INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

QPAL – A solver of convex quadratic optimization problems, using an augmented Lagrangian approach

Version 0.6.1 (May, 2009)

J. Charles GILBERT

N° 0377

19 décembre 2009

Thème NUM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0377--FR+ENG

QPAL – A solver of convex quadratic optimization problems, using an augmented Lagrangian approach

Version 0.6.1 (May, 2009)

J. Charles Gilbert[†]

Thème NUM — Systèmes numériques
Projet Estime

Rapport technique n° 0377 — 19 décembre 2009 — 22 pages

Abstract: QPAL is a piece of software that aims at solving a convex quadratic optimization problem with linear equality and inequality constraints. The implemented algorithm uses an augmented Lagrangian approach, which relaxes the equality constraints and deals explicitely with the bound constraints on the original and slack variables. The generated quadratic functions are minimized on the activated faces by a truncated conjugate gradient algorithm, interspersed with gradient projection steps. When the optimal value is finite, convergence occurs at a linear speed that can be prescribed by the user. Matrices can be strored in dense or sparse structures; in addition, the Hessian of the quadratic objective function may have a direct of inverse ℓ-BFGS form. QPAL is written in Fortran-2003.

Key-words: augmented lagrangian, convex quadratic optimization, dense and sparse matrices, gradient projection, ℓ-BFGS matrix, linear convergence.

[†] 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

QPAL – Un solveur de problèmes d'optimisation quadratique convexe, fondé sur une approche par lagrangien augmenté

Version 0.6.1 (Mai, 2009)

Résumé : QPAL est un code destiné à minimiser une fonction quadratique convexe sous des contraintes linéaires d'égalité et d'inégalité. L'algorithme implémenté utilise une technique de lagrangien augmenté, qui relaxe les contraintes d'égalité et traite explicitement les bornes sur les variables originales et d'écart. Les fonctions quadratiques générées sont minimisées sur les faces activées par un algorithme de gradient conjugué tronqué, intrecoupé de phases de gradient projeté. Lorsque la valeur optimale est finie, la convergence est linéaire à une vitesse qui peut être prescrite par l'utilisateur. Les matrices peuvent être stockées dans des tructures denses ou creuses ; de plus le hessien de la fonction à minimiser peut prendre la forme d'une matrice ℓ-BFGS directe ou inverse. QPAL est écrit en Fortran-2003.

Mots-clés : convergence linéaire, gradient projeté, lagrangien augmenté, matrice ℓ-BFGS, matrices dense et creuse, optimisation quadratique convexe.

1 Presentation

1.1 Scope of the program

The semi-acronym QPAL (pronounce Q-pal) stands for Quadratic Programming with an Augmented Lagrangian. QPAL can indeed solve a *convex* quadratic optimization problem with linear constraints. The form of the problem considered by QPAL is the following

$$(\mathrm{QP}) \quad \left\{ \begin{array}{l} \min \left(f(x) := g^\top x + \frac{1}{2}\, x^\top H x \right) \\ l \leqslant (x, A_I x) \leqslant u \\ A_E x = e. \end{array} \right. \tag{1.1}$$

The function f in (1.1) is called the *objective* of the problem. Its *Hessian* H must be an $n \times n$ positive semidefinite symmetric matrix (hence f is a convex function) and its *gradient* at the origin is the vector $g \in \mathbb{R}^n$. The notation $l \leqslant (x, A_I x) \leqslant u$ expresses in compact form bound constraints on x and on $A_I x$: A_I is $m_I \times n$ and the $(n{+}m_I)$-dimensional vectors l and u satisfy $l < u$ (i.e., $l_i < u_i$ for all indices i) and may have infinite components. Hence the inequality constraints read

$$\begin{aligned} l_i \leqslant x_i \leqslant u_i, &\quad \text{for } i = 1, \ldots, n, \\ l_{n+i} \leqslant (A_I x)_i \leqslant u_{n+i}, &\quad \text{for } i = 1, \ldots, m_I. \end{aligned}$$

Finally, the equality constraints are expressed with an $m_E \times n$ matrix A_E and a vector $e \in \mathbb{R}^{m_E}$. The matrix A_E need not be surjective.

Since H can vanish, (1.1) can also model a linear optimization problem.

QPAL is flexible with respect to the memory representation of the matrices defining the problem. Indeed the Hessian matrix H may be dense, or sparse, or may have a direct or inverse ℓ-BFGS structure. The constraint Jacobians A_I and A_E 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.

It is convenient to denote by $B := \{1, \ldots, n\}$ the index set of the variables x and by $A_B = I_n$ the identity matrix of order n. Then the inequality $l \leqslant (x, A_I x) \leqslant u$ reads in a similar manner $l \leqslant A_{B\cup I} x \leqslant u$. We also adopt the notation

$$m = n + m_I + m_E \qquad \text{and} \qquad A = \begin{pmatrix} A_B \\ A_I \\ A_E \end{pmatrix}.$$

An inequality constraint with the index $i \in B \cup I$ is said to be *active* at $x \in X$ if $A_i x = l_i$ or u_i. It is said to be *inactive* otherwise. The *active set* at $x \in X$ is defined by

$$\mathcal{A}(x) = \{i \in B \cup I : A_i x = l_i \text{ or } u_i\}.$$

The *feasible set* of (1.1) is denoted by

$$X = \{x \in \mathbb{R}^n : l \leqslant A_{B\cup I} x \leqslant u,\ A_E x = e\}.$$

QPAL 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. QPAL compiles with `gfortran` [5], version "`4.4.0 20090321 (experimental)`" or higher.

