GALAHAD QPA

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package uses a working-set method to solve the ℓ_1 **quadratic programming problem**

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad q(\mathbf{x}) + \rho_g v_g(\mathbf{x}) + \rho_b v_b(\mathbf{x}) \tag{1.1}$$

involving the quadratic objective

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

and the infeasibilities

$$v_g(\mathbf{x}) = \sum_{i=1}^{m} \max(c_i^l - \mathbf{a}_i^T\mathbf{x}, 0) + \sum_{i=1}^{m} \max(\mathbf{a}_i^T\mathbf{x} - c_i^u, 0)$$

and

$$v_b(\mathbf{x}) = \sum_{j=1}^{n} \max(x_j^l - x_j, 0) + \sum_{j=1}^{n} \max(x_j - x_j^u, 0),$$

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 scalars f, ρ_g and ρ_b 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 package may also be used to solve the **quadratic programming problem**

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad q(\mathbf{x}), \tag{1.2}$$

subject to the general linear constraints

$$c_i^l \leq \mathbf{a}_i^T\mathbf{x} \leq c_i^u, \quad i = 1,\ldots,m, \tag{1.3}$$

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1,\ldots,n, \tag{1.4}$$

by automatically adjusting the parameters ρ_g and ρ_b in (1.1).

Similarly, the package is capable of solving the **bound-constrained ℓ_1 quadratic programming problem**

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad q(\mathbf{x}) + \rho_g v_g(\mathbf{x}), \tag{1.5}$$

subject to the simple bound constraints (1.4), by automatically adjusting ρ_b in (1.1).

If the matrix $\mathbf{H}$ is positive semi-definite, a global solution is found. However, if $\mathbf{H}$ is indefinite, the procedure may find a (weak second-order) critical point that is not the global solution to the given problem.

N.B. In many cases, the alternative GALAHAD quadratic programming package `GALAHAD_QPB` is faster, and thus to be preferred.

ATTRIBUTES — Versions: `GALAHAD_QPA_single, GALAHAD_QPA_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NORMS`, `GALAHAD_SPACE`, `GALAHAD_RAND`, `GALAHAD_TOOLS`, `GALAHAD_ROOTS`, `GALAHAD_SORT`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPP`, `GALAHAD_QPD`, `GALAHAD_SLS`, `GALAHAD_SPECFILE`, `GALAHAD_SCU`. **Date:** October 2001. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

The package is available using both single and double 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_QPA_single
```

with the obvious substitution `GALAHAD_QPA_double`, `GALAHAD_QPA_single_64` and `GALAHAD_QPA_double_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`, `QPA_time_type`, `QPA_control_type`, `QPA_inform_type` and `QPA_data_type` (Section 2.4) and the subroutines `QPA_initialize`, `QPA_solve`, `QPA_terminate`, (Section 2.5) and `QPA_read_specfile` (Section 2.7) 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 $\mathbf{a}_i^T$, $i = 1,\ldots,m$, 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`. 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), $\ldots$,`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 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 and DOUBLE PRECISION for the double precision cases, and correspond to rp_ = real32 and rp_ = real64, respectively, as supplied 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 Parallel usage

OpenMP may be used by the GALAHAD_QPA package to provide parallelism for some solvers 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.

MPI may also be used by the package to provide parallelism for some solvers in a distributed memory environment. To use this form of parallelism, MPI must be enabled at runtime by using the correct compiler flag (usually some variant of -lmpi). Although the MPI process will be started automatically when required, it should be stopped by the calling program once no further use of this form of parallelism is needed. Typically, this will be via statements of the form

```
CALL MPI_INITIALIZED( flag, ierr )
IF ( flag ) CALL MPI_FINALIZE( ierr )
```

The code may be compiled and run in serial mode.

2.4 The derived data types

Six derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrices **A** and **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 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 §2.4.2).

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 **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 hold 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 m + 1, that may hold the pointers to the first entry in each row (see §2.1.3).

2.4.2 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:

new_problem_structure is a scalar variable of type default LOGICAL, that is .TRUE. if this is the first (or only) problem in a sequence of problems with identical "structure" to be attempted, and .FALSE. if a previous problem with the same "structure" (but different numerical data) has been solved. Here, the term "structure" refers both to the sparsity patterns of the Jacobian matrices **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).

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.

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H**. 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 QPA_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', istat )
```

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 **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 **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 **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 **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 **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 REAL(rp_), that holds the gradient **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 REAL(rp_), that holds the constant term, f, in the objective function.

rho_g is a scalar variable of type REAL(rp_), that holds the parameter, ρ_g, used to weight the infeasibility term $v_g(\mathbf{x})$.

rho_b is a scalar variable of type REAL(rp_), that holds the parameter, ρ_b, used to weight the infeasibility term $v_b(\mathbf{x})$.

