



Science and
Technology
Facilities Council



GALAHAD

QPP

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

This package reorders to a standard form the variables and constraints for the **quadratic programming problem**

$$\text{minimize } \frac{1}{2} \mathbf{x}^T \mathbf{H} \mathbf{x} + \mathbf{g}^T \mathbf{x} + f \quad (1.1)$$

or sometimes the (shifted, squared) **least-distance problem**

$$\text{minimize } \frac{1}{2} \sum_{i=1}^n w_i^2 (x_i - x_i^0)^2 + \mathbf{g}^T \mathbf{x} + f, \quad (1.2)$$

subject to the general linear constraints

$$c_i^l \leq \mathbf{a}_i^T \mathbf{x} \leq c_i^u, \quad i = 1, \dots, m,$$

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1, \dots, n,$$

where the n by n symmetric matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{w}$, $\mathbf{x}^0$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the scalar f are given. Full advantage is taken of any zero coefficients in the matrix $\mathbf{H}$ or the vectors $\mathbf{a}_i$. Any of the constraint bounds c_i^l , c_i^u , x_j^l and x_j^u may be infinite.

The variables are reordered so that any free variables (ie, those without bounds) occur first, followed respectively by non-negativities (i.e., those for which the only bounds are that $x_j \geq 0$), lower-bounded variables (i.e., those for which the only bounds are that $x_j \geq x_j^l \neq 0$), range-bounded variables (i.e., those for which the bounds satisfy $-\infty < x_j^l < x_j^u < \infty$) upper-bounded variables (i.e., those for which the only bounds are that $x_j \leq x_j^u \neq 0$), and finally non-positivities (i.e., those for which the only bounds are that $x_j \leq 0$). Fixed variables will be removed. Within each of the above categories, the variables are further ordered so that those with non-zero diagonal Hessian entries occur before the remainder.

The constraints are reordered so that equality constraints (i.e., those for which $c_i^l = c_i^u$) occur first, followed respectively by those which are lower-bounded (i.e., those for which the only bounds are that $\mathbf{a}_i^T \mathbf{x} \geq c_i^l$), those which have ranges (i.e., those for which the bounds satisfy $-\infty < c_j^l < c_j^u < \infty$), and finally those which are upper-bounded (i.e., those for which the only bounds are that $\mathbf{a}_i^T \mathbf{x} \leq c_i^u$). Free constraints, that is those for which $c_i^l = -\infty$ and $c_i^u = \infty$, are removed.

Procedures are provided to determine the required ordering, to reorder the problem to standard form, and to recover the problem, or perhaps just the values of the original variables, once it has been converted to standard form.

The derived type is also capable of supporting *parametric* quadratic programming problems, in which an additional objective term $\theta \delta \mathbf{g}^T \mathbf{x}$ is included, and the trajectory of solution are required for all $0 \leq \theta \leq \theta_{\max}$ for which

$$c_i^l + \theta \delta c_i^l \leq \mathbf{a}_i^T \mathbf{x} \leq c_i^u + \theta \delta c_i^u, \quad i = 1, \dots, m,$$

and

$$x_j^l + \theta \delta x_j^l \leq x_j \leq x_j^u + \theta \delta x_j^u, \quad j = 1, \dots, n.$$

It is anticipated that this module will principally be used as a pre- and post-processing tool for other GALAHAD packages.