1.2 Detecting optimality

The *Lagrangian* of problem (1.1) is the function $\ell : \mathbb{R}^n \times \mathbb{R}^{n+2m_I+m_E} \to \mathbb{R}$ defined at (x, λ) by

$$\begin{aligned} \ell(x,\lambda) = {} & g^\top x + \frac{1}{2} x^\top H x \\ & + (\lambda^l_{B\cup I})^\top (l - A_{B\cup I} x) + (\lambda^u_{B\cup I})^\top (A_{B\cup I} x - u) + \lambda_E^\top (A_E x - e), \end{aligned} \tag{1.2}$$

in which $\lambda = (\lambda^u_{B\cup I}, \lambda^l_{B\cup I}, \lambda_E)$.

Since problem (1.1) is convex, its optimality conditions read: x is a solution to (1.1) if and only if there is a $\lambda \in \mathbb{R}^{n+m_I+m_E}$ such that

$$\left\{ \begin{array}{l} g + Hx + A^\top \lambda = 0 \\ l \leqslant A_{B\cup I} x \leqslant u \\ (\lambda^+_{B\cup I})^\top (A_{B\cup I} x - u) = 0 \\ (\lambda^-_{B\cup I})^\top (A_{B\cup I} x - l) = 0 \\ A_E x = e, \end{array} \right. \tag{1.3}$$

where $\lambda^+ := \max(0, \lambda)$ and $\lambda^- := \max(0, -\lambda)$, componentwise. In the table below, we have shown the correspondence existing between the multipliers used in (1.2) and in (1.3):

multipliers in (1.2)	multipliers in (1.3)
$\lambda^u_{B\cup I}$	$\lambda^+_{B\cup I}$
$\lambda^l_{B\cup I}$	$\lambda^-_{B\cup I}$
$\lambda^u_{B\cup I} - \lambda^l_{B\cup I}$	$\lambda_{B\cup I}$
λ_E	λ_E

1.3 Brief description of the method

In QPAL, problem (1.1) is transformed, using an auxiliary variable $y \in \mathbb{R}^{m_I}$, into

$$(\text{QP'}) \qquad \begin{cases} \min \; g^\top x + \frac{1}{2} x^\top H x \\ l \leqslant (x, y) \leqslant u \\ A_I x = y \\ A_E x = e. \end{cases} \tag{1.4}$$

Note that an auxiliary variable is only associated with the complex inequalities $l_I \leqslant A_I x \leqslant u_I$, not with the bound constraints $l_B \leqslant x \leqslant u_B$. As we will now see, this choice has for consequence that the two sets of inequalities are not treated in the same manner. The user of QPAL can always declare the bound constraint $l_B \leqslant x \leqslant u_B$ as being part of the complex inequalities by enlarging the matrix A_I with an identity matrix (the converse is not possible though), but this is not recommended since the solver is more efficient on bound constraints than on general linear inequality constraints.

The algorithm used by QPAL generates a sequence $\{\lambda_k\}_{k \geqslant 1}$ of multipliers associated with the equality constraints of (QP') in the following manner [3, 4] (we describe a typical iteration).

- At iteration k, the algorithm solves

$$\min_{l \leqslant (x,y) \leqslant u} \; \ell_{r_k}(x, y, \lambda_k), \tag{1.5}$$

where ℓ_r is the *augmented Lagrangian* (AL) defined for $x \in \mathbb{R}^n$, $y \in \mathbb{R}^{m_I}$, and $\lambda = (\lambda_I, \lambda_E) \in \mathbb{R}^{m_E} \times \mathbb{R}^{m_I}$ by

$$\begin{aligned} \ell_r(x, y, \lambda_{I \cup E}) = g^\top x + \frac{1}{2} x^\top H x + \lambda_I^\top (A_I x - y) + \frac{r}{2} \|A_I x - y\|_2^2 \\ + \lambda_E^\top (A_E x - e) + \frac{r}{2} \|A_E x - e\|_2^2. \end{aligned} \tag{1.6}$$

The scalar $r > 0$ is called the *augmentation parameter*. The convex quadratic problem (1.5) is *a priori* easier to solve than (QP) since it has only bound constraints. Note that the bound constraints on x are treated explicitly in (1.5), while the complex inequalities $l_I \leqslant A_I x \leqslant u_I$ are relaxed into the AL. The minimization in (1.5) is done by a combination of various algorithms: gradient projection, active set, and conjugate gradient. Under the sole assumption that problem (1.1) has a solution, problem (1.5) has also a solution, say (x_{k+1}, y_{k+1}). This solution is not necessary unique however.

- Then the multiplier λ_k is updated by the formula

$$\lambda_{k+1} = \lambda_k + r_k \begin{pmatrix} A_I x_{k+1} - y_{k+1} \\ A_E x_{k+1} - e \end{pmatrix}. \tag{1.7}$$

Even though (x_{k+1}, y_{k+1}) is not uniquely determined as a solution to (1.5), the constraint value (factor of r_k above) is independent of that solution, so that the multipliers λ_k are unambiguously generated.

On entry in `qpal_solve`, x and λ are supposed given: x is a primal variable that can be reasonably initialized by the user and λ is a multiplier that can be adequately initialized in a context like SQP. If one has no idea of the correct x and/or λ, one can just set $x = P_{[l,u]}0$ (the projection of 0 on the interval $[l, u]$) and/or $\lambda = 0$. The augmentation parameter r is set to some positive value (actually 1), unless a positive value has been given on entry in `qpal_solve` through the argument `options%r`. When solving a sequence of similar QP's, like in the SQP algorithm, the value of r can be set to the one determined by `qpal_solve` at the end of the previous run, which is available in the output variable `info%r`.

1.4 The package

1.4.1 Description

The QPAL package is formed of the files and directories described below. In this description, `PLAT` is the name of an environment variable that designates the platform for which QPAL 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 for the details).

