GALAHAD QP

USER DOCUMENTATION GALAHAD Optimization Library version 3.0

1 SUMMARY

This package provides a common interface to other GALAHAD packages for solving the **quadratic programming problem**

$$\text{minimize } q(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{g}^T\mathbf{x} + f$$

subject to the general linear constraints

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

and the simple bound constraints

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

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

ATTRIBUTES — Versions: `GALAHAD_QP_single`, `GALAHAD_QP_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_TOOLS`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPD`, `GALAHAD_SORT`, `GALAHAD_SCALE`, `GALAHAD_PRESOLVE`. `GALAHAD_MOP`, `GALAHAD_QPA`, `GALAHAD_QPB`, `GALAHAD_QPC`, `GALAHAD_CQP`, `GALAHAD_DQP`, **Date:** January 2011. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

Access to the package requires a `USE` statement such as

Single precision version

```
USE GALAHAD_QP_single
```

Double precision version

```
USE GALAHAD_QP_double
```

If it is required to use both modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `QP_time_type`, `QP_control_type`, `QP_inform_type` and `QP_data_type` (Section 2.3) and the subroutines `QP_initialize`, `QP_solve`, `QP_terminate`, (Section 2.4) and `QP_read_specfile` (Section 2.6) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

When they are explicitly available, Both the Hessian matrix $\mathbf{H}$ and the constraint Jacobian $\mathbf{A}$, the matrix whose rows are the vectors $\mathbf{a}_i^T$, $i = 1,\ldots,m$, 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 $i = 1,\ldots,m$, $j = 1,\ldots,n$. 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 a 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 $l =$ `A%ptr`(i), ..., `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 diagonals 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 $\mathbf{A}$.

2.2 OpenMP

OpenMP may be used by the `GALAHAD_QP` package to provide parallelism for some solver options in shared memory environments. See the documentation for the GALAHAD package `SLS` for more details. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of `-openmp`). The number of threads may be controlled at runtime by setting the environment variable `OMP_NUM_THREADS`.

The code may be compiled and run in serial mode.

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 default `INTEGER`, that holds the number of rows in the matrix.

`n` is a scalar component of type default `INTEGER`, that holds the number of columns in the matrix.

`ne` is a scalar variable of type default `INTEGER`, that holds the number of matrix entries.

`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 default `REAL` (double precision in `GALAHAD_QP_double`) 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 default `INTEGER`, 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 default `INTEGER`, 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 default `INTEGER`, and dimension at least `m + 1`, that may hold the pointers to the first entry in each row (see §2.1.3).

2.3.2 The derived data type for holding the problem

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

`n` is a scalar variable of type default `INTEGER`, that holds the number of optimization variables, n.

`m` is a scalar variable of type default `INTEGER`, that holds the number of general linear constraints, m.

`H` is scalar variable of type `SMT_TYPE` that holds the Hessian matrix **H** when it is available explicitly. 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 `QP_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 default `INTEGER`, that holds the number of entries in the **lower triangular** part of **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 default `REAL` (double precision in `GALAHAD_QP_double`), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.1.

`H%row` is a rank-one allocatable array of type default `INTEGER`, that holds the row indices of the **lower triangular** part of **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 default `INTEGER`, that holds the column indices of the **lower triangular** part of **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 default `INTEGER`, that holds the starting position of each row of the **lower triangular** part of **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.

G is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_QP_double), that holds the gradient $\mathbf{g}$ of the linear term of the quadratic objective function. The j-th component of G, $j = 1,\ldots,n$, contains $\mathbf{g}_j$.

f is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that holds the constant term, f, in the objective function.

A is scalar variable of type SMT_TYPE that holds the Jacobian matrix $\mathbf{A}$ when it is available explicitly. 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 QP_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 default INTEGER, 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 default REAL (double precision in GALAHAD_QP_double), 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 default INTEGER, 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 default INTEGER, 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 default INTEGER, 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 default REAL (double precision in GALAHAD_QP_double), that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i-th component of C_l, $i = 1,\ldots,m$, contains $\mathbf{c}_i^l$. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

C_u is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_QP_double), that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains $\mathbf{c}_i^u$. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than infinity, where infinity is a component of the control array control (see Section 2.3.3).