ATTRIBUTES — Versions: GALAHAD_QPP_single, GALAHAD_QPP_double. **Uses:** GALAHAD_SYMBOLS, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SORT. **Date:** December 1999. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_QPP_single
```

with the obvious substitution `GALAHAD_QPP_double`, `GALAHAD_QPP_quadruple`, `GALAHAD_QPP_single_64`, `GALAHAD_QPP_double_64` and `GALAHAD_QPP_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_TYPE`, `QPT_problem_type`, `QPT_dimensions_type`, `QPP_control_type`, `QPP_inform_type` and `QPP_map_type` (Section 2.3) and the subroutines `QPP_initialize`, `QPP_reorder`, `QPP_apply`, `QPP_get_values`, `QPP_restore` and `QPP_terminate` (Section 2.4) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Both the Hessian matrix $\mathbf{H}$ and the constraint Jacobian $\mathbf{A}$, the matrix whose rows are the vectors $\mathbf{a}_i^T$, $i = 1, \dots, m$, may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix $\mathbf{A}$ is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n * (i - 1) + j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1, \dots, m$, $j = 1, \dots, n$. Since $\mathbf{H}$ is symmetric, only the lower triangular part (that is the part h_{ij} for $1 \leq j \leq i \leq n$) need be held. In this case the lower triangle will be stored by rows, that is component $i * (i - 1) / 2 + j$ of the storage array `H%val` will hold the value h_{ij} (and, by symmetry, h_{ji}) for $1 \leq j \leq i \leq n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of $\mathbf{A}$, its row index i , column index j and value a_{ij} are stored in the l -th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val` and an integer value `H%ne`), except that only the entries in the lower triangle need be stored.

2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i + 1$. For the i -th row of $\mathbf{A}$, the i -th component of an integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr(m + 1)` holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i -th row are stored in components $l = \text{A\%ptr}(i), \dots, \text{A\%ptr}(i + 1) - 1$ of the integer array `A%col`, and real array `A%val`, respectively. The same scheme is applicable to $\mathbf{H}$ (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle need be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.1.4 Diagonal storage format

If $\mathbf{H}$ is diagonal (i.e., $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonal entries h_{ii} , $1 \leq i \leq n$, need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square $\mathbf{A}$.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.3 The derived data types

Six derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrices $\mathbf{A}$ and $\mathbf{H}$. The components of `SMT_TYPE` used here are:

- `m` is a scalar component of type `INTEGER(ip_)`, that holds the number of rows in the matrix.
- `n` is a scalar component of type `INTEGER(ip_)`, that holds the number of columns in the matrix.
- `ne` is a scalar variable of type `INTEGER(ip_)`, that either holds the number of matrix entries or is used to flag the storage scheme used.
- `type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.3.2).
- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix $\mathbf{H}$ is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.
- `row` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.2).
- `col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may the column indices of the entries (see §2.1.2–2.1.3).
- `ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `n + 1`, that may hold the pointers to the first entry in each column (see §2.1.3).

2.3.2 The derived data type for holding quadratic programs

The derived data type `QPT_problem_type` is used to hold the data that defines the problem. The components of `QPT_problem_type` used here are:

- `n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables, n .
- `m` is a scalar variable of type `INTEGER(ip_)`, that holds the number of general linear constraints, m .
- `Hessian_kind` is a scalar variable of type `INTEGER(ip_)`, that is used to indicate what type of Hessian the problem involves. Possible values for `Hessian_kind` are:

<0 In this case, a general quadratic program of the form (1.1) is given. The Hessian matrix $\mathbf{H}$ will be provided in the component `H` (see below).

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

- 0 In this case, a linear program, that is a problem of the form (1.2) with weights $\mathbf{w} = 0$, is given.
 - 1 In this case, a least-distance problem of the form (1.2) with weights $w_j = 1$ for $j = 1, \dots, n$ is given.
 - >1 In this case, a weighted least-distance problem of the form (1.2) with general weights $\mathbf{w}$ is given. The weights will be provided in the component `WEIGHT` (see below).
- H is scalar variable of type `SMT_TYPE` that contains the Hessian matrix $\mathbf{H}$ whenever `Hessian_kind` > 1 . The following components are used:
- H%type is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of H%type must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of H%type must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of H%type must contain the string `SPARSE_BY_ROWS`, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of H%type must contain the string `DIAGONAL`.
- For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into H%type. For example, if `prob` is of derived type `QPP_problem_type` and involves a Hessian we wish to store using the co-ordinate scheme, we may simply
- ```
CALL SMT_put(prob%H%type, 'COORDINATE')
```
- See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.
- H%ne is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in the **lower triangular** part of  $\mathbf{H}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.
- H%val is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the **lower triangular** part of the Hessian matrix  $\mathbf{H}$  in any of the storage schemes discussed in Section 2.1.
- H%row is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of  $\mathbf{H}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.
- H%col is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of  $\mathbf{H}$  in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.
- H%ptr is a rank-one allocatable array of dimension  $n+1$  and type `INTEGER(ip_)`, that holds the starting position of each row of the **lower triangular** part of  $\mathbf{H}$ , as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.
- If `Hessian_kind`  $\geq 0$ , the components of H need not be set.
- WEIGHT is a rank-one allocatable array type `REAL(rp_)`, that should be allocated to have length  $n$ , and its  $j$ -th component filled with the value  $w_j$  for  $j = 1, \dots, n$ , whenever `Hessian_kind`  $> 1$ . If `Hessian_kind`  $\leq 1$ , `WEIGHT` need not be allocated.
- target\_kind is a scalar variable of type `INTEGER(ip_)`, that is used to indicate whether the components of the targets  $\mathbf{x}^0$  (if they are used) have special or general values. Possible values for `target_kind` are:
- 0 In this case,  $\mathbf{x}^0 = 0$ .
  - 1 In this case,  $x_j^0 = 1$  for  $j = 1, \dots, n$ .
  - $\neq 0, 1$  In this case, general values of  $\mathbf{x}^0$  will be used, and will be provided in the component `X0` (see below).

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
 See <https://www.galahad.rl.ac.uk/download/> for full details.

`X0` is a rank-one allocatable array type `REAL(rp_)`, that should be allocated to have length `n`, and its  $j$ -th component filled with the value  $x_j^0$  for  $j = 1, \dots, n$ , whenever `Hessian_kind`  $> 0$ . If `Hessian_kind`  $\leq 0$  or `target_kind`  $= 0, 1$ , `X0` need not be allocated.

`gradient_kind` is a scalar variable of type `INTEGER(ip_)`, that is used to indicate whether the components of the gradient  $\mathbf{g}$  have special or general values. Possible values for `gradient_kind` are:

0 In this case,  $\mathbf{g} = 0$ .

1 In this case,  $g_j = 1$  for  $j = 1, \dots, n$ .

$\neq 0, 1$  In this case, general values of  $\mathbf{g}$  will be used, and will be provided in the component `G` (see below).

`G` is a rank-one allocatable array type `REAL(rp_)`, that should be allocated to have length `n`, and its  $j$ -th component filled with the value  $g_j$  for  $j = 1, \dots, n$ , whenever `gradient_kind`  $\neq 0, 1$ . If `gradient_kind`  $= 0, 1$ , `G` need not be allocated.

`DG` is a rank-one allocatable array of dimension `n` and type `REAL(rp_)`, that may hold the gradient  $\delta \mathbf{g}$  of the parametric linear term of the quadratic objective function. The  $j$ -th component of `DG`,  $j = 1, \dots, n$ , contains  $\delta g_j$ .

`f` is a scalar variable of type `REAL(rp_)`, that holds the constant term,  $f$ , in the objective function.

`A` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix  $\mathbf{A}$ . The following components are used:

`A%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of `A%type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of `A%type` must contain the string `COORDINATE`, while for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of `A%type` must contain the string `SPARSE_BY_ROWS`.

Just as for `H%type` above, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `A%type`. Once again, if `prob` is of derived type `QPP_problem_type` and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