- The files `COPYRIGHT.*` give the conditions of use of QPAL. 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 solver.
- The file `README` gives a short description of the package and tells how to use it.
- The directory `bin` is originally empty and will contain the binaries gathered in libraries, named `lib*.a`, after compilation of QPAL:
  - `libblas.$PLAT.a` contains the BLAS routines used in QPAL, those in the directory `blas` (see below); this archive can be used if the compiler does not provide a BLAS library;
  - `liblapack.$PLAT.a` contains the LAPACK routines used in QPAL, those in the directory `lapack` (see below); this archive can be used if the compiler does not provide a LAPACK library;
  - `libqpal.$PLAT.a` contains the routines peculiar to QPAL, those in the directory `src` (see below).
- The directory `blas` contains the BLAS [1] routines used in QPAL and a `Makefile` to compile them. These compiled routines may be replaced by versions that have been tuned to a particular platform, for example by using the option `-lblas` of the compiler, if any.
- The directory `cuter` contains all the files that are useful to install QPAL in CUTEr [2, 7]; see how to proceed in the `README` file of the `cuter` directory.
- The directory `doc` contains this documentation in PDF (`doc.pdf`). It also contains the file `VERSIONS.txt` that briefly describes the successive versions of the solver.
- The directory `example` gives an elementary example of convex quadratic optimization problem, that is solved by QPAL. The goal of this example is to make concrete the way of encoding the problem and to call the QP solver. The example is implemented using both dense and sparse storage.
- The directory `lapack` contains the LAPACK routines used in QPAL and a `Makefile` to compile them. These compiled routines may be replaced by versions that have been tuned to a particular platform, for example by using the option `-llapack` of the compiler, if any.
- The directory `libopt` contains all the files that are useful to install QPAL in the LIBOPT environment [6]; see how to proceed in the `README` file of the `libopt` directory.
- The file `make.a_platform` is used by the `Makefile`'s of the package to compile various programs. You will probably need to adapt this file to your own platform, by redefining some of its variables; more is said on this subject in stage 2 of section 1.4.2.
- The directory `mod` is originally empty and will contain, after compilation of QPAL, the module descriptor `qpal_mod.$PLAT.mod` of the `qpal_mod` module. It provides, in particular, the description of the public derived types discussed in section 2.1.
- The directory `src` contains the peculiar routines of the QPAL solver and a `Makefile` to compile them.

1.4.2 Installation

The installation of QPAL can be made by following the stages below.

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

   ```
   QPAL-xxx-distrib.tgz
   ```

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

   ```
   tar -zxvf QPAL-xxx-distrib.tgz
   ```

   This creates the directory `QPAL-xxx-distrib`, which can be renamed. Let us call it the *`qpal` directory*.

2. The second stage deals with platform matters. To know the chosen compiler, linker and their options, the Makefile's of the QPAL package include a file named `make.$PLAT`, located in the `qpal` directory. Here, `PLAT` is the name 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 the `qpal` 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 QPAL `Makefile`'s:

   ```
   F03        = # name of the Fortran 2003 compiler
   F03DBGFLAG = # option requiring generating code for debugging
   F03FLAGS   = # usual options to use with $(F03)
   F03NOLD    = # option preventing from making a load object
   ```

3. You are now ready to compile QPAL. Go into the directory `src` and type

   ```
   make
   ```

   As said above, this command places the QPAL libraries in the directory `bin` and module descriptors in the directory `mod`. These can now be linked to a program that uses QPAL as a convex quadratic solver.

2 Usage

We start in section 2.1 by specifying the data structures used by the QPAL solver, which is revealed by its various Fortran types. In section 2.2, we describe the arguments of the subroutines or function `qpal_allocate`, `qpal_solve`, `qpal_conv`, and `qpal_deallocate`, which are various tools making possible to run and control of the solver. In section 2.2.6, we give the typical sequence of statements that must precede a call to the solver.

2.1 Data structure

The QPAL solver is structured as a Fortran module, named `qpal_mod`, with its private/public derived type definitions, data, functions, and subroutines. The entry point to the solver is `qpal_solve`. The associated subroutine `qpal_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 QPAL public derived types are described below.

A structure with the `qpal_data_type` public derived type has allocatable components that are allocated by the subroutine `qpal_allocate` (section 2.2.1). This feature is a little peculiar since the structure must be declared in the *user* space, while it is allocated by the *solver* subroutine `qpal_allocate`. The use of this F03 feature is motivated by the desire to discharge the user from taking care of the boring, humdrum, and error-prone allocation instructions. You will be able to use a `qpal_data_type` structure only after having called `qpal_allocate`. Useless components are not allocated by `qpal_allocate`. Deallocation is realized by `qpal_deallocate` (section 2.2.5).

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

2.1.1 Data types

QPAL can solve convex quadratic optimization problems having various data structures. The matrices A_I and A_E defining the constraints can indeed be dense or sparse, while the Hessian H of the objective can be dense, sparse, or can have an ℓ-BFGS structure. 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 :: qpal_sparse_type
  sequence
  integer                       :: nnz
  integer, allocatable          :: i(:), j(:)
  double precision, allocatable :: v(:)
end type qpal_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 `qpal_sparse_type` (also with the keyword `sequence`) but with a different type name will be correctly identified with a structure of type `qpal_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 `qpal_hessian_type`, which describes the memory representation of the Hessian H of the objective of problem (1.1). The Hessian can be dense, sparse, or have a direct or inverse ℓ-BFGS representation.

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

Description of the components.