X_l is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_QP_double), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}_j^l$. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

X_u is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_QP_double), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}_j^u$. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.3.3).

X is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_QP_double), that holds the values $\mathbf{x}$ of the optimization variables. The j-th component of X, $j = 1, \ldots, n$, contains x_j.

C is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_QP_double), that holds the values $\mathbf{Ax}$ of the constraints. The i-th component of C, $i = 1, \ldots, m$, contains $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$.

Y is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_QP_double), 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, $i = 1, \ldots, m$, contains y_i.

Z is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_QP_double), 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, $j = 1, \ldots, n$, contains z_j.

2.3.3 The derived data type for holding control parameters

The derived data type QP_control_type is used to hold controlling data. Default values may be obtained by calling QP_initialize (see Section 2.4.1), while components may also be changed by calling QP_read_specfile (see Section 2.6.1). The components of QP_control_type are:

error is a scalar variable of type default INTEGER, that holds the stream number for error messages. Printing of error messages in QP_solve and QP_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type default INTEGER, that holds the stream number for informational messages. Printing of informational messages in QP_solve is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type default INTEGER, that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

maxit is a scalar variable of type default INTEGER, that holds the maximum number of iterations which will be allowed in QP_solve. The default is maxit = 1000.

scale is a scalar variable of type default INTEGER, that is used to control problem scaling prior to solution. Possible values and their consequences are:

0. no scaling will be performed
1. scaling will be performed to try to map all variables and constraints to have values between 0 and 1.
2. the symmetric Curtis-Reid method will be applied to normalize the rows of the matrix

$$\mathbf{K} = \begin{pmatrix} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & 0 \end{pmatrix}.$$

3. the unsymmetric Curtis-Reid method will be applied to normalize the rows and columns of $\mathbf{A}$.
4. scaling will be applied to equilibrate the norms of the rows of $\mathbf{A}$.
5. strategy 2 will be followed by strategy 4.
6. strategy 3 will be followed by strategy 4.
7. scaling will be applied to equilibrate the rows and columns of $\mathbf{K}$ using the Sinkhorn-Knopp strategy.

If the negative of one of the above values is given, scaling will be performed prior to (rather than after) any pre-solving (see %presolve below).

While scaling may improve the performance of the algorithm, it might also degrade it, so scaling should be used with caution. The default is scale = 0.

`infinity` is a scalar variable of type default `REAL` (double precision in `GALAHAD_QP_double`), 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` = 10^{19}.

`presolve` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if a pre-solve phase is to be applied to the data prior to the actual solution and `.FALSE.` otherwise. Pre-solving aims to reduce the size of the problem using the data to deduce at optimality which variables must be active, which constraints are inactive, etc. This may sometimes result in a worse-conditioned problem. The default is `presolve = .FALSE.`.

`space_critical` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if space is critical when allocating arrays and `.FALSE.` otherwise. The package may run faster if `space_critical` is `.FALSE.` but at the possible expense of a larger storage requirement. The default is `space_critical = .FALSE.`.

`deallocate_error_fatal` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes to terminate execution if a deallocation fails, and `.FALSE.` if an attempt to continue will be made. The default is `deallocate_error_fatal = .FALSE.`.

`quadratic_programming_solver` is a scalar variable of type default `CHARACTER` and length 30, that specifies which quadratic programming solver to use. Possible values are

`qpa` if the GALAHAD active-set solver `QPA` is desired.

`qpb` if the GALAHAD interior-point solver `QPB` is desired.

`qpc` if the GALAHAD interior-point/active-set crossover solver `QPC` is desired.

`cqp` if the GALAHAD convex interior-point solver `CQP` is desired.