```
CALL SMT_put(prob%A%type, 'SPARSE_BY_ROWS')
```

`A%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in  $\mathbf{A}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for either of the other two schemes.

`A%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the Jacobian matrix  $\mathbf{A}$  in any of the storage schemes discussed in Section 2.1.

`A%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of  $\mathbf{A}$  in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.

`A%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of  $\mathbf{A}$  in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense storage scheme is used.

`A%ptr` is a rank-one allocatable array of dimension `m+1` and type `INTEGER(ip_)`, that holds the starting position of each row of  $\mathbf{A}$ , as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

`C_l` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the vector of lower bounds  $\mathbf{c}^l$  on the general constraints. The  $i$ -th component of `C_l`,  $i = 1, \dots, m$ , contains  $c_i^l$ . Infinite bounds are allowed by setting the corresponding components of `C_l` to any value smaller than `-infinity`, where `infinity` is a solver-dependent value that will be recognised as infinity.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

- `C_u` is a rank-one allocatable array of dimension  $m$  and type `REAL(rp_)`, that holds the vector of upper bounds  $\mathbf{c}''$  on the general constraints. The  $i$ -th component of `C_u`,  $i = 1, \dots, m$ , contains  $c_i''$ . Infinite bounds are allowed by setting the corresponding components of `C_u` to any value larger than `infinity`, where `infinity` is a solver-dependent value that will be recognised as infinity.
- `DC_l` is a rank-one allocatable array of dimension  $m$  and type `REAL(rp_)`, that may hold the vector of parametric lower bounds  $\delta\mathbf{c}^l$  on the general constraints. The  $i$ -th component of `DC_l`,  $i = 1, \dots, m$ , contains  $\delta c_i^l$ . Only components corresponding to finite lower bounds  $c_i^l$  need be set.
- `DC_u` is a rank-one allocatable array of dimension  $m$  and type `REAL(rp_)`, that may hold the vector of parametric upper bounds  $\delta\mathbf{c}''$  on the general constraints. The  $i$ -th component of `DC_u`,  $i = 1, \dots, m$ , contains  $\delta c_i''$ . Only components corresponding to finite upper bounds  $c_i''$  need be set.
- `X_l` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the vector of lower bounds  $\mathbf{x}^l$  on the variables. The  $j$ -th component of `X_l`,  $j = 1, \dots, n$ , contains  $x_j^l$ . Infinite bounds are allowed by setting the corresponding components of `X_l` to any value smaller than `-infinity`, where `infinity` is a solver-dependent value that will be recognised as infinity.
- `X_u` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the vector of upper bounds  $\mathbf{x}''$  on the variables. The  $j$ -th component of `X_u`,  $j = 1, \dots, n$ , contains  $x_j''$ . Infinite bounds are allowed by setting the corresponding components of `X_u` to any value larger than `infinity`, where `infinity` is a solver-dependent value that will be recognised as infinity.
- `DX_l` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that may hold the vector of parametric lower bounds  $\delta\mathbf{x}^l$  on the variables. The  $j$ -th component of `DX_l`,  $j = 1, \dots, n$ , contains  $\delta x_j^l$ . Only components corresponding to finite lower bounds  $x_j^l$  need be set.
- `DX_u` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that may hold the vector of parametric upper bounds  $\delta\mathbf{x}''$  on the variables. The  $j$ -th component of `DX_u`,  $j = 1, \dots, n$ , contains  $\delta x_j''$ . Only components corresponding to finite upper bounds  $x_j''$  need be set.
- `X` is a rank-one allocatable array of dimension  $n$  and type `REAL(rp_)`, that holds the values  $\mathbf{x}$  of the optimization variables. The  $j$ -th component of `X`,  $j = 1, \dots, n$ , contains  $x_j$ .
- `Z` is a rank-one allocatable array of dimension  $n$  and type default `REAL(rp_)`, that holds the values  $\mathbf{z}$  of estimates of the dual variables corresponding to the simple bound constraints (see Section 4). The  $j$ -th component of `Z`,  $j = 1, \dots, n$ , contains  $z_j$ .
- `Z_l` is a rank-one allocatable array of dimension  $n$  and type default `REAL(rp_)`, that holds the values  $\mathbf{z}^l$  of estimates of the dual variables corresponding to the lower simple bound constraints  $\mathbf{x}^l \leq \mathbf{x}$  (see Section 4). The  $j$ -th component of `Z_l`,  $j = 1, \dots, n$ , contains  $z_j^l$ .
- `Z_u` is a rank-one allocatable array of dimension  $n$  and type default `REAL(rp_)`, that holds the values  $\mathbf{z}''$  of estimates of the dual variables corresponding to the upper simple bound constraints  $\mathbf{x} \leq \mathbf{x}''$  (see Section 4). The  $j$ -th component of `Z_u`,  $j = 1, \dots, n$ , contains  $z_j^l$ .
- `C` is a rank-one allocatable array of dimension  $m$  and type default `REAL(rp_)`, that holds the values  $\mathbf{Ax}$  of the constraints. The  $i$ -th component of `C`,  $i = 1, \dots, m$ , contains  $\mathbf{a}_i^T \mathbf{x} \equiv (\mathbf{Ax})_i$ .
- `Y` is a rank-one allocatable array of dimension  $m$  and type `REAL(rp_)`, that holds the values  $\mathbf{y}$  of estimates of the Lagrange multipliers corresponding to the general linear constraints (see Section 4). The  $i$ -th component of `Y`,  $i = 1, \dots, m$ , contains  $y_i$ .

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

- `Y_l` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the values  $\mathbf{y}^l$  of estimates of the Lagrange multipliers corresponding to the lower general constraints  $\mathbf{c}^l \leq \mathbf{Ax}$  (see Section 4). The  $i$ -th component of `Y_l`,  $i = 1, \dots, m$ , contains  $y_i^l$ .
- `Y_u` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the values  $\mathbf{y}^u$  of estimates of the Lagrange multipliers corresponding to the upper general constraints  $\mathbf{Ax} \leq \mathbf{c}^u$  (see Section 4). The  $i$ -th component of `Y_u`,  $i = 1, \dots, m$ , contains  $y_i^u$ .

### 2.3.3 The derived data type for holding the problem dimensions

The derived data type `QPT_dimensions_type` is used to hold scalar data that defines the problem partitioning for the reordered problem. The components of `QPT_dimensions_type` are:

- `x_free` is a scalar variable of type `INTEGER(ip_)`, that holds the number of free variables.
- `x_l_start` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the first variable with a nonzero lower (or lower range) bound.
- `x_l_end` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last variable with a nonzero lower (or lower range) bound.
- `x_u_start` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the first variable with a nonzero upper (or upper range) bound.
- `x_u_end` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last variable with a nonzero upper (or upper range) bound.
- `c_equality` is a scalar variable of type `INTEGER(ip_)`, that holds the number of equality constraints.
- `c_l_start` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the first inequality constraint with a lower (or lower range) bound.
- `c_l_end` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last inequality constraint with a lower (or lower range) bound.
- `c_u_start` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the first inequality constraint with an upper (or upper range) bound.
- `c_u_end` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last inequality constraint with an upper (or upper range) bound.
- `h_diag_end_free` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last free variable for which the Hessian has a diagonal entry.
- `h_diag_end_nonneg` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last non-negative variable for which the Hessian has a diagonal entry
- `h_diag_end_lower` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last lower-bounded variable for which the Hessian has a diagonal entry
- `h_diag_end_range` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last range variable for which the Hessian has a diagonal entry
- `h_diag_end_upper` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last upper-bounded variable for which the Hessian has a diagonal entry
- `h_diag_end_nonpos` is a scalar variable of type `INTEGER(ip_)`, that holds the index of the last non-positive variable for which the Hessian has a diagonal entry

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

### 2.3.4 The derived data type for holding control parameters

The derived data type `QPP_control_type` is used to hold controlling data. Default values may be obtained by calling `QPP_initialize` (see Section 2.4.1). The components of `QPP_control_type` are:

`error` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for error messages. Printing of error messages in `QPP_reorder` and `QPP_terminate` is suppressed if  $\text{error} \leq 0$ . The default is `error = 6`.

`infinity` is a scalar variable of type `REAL(rp_)`, that is used to specify which constraint bounds are infinite. Any bound larger than `infinity` in modulus will be regarded as infinite. The default is `infinity = 1019`.

`treat_zero_bounds_as_general` is a scalar variable of type default `LOGICAL`. If it is set to `.FALSE.`, variables which are only bounded on one side, and whose bound is zero, will be recognised as non-negativities/non-positivities rather than simply as lower- or upper-bounded variables. If it is set to `.TRUE.`, any variable bound  $x_j^l$  or  $x_j^u$  which has the value 0.0 will be treated as if it had a general value. Setting `treat_zero_bounds_as_general` to `.TRUE.` has the advantage that if a sequence of problems are reordered, then bounds which are “accidentally” zero will be considered to have the same structure as those which are nonzero. However, `GALAHAD_QPP` is able to take special advantage of non-negativities/non-positivities, so if a single problem, or if a sequence of problems whose bound structure is known not to change, is/are to be solved, it will pay to set the variable to `.FALSE.`. The default is `treat_zero_bounds_as_general = .FALSE.`

### 2.3.5 The derived data type for holding informational parameters

The derived data type `QPP_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `QPP_inform_type` are:

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See Sections 2.5 and 2.6 for details.

`alloc_status` is a scalar variable of type `INTEGER(ip_)`, that gives the status of the last attempted array allocation or deallocation.

### 2.3.6 The derived data type for holding reordering data

The derived data type `QPP_map_type` is used to hold all the reordering and workspace data for a particular problem, or sequences of problems with the same structure, between calls of `QPP` procedures. This data should be preserved, untouched, from the initial call to `QPP_initialize` to the final call to `QPP_terminate`.

## 2.4 Argument lists and calling sequences

There are six procedures for user calls:

1. The subroutine `QPP_initialize` is used to set default values, and initialize private data, before reordered one or more problems with the same sparsity and bound structure. Here, the term “structure” refers both to the sparsity patterns of the Hessian matrices  $\mathbf{H}$  and Jacobian matrices  $\mathbf{A}$  involved (but not their numerical values), to the zero/nonzero/infinity patterns (a bound is either zero,  $\pm$  infinity, or a finite but arbitrary nonzero) of each of the constraint bounds, and to the variables and constraints that are fixed (both bounds are the same) or free (the lower and upper bounds are  $\pm$  infinity, respectively).
2. The subroutine `QPP_reorder` is called to reorder a problem, or the first of a sequence of structurally identical problems.
3. The subroutine `QPP_apply` may be called to reorder real data for subsequent structurally identical problems.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

4. The subroutine `QPP_get_values` may be used to obtain the values of the original primal and dual variables and Lagrange multipliers from those for the reordered problem.
5. The subroutine `QPP_restore` may be used to recover the original problem from the data for the reordered one.
6. The subroutine `QPP_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `QPP_reorder`, at the end of the reordering process. It is important to do this if the data object is re-used for another problem **with a different structure** since `QPP_initialize` cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

### 2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL QPP_initialize(map, control)
```