- `id`: this `integer` component specifies the *Hessian identificator*. The following 5 values can be used, named by public parameters:
  - `missing` informs QPAL that $H = 0$, so that the quadratic problem (1.1) is actually a linear optimization problem;
  - `dense` informs QPAL that H is a dense matrix that must be stored in the component `dense`, which is a `double precision` array of dimension `(n,n)`;
  - `sparse` informs QPAL that H is a sparse matrix that must be stored in the component `sparse`, which is a structure of type `qpal_sparse_type`;
  - `lbfgs_dir` informs QPAL that H is a *direct* ℓ-BFGS matrix that is stored in the data structure `lbfgs_dir`, as explained in the module `modulopt_lbfgs_mod` (the features of this module need not be known by the user of QPAL);
  - `lbfgs_inv` informs QPAL that H is an *inverse* ℓ-BFGS matrix; this data structure is essentially used for unconstrained problem, so that QPAL does not provide any code for dealing with that case, in particular no data structure.
- `dense`: this `allocatable double precision` array is used to store the Hessian when `id` = `dense`.
- `sparse`: this memory structure of type `qpal_sparse_type` is used to store a sparse Hessian when `id` = `sparse`. Since a Hessian is a symmetric matrix, only the lower triangular part (with row index `i(k)` $\leqslant$ the column index `j(k)`) must be filled in and will be used (values `v(k)` with `i(k)` $>$ `j(k)` are ignored).
- `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, A_I and A_E respectively. These Jacobians can be dense or sparse.

```
type, public :: qpal_constraint_type
  integer                        :: id
  double precision, allocatable :: dense(:,:)
  type(qpal_sparse_type)         :: sparse
end type qpal_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 QPAL that the Jacobian is a dense matrix that must be stored in the component `dense`;
  - `sparse` informs QPAL 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 `qpal_sparse_type` is used to store a sparse constraint Jacobian when `id` = `sparse`.

2.1.2 Problem data

The public derived type `qpal_data_type` must be used to declare and store the data of problem (1.1). Its allocatable components are allocated by `qpal_allocate` (section 2.2.1); only the useful variables for the considered problem are allocated.

```
type, public :: qpal_data_type
  integer                         :: n, nb, mi, me
  double precision, allocatable :: g(:)
  type(qpal_hessian_type)         :: h
  double precision, allocatable :: lb(:), ub(:)
  type(qpal_constraint_type)      :: ai
  double precision, allocatable :: li(:), ui(:)
  type(qpal_constraint_type)      :: ae
  double precision, allocatable :: e(:)
end type qpal_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 variables with a finite lower or upper bound, the number m_I of inequality constraints, and the number m_E of equality constraints.
- `g(:)`: this allocatable `double precision` array of dimension `n` is used to store the vector $g \in \mathbb{R}^n$ giving the linear part of the objective of problem (1.1).
- `h`: this data structure is used to store the Hessian H of the objective of problem (1.1), using the type `qpal_hessian_type` previously defined.
- `lb(:)`, `ub(:)`: these allocatable `double precision` arrays of dimension `n` 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.
- `ai`: this data structure is used to store the Jacobian A_I of the inequality constraints of problem (1.1), using the type `qpal_constraint_type` previously defined.
- `li(:)`, `ui(:)`: these allocatable `double precision` arrays of dimension `mi` are used to store the vectors l_I and $u_I \in \mathbb{R}^{m_I}$, giving respectively the lower and upper bounds on $A_I x$.
- `ae`: this data structure is used to store the Jacobian A_E of the equality constraints of problem (1.1), using the type `qpal_constraint_type` previously defined.

- `e(:)`: this allocatable `double precision` array of dimension `me` is used to store the vector $e \in \mathbb{R}^{m_E}$, giving the RHS of the equality constraints.

2.1.3 Solver options

The public derived type `qpal_options_type` is used to describe the options of the QPAL solver, i.e., those parameters that can be used to tune the behavior of the solver.

```
type, public :: qpal_options_type
  integer          :: iout, verb, iter, cgit, hvpd, avpd
  character(len=3) :: optimality_norm
  double precision :: glan_abs, feas_abs, mpos_abs
  double precision :: inf, dxmin, dcmin
  double precision :: r, dcrt, infs
end type qpal_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.

- `avpd`: `integer` variable specifying the maximal number of constraint-vector products ($A_I v$ and $A_E v$) allowed.

  Valid value: > 0. *Default value:* 1000.

- `cgit`: `integer` variable specifying the maximal number of conjugate gradient (CG) iterations allowed.

  Valid value: > 0. *Default value:* 1000.

- `dcmin`: positive `double precision` variable that is used to detect active bounds with index $i \in I$. To this respect, there must also hold

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

  Valid values: > 0. *Default value:* 10^{-10}.

- `dcrt`: `double precision` variable giving the desired decrease factor for the constraint norm. A small value looks better but it forces QPAL to take a large AL parameter r, inducing ill-conditioning. QPAL will not try to decrease the constraint norm by a factor smaller than `dcrt`. For a small easy problem, `dcrt` can be chosen rather small. For a large scale ill-conditioned problem, choose `dcrt` close to 1.

  Valid values: $]0, 1[$. *Default value:* 10^{-1}.

- `dxmin`: positive `double precision` variable that is used to detect active bounds with index $i \in B$. To this respect, there must also hold

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

  Valid values: > 0. *Default value:* 10^{-10}.

- `feas_abs`: `double precision` variable specifying the required accuracy on the constraint norm (see the component `optimality_norm` for the definition of the norm $\|\cdot\|$). Knowing that the bound constraints on x are satisfied along the iterations, hence at the final point, only the feasibility with respect to the constraints $B \cup I$ are considered. If the solver stops at x with `info%flag` $= 0$, there hold

$$\max\Big(\|[l_I - A_I x]^+ + [A_I x - u_I]^+\|, \|A_E x - e)\|\Big) \leqslant \texttt{feas\_abs}. \tag{2.1}$$

Valid value: $\geqslant 0$. *Default value:* 10^{-8}.

- `glan_abs`: `double precision` variable specifying the required *absolute* accuracy on the gradient of the Lagrangian norm (see the component `optimality_norm` for the definition of the norm $\|\cdot\|$). If the solver stops at (x, λ) with `info%flag` $= 0$, there holds

$$\|\nabla_x \ell(x, \lambda)\| \leqslant \texttt{glan\_abs}. \tag{2.2}$$

  Valid value: $\geqslant 0$. *Default value:* 10^{-8}.

- `hvpd`: `integer` variable specifying the maximal number of Hessian-vector products (Hv) allowed.

  Valid value: > 0. *Default value:* 1000.

- `inf`: `double precision` variable. This is the value used by QPAL to detect inexistent bounds on the variables or in the inequality constraints. In other words, if after having defined `inf`, you set `ub(i)` [resp. `lb(i)`] (these are components of a variable of type `qpal_data_type`) to a value $\geqslant$ `inf` [resp. $\leqslant$ `-inf`], QPAL will consider that the ith variable is not subject to an upper bound [resp. to a lower bound]. The same value and rule are used to locate bounds on $A_I x$. As a result, it is recommended to use a large value.

  Valid value: > 0. *Default value:* `huge(1.d0)`.

- `infs`: `double precision` variable specifying what is an infinite stepsize. If such a stepsize along the current search direction leads to a decrease in the objective f without encountering a bound, the direction is considered to be a direction of unboundedness and the solver stops with `info%flag` $= 2$.

  Valid value: > 0. *Default value:* 10^{20}.

- `iout`: `integer` variable that is taken as the channel number for the written outputs. These are indeed written by