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 QPA_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', istat )
```

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, $i = 1,\ldots,m$, contains $\mathbf{c}^l_i$. 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 2.4.3).

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), 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 $\mathbf{c}^u_i$. 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 2.4.3).

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 the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}^l_j$. 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 2.4.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), 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 $\mathbf{x}^u_j$. 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 2.4.3).

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, $j = 1,\ldots,n$, 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, $j = 1,\ldots,n$, contains z_j.

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, $i = 1,\ldots,m$, contains $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$.

Y is a rank-one allocatable array of dimension m and type 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, $i = 1,\ldots,m$, contains y_i.

2.4.3 The derived data type for holding control parameters

The derived data type QPA_control_type is used to hold controlling data. Default values may be obtained by calling QPA_initialize (see Section 2.5.1), while components may also be changed by calling GALAHAD_QPA_read_spec (see Section 2.7.1). The components of QPA_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 QPA_solve and QPA_terminate is suppressed if error ≤ 0. The default is error = 6.

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

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

start_print is a scalar variable of type INTEGER(ip_), that specifies the first iteration for which printing will occur in QPA_solve. If start_print is negative, printing will occur from the outset. The default is start_print = -1.

stop_print is a scalar variable of type INTEGER(ip_), that specifies the last iteration for which printing will occur in QPA_solve. If stop_print is negative, printing will occur once it has been started by start_print. The default is stop_print = -1.

factor is a scalar variable of type INTEGER(ip_), that indicates the type of factorization of the preconditioner to be used. Possible values are:

0 the type is chosen automatically on the basis of which option looks likely to be the most efficient.

1 a Schur-complement factorization will be used.

2 an augmented-system factorization will be used.

The default is factor = 0.

max_col is a scalar variable of type INTEGER(ip_), that specifies the maximum number of nonzeros in a column of $\mathbf{A}$ which is permitted by the Schur-complement factorization. The default is max_col = 35.

max_sc is a scalar variable of type INTEGER(ip_), that specifies the maximum number of columns permitted in the Schur complement of the reference matrix (see Section 4) before a refactorization is triggered. The default is max_sc = 75.

itref_max is a scalar variable of type INTEGER(ip_), that specifies the maximum number of iterative refinements allowed with each application of the preconditioner. The default is itref_max = 1.

cg_maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of conjugate-gradient inner iterations that may be performed during the computation of each search direction in QPA_solve. If cg_itmax is set to a negative number, it will be reset by QPA_solve to the dimension of the relevant linear system +1. The default is cg_itmax = -1.

precon is a scalar variable of type INTEGER(ip_), that specifies which preconditioner to be used to accelerate the conjugate-gradient inner iteration. Possible values are:

- 0 the type is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different preconditioners may be used at different stages.
- 1 a full factorization using the Hessian, which is equivalent to replacing the conjugate gradient inner iteration by a direct method. The Hessian may be perturbed to ensure that the resultant matrix is a preconditioner.
- 2 the Hessian matrix is replaced by the identity matrix.
- 3 the Hessian matrix is replaced by a band of given semi-bandwidth (see nsemib below).
- 4 the Hessian matrix terms in the current reference matrix (see Section 4) are replaced by the identity matrix.
- 5 the Hessian matrix terms outside a band of given semi-bandwidth in the current reference matrix are replaced by zeros (see nsemib below).

The default is precon = 0.

nsemib is a scalar variable of type INTEGER(ip_), that specifies the semi-bandwidth of the band preconditioner when precon = 3, if appropriate. The default is nsemib = 5.

full_max_fill is a scalar variable of type INTEGER(ip_). If the ratio of the number of nonzeros in the factors of the reference matrix (see Section 4) to the number of nonzeros in the matrix itself exceeds full_max_fil, and the preconditioner is being selected automatically (precon = 0), a banded approximation (see precon = 3) will be used instead. The default is full_max_fill = 10.

deletion_strategy is a scalar variable of type INTEGER(ip_), that specifies the rules used to determine which constraint to remove from the working set (see Section 4) when necessary to ensure further progress towards the solution. Possible values are:

- 0 the constraint whose Lagrange multiplier most violates its required optimality bound will be removed.
- 1 the most-recently added constraint whose Lagrange multiplier violates its required optimality bound will be removed.
- k $>$ 1 among the k most-recently added constraints whose Lagrange multipliers violates their required optimality bounds, the one which most violates its bound will be removed.

The default is deletion_strategy = 0.

restore_problem is a scalar variable of type INTEGER(ip_), that specifies how much of the input problem is to be retored on output. Possible values are:

- 0 nothing is restored.
- 1 the vector data $\mathbf{g}$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, and $\mathbf{x}^u$ will be restored to their input values.
- 2 the entire problem, that is the above vector data along with the Hessian matrix $\mathbf{H}$ and the Jacobian matrix $\mathbf{A}$, will be restored.

The default is restore_problem = 2.

monitor_residuals is a scalar variable of type INTEGER(ip_), that specifies the frequency at which working constraint residuals will be monitored to ensure feasibility. The residuals will be monitored every monitor_residual iterations. The default is monitor_residuals = 1.

cold_start is a scalar variable of type INTEGER(ip_), that controls the initial working set (see Section 4)). Possible values are:

- 0 a "warm start" will be performed. The values set in C_stat and B_stat indicate which constraints will be included in the initial working set. (see C_stat and B_stat in Section 2.5.2).
- 1 the constraints "active" at p%X (see Section 2.5.2) will determine the initial working set.
- 2 the initial working set will be empty.
- 3 the initial working set will only contain equality constraints.
- 4 the initial working set will contain as many active constraints as possible, chosen (in order) from equality constraints, simple bounds, and finally general linear constraints.

The default is cold_start = 3.

infeas_check_interval is a scalar variable of type INTEGER(ip_), that gives the number of iterations that are permitted before the infeasibility is checked for improvement. The default is infeas_check_interval = 100.

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 = 10^{19}.

feas_tol is a scalar variable of type REAL(rp_), that specifies the maximum amount by which a constraint may be violated and yet still be considered to be satisfied. The default is feas_tol = $u^{3/4}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPA_double).

obj_unbounded is a scalar variable of type default REAL(rp_), that specifies smallest value of the objective function that will be tolerated before the problem is declared to be unbounded from below. The default is potential_unbounded = $-u^{-2}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPA_double).

increase_rho_g_factor is a scalar variable of type REAL(rp_), that gives the factor by which the current penalty parameter ρ_g for the general constraints may be increased when solving quadratic programs. The default is increase_rho_g_factor = 2.

increase_rho_b_factor is a scalar variable of type REAL(rp_), that gives the factor by which the current penalty parameter ρ_b for the simple bound constraints may be increased when solving quadratic programs. The default is increase_rho_b_factor = 2.

infeas_g_improved_by_factor is a scalar variable of type default REAL(rp_), that specifies the relative improvement in the infeasibility that must be achieved when solving quadratic programs if the current value of ρ_g is to be maintained. Specifically if the infeasibility of the general constraints has not fallen by at least a factor infeas_g_improved_by_factor during the previous infeas_check_interval iterations, the penalty parameter will be increased by a factor increase_rho_g_factor. The default is infeas_improved_g_by_factor = 0.75.

infeas_b_improved_by_factor is a scalar variable of type default REAL(rp_), that specifies the relative improvement in the infeasibility that must be achieved when solving quadratic programs if the current value of ρ_b is to be maintained. Specifically if the infeasibility of the simple bound constraints has not fallen by at least a factor infeas_b_improved_by_factor during the previous infeas_check_interval iterations, the penalty parameter will be increased by a factor increase_rho_b_factor. The default is infeas_improved_b_by_factor = 0.75.

pivot_tol_for_dependencies is a scalar variable of type default REAL(rp_), that holds the relative threshold pivot tolerance used by the matrix factorization when attempting to detect linearly dependent constraints. A value larger than pivot_tol is appropriate. See the documentation for the package SLS for details. The default is pivot_tol_for_dependencies = 0.5.

multiplier_tol is a scalar variable of type REAL(rp_). Any dual variable or Lagrange multiplier which is less than multiplier_tol outside its optimal interval will be regarded as being acceptable when checking for optimality. The default is zero_pivot = $\sqrt{u}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPA_double).

inner_stop_relative and inner_stop_absolute are scalar variables of type REAL(rp_), that hold the relative and absolute convergence tolerances for the inner iteration (search direction) problem. and correspond to the values control%stop_relative and control%stop_absolute in the GALAHAD package GALAHAD_GLTR. The defaults are inner_stop_relative = 0.0 and inner_stop_absolute = $\sqrt{u}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPA_double).

cpu_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is cpu_time_limit = - 1.0.

clock_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is clock_time_limit = - 1.0.

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_QPA 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..

solve_qp is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm will aim to solve the quadratic programming problem (1.2)–(1.4) (by adjusting ρ_g and ρ_b as necessary), and and .FALSE. if the solution of the l_1-quadratic program for the specified values of ρ_g and ρ_b is required. The default is solve_qp = .FALSE..

solve_within_bounds is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm will aim to solve the bound-constrained ℓ_1 quadratic programming problem (1.4)–(1.5) (by adjusting ρ_b as necessary), and and .FALSE. otherwise. If solve_qp is .TRUE., the value of solve_within_bounds will be ignored. The default is solve_within_bounds = .FALSE..