`map` is a scalar INTENT(OUT) argument of type `QPP_map_type` (see Section 2.3.6). It is used to hold all the reordering and workspace data for the problem.

`control` is a scalar INTENT(OUT) argument of type `QPP_control_type` (see Section 2.3.4). On exit, `control` contains default values for the components as described in Section 2.3.4. These values should only be changed after calling `QPP_initialize`.

### 2.4.2 The initial reordering subroutine

The initial reordering algorithm is applied as follows:

```
CALL QPP_reorder(map, control, info, dims, prob, &
 get_x, get_y, get_z [, parametric])
```

`map` is a scalar INTENT(INOUT) argument of type `QPP_map_type`. It is used to hold reordering and workspace data for the problem. It must not have been altered **by the user** since the last call to `QPP_initialize`.

`control` is a scalar INTENT(IN) argument of type `QPP_control_type` (see Section 2.3.4). Default values may be assigned by calling `QPP_initialize` prior to the first call to `QPP_reorder`.

`info` is a scalar INTENT(OUT) argument of type `QPP_inform_type` (see Section 2.3.5). A successful call to `QPP_reorder` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.5.

`dims` is a scalar INTENT(OUT) argument of type `QPT_dimensions_type` that is used to hold scalar data that defines the reordered problem. On successful exit, all components will have been set to values that define the reordered problem (see Section 2.3.3).

`prob` is a scalar INTENT(INOUT) argument of type `QPT_problem_type` that is used to hold data that defines the original and reordered problem. On entry, components `f`, `gradient_kind`, `G`, `A`, `C_l`, `C_u`, `X_l` and `X_u` must be appropriately allocated and set (see Section 2.3.2). The same is true of component `H` in the quadratic programming case and components `Hessian_kind`, `target_kind`, `WEIGHT` and `X0` in the least-distance case. In addition, for parametric problems `DG`, `DC_l`, `DC_u`, `DX_l` and `DX_u` must be allocated appropriately and set. If the user wishes to provide suitable starting values for  $\mathbf{x}$ ,  $\mathbf{y}$  (or alternatively  $\mathbf{y}^l$  and  $\mathbf{y}^u$ ) and  $\mathbf{z}$  (or alternatively  $\mathbf{z}^l$  and  $\mathbf{z}^u$ ), they should be placed in `X`, `Y` (or `Y_l` and `Y_u`) and `Z` (or `Z_l` and `Z_u`) respectively, and the arguments `get_x`, `get_y` and `get_z` set appropriately (see below).

On successful exit, all provided components will have been set to values that define the reordered problem (see Section 2.2.1). The reordered arrays  $\mathbf{A}$  and, if appropriate,  $\mathbf{H}$  will be stored using the row-wise scheme. In

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
See <https://www.galahad.rl.ac.uk/download/> for full details.

addition the components  $X$ ,  $Y$ ,  $Z$  and  $C$  will contain values of  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{z}$  and  $\mathbf{Ax}$  for the reordered problem. The user should be aware that fixed variables and free constraints will have been removed, and thus that the components `prob%n` and `prob%m` may be smaller than their values on entry.

`get_x` is a scalar `INTENT(IN)` argument of type default `LOGICAL`, that must be set `.FALSE.` if the user wishes to provide suitable values for the primal variables in  $X$ , and `.TRUE.`, if appropriate values should be calculated by the subroutine.

`get_y` is a scalar `INTENT(IN)` argument of type default `LOGICAL`, that must be set `.FALSE.` if the user wishes to provide suitable values for the Lagrange multipliers for the general linear constraints in  $Y$  (or alternatively in  $Y_{\text{L}}$  and  $Y_{\text{U}}$ ), and `.TRUE.`, if appropriate values should be calculated by the subroutine. In the latter case, the array  $Y$  (or the arrays  $Y_{\text{L}}$  and  $Y_{\text{U}}$ ) must have been allocated.

`get_z` is a scalar `INTENT(IN)` argument of type default `LOGICAL`, that must be set `.FALSE.` if the user wishes to provide suitable values for the dual variables for the simple bound constraints in  $Z$  (or alternatively in  $Z_{\text{L}}$  and  $Z_{\text{U}}$ ), and `.TRUE.`, if appropriate values should be calculated by the subroutine. In the latter case, the array  $Z$  (or the arrays  $Z_{\text{L}}$  and  $Z_{\text{U}}$ ) must have been allocated.

`parametric` is an `OPTIONAL` scalar `INTENT(IN)` argument of type default `LOGICAL`. If `parametric` is present, the problem will be assumed to include parametric data  $\delta \mathbf{g}$ ,  $\delta \mathbf{x}_l$ ,  $\delta \mathbf{x}_u$ ,  $\delta \mathbf{x}_l$  and  $\delta \mathbf{x}_u$  as part of `prob`, and this data will be reordered. If `parametric` is absent, no parametric data will be processed.

### 2.4.3 The subsequent reordering subroutine

The reordering calculated by a previous call to `QPP_reorder` may be applied to a structurally identical problem with different real data as follows:

```
CALL QPP_apply(map, info, dims, prob [, get_all, get_all_parametric, &
 get_g, get_dg, get_x, get_y, get_z, &
 get_x_bounds, get_dx_bounds, get_c, get_c_bounds, &
 get_dc_bounds, get_A, get_H])
```

The arguments `map`, and `info` are exactly as for `QPP_reorder`. The values of the integers `prob%n`, `prob%m`, `prob%h_ne`, `prob%a_ne`, `prob%gradient%type` and `prob%weight%type`, the integer arrays `prob%H%col`, `prob%H%ptr`, `prob%A%col`, `prob%A%ptr`, and the remaining (integer) components of `dims` must have been preserved exactly as they were on exit from the most recent call to `QPP_reorder` or `QPP_restore`, and are not altered by the subroutine.