```
write (iout,...) ...
```

  Valid value: > 0. *Default value:* 6.

- `iter`: `integer` variable specifying the maximal number of AL iterations allowed.

  Valid value: > 0. *Default value:* 100.

- `mpos_abs`: `double precision` variable giving the required accuracy on the multiplier positivity. If the solver stops with `info%flag` $= 0$, there holds

$$\lambda^l \geqslant -\texttt{mpos\_abs} \qquad \text{and} \qquad \lambda^u \geqslant -\texttt{mpos\_abs}.$$

  Valid value: $\geqslant 0$. *Default value:* 10^{-8}.

- `optimality_norm`: this string, made of at most 3 characters, specifies the type of vector norm $\|\cdot\|$ that must be used to check optimality:
  - `'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 solver stops more rapidly with the sup norm than with the Euclidean norm.

Valid values: `'2'`, `'euc'`, `'inf'`. *Default value:* `'inf'`.

- `r`: `double precision` variable giving the initial value of the AL parameter r. This value is increased by QPAL if the decrease in the constraint norm is not fast enough (compared to `dcrt`).

  Valid value: > 0. *Default value:* 1.

- `verb`: `integer` variable that specifies the verbosity level of the solver, i.e., the amount of information that is written on channel `iout`. The following values are meaningful:

  $= 0$: nothing is written;
  $= 1$: in addition to the initial and final outputs, the solver writes one line at each AL iteration;
  $= 2$: one more line for each gradient projection (GP), descent projection (DP), and conjugate gradient (CG) phase;
  $= 3$: more details are given, including those from the GP-AS-CG inner algorithm;
  $\geqslant 4$: additional possibly long lists of numbers are written.

  Valid value: $\geqslant 0$. *Default value:* 0.

2.1.4 Solver diagnostics

The public derived type `qpal_info_type` must be used to define the variable getting information on the problem and its possible solution, on return from the solver. Its allocatable components are allocated by `qpal_allocate` (section 2.2.1) and `qpal_solve`; only the useful variables for the considered problem are allocated.

```
type, public :: qpal_info_type
  integer                         :: flag, iter, hvpd, avpd
  double precision                :: f, glan, feas, r, estl
  double precision, allocatable :: glag(:), aix(:), aex(:)
  double precision, allocatable :: dx(:), dy(:), res_bie(:)
end type qpal_info_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. A variable that is said non available below has no memory allocated for it; hence do not use it if you want to avoid a bus error.

- `aex`: `double precision` array of dimension m_E. It gives $A_E x_f$, the equality constraint value at the final point x_f.

  Not available if $m_E = 0$.

- `aix`: `double precision` array of dimension m_I. It gives $A_I x_f$, the inequality constraint value at the final point x_f.

  Not available if $m_I = 0$.

- `avpd`: `integer` variable. It gives the number of matrix-vector products, with the constraint matrices A_E and A_I, realized during the run.

- `dx`: `double precision` array of dimension n. In case `info%flag` $= 2$ (unbounded problem), it gives a direction in x along which problem (1.1) is unbounded, when it is feasible.

Not available if `info%flag` $\neq 2$.

- `dy`: `double precision` array of dimension m_I. In case `info%flag` $= 2$ (unbounded problem), it gives $A_I dx$, where dx is given by `info%dx`.

  Not available if `info%flag` $\neq 2$.

- `est`: `double precision` variable. It is known from [3] that there is a positive constant L such that the ℓ_2 norm of the constraints decreases at least by the factor $\min(L/r, 1)$ at each iteration. The value of `estl` provides a lower estimate of L.
- `f`: `double precision` variable. It gives the value $f(x_f)$ of the quadratic objective at the final point x_f.
- `feas`: `double precision` variable. It gives the value of the left hand side of (2.1) at the final point x_f found by the solver, measuring the realized accuracy on the constraints. The norm $\|\cdot\|$ is the one specified by the option `optimality_norm` on entry in QPAL.
- `flag`: `integer` variable that specifies the return status of QPAL. The following values are possible:

  $= 0$: a solution is found up to the required accuracy;
  $= 1$: erroneous argument (possibly unallocated variables or trivially inconsistent constraints);
  $= 2$: the QP is likely to be either unbounded or infeasible; in the latter case, a shift of the constraint bounds to make it feasible would also make it unbounded;
  $= 3$: maximum iterations reached (`iter` or `cgit` counters);
  $= 4$: maximum matrix-vector products reached (`hvpd` or `avpd` counters);
  $= 5$: H is not positive semidefinite;
  $= 6$: failure in linesearch;
  $= 7$: diagonal preconditioning is required but the Hessian of the AL is not positive definite;
  $= 8$: impossible to minimize the AL function on the current active face and the AL parameter r cannot be decreased;
  $= 9$: something wrong in the linear algebra subroutines;
  $= 99$: something wrong, contact your guru.

