to control the precision on the direction computed by the QP solver. See section 2.4.1 for more information.
  Valid values: $(0, 1)$. *Default value:* 0.5.
- `qp_precision_id`: this `integer` component specifies the *QP precision identificator*, which is used to indicate the method that must be used to control the precision to which the QP's (1.3) must be solved at each iteration. The following 2 values can be used, named by public parameters:
  - `fixed` requires from SQPpro that it imposes to the QP solver to solve the QP's with a precision that is identical at each iteration and that is specified by the `qp_tol_glan` and `qp_tol_feas` components;
  - `variable` requires from SQPpro that it imposes to the QP solver to solve the QP's with a precision that depends on the presision reached on the nonlinear problem (1.1) at the current iteration, using the parameter `qp_forcing_factor` and ensuring nevertheless that the computed direction be a descent direction of the merit function; this strategy is in the spirit of the *truncated Newton method*.

  See section 2.4.1 for more information.
  Valid values: `fixed`, `variable`. *Default value:* `fixed`.
- `qp_tol_glan`, `qp_tol_feas`: when `qp_precision_id` is set to `fixed`, these `double precision` components are used to specify the tolerances on the KKT conditions of the QP (1.3), using the norm $\|\cdot\|_{\text{QP}}$: `qp_tol_glan` refers to the norm of the gradient of the QP Lagrangian and `qp_tol_feas` refers to the norm of the QP constraints.
  Valid values: > 0. *Default value:* 10^{-6}.

2.1.4 Solver diagnostics

The public derived type `sqppro_info_type` contains the information on the problem (P_{EI}) returned by the solver SQPpro.

```
type, public :: sqppro_info_type
  integer          :: nb, iter, nsim, info
  double precision :: kkt_glan, kkt_feas, kkt_cmpl
end type sqppro_info_type
```

Here are the components of this type.

- `nb`: this `integer` variable contains the number of variables x_i with a lower and/or an upper bound (i.e., the number of $i \in B$ with either l_i or u_i finite).

- `iter`: this `integer` variable is the number of iterations performed by the solver.
- `nsim`: this `integer` variable is the number of times the *simulator* has been called.
- `info`: this `integer` variable is the return value of the solver, which specifies the reason why it stopped. Here are the possible values:
  = 0: the stopping criterion is verified.
  = 1: failure in the initialization;
  = 2: an argument is wrong;
  = 5: maximal number of iterations has been reached;
  = 6: maximal number of simulations has been reached;
  = 7: stop required by the simulator;
  = 9: too many stepsize trials in the linesearch;
  = 10: the QP solver has computed a null step, although x is not optimal (strange);
  = 20: maximal number of iterations has been reached in the QP solver;
  = 21: maximal number of matrix-vector products has been reached in the QP solver;
  = 22: nonconvex quadratic problem (1.3);
  = 23: linesearch failure in the QP solver;
  = 24: an infeasible QP has been encountered;
  = 25: if feasible, the quadratic problem (1.3) is likely to be unbounded;
  = 99: abnormal failure, call your guru.

  More information can usually be obtained on a possible failure of the optimization by running the solver with `options%plevel` > 0.
- `kkt_glan`: `double precision` variable giving the ℓ_∞-norm of the gradient of the Lagrangian at the output primal-dual variable (x, λ):

$$\texttt{info\%kkt\_glan}(x,\lambda) := \|\nabla_x \ell(x,\lambda)\|_\infty. \tag{2.1}$$

  This value can be viewed as an optimality measure.
- `kkt_feas`: `double precision` variable giving the infinity norm of the constraint violation at the final primal-dual variable (x, λ):

$$\texttt{info\%kkk\_feas}(x) := \left\| \begin{pmatrix} \max(0, l - c_{B\cup I}(x), c_{B\cup I}(x) - u) \\ c_E(x) \end{pmatrix} \right\|_\infty. \tag{2.2}$$