New `REAL(rp_)` values may be assigned to the arguments `prob%f`, `prob%H%val`, `prob%G`, `prob%A%val`, `prob%WEIGHT`, `prob%X0`, `prob%CL`, `prob%CL`, `prob%X_L`, `prob%X_U`, `prob%X`, `prob%Y` (or alternatively `prob%Y_L` and `prob%Y_U`), and `prob%Z` (or alternatively `prob%Z_L` and `prob%Z_U`), (and optionally `prob%DC`, `prob%DC_L`, `prob%DC_U`, `prob%DX_L` and `prob%DX_U` for parametric problems), but the components of these arrays must be in exactly the same order as originally presented to `QPP_reorder`. The exit values of all of these real values depend on the following, remaining arguments:

`get_all` is an `OPTIONAL` scalar `INTENT(IN)` argument of type default `LOGICAL`. If `get_all` is present, the entire non-parametric problem input in `prob` will be reordered according to the mappings generated by the last successful call to `QPP_reorder`. Any parametric data will be ignored.

`get_all_parametric` is an `OPTIONAL` scalar `INTENT(IN)` argument of type default `LOGICAL`. If `get_all_parametric` is present, the entire parametric problem input in `prob` will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_f` is an `OPTIONAL` scalar `INTENT(IN)` argument of type default `LOGICAL`. If `get_f` is present, the constant objective term  $f$ , input in `prob%f` will be adjusted for the the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`get_g` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_g` is present, the gradient  $\mathbf{g}$ , input in `prob%G` will be adjusted for the the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

`get_dg` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_dg` is present, the parametric gradient  $\delta\mathbf{g}$ , input in `prob%DG`, will be adjusted for the the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

`get_x` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_x` is present, the vector of primal variables  $\mathbf{x}$ , input in `prob%X`, will be adjusted for the the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

`get_y` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_y` is present, the Lagrange multipliers  $\mathbf{y}$ , input in `prob%Y` (or alternatively in `prob%Y_l` and `prob%Y_u`), will be adjusted for the the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

`get_z` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_z` is present, the vector of dual variables  $\mathbf{z}$ , input in `prob%Z` (or alternatively in `prob%Z_l` and `prob%Z_u`), will be adjusted for the reordered problem according to the mappings generated by the last successful call to `QPP_reorder`.

`get_x_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_x_bounds` is present, the vectors of variable bounds  $\mathbf{x}_l$  and  $\mathbf{x}_u$ , input in `prob%X_l` and `prob%X_u` respectively will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_dx_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_dx_bounds` is present, the vectors of parametric variable bounds  $\delta\mathbf{x}_l$  and  $\delta\mathbf{x}_u$ , input in `prob%DX_l` and `prob%DX_u` respectively, will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_c` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_c` is present, the vector  $\mathbf{Ax}$  for the reordered problem will be returned in `prob%C`.

`get_c_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_c_bounds` is present, the vectors of constraint bounds  $\mathbf{c}_l$  and  $\mathbf{c}_u$ , input in `prob%C_l` and `prob%C_u` respectively will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_dc_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_dc_bounds` is present, the vectors of parametric constraint bounds  $\delta\mathbf{c}_l$  and  $\delta\mathbf{c}_u$ , input in `prob%DC_l` and `prob%DC_u` respectively, will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_A` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_A` is present, the entries of the matrix  $\mathbf{A}$ , input in `prob%A_val`, will be reordered according to the mappings generated by the last successful call to `QPP_reorder`.

`get_H` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_H` is present, the Hessian matrix will be reordered according to the mappings generated by the last successful call to `QPP_reorder`. Specifically, if `Hessian_kind`  $< 0$ , the entries of the lower triangular part of the matrix  $\mathbf{H}$ , input in `prob%H_val`, will be reordered. If `Hessian_kind`  $> 0$ , the components of  $\mathbf{x}^0$  input in `prob%X0` will be reordered, and additionally if `Hessian_kind`  $> 1$  so will the weights  $\mathbf{w}$  input in `prob%WEIGHT`.

#### 2.4.4 The variable recovery reordering subroutine

The values of minimization variables that have been determined for the reordered problem may be recovered for the original problem as follows:

```
CALL QPP_get_values(map, info, prob [, X_val, Y_val, Z_val])
```

---

All use is subject to the conditions of a BSD-3-Clause License.  
See <https://www.galahad.rl.ac.uk/download/> for full details.

The arguments `map` and `info` are exactly as for `QPP_reorder`. The `INTENT (IN)` argument `prob` must contain reordered problem data from a previous call to `QPP_reorder` or `QPP_apply`.

`X_val` is an OPTIONAL rank-one `INTENT (OUT)` array argument of type `REAL (rp_)`. If present, it will be filled with the values of the primal variables  $\mathbf{x}$  for the original problem, corresponding to those for the reordered problem input in `X`.

`Y_val` is an OPTIONAL rank-one `INTENT (OUT)` array argument of type `REAL (rp_)`. If present, it will be filled with the values of the Lagrange multipliers  $\mathbf{y}$  for the original problem, corresponding to those for the reordered problem input in `Y` (or `Y_l` + `Y_u`).

`Z_val` is an OPTIONAL rank-one `INTENT (OUT)` array argument of type `REAL (rp_)`. If present, it will be filled with the values of the primal variables  $\mathbf{z}$  for the original problem, corresponding to those for the reordered problem input in `Z` (or `Z_l` + `Z_u`).

#### 2.4.5 The problem restoration subroutine

The data for the original problem may be recovered from its reordered variant as follows:

```
CALL QPP_restore(map, info, dims, prob [, get_all, get_all_parametric, &
 get_g, get_dg, get_x, get_y, get_z,
 get_x_bounds, get_dx_bounds, get_c, get_c_bounds,
 get_dc_bounds, get_A, get_H])
```

The arguments `map`, `info`, `dims` and `prob` are exactly as described as output from `QPP_reorder` or `QPP_apply`, and correspond to data for the reordered problem. They may be restored to data for the original problem by appropriate settings for the remaining arguments:

`get_all` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_all` is present, the entire non-parametric problem input in `prob` and `dims` will be restored using the mappings generated by the last successful call to `QPP_reorder`. Any parametric data will be ignored.

`get_all_parametric` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_all_parametric` is present, the entire parametric problem input in `prob` and `dims` will be recovered from the mappings generated by the last successful call to `QPP_reorder`.

`get_g` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_g` is present, the gradient  $\mathbf{g}$  will be recovered from the reordered problem and placed in `prob%G` using the mappings generated by the last successful call to `QPP_reorder`.

will be recovered from the reordered problem using the mappings generated by the last successful call to `QPP_reorder`.

`get_dg` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_dg` is present, the parametric gradient  $\delta\mathbf{g}$  will be recovered from the reordered problem and placed in `prob%DG` using the mappings generated by the last successful call to `QPP_reorder`.

`get_x` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_x` is present, the vector of primal variables  $\mathbf{x}$  will be recovered from the reordered problem and placed in `prob%X` using the mappings generated by the last successful call to `QPP_reorder`.