- `glag`: `double precision` array of dimension n. It gives $\nabla_x \ell(x_f, \lambda_f)$, the gradient with respect to x of the Lagrangian ℓ at the final primal-dual pair (x_f, λ_f).
- `glan`: `double precision` variable. It gives the norm of the gradient of the Lagrangian, $\|\nabla_x \ell(x_f, \lambda_f)\|$, at the final primal-dual pair (x_f, λ_f). The norm $\|\cdot\|$ is the one specified by the option `optimality_norm` on entry in QPAL.
- `hvpd`: `integer` variable. It gives the number of Hessian-vector products realized during the run.
- `iter`: `integer` variable. It gives the number of AL iterations realized during the run.
- `r`: `double precision` variable. It is the final value of the AL parameter r.
- `res_bie`: `double precision` vector of dimension $n + m_I + m_E$. It provides the final constraint residual :

$$
\begin{aligned}
\texttt{res\_bie(1:n)} &= \max(0, l_B - x, x - u_B) \\
\texttt{res\_bie(n+1:n+mi)} &= \max(0, l_I - A_I x, A_I x - u_I) \\
\texttt{res\_bie(n+mi+1:n+mi+me)} &= A_E x - e.
\end{aligned}
$$

2.2 Running the solver

The QPAL solver makes available four subroutines (`qpal_allocate`, `qpal_default_options`, `qpal_solve`, `qpal_deallocate`) and a function (`qpal_conv`). The subroutine `qpal_allocate` can be used to allocate variables of a data and an information structure (section 2.2.1). The subroutine `qpal_default_options` can be used to get the default options of the solver, before tuning these to a particular run (section 2.2.2). The subroutine `qpal_solve` is used to solve a particular instance of problem (1.1) (section 2.2.3). The function `qpal_conv` is provided to adapt the inside stopping criterion of the solver (section 2.2.4). Finally, the subroutine `qpal_deallocate` can be used to free the memory allocated by `qpal_allocate` and `qpal_solve` (section 2.2.5).

In the description of the subroutines/functions, an argument flagged with (I) means that it is an *input* or `intent(in)` variable, which has to be initialized before calling the subroutine/function; 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/function; 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/function.

2.2.1 Data allocation with `qpal_allocate`

It is recommended to call the subroutine `qpal_allocate` before calling `qpal_solve`. Its role is to allocate the allocatable components of the following two arguments: `data`, which is a structure containing the data of the quadratic problem to solve (see section 2.1.2), and `info`, which is a structure that will contain information on the run after having called the solver (see section 2.1.4). This allocation can be done without calling `qpal_allocate`, but it is probably better to let a procedure do the job. After having called `qpal_allocate`, you will be able to fill in the `data` structure with the problem data; note that the scalar quantities of this structure (dimensions, matrix type identificators, and numbers of nonzero elements if this is relevant) are set by `qpal_allocate`, so that you only have to take care of the array components. On the other hand, the `info` structure will be fill in by the QP solver `qpal_solve`, so that there is no reason to modify that structure.

```
subroutine qpal_allocate (n, nb, mi, me, &
                          h_id, h_nnz, h_mys, h_scale, h_cold, &
                          ai_id, ai_nnz, ae_id, ae_nnz, &
                          fout, plevel, prest, flag, data, info)
```

`n`, `nb`, `mi`, `me` (I): positive `integer` variables. They define the dimensions `n` $= n$, `mi` $= m_I$, and `me` $= m_E$ of problem (1.1), as well as the number `nb` of bounds on the primal variables x. 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, `qpal_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)`. The other dimensions must be set with care, since they directly intervene in the allocation of variables in the structure `data` and are used as the dimensions of the problem in the solver `qpal_solve` (section 2.2.3).

`h_id` (I): `integer` variable. It identifies the type of memory space used to store the Hessian H of the objective of problem (1.1). See the description of the `id` component of the type `qpal_hessian_type` in section 2.1.1 for the possible values of this variable, which can be `missing`, `dense`, `sparse`, `lbfgs_dir`, or `lbfgs_inv`.

`h_nnz` (I): `integer` variable. It gives the number of nonzero elements in the lower triangular part of the Hessian H in case this one is stored in a sparse structure (`h_id` has been set to `sparse`); it is meaningless otherwise.

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

`h_scale` (I): `integer` variable that determines the type of scaling that will be used in the ℓ-BFGS Hessian matrix (if `h_id` has been set to `lbfgs_dir` or `lbfgs_inv`). This one can be set to the public values `lbfgs_scal_scaling` (matrix scaled by a scalar) or `lbfgs_diag_scaling` (matrix scaled by a diagonal matrix).

`h_cold` (I): `logical` variable that tells the ℓ-BFGS initialization procedure whether the ℓ-BFGS Hessian matrix will be used from scratch (set `cold` to `.true.`) or a warm restart will be done (set `cold` to `.false.`). This is only useful when `h_id` has been set to `lbfgs_dir` or `lbfgs_inv`.