ramdomize is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to perturb the constraint bounds by small random quantities during the first stage of the solution process, and .FALSE. otherwise. Any randomization will ultimately be removed. Randomization helps when solving degenerate problems and is usually to be recommended. The default is randomize = .TRUE..

symmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any symmetric linear system that might arise. Current possible choices are 'sils', 'ma27', 'ma57', 'ma77', 'ma86', 'ma97', ssids, 'pardiso' and 'wsmp', although only 'sils' and, for OMP 4.0-compliant compilers, 'ssids' are installed by default. See the documentation for the GALAHAD package SLS for further details. The default is symmetric_linear_solver = 'sils'.

definite_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any symmetric positive-definite linear system that might arise. Current possible choices are 'sils', 'ma27', 'ma57', 'ma77', 'ma86', 'ma87', 'ma97', ssids, 'pardiso' and 'wsmp', although only 'sils' and, for OMP 4.0-compliant compilers, 'ssids' are installed by default. See the documentation for the GALAHAD package SLS for further details. The default is definite_linear_solver = 'sils'.

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 = "".

SLS_control is a scalar variable argument of type SLS_control_type that is used to pass control options to external packages used to factorize relevant symmetric matrices that arise. See the documentation for the GALAHAD package SLS for further details. In particular, default values are as for SLS, except that SLS_control%relative_pivot_tolerance is reset to pivot_tol.

2.4.4 The derived data type for holding timing information

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

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

preprocess is a scalar variable of type REAL(rp_), that gives the CPU time spent reordering the problem to standard form prior to solution.

analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing the required matrices prior to factorization.

factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type REAL(rp_), that gives the CPU time spent computing the search direction.

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

clock_preprocess is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent reordering the problem to standard form prior to solution.

clock_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing the required matrices prior to factorization.

clock_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent computing the search direction.

2.4.5 The derived data type for holding informational parameters

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

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Section 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. 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.

major_iter is a scalar variable of type INTEGER(ip_), that gives the total number of major iterations required.

iter is a scalar variable of type INTEGER(ip_), that gives the total number of iterations required.

cg_iter is a scalar variable of type INTEGER(ip_), that gives the total number of conjugate-gradient inner iterations required.

factorization_status is a scalar variable of type INTEGER(ip_), that gives the return status from the matrix factorization.

factorization_integer is a scalar variable of type long INTEGER(ip_), that gives the amount of integer storage used for the matrix factorization.

factorization_real is a scalar variable of type INTEGER(int64), that gives the amount of real storage used for the matrix factorization.

nfacts is a scalar variable of type INTEGER(ip_), that gives the total number of factorizations performed.

nmods is a scalar variable of type INTEGER(ip_), that gives the total number of factorizations which were modified to ensure that the matrix is an appropriate preconditioner.

num_g_infeas is a scalar variable of type INTEGER(ip_), that gives the total number of general constraints that are violated.

num_b_infeas is a scalar variable of type INTEGER(ip_), that gives the total number of simple bound constraints that are violated.

obj is a scalar variable of type REAL(rp_), that holds the value of the objective function at the best estimate of the solution found.

infeas_g is a scalar variable of type REAL(rp_), that holds the value of the infeasibility $v_g(\mathbf{x})$.

infeas_b is a scalar variable of type REAL(rp_), that holds the value of the infeasibility $v_b(\mathbf{x})$.

merit is a scalar variable of type REAL(rp_), that holds the value of the merit function $q(\mathbf{x}) + \rho_g v_g(\mathbf{x}) + \rho_b v_b(\mathbf{x})$ at the best estimate of the solution found.

time is a scalar variable of type QPA_time_type whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

SLS_inform is a scalar variable argument of type SLS_inform_type that is used to pass information concerning the progress of the external packages used to factorize relevant symmetric matrices that arise. See the documentation for the GALAHAD package SLS for further details.

2.4.6 The derived data type for holding problem data

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

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.7 for further features):

1. The subroutine QPA_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 QPA_solve is called to solve the problem.
3. The subroutine QPA_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by QPA_solve, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since QPA_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

data is a scalar INTENT(INOUT) argument of type QPA_data_type (see Section 2.4.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type QPA_control_type (see Section 2.4.3). On exit, control contains default values for the components as described in Section 2.4.3. These values should only be changed after calling QPA_initialize.

inform is a scalar INTENT(INOUT) argument of type QPA_inform_type (see Section 2.4.5). A successful call to QPA_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.6.

2.5.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

