


GALAHAD DQP

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package uses a dual gradient-projection method to solve the **strictly-convex quadratic-programming problem**

$$\text{minimize } q(\mathbf{x}) \stackrel{\text{def}}{=} \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{g}^T\mathbf{x} + f \tag{1.1}$$

or the **shifted least-distance problem**

$$\text{minimize } s(\mathbf{x}) \stackrel{\text{def}}{=} \tfrac{1}{2}\sum_{j=1}^{n} w_j^2(x_j - x_j^0)^2 + \mathbf{g}^T\mathbf{x} + f \tag{1.2}$$

subject to the general linear constraints

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

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1,\ldots,n, \tag{1.3}$$

where the n by n symmetric, positive-definite matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{w}$, $\mathbf{x}^0$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the scalar f are given. 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 $\mathbf{a}_i$.

The package may also be used to minimize the penalty functions

$$q(\mathbf{x}) + \rho\sum_{i=1}^{m}\max\left(c_i^l - \mathbf{a}_i^T\mathbf{x}, \mathbf{a}_i^T\mathbf{x} - c_i^u, 0\right) \text{ or } s(\mathbf{x}) + \rho\sum_{i=1}^{m}\max\left(c_i^l - \mathbf{a}_i^T\mathbf{x}, \mathbf{a}_i^T\mathbf{x} - c_i^u, 0\right)$$

subject to the simple bound constraints (1.3).

ATTRIBUTES — Versions: `GALAHAD_DQP_single`, `GALAHAD_DQP_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_STRING`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPP`, `GALAHAD_QPD`, `GALAHAD_SORT`, `GALAHAD_FDC`, `GALAHAD_SLS`, `GALAHAD_SBLS`, `GALAHAD_SCU`, `GALAHAD_GLTR`, `GALAHAD_NORMS`. **Date:** August 2012. **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