`ai_id` (I): `integer` variable. It identifies the type of memory space used to store the Jacobian A_I of the inequality constraint of problem (1.1). See the description of the `id` component of the type `qpal_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 matrix A_I 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 Jacobian A_E of the equality constraint of problem (1.1). See the description of the `id` component of the type `qpal_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 matrix A_E 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 `qpal_allocate`.

`plevel` (I): `integer` variable. It specifies the verbosity level of `qpal_allocate`. The following values are meaningful:
- $\leqslant 0$: silent mode;
- > 0: error and warning messages are printed on channel `fout`.

`prestr` (I): `character` string of arbitrary length that will precede the lines printed by QPAL (if any).

`flag` (O): `integer` variable. It provides information on the success of the allocations. Possible values are:
- $= 0$: allocations done;
- $= 1$: some allocation failed.

`data` (O): variable of type `qpal_data_type` (see section 2.1.2). The usefull allocatable components of the variable are allocated by `qpal_allocate`.

`info` (O): variable of type `qpal_info_type` (see section 2.1.4). The usefull allocatable components of the variable are allocated by `qpal_allocate`.

2.2.2 Setting default options with `qpal_default_options`

The subroutine `qpal_default_options` can be called to set the default options in a structure of `qpal_options_type` type. This call is normally made before setting the various component of the structure to values that are appropriate to the quadratic problem to solve. It allows the user not to have to set all the options, when the default ones are appropriate.

```
subroutine qpal_default_options (options)
```

`options` (O): variable of type `qpal_options_type` (see section 2.1.3).

2.2.3 Solving the problem with `qpal_solve`

The solver of the QPAL package is `qpal_solve`. Here is the subroutine definition statement.

```
subroutine qpal_solve (x, lm, data, info, options)
```

`x` (IO): `double precision` array of dimension n. It is the vector of variables x to optimize.

- On entry, it is an initial guess of the solution to (QP). It is taken as starting point by QPAL. This point need not be feasible.
- On return, when `info%flag` $= 0$, it is the optimal solution x_f found by QPAL.

`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 (QP). The first n components are associated with the bounds on x. The next m_I components are associated with the inequality constraints $l_I \leqslant A_I x \leqslant u_I$. The last m_E components are associated with the equality constraints $A_E x = e$. 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

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

where $t^+ := \max(t, 0)$ and $t^- := \max(-t, 0)$ for $t \in \mathbb{R}$ and $()^+$ and $()^-$ act componentwise for vectors.

- On entry, it is an initial guess of the dual solution to (QP). QPAL uses this vector to initialize the AL iterations (1.7).
- On return, when `info%flag` $= 0$, it is the vector of optimal dual variables λ_f found by the optimizer.

`data` (I): variable of type `qpal_data_type` (see section 2.1.2) that contains the data of problem (1.1).

`info` (O): this is a variable of type `qpal_info_type`, whose components are described in section 2.1.4. It is used to get information on the problem and its possible solution, on return from the solver.

`options` (I): this is a variable of type `qpal_options_type`, whose components are described in section 2.1.3. It is used to tune the behavior of the solver.

2.2.4 Convergence control with `qpal_conv`

QPAL has its own convergence test, which uses the tolerances `glan_abs`, `feas_abs`, and `mpos_abs` described in section 2.1.3. One may not be satisfied with this way of verifying convergence or one may want to add conditions. QPAL has anticipated this possible opinion by providing a mechanism that allows the user to modify the convergence criterion, which we now discuss.

Each time QPAL observes that its convergence tests are satisfied, it calls the logical function `qpal_conv` described below. If this one returns `.true.` (this is the case for the function in the standard distribution), the solver interrupts its iterative process and declares convergence; otherwise the iterations are pursued until another convergence test occurs. As a result, the user can modify the stopping criterion by writing a new function `qpal_conv` (not the one in the standard distribution) as follows.

- If a more stringent criterion is desired, the additional conditions can simply be implemented in the new function `qpal_conv`.
- If a completely different stopping criterion is desired, it suffices to implement this criterion in the new function `qpal_conv` and to set the tolerances `glan_abs`, `feas_abs`, and `mpos_abs` to a large number on entry into `qpal_solve`. Then the internal stopping criterion will always be satisfied and the one in `qpal_conv` will decide when to stop the iterations.

The compiled version of the function `qpal_conv` in the standard distribution is in the archive `bin/libqpal.$PLAT.a`. There is no reason to recompile this archive with the new function `qpal_conv` but to introduce its compiled version `qpal_conv.o` in the link command, as in

```
$(F03) -o main ...  main.o qpal_conv.o .../bin/libqpal.$PLAT.a ...
```

Since the new version of `qpal_conv.o` is in the command line the one in the archive `bin/libqpal.$PLAT.a` will not be selected at link time. Another possibility is to remove the default version of `qpal_conv.o` from the archive `bin/libqpal.$PLAT.a`, using

```
ar -d bin/libqpal.$PLAT.a qpal_conv.o
```

and then to provide a new version of `qpal_conv.o` at link time.

Here is the definition statement of the logical function `qpal_conv`.

```
logical function qpal_conv (data, x, lm, aix, aex, glag, qp_inf, qp_out)
```

`qpal_conv` (O): `logical` output value. Set it to `.true.` is the primal-dual variables (`x`, `lm`) are appropriate; to `.false.` otherwise.

`data` (I): variable of type `qpal_data_type` (see section 2.1.2) that contains the data of problem (1.1).

`x` (I): `double precision` array of dimension n. It is the value of the vector of variables x to optimize at the time the function is called by QPAL.

`lm` (I): `double precision` array of dimension $n + m_I + m_E$. It is the value of the dual variable or KKT multiplier associated with the constraints of problem (1.1) at the time the function is called by QPAL. The first n components are associated with the bounds on x. The next m_I components are associated with the inequality constraints $l_I \leqslant A_I x \leqslant u_I$. The last m_E components are associated with the equality constraints $A_E x = e$. The multiplier $\lambda_{B\cup I}$ associated with the bound constraints can be viewed as 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.