`get_y` is an OPTIONAL scalar `INTENT (IN)` argument of type default `LOGICAL`. If `get_y` is present and `prob%Y` is allocated, the Lagrange multipliers  $\mathbf{y}$  will be recovered from the reordered problem and placed in `prob%Y` using the mappings generated by the last successful call to `QPP_reorder`. If `prob%Y_l` and `prob%Y_u` are allocated, the Lagrange multipliers  $\mathbf{y}'$  and  $\mathbf{y}''$  will be recovered from the reordered problem and placed in `prob%Y_l` and `prob%Y_u` respectively.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

`get_z` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_z` is present and `prob%Z` is allocated, the vector of dual variables  $\mathbf{z}$  will be recovered from the reordered problem and placed in `prob%Z` using the mappings generated by the last successful call to `QPP_reorder`. If `prob%Z_l` and `prob%Z_u` are allocated, the dual variables  $\mathbf{z}^l$  and  $\mathbf{z}^u$  will be recovered from the reordered problem and placed in `prob%Z_l` and `prob%Z_u` respectively.

`get_x_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_x_bounds` is present, the vectors of variable bounds  $\mathbf{x}_l$  and  $\mathbf{x}_u$  will be recovered from the reordered problem and placed in `prob%X_l` and `prob%X_u` using the mappings generated by the last successful call to `QPP_reorder`.

`get_dx_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_dx_bounds` is present, the vectors of parametric variable bounds  $\delta\mathbf{x}_l$  and  $\delta\mathbf{x}_u$  will be recovered from the reordered problem and placed in `prob%DX_l` and `prob%DX_u` using the mappings generated by the last successful call to `QPP_reorder`.

`get_c` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_c` is present, the vector  $\mathbf{Ax}$  will be recovered from the reordered problem and placed in `prob%C` using the mappings generated by the last successful call to `QPP_reorder`.

`get_c_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_c_bounds` is present, the vectors of constraint bounds  $\mathbf{c}_l$  and  $\mathbf{c}_u$  will be recovered from the reordered problem and placed in `prob%C_l` and `prob%C_u` using the mappings generated by the last successful call to `QPP_reorder`.

`get_dc_bounds` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_dc_bounds` is present, the vectors of parametric constraint bounds  $\delta\mathbf{c}_l$  and  $\delta\mathbf{c}_u$  will be recovered from the reordered problem and placed in `prob%DC_l` and `prob%DC_u` using the mappings generated by the last successful call to `QPP_reorder`.

`get_A` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_A` is present, the matrix  $\mathbf{A}$  will be recovered from the reordered problem and placed in `prob%A` using the mappings generated by the last successful call to `QPP_reorder`.

`get_H` is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If `get_H` is present, the Hessian matrix will be recovered from the reordered problem using the mappings generated by the last successful call to `QPP_reorder`. Specifically, if `Hessian_kind`  $< 0$ , the entries of the recovered lower triangular part of the matrix  $\mathbf{H}$  will be placed in `prob%H_val`. If `Hessian_kind`  $> 0$ , the components of  $\mathbf{x}^0$  will be recovered and placed in `prob%X0`, and additionally if `Hessian_kind`  $> 1$  the weights  $\mathbf{w}$  will be placed in `prob%WEIGHT`.