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_DQP_single
```

with the obvious substitution `GALAHAD_DQP_double`, `GALAHAD_DQP_single_64` and `GALAHAD_DQP_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`, `NLPT_userdata_type`, `DQP_time_type`, `DQP_control_type`, `DQP_inform_type` and `DQP_data_type` (Section 2.4) and the subroutines `DQP_initialize`, `DQP_solve`, `DQP_terminate`, (Section 2.5) and `DQP_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`, 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 $\mathbf{A}$.

2.1.5 Scaled-identity-matrix storage format

If $\mathbf{H}$ is a scalar multiple of the identity matrix (i.e., $h_{ii} = h_{11}$ and $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the first diagonal entry h_{11} needs be stored, and the first component of the array `H%val` may be used for the purpose. Again, there is no sensible equivalent for the non-square $\mathbf{A}$.

2.1.6 Identity-matrix storage format

If $\mathbf{H}$ is the identity matrix (i.e., $h_{ii} = 1$ and $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$), no explicit entries needs be stored.

2.1.7 Zero-matrix storage format

If $\mathbf{0} = 0$ (i.e., $h_{ij} = 0$ for all $1 \leq i, j \leq n$), no explicit entries needs be stored.

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_DQP` 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

Ten 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 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:

n is a scalar variable of type INTEGER(ip_), that holds the number of optimization variables, n.

m is a scalar variable of type INTEGER(ip_), that holds the number of general linear constraints, m.

Hessian_kind is a scalar variable of type INTEGER(ip_), that is used to indicate what type of Hessian the problem involves. Possible values for Hessian_kind are:

- <0 In this case, a general quadratic program of the form (1.1) is given. The Hessian matrix **H** will be provided in the component H (see below).
- 0 In this case, a linear program, that is a problem of the form (1.2) with weights $\mathbf{w} = 0$, is given.
- 1 In this case, a least-distance problem of the form (1.2) with weights $w_j = 1$ for $j = 1, \ldots, n$ is given.
- >1 In this case, a weighted least-distance problem of the form (1.2) with general weights **w** is given. The weights will be provided in the component WEIGHT (see below).

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H** whenever Hessian_kind < 0. 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 the scaled-identity matrix storage scheme (see Section 2.1.5), the first fifteen components of H%type must contain the string SCALED_IDENTITY, for the identity matrix storage scheme (see Section 2.1.6), the first eight components of H%type must contain the string IDENTITY, and for the zero matrix storage scheme (see Section 2.1.7), the first four components of H%type must contain the string ZERO.

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 DQP_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 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 non-trivial storage schemes mentioned in Sections 2.1.2–2.1.4. For the scaled-identity scheme (see Section 2.1.5), the first component, H%val(1), holds the scale factor h_{11}. It need not be allocated for any of the remaining schemes.

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 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 any of the other 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.

If Hessian_kind ≥ 0, the components of H need not be set.

WEIGHT is a rank-one allocatable array type REAL(rp_), that should be allocated to have length n, and its j-th component filled with the value w_j for $j = 1,\ldots,n$, whenever Hessian_kind > 1. If Hessian_kind ≤ 1, WEIGHT need not be allocated.

target_kind is a scalar variable of type INTEGER(ip_), that is used to indicate whether the components of the targets $\mathbf{x}^0$ (if they are used) have special or general values. Possible values for target_kind are:

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

1 In this case, $x^0_j = 1$ for $j = 1,\ldots,n$.

$\neq$ 0,1 In this case, general values of $\mathbf{x}^0$ will be used, and will be provided in the component X0 (see below).

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

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

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

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

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

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

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

A is scalar variable of type SMT_TYPE that holds the Jacobian matrix **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 DQP_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 **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for either of the other two appropriate schemes.

A%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix **A** in any of the appropriate 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 **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two appropriate schemes.

A%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **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 **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 appropriate 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.

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 § 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(rp_), that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the simple bound constraints (see § 4). The j-th component of Z, $j = 1,\ldots,n$, contains z_j.

2.4.3 The derived data type for holding control parameters

The derived data type DQP_control_type is used to hold controlling data. Default values may be obtained by calling DQP_initialize (see Section 2.5.1), while components may also be changed by calling DQP_read_specfile (see Section 2.7.1). The components of DQP_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 DQP_solve and DQP_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 DQP_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 $\leq$ 0. If print_level $=$ 1, a single line of output will be produced for each iteration of the process. If print_level $\geq$ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

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