  The value of `kkt_feas` can be viewed as a feasibility measure.
- `kkt_cmpl`: `double precision` variable measuring the complementarity. It is defined as the minimum of two measures, specifically by

$$\texttt{info\%kkk\_cmpl}(x,\lambda) := \|\min(\texttt{c1}, \texttt{c2})\|_\infty, \tag{2.3}$$

  where the components of the vectors `c1` and `c2` are defined for $i \in B$ by (a similar definition is used for $i \in I$):

$$\texttt{c1}(i) := \begin{cases} \lambda_i^- & \text{if } |l_i - x_i| > \texttt{options\%dxmin} \\ \lambda_i^+ & \text{if } |u_i - x_i| > \texttt{options\%dxmin} \end{cases}$$

  and

$$\texttt{c2}(i) := |l_i - x_i|\lambda_i^- + |u_i - x_i|\lambda_i^+.$$

  Hence, `info%kkk_cmpl` takes, for each component, the best of two complementarity measures. Expression `c1` is based on the fact that if the ith lower [resp. upper] bound is inactive, the associated multiplier should be nonnegative [resp. nonpositive], hence λ_i^- [resp. λ_i^+] should be zero. Expression `c2` is based on the fact that $\lambda_i^-(l_i - x_i)$ and $\lambda_i^+(x_i - u_i)$ should be zero, see (1.2–c).

2.1.5 User data

The public derived type `sqppro_user_type` is used to allow the user to pass information to the simulator through `sqppro_solve`. This structure is not used by the solver; it is not manadged either, meaning that its allocatable variables must be allocated by the user of the solver.

```
type, public :: sqppro_user_type
  integer, allocatable          :: izs(:)
  real, allocatable             :: rzs(:)
  double precision, allocatable :: dzs(:)
end type sqppro_user_type
```

Here are the components of this type.

- `izs`: this `integer` array is dedicated to pass `integer` values to the simulator.
- `rzs`: this `real` array is dedicated to pass `real` values to the simulator.
- `dzs`: this `double precision` array is dedicated to pass `double precision` values to the simulator.

2.2 Simulator

The *simulator* is the part of the program that evaluates the value of the functions defining problem (P_{EI}) and their derivatives. SQPpro get information on the problem to solve through *direct communication*: the simulator receives a message from `sqppro_solve`, which tells what has to be computed, using the flag `indic`; then the simulator fills in parts of the structure `data` and uses `indic` to tell `sqppro_solve` whether the required computation has been reallized.

Here is the subroutine definition statement that is assumed by SQPpro.

```
subroutine simul (indic, x, data, user)
```

`indic`: `integer` variable organizing the communication between the solver `sqppro_solve` and the simulator `simul`.

- On entry, `indic` is used by `sqppro_solve` to tell the simulator `simul` what it has to do.
  - $= 1$: The simulator can do anything except changing the value of x (doing nothing is also fine). Typically it prints some information on the screen, in a file, or on a plotter. The solver `sqppro_solve` calls the simulator with this value of `indic` at each iteration, so that the simulator can also count the iterations.
  - $= 4$: The simulator is asked to compute `data%f` $= f(x)$, `data%ci` $= c_I(x)$, and `data%ce` $= c_E(x)$ at a given point x, as well as the gradient `data%g` $= \nabla f(x) \in \mathbb{R}^n$, and the Jacobian matrices `data%ai` $= c'_I(x)$ and `data%ae` $= c'_E(x)$ at a given point x.
- On return, it contains a message from the simulator to `sqppro_solve`.
  - $\geqslant 0$: 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. In that case, `sqppro_solve` backtracks along the search direction, until the computation can be done.

This feature can be used when *implicit constraints* are present, i.e., strict inequality constraints or inequalities that are known to be inactive at the solution. By no way this feature can handle inequality constraints that are active (satisfied with equality) at the solution.

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

`x` (I): `double precision` array of dimension n. It is the vector of primal variables x at which the functions defining (P_{EI}) have to be evaluated. The vector x cannot be modified by the simulator.

`user` (I): this is the same variable as the one with the same name, given as argument of `sqppro_solve`. The solver `sqppro_solve` does not touch it and transmits it to the simulator as an argument of `simul`. See section 2.1.5 for a description of its components.

2.3 Running the solver

The SQPpro solver makes available three subroutines: `sqppro_allocate`, `sqppro_default_options`, and `sqppro_solve`. The subroutine `sqppro_allocate` must be used to allocate variables of a data structure (section 2.3.1). The subroutine `sqppro_default_options` can be used to get the default options of the solver, before tuning these to a particular run (section 2.3.2). The subroutine `sqppro_solve` is used to solve a particular instance of problem (P_{EI}) (section 2.3.3)

In the description of the subroutines, an argument flagged with (I) means that it is an *input* or `intent(in)` variable, which has to be initialized before calling the subroutine; an argument flagged with (O) means that it is an *output* or `intent(out)` variable, which has only a meaning on return from the subroutine; and an argument flagged with (IO) is an *input-output* or `intent(inout)` argument, which has to be initialized and has a meaning on return from the subroutine.