```
CALL QPA_solve( p, C_stat, B_stat, data, control, info )
```

p is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.4.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for all components except p%C. p%new_problem_structure must be set .TRUE., but will have been reset to .FALSE. on exit from QPA_solve. Users are free to choose whichever of the matrix formats described in Section 2.1 is appropriate for **A** and **H** for their application—different formats may be used for the two matrices.

For a problem with the same structure as one that has just been solved, the user may set p%new_problem_structure to .FALSE., so long as QPA_terminate has not been called in the interim. The INTEGER(ip_) components must be unaltered since the previous call to QPA_solve, but the REAL(rp_) may be altered to reflect the new problem.

The components p%X, p%Y and p%Z must be set to initial estimates, $\mathbf{x}^0$, of the primal variables, **x**, Lagrange multipliers for the general constraints, **y**, 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 p%X=0.0, p%Y=0.0 and p%Z=0.0. The component p%C need not be set on entry.

On exit, the components p%X, p%Y, p%Z and p%C will contain the best estimates of the primal variables **x**, Lagrange multipliers for the general constraints **y**, dual variables for the bound constraints **z**, and values of the constraints **Ax** respectively. The components p%rho_g and p%rho_b may have been altered if either of

control%solve_qp or control%solve_within_bounds have been set .TRUE. and will reflect the final values of ρ_g and ρ_b used. What of the remaining problem data has been restored depends upon the input value of the control parameter control%restore_problem. The return format for a restored array component will be the same as its input format. **Restrictions:** p%n > 0, p%m ≥ 0, p%A%ne ≥ -2 and p%H%ne ≥ -2.

C_stat is a rank-one INTENT(INOUT) array argument of dimension p%m and type INTEGER(ip_), 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%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 INTEGER(ip_), that indicates which of the simple bound constraints are in the current working set. Possible values for B_stat(j), j$= 1,\ldots,$ 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%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.

data is a scalar INTENT(INOUT) argument of type QPA_data_type (see Section 2.4.6). 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 QPA_initialize.

control is a scalar INTENT(IN) argument of type QPA_control_type (see Section 2.4.3). Default values may be assigned by calling QPA_initialize prior to the first call to QPA_solve.

inform is a scalar INTENT(OUT) argument of type QPA_inform_type (see Section 2.4.5). A successful call to QPA_solve is indicated when the component status has the value 0. For other return values of status, see Section 2.6.

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL QPA_terminate( data, control, info )
```