print_gap is a scalar variable of type INTEGER(ip_). Once printing has been started, output will occur once every print_gap iterations. If print_gap is no larger than 1, printing will be permitted on every iteration. The default is print_gap = 1.

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

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 when there is no Fredholm Alternative. The default is max_sc = 100.

cauchy_only is a scalar variable of type INTEGER(ip_), that specifies the maximum number of changes in the active set that may occur in the first-phase projected-dual-gradient arc search during an iteration before attempting a second phase unconstrained minimization in the space of free dual variables § 4). If cauchy_only is negative, the second phase will always be tried. The default is cauchy_only = -1.

arc_search_maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of steps that may be performed by the arc-search every iteration. If arc_search_maxit is set to a negative number, as many steps as are necessary will be performed. The default is arc_search_maxit = -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 DQP_solve. If cg_maxit is set to a negative number, it will be reset by DQP_solve to the dimension of the relevant linear system $+1$. The default is cg_maxit = 1000.

dual_starting_point is a scalar variable of type INTEGER(ip_), that specifies how the algorithm computes its staring point. Possible values are:

- 0 the values **y** and **z** provided by the user in components Y and Z of the derived data type QPT_problem_type will be used.
- 1 values obtained by minimizing a linearized version of the dual will be used.
- 2 values obtained by minimizing a simplified quadratic version of the dual will be used.
- 3 values will be chosen so that all dual variables lie away from their bounds if possible. This corresponds to trying to start from a point in which all primal constraints are active.
- 4 values will be chosen so that all dual variables lie on their bounds if possible. This corresponds to trying to start from a point in which all primal constraints are inactive.

Any other value will be interpreted as dual_starting_point = 0, and this is the default.

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

0 nothing is restored.

1 the vector data $\mathbf{w}$, $\mathbf{g}$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, and $\mathbf{x}^u$ will be restored to their input values.

2 the entire problem, that is the above vector data along with the Jacobian matrix $\mathbf{A}$, will be restored.

The default is restore_problem = 2.

rho is a scalar variable of type REAL(rp_), that holds the penalty weight, ρ. If rho > 0, the general linear constraints are not enforced explicitly, but instead included in the objective as a penalty term weighted by ρ. If rho ≤ 0, the general linear constraints are explicit (that is, there is no penalty term in the objective function) The default is rho = 0.0.

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}.

stop_abs_p and stop_rel_p are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the primal infeasibility (see Section 4). The absolute value of each component of the primal infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_p = stop_rel_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_double).

stop_abs_d and stop_rel_d are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the dual infeasibility (see Section 4). The absolute value of each component of the dual infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_d = stop_rel_d = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_double).

stop_abs_c and stop_rel_c are scalar variables of type REAL(rp_), that hold the required absolute and relative accuracy for the violation of complementary slackness (see Section 4). The absolute value of each component of the complementary slackness on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_c = stop_rel_c = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_double).

stop_cg_relative and stop_cg_absolute are scalar variables of type REAL(rp_), that hold the relative and absolute convergence tolerances for the conjugate-gradient iteration that occurs in the face of currently-active constraints that may be used to construct the search direction. _stop_cg_relative = 0.01 and stop_cg_absolute = $\sqrt{u}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_double).

cg_zero_curvature is a scalar variable of type REAL(rp_) that specifies the threshold below which any curvature encountered by the conjugate-gradient iteration is regarded as zero. The default is cg_zero_curvature = $10u$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_double).

identical_bounds_tol is a scalar variable of type REAL(rp_). Each pair of variable bounds (x_j^l, x_j^u) that is closer than identical_bounds_tol will be reset to the average of their values, $\frac{1}{2}(x_j^l + x_j^u)$. The default is identical_bounds_tol = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_DQP_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.

remove_dependencies is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm is to attempt to remove any linearly dependent constraints before solving the problem, and .FALSE. otherwise. We recommend removing linearly dependencies. The default is remove_dependencies = .TRUE..

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_DQP 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..

exact_arc_search is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to perform an exact arc search and .FALSE. if an inexact search suffices. Usually the exact search is beneficial, but occasionally it may be more expensive. The default is exact_arc_search = .TRUE..

subspace_direct is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to compute subspace steps using matrix factorization, and .FALSE. if conjugate-gradient steps are preferred. Factorization often produces a better step, but sometimes the conjugate-gradient method may be less expensive and less demanding on storage. The default is subspace_direct = .FALSE., but if more advanced symmetric linear solvers such as MA57 or MA97 available, we recommend setting subspace_direct = .TRUE. and changing symmetric_linear_solver (see below) appropriately.