2.3.1 Memory allocation with `sqppro_allocate`

The subroutine `sqppro_allocate` must be called before calling `sqppro_solve`. Its role is to allocate memory to the data structure of the dummy argument `data`, which is aimed at containing data on problem (P_{EI}). It is only after having called `sqppro_allocate` that it will be possible to fill in the variable `data`. This allocation depends on the type of approximation chosen for the Hessian of the Lagrangian (variables `h_type` and `h_mys`); since this one can be specified by a specification file, this file is read and scrutinized by `sqppro_allocate`. The values of `h_type` and `h_mys` in that file prevail on the values given on entry in `sqppro_allocate`.

```
subroutine sqppro_allocate (n, nb, mi, me, &
                            h_type, h_mys, ai_id, ai_nnz, ae_id, ae_nnz, &
                            fout, plevel, flag, data)
```

`n`, `nb`, `mi`, `me` (I): positive `integer` variables specifying the dimensions of the problem (P_{EI}): `n` = n, `mi` = m_I, `me` = m_E, and `nb` is the number of primal variables with a lower or an upper bound (hence `nb` $\in [0, n]$). Actually, the latter must only be vaguely defined: if there is no bound on x, set `nb` = 0, otherwise give `nb` an arbitrary positive (> 0) value. Indeed, `sqppro_allocate` only trusts the sign of `nb` to decide whether memory must be allocated for the bounds on x, i.e., for the variables `data%lb(1:n)` and `data%ub(1:n)`.

`h_type` (I): `integer` variable specifying the *Hessian type*. In the current version, SQPpro can only use approximations of the Hessian of the Lagrangian, either using the BFGS formula

(for small problems, say less than a few hundred variables) or the ℓ-BFGS formula (for larger problems). The type of approximation is decided by `sqppro_allocate` and stored in a variable in the `sqppro_mod` module. The following values are possible:

- `bfgs`: a BFGS approximation of the Hessian of the Lagrangian is generated by the solver,
- `lbfgs`: an ℓ-BFGS approximation of the Hessian of the Lagrangian is generated by the solver; the inverse formula is used if there is no constraint, in which case there is no need to solve an osculating QP (faster); the direct formula is used if there are constraints.

The value of `h_type` given in the specification file prevails on the one given in argument to `sqppro_allocate`.

`h_mys` (I): positive `integer` variable. It is only used in case an ℓ-BFGS Hessian approximation is declared with `h_type`, to specify the number of pairs (y_k, s_k) that are used to form the Hessian approximation.

`ai_id` (I): `integer` variable. It identifies the type of memory space used to store the inequality constraint Jacobian $c_I'(x)$ at a given point x. See the description of the `id` component of the type `sqppro_constraint_type` in section 2.1.1 for the possible values of this variable, which can be `dense` or `sparse`.

`ai_nnz` (I): `integer` variable. It gives the number of nonzero elements in the inequality constraint Jacobians $c_I'(\cdot)$ in case this one is stored in a sparse structure (`ai_id` has been set to `sparse`); it is meaningless otherwise.

`ae_id` (I): `integer` variable. It identifies the type of memory space used to store the equality constraint Jacobian $c_E'(x)$ at a given point x. See the description of the `id` component of the type `sqppro_constraint_type` in section 2.1.1 for the possible values of this variable, which can be `dense` or `sparse`.

`ae_nnz` (I): `integer` variable. It gives the number of nonzero elements in the equality constraint Jacobians $c_E'(\cdot)$ in case this one is stored in a sparse structure (`ae_id` has been set to `sparse`); it is meaningless otherwise.

`fout` (I): `integer` variable. This is the channel number for the written outputs in `sqppro_allocate`.

`plevel` (I): `integer` variable. It specifies the printing level of `sqppro_allocate`. The following values are meaningful:

$\leqslant 0$: silent mode;
> 0: error and warning messages are printed on channel `fout`.