`aix` (I): `double precision` array of dimension m_I. It gives the value of the product $A_I x$ at the time the function is called by QPAL.

`aex` (I): `double precision` array of dimension m_E. It gives the value of the product $A_E x$ at the time the function is called by QPAL.

`glag` (I): `double precision` array of dimension n. It gives the value of the gradient of the Lagrangian at the time the function is called by QPAL.

`qp_inf` (I): `double precision` variable used to detect an infinite bound in the QP.

`qp_iout` (I): `integer` variable specifying the output channel used by QPAL for printing.

2.2.5 Deallocation with `qpal_deallocate`

The subroutine **`qpal_allocate`** allocates memory for storing the data of problem (1.1) and another one to store the solver diagnostics. This memory allocation depends on the dimensions of the problem (n, m_I, and m_E). The subroutine **`qpal_deallocate`** can be used to deallocate these variables.

Here is the definition statement of the subroutine **`qpal_deallocate`**.

```
subroutine qpal_deallocate (data, info, fout, plevel, flag)
```

`data` (IO): variable of type **`qpal_data_type`** (see section 2.1.2) that contains the data of problem (1.1) and was previously used as argument of **`qpal_allocate`** and **`qpal_solve`**. Its allocated components are deallocated by **`qpal_deallocate`**.

`info` (IO): variable of type **`qpal_info_type`** (see section 2.1.4) that contains information on problem (1.1) and was previously used as argument of **`qpal_solve`**. Its allocated components are deallocated by **`qpal_deallocate`**.

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

`plevel` (I): **`integer`** variable. It specifies the verbosity level of **`qpal_deallocate`**. The following values are meaningful:
- $= 0$: silent mode;
- > 0: error and warning messages are printed on channel **`fout`**.

`flag` (O): **`integer`** variable. It provides information on the success of the deallocations. Possible values are:
- $= 0$: deallocations done;
- $= 1$: some problem with a deallocation.

2.2.6 Calling sequence

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

1. Specification of the use of the QPAL module **`qpal_mod`**:

   ```
   use qpal_mod
   ```

   If the name of some of the public types, parameters, or procedures of **`qpal_mod`** are in conflict with your own variables, you can rename them. For example, to rename the QPAL public parameter **`dense`** (resp. **`sparse`**) into **`qpal_dense`** (resp. **`qpal_sparse`**), write instead

   ```
   use qpal_mod, qpal_dense  => dense, &
                 qpal_sparse => sparse
   ```

2. Declare the variables **`n`**, **`nb`**, **`mi`**, **`me`**, **`h_id`**, **`h_nnz`**, **`h_mys`**, **`h_scale`**, **`h_cold`**, **`ai_id`**, **`ai_nnz`**, **`ae_id`**, **`ae_nnz`**, **`fout`**, **`plevel`**, **`prest`**, **`flag`**, **`data`**, and **`info`**, set those that are relevant to the current run (except **`data`** and **`info`**). Then allocate the relevant allocated components of the **`data`** and **`info`** structures by

   ```
   call qpal_allocate (n, nb, mi, me, &
                       h_id, h_nnz, h_mys, h_scale, h_cold, &
                       ai_id, ai_nnz, ae_id, ae_nnz, &
                       fout, plevel, prest, flag, data, info)
   ```

The `flag` argument should be zero on return from `qpal_allocate`. See section 2.2.1 for more information.

3. After execution of `qpal_allocate`, the allocatable components of the variable `data` have been allocated and you can fill in them:

```
data%g  = ...
data%lb = ...
data%ub = ...
...
```

4. Before calling the solver `qpal_solve`, you still have to give an initial value to the primal-dual variables (of course these must have been declared in the calling subroutine/function as `double precision` vectors with the respective dimensions `n` and `n+mi+me`):

```
x  = ...
lm = ...
```

You also have to set in a structure here called `options`, whose type is `qpal_options_type` (section 2.1.3), the options of the solver that should not have the default value set by `qpal_default_options` (section 2.2.2):

```
call qpal_default_options (options)
options%avpd = ...
options%cgit = ...
...
```

Of particular importance are of course the variables `options%glan_abs`, `options%feas_abs`, and `options%mpos_abs` specifying the stopping criterion.

5. You can now call `qpal_solve` (section 2.2.3):

```
call qpal_solve (x, lm, data, info, options)
```

The primal-dual solution found by `qpal_solve` is in (`x`, `lm`) and the information `qpal_solve` gives on its run is in the argument `info` (section 2.1.4).

6. If the structures `data` and `info` allocated by `qpal_allocate` are no longer useful, you may want to deallocate them by using `qpal_deallocate` (section 2.2.5):

```
call qpal_deallocate (data, info, fout, plevel, flag)
```

3 Current limitations and perspectives

We list below the limitations of QPAL we are aware of, together with planned improvements. The list is certainly not exhaustive.

1. There is no projection stage along the last direction found by the conjugate gradient.
2. The augmented parameter r is only increased.
3. The code cannot detect constraint incompatibility. For the while this is reflected by an augmentation parameter r that blows up.
4. Positive semi-definite linear systems are currently solved by a preconditioned conjugate gradient algorithm. We plan to offer the possibility to solve linear systems by updated Cholesky factorizations.

5. The parallelisation of the loops could be done with OpenMP directives [8].

References

[1] Blas. `http://www.netlib.org/blas`. 6

[2] 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. 6

[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, 14

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

[7] 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`. 6

[8] OpenMP Architecture Review Board (2008). OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/spec30.pdf`. 21

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