`dqp` if the GALAHAD strictly-convex dual gradient projection solver `DQP` is desired.

Other solvers may be added in the future.

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

`SCALE_control` is a scalar variable of type `SCALE_control_type` whose components are used to control any problem scaling performed by the package `GALAHAD_SCALE`. See the specification sheet for the package `GALAHAD_SCALE` for details, and appropriate default values.

`PRESOLVE_control` is a scalar variable of type `PRESOLVE_control_type` whose components are used to control any pre-solving performed by the package `GALAHAD_PRESOLVE`. See the specification sheet for the package `GALAHAD_PRESOLVE` for details, and appropriate default values.

`QPA_control` is a scalar variable of type `QPA_control_type` whose components are used to control the package `GALAHAD_QPA` if selected by `quadratic_programming_solver` (see above). See the specification sheet for the package `GALAHAD_QPA` for details, and appropriate default values.

`QPB_control` is a scalar variable of type `QPB_control_type` whose components are used to control the package `GALAHAD_QPB` if selected by `quadratic_programming_solver` (see above). See the specification sheet for the package `GALAHAD_QPB` for details, and appropriate default values.

`QPC_control` is a scalar variable of type `QPC_control_type` whose components are used to control the package `GALAHAD_QPC` if selected by `quadratic_programming_solver` (see above). See the specification sheet for the package `GALAHAD_QPC` for details, and appropriate default values.

CQP_control is a scalar variable of type CQP_control_type whose components are used to control the package GALAHAD_CQP if selected by quadratic_programming_solver (see above). See the specification sheet for the package GALAHAD_CQP for details, and appropriate default values.

DQP_control is a scalar variable of type DQP_control_type whose components are used to control the package GALAHAD_DQP if selected by quadratic_programming_solver (see above). See the specification sheet for the package GALAHAD_DQP for details, and appropriate default values.

2.3.4 The derived data type for holding timing information

The derived data type QP_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of QP_time_type are:

total is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the total CPU time spent in the package.

presolve is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the CPU time spent pre-solving the problem.

scale is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the CPU time spent scaling the problem.

solve is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the CPU time spent actually solving the quadratic program.

clock_total is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the total elapsed system clock time spent in the package.

clock_presolve is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the elapsed system clock time spent pre-solving the problem.

clock_scale is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the elapsed system clock time spent scaling the problem.

clock_solve is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that gives the elapsed system clock time spent actually solving the quadratic program.

2.3.5 The derived data type for holding informational parameters

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

status is a scalar variable of type default INTEGER, that gives the exit status of the algorithm. See Section 2.5 for details.

alloc_status is a scalar variable of type default INTEGER, that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

obj is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that holds the value of the objective function at the best estimate of the solution found.

primal_infeasibility is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that holds the norm of the violation of primal optimality (see Section 2.3.4) at the best estimate of the solution found.

dual_infeasibility is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that holds the norm of the violation of dual optimality (see Section 2.3.4) at the best estimate of the solution found.

complementary_slackness is a scalar variable of type default REAL (double precision in GALAHAD_QP_double), that holds the norm of the violation of complementary slackness (see Section 2.3.4) at the best estimate of the solution found.

time is a scalar variable of type QP_time_type whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.3.4).

SCALE_inform is a scalar variable of type SCALE_inform_type whose components are used to provide information about any problem scaling performed by the package GALAHAD_SCALE. See the specification sheet for the package GALAHAD_SCALE for details.

PRESOLVE_inform is a scalar variable of type PRESOLVE_inform_type whose components are used to provide information about any pre-solving performed by the package GALAHAD_PRESOLVE. See the specification sheet for the package GALAHAD_PRESOLVE for details.

QPA_inform is a scalar variable of type QPA_inform_type whose components are used to provide information about the progress of the quadratic programming package GALAHAD_QPA, iff used. See the specification sheet for the package GALAHAD_QPA for details.