subspace_arc_search is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to perform an arc search following the subspace step and .FALSE. if a step to the nearest inactive bound suffices. As before, the exact search is usually beneficial, but it is more expensive. The default is subspace_arc_search = .TRUE..

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..

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'. See the documentation for the GALAHAD package SLS for further details. Since 'sils' does not currently provide the required Fredholm Alternative option, the default is symmetric_linear_solver = 'ma57', but we recommend 'ma97' if it is available.

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'. See the documentation for the GALAHAD package SLS for further details. The default is definite_linear_solver = 'sils', but we recommend 'ma87' if it available.

unsymmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any unsymmetric linear systems that might arise. Possible choices are 'gls', 'ma28' and 'ma48', although only 'gls' is installed by default. See the documentation for the GALAHAD

package ULS for further details. The default is unsymmetric_linear_solver = 'gls', but we recommend 'ma48' if it available.

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 = "".

FDC_control is a scalar variable of type FDC_control_type whose components are used to control any detection of linear dependencies performed by the package GALAHAD_FDC. See the specification sheet for the package GALAHAD_FDC for details, and appropriate default values.

SLS_control is a scalar variable of type SLS_control_type whose components are used to control factorizations performed by the package GALAHAD_SLS. See the specification sheet for the package GALAHAD_SLS for details, and appropriate default values.

SBLS_control is a scalar variable of type SBLS_control_type whose components are used to control factorizations performed by the package GALAHAD_SBLS. See the specification sheet for the package GALAHAD_SBLS for details, and appropriate default values.

GLTR_control is a scalar variable of type GLTR_control_type whose components are used to control conjugate-gradient solves performed by the package GALAHAD_GLTR. See the specification sheet for the package GALAHAD_GLTR for details, and appropriate default values.

2.4.4 The derived data type for holding timing information

The derived data type DQP_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of DQP_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 preprocess the problem prior to solution.

find_dependent is a scalar variable of type REAL(rp_), that gives the CPU time spent detecting and removing dependent constraints 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.

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

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 preprocess the problem prior to solution.

clock_find_dependent is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent detecting and removing dependent constraints 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.

clock_search is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent in the arc search.

2.4.5 The derived data type for holding informational parameters

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

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

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.

threads is a scalar variable of type INTEGER(ip_), that gives the total number of threads used for parallel execution.

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.

primal_infeasibility is a scalar variable of type REAL(rp_), that holds the ℓ_∞-norm of the violation of primal optimality (see Section 2.4.4) at the best estimate of the solution found.

dual_infeasibility is a scalar variable of type REAL(rp_), that holds the ℓ_∞-norm of the violation of dual optimality (see Section 2.4.4) at the best estimate of the solution found.

complementary_slackness is a scalar variable of type REAL(rp_), that holds the absolute value of the violation of complementary slackness (see Section 2.4.4) at the best estimate of the solution found.

non_negligible_pivot is a scalar variable of type REAL(rp_), that holds the value of the smallest pivot that was not judged to be zero when searching for dependent linear constraints.

feasible is a scalar variable of type default LOGICAL, that has the value .TRUE. if the output value of **x** satisfies the constraints, and the value .FALSE. otherwise.

time is a scalar variable of type DQP_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).

FDC_inform is a scalar variable of type FDC_inform_type whose components are used to provide information about any detection of linear dependencies performed by the package GALAHAD_FDC. See the specification sheet for the package GALAHAD_FDC for details.

SLS_inform is a scalar variable of type SLS_inform_type whose components are used to provide information about factorizations performed by the package GALAHAD_SLS. See the specification sheet for the package GALAHAD_SLS for details.

SBLS_inform is a scalar variable of type SBLS_inform_type whose components are used to provide information about factorizations performed by the package GALAHAD_SBLS. See the specification sheet for the package GALAHAD_SBLS for details.

GLTR_inform is a scalar variable of type GLTR_info_type whose components are used to provide information about the step calculation performed by the package GALAHAD_GLTR. See the specification sheet for the package GALAHAD_GLTR for details.

2.4.6 The derived data type for holding problem data

The derived data type DQP_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of DQP procedures. This data should be preserved, untouched, from the initial call to DQP_initialize to the final call to DQP_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 DQP_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 DQP_solve is called to solve the problem.
3. The subroutine DQP_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by DQP_solve, at the end of the solution process.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

data is a scalar INTENT(INOUT) argument of type DQP_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 DQP_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 DQP_initialize.

inform is a scalar INTENT(OUT) argument of type DQP_inform_type (see Section 2.4.5). A successful call to DQP_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 DQP_solve( prob, data, control, inform[, C_stat, X_stat] )
```