#### 2.4.6 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL QPP_terminate(map, control, info)
```

`map` is a scalar INTENT(INOUT) argument of type `QPP_map_type` exactly as for `QPP_reorder` which must not have been altered **by the user** since the last call to `QPP_initialize`. On exit, array components will have been deallocated.

`control` is a scalar INTENT(IN) argument of type `QPP_control_type` exactly as for `QPP_reorder`.

`info` is a scalar INTENT(OUT) argument of type `QPP_inform_type` exactly as for `QPP_reorder`. Only the component `status` will be set on exit, and a successful call to `QPP_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.5.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

## 2.5 Warning and error messages

A negative value of `info%status` on exit from `QPP_solve` or `QPP_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

- 1. An allocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status is given by the value `inform%alloc_status`.
- 3. One of the restrictions `prob%n > 0` or `prob%m ≥ 0` or requirements that `prob%A%type` and `prob%H%type` contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE\_BY\_ROWS' or 'DIAGONAL' has been violated.
- 5. The constraints are inconsistent.
- 23. An entry from the strict upper triangle of  $\mathbf{H}$  has been specified.
- 31. An attempt to use `QPP_apply`, `QPP_get_values` or `QPP_restore` has been made before a successful call to `QPP_reorder`.
- 52. An attempt to change a matrix storage format has been made without first recalling `QPP_reorder`.
- 53. At least one of the matrices  $\mathbf{A}$  or  $\mathbf{H}$  has not been reordered, while the current subroutine call requires it to have been.
- 54. Neither the array `prob%Y` nor the pair `prob%Y_l` and `prob%Y_u` have been allocated.
- 55. Neither the array `prob%Z` nor the pair `prob%Z_l` and `prob%Z_u` have been allocated.

## 2.6 Information printed

The only information printed will be error messages, corresponding to nonzero values of `info%status`, on unit `control%error`.

## 3 GENERAL INFORMATION

**Use of common:** None.

**Workspace:** Provided automatically by the module.

**Other routines called directly:** None.

**Other modules used directly:** `GALAHAD_SYMBOLS`, `GALAHAD_SMT`, `GALAHAD_QPT`, and `GALAHAD_SORT`.

**Input/output:** Output is under control of the argument `control%error`.

**Restrictions:** `prob%n > 0`, `prob%m ≥ 0`, `prob%A%type` and `prob%H%type`  $\in \{\text{'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS'}\}$ .

**Portability:** ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
 See <https://www.galahad.rl.ac.uk/download/> for full details.

## 4 METHOD

The required solution  $\mathbf{x}$  necessarily satisfies the primal optimality conditions

$$\mathbf{Ax} = \mathbf{c}, \quad (4.1)$$

where

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u \text{ and } \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \quad (4.2)$$

the dual optimality conditions

$$\mathbf{Hx} + \mathbf{g} = \mathbf{A}^T \mathbf{y} + \mathbf{z} \text{ (or } \mathbf{W}^2(\mathbf{x} - \mathbf{x}^0) + \mathbf{g} = \mathbf{A}^T \mathbf{y} + \mathbf{z} \text{ for the least-distance type objective),} \quad (4.3)$$

where

$$\mathbf{y} = \mathbf{y}^l + \mathbf{y}^u, \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \mathbf{y}^l \geq 0, \mathbf{y}^u \leq 0, \mathbf{z}^l \geq 0 \text{ and } \mathbf{z}^u \leq 0, \quad (4.4)$$

and the complementary slackness conditions

$$(\mathbf{Ax} - \mathbf{c}^l)^T \mathbf{y}^l = 0, (\mathbf{Ax} - \mathbf{c}^u)^T \mathbf{y}^u = 0, (\mathbf{x} - \mathbf{x}^l)^T \mathbf{z}^l = 0 \text{ and } (\mathbf{x} - \mathbf{x}^u)^T \mathbf{z}^u = 0, \quad (4.5)$$

where the diagonal matrix  $\mathbf{W}^2$  has diagonal entries  $w_j^2$ ,  $j = 1, \dots, n$ , where the vectors  $\mathbf{y}$  and  $\mathbf{z}$  are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise.

Two passes are made through the sets of bounds on the variables. In the first, the number belonging to each of the required categories (free, non-negativities, lower-bounded, range-bounded, upper-bounded, non-positivities and fixed) is computed, with further subdivisions within each categories according to those which have nonzero diagonal Hessian entries being recorded. On the second pass, a permutation of the variables to rearrange them into the required standard form is obtained. A mapping array of the original Hessian entries into their permuted form is then obtained, and the permutations applied in place (ie, without resorting to further storage) to  $\mathbf{H}$ ,  $\mathbf{x}$ ,  $\mathbf{z}$ ,  $\mathbf{g}$ ,  $\mathbf{x}^l$  and  $\mathbf{x}^u$ , suitable values of  $\mathbf{x}$  and  $\mathbf{z}$  satisfying (4.2) and (4.4) having optionally been computed.

Next, two passes are made through the sets of constraint bounds. In the first, the number belonging to each of the required categories (equality, lower-bounded, range-bounded, upper-bounded, and free) is computed, while in the second the required permutation of the constraints into the required standard form is obtained. A mapping array of the original Jacobian entries into their permuted form is then obtained, and the permutations applied in place to  $\mathbf{A}$ ,  $\mathbf{c}$ ,  $\mathbf{y}$ ,  $\mathbf{c}^l$  and  $\mathbf{c}^u$ , suitable values of  $\mathbf{c}$  and  $\mathbf{y}$ , satisfying (4.4), having, as before, optionally been computed. Both sets of permutations, and the matrix mapping arrays are saved for possible later use.

Any fixed variables and free constraints are removed. Fixing variables results in changes to the values of  $f$ ,  $\mathbf{g}$ ,  $\mathbf{c}^l$  and  $\mathbf{c}^u$ . Subsequent reorderings for structurally similar problems, or restorations of data from reordered problems, are easily obtained from the permutation and mapping arrays, and their inverses.

## 5 EXAMPLE OF USE

Suppose we wish to minimize  $\frac{1}{2}x_1^2 + x_2^2 + \frac{5}{2}x_3^2 + \frac{3}{2}x_4^2 - x_2x_3 + 4x_1x_4 + x_1 + 2x_2 + 3x_3 + 4x_4 + 1$  subject to the general linear constraints  $1 \leq 2x_1 + x_2 \leq 2$ ,  $x_2 + x_3 + x_4 = 2$ , and simple bounds  $-1 \leq x_1 \leq 1$ ,  $x_3 = 1$  and  $x_4 \leq 2$ , but first wish to convert the problem to our standard form. Then, on writing the data for this problem as

$$\mathbf{H} = \begin{pmatrix} 1 & & & 4 \\ & 2 & -1 & \\ & -1 & 5 & \\ 4 & & & 3 \end{pmatrix}, \mathbf{g} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}, \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ 1 \\ -\infty \end{pmatrix} \text{ and } \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 1 \\ 2 \end{pmatrix},$$

and

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & & \\ & 1 & 1 & 1 \end{pmatrix}, \mathbf{c}^l = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \text{ and } \mathbf{c}^u = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

we may use the following code:

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```

PROGRAM GALAHAD_QPP_EXAMPLE
USE GALAHAD_QPP_double ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND(1.0D+0) ! set precision
REAL (KIND = wp), PARAMETER :: infinity = 10.0_wp ** 20
TYPE (QPT_dimensions_type) :: d
TYPE (QPP_map_type) :: map
TYPE (QPP_control_type) :: control
TYPE (QPP_inform_type) :: info
TYPE (QPT_problem_type) :: p
INTEGER :: i, j
INTEGER, PARAMETER :: n = 4, m = 2, h_ne = 5, a_ne = 5
REAL (KIND = wp) :: X_orig(n)
! sparse co-ordinate storage format
CALL SMT_put(p%H%type, 'COORDINATE') ! Specify co-ordinate
CALL SMT_put(p%A%type, 'COORDINATE') ! storage for H and A
ALLOCATE(p%H%val(h_ne), p%H%row(h_ne), p%H%col(h_ne))
ALLOCATE(p%A%val(a_ne), p%A%row(a_ne), p%A%col(a_ne))
p%H%val = (/ 1.0_wp, 2.0_wp, -1.0_wp, 5.0_wp, 4.0_wp /) ! Hessian H
p%H%row = (/ 1, 2, 3, 3, 4 /) ! NB lower triangle
p%H%col = (/ 1, 2, 2, 3, 1 /) ; p%H%ne = h_ne
p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%A%row = (/ 1, 1, 2, 2, 2 /)
p%A%col = (/ 1, 2, 2, 3, 4 /) ; p%A%ne = a_ne
! arrays complete
ALLOCATE(p%G(n), p%X_l(n), p%X_u(n))
ALLOCATE(p%C_l(m), p%C_u(m))
ALLOCATE(p%X(n), p%Y(m), p%Z(n))
p%n = n ; p%m = m ; d%f = 1.0_wp ! dimensions & objective constant
p%Hessian_kind = - 1 ; p%gradient_kind = - 1 ! generic quadratic program
p%G = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /) ! objective gradient
p%C_l = (/ 1.0_wp, 2.0_wp /) ! constraint lower bound
p%C_u = (/ 2.0_wp, 2.0_wp /) ! constraint upper bound
p%X_l = (/ - 1.0_wp, - infinity, 1.0_wp, - infinity /) ! variable lower bound
p%X_u = (/ 1.0_wp, infinity, 1.0_wp, 2.0_wp /) ! variable upper bound
CALL QPP_initialize(map, control) ! Initialize control parameters
control%infinity = infinity ! Set infinity
! reorder problem
CALL QPP_reorder(map, control, info, d, p, .TRUE., .TRUE., .TRUE.)
IF (info%status /= 0) & ! Error returns
WRITE(6, "(' QPP_solve exit status = ', I6) ") info%status
WRITE(6, "(' problem now involves ', I1, ' variables and ', &
& I1, ' constraints. f is now', ES12.4)") p%n, p%m, d%f
! re-ordered variables
WRITE(6, "(/, 5X, 'i', 6X, 'v', 11X, 'l', 11X, 'u', 11X, 'z', 11X, &
& 'g', 6X, 'type')")
DO i = 1, d%x_free ! free variables
WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), ' '
END DO
DO i = d%x_free + 1, d%x_l_start - 1 ! non-negativities
WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), '0< '
END DO
DO i = d%x_l_start, d%x_u_start - 1 ! lower-bounded variables
WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), 'l< '
END DO