`flag` (O): `integer` variable. It provides information on the allocation process. Possible values are:

$= 0$: allocation done;
$= 1$: allocation was already done in a previous call to `sqppro_allocate`;
$= 2$: QPAL allocation failed;
$= 3$: SQPpro allocation failed;
$= 4$: some of the allocations were already done; allocation is probably correct is the previous allocated variables had the appropriate dimensions.

`data` (O): variable of type `sqppro_data_type` (see section 2.1.2). The useful allocatable components of the variable are allocated by `sqppro_allocate`.

2.3.2 Setting default options with `sqppro_default_options`

The subroutine `sqppro_default_options` can be called to set the default options in a structure of `sqppro_options_type` type, named `options` below. This call is normally made before setting

the various component of `options` to values that are appropriate to the optimization problem to solve. It allows the user not to have to set all the componentws in `options`, when the default ones are appropriate.

The options determined by `sqppro_default_options` and possibly modified after having called that subroutine are overwritten by `sqppro_solve` with the options given in the specification file named `sqppro.spc` in the working directory. An example of *specification file* is provided in

sqppro_directory`/doc/sqppro.spc`

with enough comments to be self-explanatory.

```
subroutine sqppro_default_options (options)
```

`options` (O): variable of type `sqppro_options_type` (see section 2.1.3), which contains the default options.

2.3.3 Solving the problem with `sqppro_solve`

The solver of the SQPpro package is `sqppro_solve`. The list of its arguments is rather short, because these have been gathered in the structures `data`, `info`, `options`, and `user`. Here is the subroutine definition statement.

```
subroutine sqppro_solve (simul, x, lm, data, info, options, user)
```

`simul`: generic name of the user-supplied simulator: see section 2.2 for more details. This name can be modified, but must be declared `external` in the subroutine calling `sqppro_solve`.

`x` (IO): `double precision` array of dimension n. It is the vector of variables $x = (x_1, \ldots, x_n)$ to optimize.

- On entry, it is an initial guess of the solution to (P_{EI}). It is taken as starting point by `sqppro_solve`. This point *need not be feasible*.
- On return, when `info%info` $= 0$ (see section 2.1.4), it is the optimal solution x found by `sqppro_solve`.

`lm` (IO): `double precision` array of dimension $n + m_I + m_E$. It is the dual variable or KKT multiplier associated with the constraints of (P_{EI}). The first n components are associated with the bounds on x; the next m_I components are associated with the inequality constraints $l_I \leqslant c_I(x) \leqslant u_I$; and the last m_E components are associated with the equality constraints $c_E(x) = 0$. The multiplier $\lambda_{B\cup I}$ associated with the bound constraints is actually the difference

$$\lambda_{B\cup I} := \lambda^u_{B\cup I} - \lambda^l_{B\cup I}$$

between the multiplier $\lambda^u_{B\cup I}$ associated with the upper bound and the multiplier $\lambda^l_{B\cup I}$ associated with the lower bound. Since $l_i < u_i$, either λ^u_i or λ^l_i vanishes, or both. Therefore, one can recover $\lambda^u_{B\cup I}$ and $\lambda^l_{B\cup I}$ by

$$\lambda^u_{B\cup I} = \lambda^+_{B\cup I} \quad \text{and} \quad \lambda^l_{B\cup I} = \lambda^-_{B\cup I}.$$

If there is no constraint (no bounds on x, $m_I = 0$, and $m_E = 0$) and `lm` is an allocated array, it may not be allocated (in `gfortran`); `sqppro_solve` will not use it.

- On entry, `lm` is an initial guess of the dual solution to (P_{EI}). This vector can just be initialized to zero, when one has no idea of the dual solution (this is often the case).
- On return, when `info%info` $= 0$ (see section 2.1.4), `lm` is the vector of *optimal multipliers* λ found by `sqppro_solve`.

`data` (IO): this is a variable of Fortran-2003 type `sqppro_data_type`, a derived public type that is defined in the module `sqppro_mod`. It must be used to give data on problem (P_{EI}) on entry in `sqppro_solve` and during the run at the generated iterates. See section 2.1.2 for a description of its components.

`info` (O): this is a variable of Fortran-2003 type `sqppro_info_type`, a derived public type that is defined in the module `sqppro_mod`. The components of `info` contain information on problem (P_{EI}) and on the course of the run. See section 2.1.4 for a description of its components.

`options` (I): this is a variable of Fortran-2003 type `sqppro_options_type`, a derived public type that is defined in the module `sqppro_mod`. The components of `options` are used to tune the behavior of the solver, by giving them a particular value before calling `sqppro_solve`. See section 2.1.3 for a description of its components.