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

control is a scalar INTENT(IN) argument of type QPA_control_type exactly as for QPA_solve.

inform is a scalar INTENT(OUT) argument of type QPA_inform_type exactly as for QPA_solve. Only the component status will be set on exit, and a successful call to QPA_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.6.

2.6 Warning and error messages

A negative value of inform%status on exit from QPA_solve or QPA_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 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 appear to have no feasible point.

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

-9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-10. The factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-16. The problem is so ill-conditioned that further progress is impossible.

-18. Too many iterations have been performed. This may happen if control%maxit is too small, but may also be symptomatic of a badly scaled problem.

-19. The elapsed CPU or system clock time limit has been reached. This may happen if either control%cpu_time_limit or control%clock_time_limit is too small, but may also be symptomatic of a badly scaled problem.

-23. An entry from the strict upper triangle of **H** has been specified.

2.7 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type QPA_control_type (see Section 2.4.3), by reading an appropriate data specification file using the subroutine QPA_read_specfile. This facility is useful as it allows a user to change QPA control parameters without editing and recompiling programs that call QPA.

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 QPA_read_specfile must start with a "BEGIN QPA" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by QPA_read_specfile .. )
  BEGIN QPA
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by QPA_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN QPA" 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 QPA SPECIFICATION
```

and

```
END QPA SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN QPA" 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 QPA_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 QPA_read_specfile.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL QPA_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type QPA_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling QPA_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 2.4.3) of control that each affects are given in Table 2.1.

device is a scalar INTENT(IN) argument of type INTEGER(ip_), 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.

2.8 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 = 1, a single line of output will be produced for each iteration of the process. The overall algorithm, for a fixed value of the penalty parameter, is divided into major iterations, at which a factorization of the current reference matrix is performed, and minor iterations, at which the the working set that defines the reference

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
maximum-number-of-iterations	%maxit	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
factorization-used	%factor	integer
maximum-column-nonzeros-in-schur-complement	%max_col	integer
maximum-dimension-of-schur-complement	%max_sc	integer
maximum-refinements	%itref_max	integer
maximum-infeasible-iterations-before-rho-increase	%infeas_check_interval	integer
maximum-number-of-cg-iterations	%cg_maxit	integer
preconditioner-used	%precon	integer
semi-bandwidth-for-band-preconditioner	%nsemib	integer
full-max-fill-ratio	%full_max_fill	integer
deletion-strategy	%deletion_strategy	integer
restore-problem-on-output	%restore_problem	integer
residual-monitor-interval	%monitor_residuals	integer
cold-start-strategy	%cold_start	integer
infinity-value	%infinity	real
feasibility-tolerance	%feas_tol	real
minimum-objective-before-unbounded	%obj_unbounded	real
increase-rho-g-factor	%increase_rho_g_factor	real
increase-rho-b-factor	%increase_rho_b_factor	real
infeasible-g-required-improvement-factor	%infeas_g_improved_by_factor	real
infeasible-b-required-improvement-factor	%infeas_b_improved_by_factor	real
pivot-tolerance-used-for-dependencies	%pivot_tol_for_dependencies	real
multiplier-tolerance	%multiplier_tol	real
inner-iteration-relative-accuracy-required	%inner_stop_relative	real
inner-iteration-absolute-accuracy-required	%inner_stop_absolute	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
treat-zero-bounds-as-general	%treat_zero_bounds_as_general	logical
solve-qp	%solve_qp	logical
solve-within-bounds	%solve_within_bounds	logical
temporarily-perturb-constraint-bounds	%randomize	logical

Table 2.1: Specfile commands and associated components of control.

matrix is gradually modified. For the major iterations, details about the factorization performed are given. Each minor iteration results in a single line summary of the progress of the method. This includes the value of the merit function, the step size taken, the size and number of the infeasibility, the number of conjugate-gradient steps taken, the dimension of the Schur complement, along with the numbers of positive and negative eigenvalues, an brief indication of what happened during the iteration (the index of the general constraint (c) or simple bound (b) which joined (+) or left (-) the active set), and the and the elapsed CPU time in seconds.

If `control%print_level` ≥ 2 this output will be increased to provide significant detail of each iteration. This extra output includes detailed progress of the linesearch and the residuals of the linear systems solved, and, for larger values of `control%print_level`, values of the primal and dual variables and Lagrange multipliers.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `QPA_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NORMS`, `GALAHAD_SPACE`, `GALAHAD_RAND`, `GALAHAD_TOOLS`, `GALAHAD_ROOTS`, `GALAHAD_SORT`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPP`, `GALAHAD_QPD`, `GALAHAD_SLS`, `GALAHAD_SPECFILE` and `GALAHAD_SCU`.

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'` }.

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

4 METHOD

At the k-th iteration of the method, an improvement to the value of the merit function $m(\mathbf{x},\rho_g,\rho_b) = q(\mathbf{x}) + \rho_g v_g(\mathbf{x}) + \rho_b v_b(\mathbf{x})$ at $\mathbf{x} = \mathbf{x}^{(k)}$ is sought. This is achieved by first computing a search direction $\mathbf{s}^{(k)}$, and then setting $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{s}^{(k)}$, where the stepsize $\alpha^{(k)}$ is chosen as the first local minimizer of $\phi(\alpha) = m(\mathbf{x}^{(k)} + \alpha\mathbf{s}^{(k)},\rho_g,\rho_b)$ as α incesases from zero. The stepsize calculation is straightforward, and exploits the fact that $\phi(\alpha)$ is a piecewise quadratic function of α.