QPB_inform is a scalar variable of type QPB_inform_type whose components are used to provide information about the progress of the quadratic programming package GALAHAD_QPB, iff used. See the specification sheet for the package GALAHAD_QPB for details.

QPC_inform is a scalar variable of type QPC_inform_type whose components are used to provide information about the progress of the quadratic programming package GALAHAD_QPC, iff used. See the specification sheet for the package GALAHAD_QPC for details.

CQP_inform is a scalar variable of type CQP_inform_type whose components are used to provide information about the progress of the quadratic programming package GALAHAD_CQP, iff used. See the specification sheet for the package GALAHAD_CQP for details.

DQP_inform is a scalar variable of type DQP_inform_type whose components are used to provide information about the progress of the quadratic programming package GALAHAD_DQP, iff used. See the specification sheet for the package GALAHAD_DQP for details.

2.3.6 The derived data type for holding problem data

The derived data type QP_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of QP procedures. This data should be preserved, untouched, from the initial call to QP_initialize to the final call to QP_terminate.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.6 for further features):

1. The subroutine QP_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.

2. The subroutine QP_solve is called to solve the problem.
3. The subroutine QP_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by QP_solve, at the end of the solution process.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL QP_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type QP_data_type (see Section 2.3.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type QP_control_type (see Section 2.3.3). On exit, control contains default values for the components as described in Section 2.3.3. These values should only be changed after calling QP_initialize.

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

2.4.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

```
CALL QP_solve( prob, data, control, inform, C_stat, B_stat )
```

prob is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.3.2). It is used to hold data about the problem being solved. The user must allocate all the array components, and set values for all components.

The components prob%X and prob%Z must be set to initial estimates of the primal variables, **x** and dual variables for the bound constraints, **z**, respectively. Inappropriate initial values will be altered, so the user should not be overly concerned if suitable values are not apparent, and may be content with merely setting prob%X=0.0 and prob%Z=0.0.

On exit, the components prob%X and prob%Z will contain the best estimates of the primal variables **x**, and dual variables for the bound constraints **z**, respectively. **Restrictions:** prob%n > 0 and (if **H** is provided) prob%H%ne ≥ -2. prob%H_type $\in$ {'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' }.

data is a scalar INTENT(INOUT) argument of type QP_data_type (see Section 2.3.6). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to QP_initialize.

control is a scalar INTENT(IN) argument of type QP_control_type (see Section 2.3.3). Default values may be assigned by calling QP_initialize prior to the first call to QP_solve.

inform is a scalar INTENT(INOUT) argument of type QP_inform_type (see Section 2.3.5). A successful call to QP_solve is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

C_stat is a rank-one INTENT(INOUT) array argument of dimension p%m and type default INTEGER, that indicates which of the general linear constraints are in the current working set. Possible values for C_stat(i), i$= 1, \ldots,$ p%m, and their meanings are

<0 the i-th general constraint is in the working set, on its lower bound,

>0 the i-th general constraint is in the working set, on its upper bound, and

0 the i-th general constraint is not in the working set.

Suitable values must be supplied if `control%qpa_control%cold_start = 0` on entry, but need not be provided for other input values of `control%cold_start`. Inappropriate values will be ignored. On exit, `C_stat` will contain values appropriate for the ultimate working set.

`B_stat` is a rank-one `INTENT(INOUT)` array argument of dimension `p%n` and type default `INTEGER`, that indicates which of the simple bound constraints are in the current working set. Possible values for `B_stat(j)`, j= 1,..., `p%n`, and their meanings are

<0 the j-th simple bound constraint is in the working set, on its lower bound,

>0 the j-th simple bound constraint is in the working set, on its upper bound, and

0 the j-th simple bound constraint is not in the working set.