`user` (I): this is a variable of Fortran-2003 type `sqppro_user_type`, a derived public type that is defined in the module `sqppro_mod`. The components of `user` are neither used nor modified by `sqppro_solve`. They are considered as user variables and are transmitted to the simulator. See section 2.1.5 for a description of its components.

2.3.4 Calling sequence

We summarize below the structure of a program that uses `sqppro_solve` to solve a nonlinear optimization problem. We assume for simplicity that all the instructions are in the same program unit.

1. Specification of the use of the SQPpro module `sqppro_mod`:

   ```
   use sqppro_mod
   ```

   If some of the public parameters of `sqppro_mod`, like `bfgs` or `lbfgs`, are in conflict with your own variables, you can rename them. For example to rename `bfgs` into `sqppro_bfgs` and `lbfgs` into `sqppro_lbfgs`, write instead

   ```
   use sqppro_mod, sqppro_bfgs  => bfgs, &
                   sqppro_lbfgs => lbfgs
   ```

2. Declare the variables `n`, `nb`, `mi`, `me`, `h_type`, `h_mys`, `fout`, `plevel`, `flag`, and `data`, and set them (except `data`, which has non-allocated components). Then allocate the useful allocatable components of `data` by

   ```
   call sqppro_allocate (n,nb,mi,me,h_type,h_mys,fout,plevel,flag,data)
   ```

   The parameter `flag` should be nonzero on return from `sqppro_allocate`. See section 2.3.1 for more information.

3. You can now fill in the constant array components of the variable `data` (see section 2.1.2 for a descritption of its components), namely

   ```
   data%lb = ...
   data%ub = ...
   data%li = ...
   data%ui = ...
   ```

These will not be modified by `sqppro_solve`. Note that `data%n`, `data%nb`, `data%mi`, and `data%me` are set in `sqppro_allocate` with the corresponding values given to its arguments.

4. Declare the variable `user` (this is an argument of the solver `sqppro_solve` and the simulator, see sections 2.1.5, 2.2, and 2.3.3). Set it if this is appropriate, i.e., if values must be transmitted to the simulator through the SQPPRO solver.

5. Declare and initialize the primal variable $\texttt{x} = x \in \mathbb{R}^n$ and the constraint multiplier $\texttt{lm} = \lambda \in \mathbb{R}^{n+m_I+m_E}$.

6. It is asked to call the simulator `simul` (section 2.2), before calling the solver `sqppro_solve`, in order to fill in the variable components of `data`:

```
indic = 4
call simul (indic, x, data, user)
```

   The variable `indic` should be $\geqslant 0$ on return from this simulation.

7. Set in `options` the options of the solver that should not have the default value given by `sqppro_allocate`:

```
call sqppro_default_options (options)
options%plevel = 5
...
```

   The options that have been set in the specification file `sqppro.spc` in the working directory (if any) will prevail on those set here.

8. The solver `sqppro_solve` can now be called:

```
call sqppro_solve (simul, x, lm, data, info, options, user)
```

   See section 2.3.3 for the details.

2.4 Fine tunings

2.4.1 Setting the precision of the QP solver

The QP solver QPAL [6] used by SQPPRO at each iteration implements an iterative process that cannot solve the QP's (1.3) with an arbitrary precision (because of rounding errors). How to specify the precision to which the QP's must be solved is a delicate task, which is monitored by various options that we now discuss (see section 2.1.3 for the list of the possible options). A fine tuning of these options is crucial for getting the best of the SQPPRO solver on difficult problems.

There are actually two ways of controlling the precision of the QP solver; they are selected by the *QP precision identificator* option

```
qp_precision_id
```

Either `qp_precision_id` = `fixed` (a parameter provided by the `sqppro_mod` module), in which case the QP's are solved with a fixed specified precision at each SQP iteration, or `qp_precision_id` = `variable` (another parameter provided by the `sqppro_mod` module), in which case the QP's are solved with a precision that depends on the SQP iteration. Note that in both cases, SQPPRO may require a possibly more important precision for the QP solution than the one specified by the options described here, in order to ensure that the computed direction is a descent direction of the merit function; when the option `plevel` is $\geqslant 4$, this situation is indicated in the output by the phrase

```
QP solve (with more stringent accuracy)
```

if such a more precise direction cannot be found by the QP solver, SQPpro stops mentioning that the QP solver failed. We now discuss the two possibilities specifiable by the option `qp_precision_id`.