The search direction is defined by a subset of the "active" terms in $v(\mathbf{x})$, i.e., those for which $\mathbf{a}_i^T\mathbf{x} = c_i^l$ or c_i^u (for $i = 1,\ldots,m$) or $x_j = x_j^l$ or x_j^u (for $j = 1,\ldots,n$). The "working" set $W^{(k)}$ is chosen from the active terms, and is such that its members have linearly independent gradients. The search direction $\mathbf{s}^{(k)}$ is chosen as an approximate solution of the equality-constrained quadratic program

$$\underset{\mathbf{s}\in\mathbb{R}^n}{\text{minimize}} \quad q(\mathbf{x}^{(k)}+\mathbf{s}) + \rho_g l_g^{(k)}(\mathbf{s}) + \rho_b l_b^{(k)}(\mathbf{s}), \tag{4.1}$$

subject to

$$\mathbf{a}_i^T\mathbf{s} = 0, \ i \in \{1,\ldots,m\}\cap W^{(k)}, \ \text{and} \ x_j = 0, \ i \in \{1,\ldots,n\}\cap W^{(k)}, \tag{4.2}$$

where

$$l_g^{(k)}(\mathbf{s}) = -\sum_{\substack{i=1\\ \mathbf{a}_i^T\mathbf{x}<c_i^l}}^{m} \mathbf{a}_i^T\mathbf{s} + \sum_{\substack{i=1\\ \mathbf{a}_i^T\mathbf{x}>c_i^u}}^{m} \mathbf{a}_i^T\mathbf{s}$$

and

$$l_b^{(k)}(\mathbf{s}) = -\sum_{\substack{j=1 \\ x_j<x_j^l}}^{n} s_j + \sum_{\substack{j=1 \\ x_j>x_j^u}}^{n} s_j.$$

The equality-constrained quadratic program (4.1)–(4.2) is solved by a projected preconditioned conjugate gradient method. The method terminates either after a prespecified number of iterations, or if the solution is found, or if a direction of infinite descent, along which $q(\mathbf{x}^{(k)}+\mathbf{s})+\rho_g l_g^{(k)}(\mathbf{s})+\rho_b l_b^{(k)}(\mathbf{s})$ decreases without bound within the feasible region (4.2), is located. Succesively more accurate approximations are required as suspected solutions of (1.1) are approached.

Preconditioning of the conjugate gradient iteration requires the solution of one or more linear systems of the form

$$\begin{pmatrix} \mathbf{M}^{(k)} & \mathbf{A}^{(k)T} \\ \mathbf{A}^{(k)} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{p} \\ \mathbf{u} \end{pmatrix} = \begin{pmatrix} \mathbf{g} \\ 0 \end{pmatrix}, \tag{4.3}$$

where $\mathbf{M}^{(k)}$ is a "suitable" approximation to $\mathbf{H}$ and the rows of $\mathbf{A}^{(k)}$ comprise the gradients of the terms in the current working set. Rather than recomputing a factorization of the preconditioner at every iteration, a Schur complement method is used, recognising the fact that gradual changes occur to successive working sets. The main iteration is divided into a sequence of "major" iterations. At the start of each major iteration (say, the overall iteration l), a factorization of the current "reference" matrix, that is the matrix

$$\begin{pmatrix} \mathbf{M}^{(l)} & \mathbf{A}^{(l)T} \\ \mathbf{A}^{(l)} & 0 \end{pmatrix} \tag{4.4}$$

is obtained using the GALAHAD matrix factorization package `GALAHAD_SLS`. This reference matrix may be factorized as a whole (the so-called "augmented system" approach), or by performing a block elimination first (the "Schur-complement" approach). The latter is usually to be preferred when a (non-singular) diagonal preconditioner is used, but may be inefficient if any of the columns of $\mathbf{A}^{(l)}$ is too dense. Subsequent iterations within the current major iteration obtain solutions to (4.3) via the factors of (4.4) and an appropriate (dense) Schur complement, obtained from the GALAHAD package `GALAHAD_SCU`. The major iteration terminates once the space required to hold the factors of the (growing) Schur complement exceeds a given threshold.