Suitable values must be supplied if `control%qpa_control%cold_start = 0` on entry, but need not be provided for other input values of `control%cold_start`. Inappropriate values will be ignored. On exit, `B_stat` will contain values appropriate for the ultimate working set.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL QP_terminate( data, control, inform )
```

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

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

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

2.5 Warning and error messages

A negative value of `inform%status` on exit from `QP_solve` or `QP_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 and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-3. One of the restrictions `prob%n` > 0 or the requirement that `prob%H_type` contain its relevant string `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'DIAGONAL'` when **H** is available, has been violated.

-4. The bound constraints are inconsistent.

-7. The objective function appears to be unbounded from below on the feasible set.

-26. The requested quadratic programming solver is not available.

-61. An error has occurred when scaling the problem. See inform%SCALE_inform%status and the documentation for the package GALAHAD_SCALE for further details.

-62. An error has occurred when pre-solving the problem. See inform%PRESOLVE_inform%status and the documentation for the package GALAHAD_PRESOLVE for further details.

-63. An error has occurred when solving the quadratic program using GALAHAD_QPA. See inform%QPA_inform%status and the documentation for the package GALAHAD_QPA for further details.

-64. An error has occurred when solving the quadratic program using GALAHAD_QPB. See inform%QPB_inform%status and the documentation for the package GALAHAD_QPB for further details.

-65. An error has occurred when solving the quadratic program using GALAHAD_QPC. See inform%QPC_inform%status and the documentation for the package GALAHAD_QPC for further details.

-66. An error has occurred when solving the quadratic program using GALAHAD_CQP. See inform%CQP_inform%status and the documentation for the package GALAHAD_CQP for further details.

-67. An error has occurred when solving the quadratic program using GALAHAD_DQP. See inform%DQP_inform%status and the documentation for the package GALAHAD_DQP for further details.

2.6 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type QP_control_type (see Section 2.3.3), by reading an appropriate data specification file using the subroutine QP_read_specfile. This facility is useful as it allows a user to change QP control parameters without editing and recompiling programs that call QP.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by QP_read_specfile must start with a "BEGIN QP" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by QP_read_specfile .. )
  BEGIN CQP
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by QP_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN QP" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN QP SPECIFICATION
```

and

```
END QP SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN QP" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when QP_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by QP_read_specfile.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL QP_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type QP_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling QP_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of control that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
scale-problem	%scale	integer
infinity-value	%infinity	real
pre-solve-problem	%presolve	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

device is a scalar INTENT(IN) argument of type default INTEGER, that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

2.7 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control-%out. If control%print_level > 0, a few lines of output indicating the progress of the scaling, pre-solve and solve phases will be given. More detailed output for these phases may be obtained by setting the values control%package_-control%print_level appropriately (see Section 2.3.5).

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `QP_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_TOOLS`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPD`, `GALAHAD_SORT`, `GALAHAD_SCALE`, `GALAHAD_PRESOLVE`. `GALAHAD_MOP`, `GALAHAD_QPA`, `GALAHAD_QPB`, `GALAHAD_QPC`, `GALAHAD_CQP` and `GALAHAD_DQP`.

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: `prob%n` > 0, `prob%m` ≥ 0, `prob%A_type` and `prob%H_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }. (if **H** and **A** are explicit).

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

4 METHOD

The required solution **x** necessarily satisfies the primal optimality conditions

$$\mathbf{A}\mathbf{x} = \mathbf{c} \tag{4.1}$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \;\; \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \tag{4.2}$$

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{A}^T\mathbf{y} + \mathbf{z}, \;\; \mathbf{y} = \mathbf{y}^l + \mathbf{y}^u \text{ and } \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \tag{4.3}$$

and

$$\mathbf{y}^l \geq 0, \;\; \mathbf{y}^u \leq 0, \;\; \mathbf{z}^l \geq 0 \text{ and } \mathbf{z}^u \leq 0, \tag{4.4}$$

and the complementary slackness conditions

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

where the vectors **y** and **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.

See the documentation for the individual quadratic programming solvers for details of how they try to attain (4.1)–(4.5).

5 EXAMPLE OF USE

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

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 2 & 1 \\ & 1 & 3 \end{pmatrix}, \;\; \mathbf{g} = \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}, \;\; \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ -\infty \end{pmatrix}, \;\; \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 2 \end{pmatrix}$$

and

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

in sparse co-ordinate format, we may use the GALAHAD quadratic programming solver `QPA` with both a pre-solve and Sinkhorn-Knopp scaling using the following code:

```
! THIS VERSION: GALAHAD 2.4 - 10/01/2011 AT 07:30 GMT.
   PROGRAM GALAHAD_QP_EXAMPLE
   USE GALAHAD_QP_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_problem_type ) :: p
   TYPE ( QP_data_type ) :: data
   TYPE ( QP_control_type ) :: control
   TYPE ( QP_inform_type ) :: inform
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: C_stat, B_stat
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ) )
   ALLOCATE( p%X( n ), p%Y( m ), p%Z( n ) )
   ALLOCATE( B_stat( n ), C_stat( m ) )
   p%new_problem_structure = .TRUE.           ! new structure
   p%n = n ; p%m = m ; p%f = 1.0_wp           ! dimensions & objective constant
   p%G = (/ 0.0_wp, 2.0_wp, 0.0_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, - infinity /) ! variable lower bound
   p%X_u = (/ 1.0_wp, infinity, 2.0_wp /)     ! variable upper bound
   p%X = 0.0_wp ; p%Y = 0.0_wp ; p%Z = 0.0_wp ! start from zero
!  sparse co-ordinate storage format
   CALL SMT_put( p%H%type, 'COORDINATE', s )     ! Specify co-ordinate
   CALL SMT_put( p%A%type, 'COORDINATE', s )     ! 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, 3.0_wp /) ! Hessian H
   p%H%row = (/ 1, 2, 2, 3 /)                     ! NB lower triangle
   p%H%col = (/ 1, 2, 1, 3 /) ; p%H%ne = h_ne
   p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
   p%A%row = (/ 1, 1, 2, 2 /)
   p%A%col = (/ 1, 2, 2, 3 /) ; p%A%ne = a_ne
! problem data complete
   CALL QP_initialize( data, control, inform )  ! Initialize control parameters
   control%infinity = infinity                  ! Set infinity
   control%quadratic_programming_solver = 'qpa' ! use QPA
   control%scale = 7                            ! Sinkhorn-Knopp scaling
   control%presolve = .TRUE.                    ! Pre-solve the problem
   CALL QP_solve( p, data, control, inform, C_stat, B_stat ) ! Solve
   IF ( inform%status == 0 ) THEN               !  Successful return
     WRITE( 6, "( ' QP: ', I0, ' QPA iterations  ', /,                        &
    &     ' Optimal objective value =',                                      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )             &
     inform%QPA_inform%iter, inform%obj, p%X
```

```
   ELSE                                        !  Error returns
     WRITE( 6, "( ' QP_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL QP_terminate( data, control, inform )  !  delete internal workspace
   END PROGRAM GALAHAD_QP_EXAMPLE
```

This produces the following output:

```
 QP: 5 QPA iterations
 Optimal objective value =  5.4706E+00
 Optimal solution =   5.8824E-02  8.8235E-01  1.1176E+00
```

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
...
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL SMT_put( p%H%type, 'SPARSE_BY_ROWS' )  ! Specify sparse-by-row
   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, 3.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 3, 3 /)                     ! NB lower triangular
   p%H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( p%H%type, 'DENSE' )  ! Specify dense
   ALLOCATE( p%H%val( n * ( n + 1 ) / 2 ) )
   p%H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 0.0_wp, 1.0_wp, 3.0_wp /) ! Hessian
! problem data complete
```

respectively.

If instead **H** had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 2 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for **H**, and in this case we would instead

```
   CALL SMT_put( prob%H%type, 'DIAGONAL' )  ! Specify dense storage for H
   ALLOCATE( p%H%val( n ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp /) ! Hessian values
```