Using *fixed precision* (`qp_precision_id` = `fixed`) is probably adequate for small problems or problems for which a rather crude solution is searched. In this case, the following two options are used for controlling the QP precision:

- `qp_tol_glan` specifies the absolute precision in the norm of the gradient of the QP Lagrangian (the used norm is specified by the option `qp_norm`);
- `qp_tol_feas`; specifies the absolute precision in the norm of the QP feasibility (the used norm is specified by the option `qp_norm`).

Using *variable precision* (`qp_precision_id` = `variable`) is often more appropriate for large-scale problems, since this option usually yields a faster run (because the QP's are solved inexactly when the iterates are far from a solution) and makes it possible to get the best precision on the solution of the nonlinear problem (1.1). In the *variable precision* case, the following two options are used for controlling the QP precision:

- `qp_forcing_factor` must be set to a number in the open interval $(0, 1)$; the QP solution is more precise when that number is closer to zero, but the precision is set relatively to the one obtained on the nonlinear problem (1.1) at the current iteration, so that the QP solution is required to be more precise when the iterate (x_k, λ_k) approaches the primal-dual solution to the nonlinear problem;
- `qp_tol_glan` specifies the absolute precision in the norm of the gradient of the QP Lagrangian (the used norm is specified by the option `qp_norm`); this value is necessary with the current version of the QPAL solver, which uses an augmented Lagrangian (AL) approach and must minimize completely (i.e., up to the precision given by `qp_tol_glan`) the QP Lagrangian at each AL iteration.

The solution offered in SQPpro 0.5 is not completely satisfactory, however, since the best way of controlling the precision of the solution computed by QPAL has not been completely clarified. In particular, it is still necessary to specify with the option `qp_tol_glan` the precision to which the AL must be minimized at each QPAL iteration. A too small value may lead to failure because the AL cannot be minimized at the required precision and a too large value may lead to the impossibility to satisfy the QP constraints.

3 Current limitations and perspectives

We list below the limitations of the software SQPpro we are aware of. The list is not comprehensive.

- The solver cannot deal with incompatible linearized constraints in the osculating tangent problem (1.3). This is intended to be fixed by shifting. The use of the Byrd-Omojokun trust region globalization technique also gives a solution to this difficulty.
- The solver can be used several times in the same program, provided the dimensions of the problems to solve are identical.
- Remedies to the Maratos effect have not been implemented yet.
- A true truncated Newton algorithm must still be implemented. This would require that the QPAL solver could provide a solution with a prescribed precision with respect to the Lagrangian gradient norm and to feasibility.

The following features are intended to be implemented in a near future. The list is not comprehensive.

- Updated Cholesky factorizations for solving linear systems (presently a conjugate gradient solver is used).
- Reverse or socket communication.
- Globalization by trust regions.
- Possibility to use second derivatives.
- Parallelisation of the loops with OpenMP directives [9].

References

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

[2] J.F. Bonnans, J.Ch. Gilbert, C. Lemaréchal, C. Sagastizábal (2006). *Numerical Optimization – Theoretical and Practical Aspects* (second edition). Universitext. Springer Verlag, Berlin. 4, 5

[3] F. Delbos, J.Ch. Gilbert (2005). Global linear convergence of an augmented Lagrangian algorithm for solving convex quadratic optimization problems. *Journal of Convex Analysis*, 12, 45–69. 5

[4] F. Delbos, J.Ch. Gilbert, R. Glowinski, D. Sinoquet (2006). Constrained optimization in seismic reflection tomography: a Gauss-Newton augmented Lagrangian approach. *Geophysical Journal International*, 164, 670–684. 5

[5] GFortran. http://gcc.gnu.org/wiki/GFortran. 4

[6] J.Ch. Gilbert (2009). QPAL – A solver of convex quadratic optimization problems, using an augmented Lagrangian approach – Version 0.6.1. Rapport Technique 0377, INRIA, BP 105, 78153 Le Chesnay, France.
http://www-rocq.inria.fr/estime/modulopt/optimization-routines/qpal/qpal.html. 5, 6, 20

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

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

[9] OpenMP Architecture Review Board (2008). OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf. 22

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