The working set changes by (a) adding an active term encountered during the determination of the stepsize, or (b) the removal of a term if $\mathbf{s}=0$ solves (4.1)–(4.2). The decision on which to remove in the latter case is based upon the expected decrease upon the removal of an individual term, and this information is available from the magnitude and sign of the components of the auxiliary vector $\mathbf{u}$ computed in (4.3). At optimality, the components of $\mathbf{u}$ for $\mathbf{a}_i$ terms will all lie between 0 and ρ_g—and those for the other terms between 0 and ρ_b—and any violation of this rule indicates further progress is possible. The components of $\mathbf{u}$ corresonding to the terms involving $\mathbf{a}_i^T\mathbf{x}$ are sometimes known as Lagrange multipliers (or generalized gradients) and denoted by $\mathbf{y}$, while those for the remaining x_j terms are dual variables and denoted by $\mathbf{z}$.

To solve (1.2)–(1.4), a sequence of problems of the form (1.1) are solved, each with a larger value of ρ_g and/or ρ_b than its predecessor. The required solution has been found once the infeasibilities $v_g(\mathbf{x})$ and $v_b(\mathbf{x})$ have been reduced to zero at the solution of (1.1) for the given ρ_g and ρ_b.

The required solution $\mathbf{x}$ to (1.2)–(1.4) necessarily satisfies the primal optimality conditions

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

and

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

the dual optimality conditions

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

and

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

and the complementary slackness conditions

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

where, as before, 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.

In order to make the solution as efficient as possible, the variables and constraints are reordered internally by the GALAHAD package `GALAHAD_QPP` prior to solution. In particular, fixed variables and free (unbounded on both sides) constraints are temporarily removed.

References:

The method is described in detail in

N. I. M. Gould and Ph. L. Toint (2001). "An iterative working-set method for large-scale non-convex quadratic programming". *Applied Numerical Mathematics* **43 (1–2)** (2002) 109–128.

5 EXAMPLE OF USE

Suppose we wish to minimize $\frac{1}{2}x_1^2 + x_2^2 + \frac{3}{2}x_3^2 + 4x_1x_3 + 2x_2 + 1$ 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

$$\mathbf{H} = \begin{pmatrix} 1 & & 4 \\ & 2 & \\ 4 & & 3 \end{pmatrix}, \ \mathbf{g} = \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}, \ \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ -\infty \end{pmatrix} \text{ and } \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 2 \end{pmatrix},$$

and

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

in sparse co-ordinate format, we may use the following code:

```
! THIS VERSION: GALAHAD 2.2 - 23/04/2008 AT 16:30 GMT.
   PROGRAM GALAHAD_QPA_EXAMPLE
   USE GALAHAD_QPA_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 ( QPA_data_type ) :: data
   TYPE ( QPA_control_type ) :: control
   TYPE ( QPA_inform_type ) :: inform
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
   INTEGER :: s
   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%rho_g = 1.0_wp ; p%rho_b = 1.0_wp        ! initial penalty parameters
   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, 3.0_wp, 4.0_wp /) ! Hessian H
   p%H%row = (/ 1, 2, 3, 3 /)                   ! NB lower triangle
   p%H%col = (/ 1, 2, 3, 1 /) ; 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 QPA_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = infinity                  ! Set infinity
   control%solve_qp = .TRUE.
!  control%print_level = 1
!  control%SLS_control%print_level = 1
   CALL QPA_solve( p, C_stat, B_stat, data, control, inform )  ! Solve problem
   IF ( inform%status == 0 ) THEN               !  Successful return
     WRITE( 6, "( ' QPA: ', I0, ' iterations. Optimal objective value =',      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, p%X
   ELSE                                         !  Error returns
     WRITE( 6, "( ' QPA_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL QPA_terminate( data, control, inform )  !  delete internal workspace
   END PROGRAM GALAHAD_QPA_EXAMPLE
```

This produces the following output:

```
 QPA: 14 iterations. Optimal objective value =  5.4459E+00
 Optimal solution =  -5.4054E-02  1.1081E+00  8.9189E-01
```

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', s )  ! Specify sparse-by-row
   CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS', s )  ! 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, 3.0_wp, 4.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 3, 1 /)                   ! NB lower triangular
   p%H%ptr = (/ 1, 2, 3, 5 /)                   ! Set row pointers
   p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
```

```
   p%A%col = (/ 1, 2, 2, 3 /)
   p%A%ptr = (/ 1, 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', s )  ! Specify dense
   CALL SMT_put( p%A%type, 'DENSE', s )  ! 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, 4.0_wp, 0.0_wp, 3.0_wp /) ! Hessian
   p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian
! problem data complete
```

respectively.

If instead **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 **H**, and in this case we would instead

```
   CALL SMT_put( prob%H%type, 'DIAGONAL', s )  ! 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.