```

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```

DO i = d%x_u_start, d%x_l_end ! range-bounded variables
 WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), 'l<u'
END DO
DO i = d%x_l_end + 1, d%x_u_end ! upper-bounded variables
 WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), ' <u'
END DO
DO i = d%x_u_end + 1, p%n ! non-positivities
 WRITE(6, 10) i, p%X(i), p%X_l(i), p%X_u(i), p%Z(i), p%G(i), ' <0'
END DO
! re-ordered constraints
WRITE(6, "(/, 5X, 'i', 5x, 'A*v', 10X, 'l', 11X, 'u', 11X, 'y', &
& 6X, 'type')")
DO i = 1, d%c_l_start - 1 ! equality constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l=u'
END DO
DO i = d%c_l_start, d%c_u_start - 1 ! lower-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l<'
END DO
DO i = d%c_u_start, d%c_l_end ! range-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l<u'
END DO
DO i = d%c_l_end + 1, d%c_u_end ! upper-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), ' <u'
END DO
! re-ordered matrices
WRITE(6, 30) 'Hessian', (('H', i, p%H%col(j), p%H%val(j), &
 j = p%H%ptr(i), p%H%ptr(i + 1) - 1), i = 1, p%n) ! Hessian
WRITE(6, 30) 'Jacobian', (('A', i, p%A%col(j), p%A%val(j), &
 j = p%A%ptr(i), p%A%ptr(i + 1) - 1), i = 1, p%m) ! Jacobian
CALL QPP_terminate(map, control, info) ! delete internal workspace
10 FORMAT(I6, 5ES12.4, 2X, A3)
20 FORMAT(I6, 4ES12.4, 2X, A3)
30 FORMAT(/, 1X, A8, /, (:, 3 (1X, A1, '(', 2I2, ')') =', ES12.4, :))
END PROGRAM GALAHAD_QPP_EXAMPLE

```

This produces the following output:

problem now involves 3 variables and 2 constraints. f is now 6.5000E+00

| i | v          | l           | u          | z           | g          | type |
|---|------------|-------------|------------|-------------|------------|------|
| 1 | 0.0000E+00 | -1.0000E+20 | 1.0000E+20 | 0.0000E+00  | 1.0000E+00 |      |
| 2 | 0.0000E+00 | -1.0000E+00 | 1.0000E+00 | 0.0000E+00  | 1.0000E+00 | l<u  |
| 3 | 1.0000E+00 | -1.0000E+20 | 2.0000E+00 | -1.0000E+00 | 4.0000E+00 | <u   |

| i | A*v        | l          | u          | y          | type |
|---|------------|------------|------------|------------|------|
| 1 | 1.0000E+00 | 1.0000E+00 | 1.0000E+00 | 1.0000E+00 | l=u  |
| 2 | 0.0000E+00 | 1.0000E+00 | 2.0000E+00 | 1.0000E+00 | l<u  |

Hessian

H( 1 1) = 2.0000E+00 H( 2 2) = 1.0000E+00 H( 3 2) = 4.0000E+00

Jacobian

A( 1 1) = 1.0000E+00 A( 1 3) = 1.0000E+00 A( 2 1) = 1.0000E+00

A( 2 2) = 2.0000E+00

and corresponds to the reordered problem of minimizing  $v_1^2 + \frac{1}{2}v_2^2 + 4v_3v_2 + v_1 + v_2 + 4v_3 + 6.5$  subject to the general

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

linear constraints  $v_1 + v_3 = 1$ ,  $1 \leq v_1 + 2v_2 \leq 2$ , and simple bounds  $-1 \leq v_1 \leq 1$  and  $v_3 \leq 2$ . Notice how the fixed variable has been removed, and  $f$ ,  $\mathbf{g}$ ,  $\mathbf{c}'$  and  $\mathbf{c}''$  adjusted appropriately.

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```
! sparse co-ordinate storage format
...
! arrays complete
```

by

```
! sparse row-wise storage format
CALL SMT_put(p%H$type, 'SPARSE_BY_ROWS') ! Specify sparse-by-row
CALL SMT_put(p%A$type, 'SPARSE_BY_ROWS') ! storage for H and A
ALLOCATE(p%H%val(h_ne), p%H%col(h_ne), p%H%ptr(n + 1))
ALLOCATE(p%A%val(a_ne), p%A%col(a_ne), p%A%ptr(m + 1))
p%H%val = (/ 1.0_wp, 2.0_wp, -1.0_wp, 5.0_wp, 4.0_wp /) ! Hessian H
p%H%col = (/ 1, 2, 2, 3, 1 /) ! NB lower triangular
p%H%ptr = (/ 1, 2, 3, 5, 6 /) ! Set row pointers
p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%A%col = (/ 1, 2, 2, 3, 4 /)
p%A%ptr = (/ 1, 3, 6 /) ! Set row pointers
! arrays complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
CALL SMT_put(p%H$type, 'DENSE') ! Specify dense
CALL SMT_put(p%A$type, 'DENSE') ! storage for H and A
ALLOCATE(p%H%val(n * (n + 1) / 2))
ALLOCATE(p%A%val(n * m))
p%H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 0.0_wp, -1.0_wp, 5.0_wp, &
 4.0_wp, 0.0_wp, 0.0_wp, 0.0_wp /) ! Hessian
p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, &
 1.0_wp, 1.0_wp /) ! Jacobian
! arrays complete
```

(If instead  $\mathbf{H}$  had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for  $\mathbf{H}$ , and in this case we would instead

```
CALL SMT_put(p%H$type, 'DIAGONAL') ! Specify dense storage for H
ALLOCATE(p%H%val(n))
p%H%val = (/ 1.0_wp, 0.0_wp, 3.0_wp /) ! Hessian values
```

Notice here that zero diagonal entries are stored.)

The solution to the reordered problem is  $(1.6, 0.2, -0.6)$ —this may be found, for example, by using the GALAHAD package GALAHAD\_QPB. To recover the solution to the original problem, insert the following lines just before the above call to QPP\_terminate:

```
p%X(: 3) = (/ 1.6_wp, 0.2_wp, -0.6_wp /)
CALL QPP_get_values(map, info, p, X_val = X_orig)
WRITE(6, "(/, ' solution = ', (4ES12.4))") X_orig(: n)
```

This yields

---

**All use is subject to the conditions of a BSD-3-Clause License.**  
**See <https://www.galahad.rl.ac.uk/download/> for full details.**

```
solution = 2.0000E-01 1.6000E+00 1.0000E+00 -6.0000E-01
```

Finally, suppose we now wish to change the linear inequality constraint to  $1 \leq 2x_1 + x_2 \leq 3$ , and to find out what effect this has on the reordered problem. Then we might insert the following lines just before the above call to QPP\_terminate:

```
! recover constraint bounds
 CALL QPP_restore(map, info, d, p, get_c_bounds = .TRUE.)
! change upper bound
 p%C_u(1) = 3.0_wp
! reorder new problem
 CALL QPP_apply(map, info, d, p, get_c_bounds = .TRUE.)
! re-ordered new constraints
 WRITE(6, "(/, 5X, 'i', 5X, 'A*v', 10X, 'l', 11X, 'u', 11X, 'y',
&
 6X, 'type')")
 DO i = 1, d%c_l_start - 1
 ! equality constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l=u'
 END DO
 DO i = d%c_l_start, d%c_u_start - 1
 ! lower-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l<'
 END DO
 DO i = d%c_u_start, d%c_l_end
 ! range-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), 'l<u'
 END DO
 DO i = d%c_l_end + 1, d%c_u_end
 ! upper-bounded constraints
 WRITE(6, 20) i, p%C(i), p%C_l(i), p%C_u(i), p%Y(i), '<u'
 END DO
```

This gives

| i | A*v        | l          | u          | y          | type |
|---|------------|------------|------------|------------|------|
| 1 | 1.0000E+00 | 1.0000E+00 | 1.0000E+00 | 1.0000E+00 | l=u  |
| 2 | 0.0000E+00 | 1.0000E+00 | 3.0000E+00 | 1.0000E+00 | l<u  |

which is to say that the inequality constraint for the reordered problem is now  $1 \leq v_1 + 2v_2 \leq 3$ .