`prob` 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. The user must allocate all the array components, and set values for all components except `prob%C`.

The components `prob%X` , `prob%Y` and `prob%Z` must be set to initial estimates of the primal variables, **x**, Lagrange multipliers, **y**, for the general constraints 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`, `prob%Y=0.0` and `prob%Z=0.0`.

On exit, the components `prob%X` , `prob%C`, `prob%Y`, and `prob%Z` will contain the best estimates of the primal variables **x**, the linear constraints **Ax**, Lagrange multipliers, **y**, for the general constraints and dual variables for the bound constraints **z**, respectively. **Restrictions:** `prob%n` > 0, `prob%m` ≥ 0, `prob%A_type` $\in$ { `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` }, 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 `DQP_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 `DQP_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `DQP_control_type` (see Section 2.4.3). Default values may be assigned by calling `DQP_initialize` prior to the first call to `DQP_solve`.

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

`C_stat` is an `OPTIONAL` rank-one `INTENT(OUT)` array argument of dimension `p%m` and type `INTEGER(ip_)`, that indicates which of the general linear constraints are in the optimal active 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 active set, on its lower bound,
- `>0` the i-th general constraint is in the active set, on its upper bound, and
- `0` the i-th general constraint is not in the active set.

`X_stat` is an `OPTIONAL` rank-one `INTENT(OUT)` array argument of dimension `p%n` and type `INTEGER(ip_)`, that indicates which of the simple bound constraints are in the current active set. Possible values for `X_stat(j)`, `j`$= 1, \ldots,$ `p%n`, and their meanings are

- `<0` the j-th simple bound constraint is in the active set, on its lower bound,
- `>0` the j-th simple bound constraint is in the active set, on its upper bound, and
- `0` the j-th simple bound constraint is not in the active set.

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type DQP_control_type exactly as for DQP_solve.

inform is a scalar INTENT(OUT) argument of type DQP_inform_type exactly as for DQP_solve. Only the component status will be set on exit, and a successful call to DQP_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 DQP_solve or DQP_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 , prob%m ≥ 0 the requirement that prob%A_type contain its relevant string 'DENSE', 'COORDINATE' or 'SPARSE_BY_ROWS', 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.

-5. The constraints appear to have no feasible point.

-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.

-17. The step is too small to make further impact.

-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.

-20. The matrix **H** does not appear to be positive definite.

-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 DQP_control_type (see Section 2.4.3), by reading an appropriate data specification file using the subroutine DQP_read_specfile. This facility is useful as it allows a user to change DQP control parameters without editing and recompiling programs that call DQP.

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

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

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

and

```
END DQP SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN DQP" 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 DQP_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 DQP_read_specfile.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL DQP_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type DQP_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling DQP_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.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
maximum-dimension-of-schur-complement	%max_schur_complement	integer
cauchy-only-until-change-level	%cauchy_only	integer
maximum-number-of-steps-per-arc-search	%arc_search_maxit	integer
maximum-number-of-cg-iterations-per-iteration	%cg_maxit	integer
dual-starting-point	%dual_starting_point	integer
restore-problem-on-output	%restore_problem	integer
penalty-weight	%rho	real
infinity-value	%infinity	real
identical-bounds-tolerance	%identical_bounds_tol	real
absolute-primal-accuracy	%stop_abs_p	real
relative-primal-accuracy	%stop_rel_p	real
absolute-dual-accuracy	%stop_abs_d	real
relative-dual-accuracy	%stop_rel_d	real
absolute-complementary-slackness-accuracy	%stop_abs_c	real
relative-complementary-slackness-accuracy	%stop_rel_c	real
cg-relative-accuracy-required	%stop_cg_relative	real
cg-absolute-accuracy-required	%stop_cg_absolute	real
zero-curvature-threshold	%cg_zero_curvature	real
identical-bounds-tolerance	%identical_bounds_tol	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
remove-linear-dependencies	%remove_dependencies	logical
treat-zero-bounds-as-general	%treat_zero_bounds_as_general	logical
perform-exact-arc-search	%exact_arc_search	logical
direct-solution-of-subspace-problem	%subspace_direct	logical
perform-subspace-arc-search	%subspace_arc_search	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
symmetric-linear-equation-solver	%symmetric_linear_solver	character
definite-linear-equation-solver	%definite_linear_solver	character
unsymmetric-linear-equation-solver	%unsymmetric_linear_solver	character
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

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. This will include values of the current dual objective value (this should converge to minus the optimal primal objective value), the primal infeasibility, the numbers of currently active constraints and the numbers that have changed during the iteration for both the arc search and the subspace step 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 a record of where in the iteration the algorithm is, 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: DQP_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_STRING, GALAHAD_SPACE, GALAHAD_SPECFILE, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_QPP, GALAHAD_QPD, GALAHAD_SORT, GALAHAD_FDC, GALAHAD_SLS, GALAHAD_SBLS, GALAHAD_SCU, GALAHAD_GLRT and GALAHAD_NORMS.

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 $\mathbf{x}$ 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

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

where

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

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

Dual gradient-projection methods solve (1.1) by instead solving the dual quadratic program

$$\begin{array}{lll} \text{minimize} & q^D(\mathbf{y}^l, \mathbf{y}^u, \mathbf{z}^l, \mathbf{z}^u) = & \frac{1}{2}[(\mathbf{y}^l + \mathbf{y}^u)^T A + (\mathbf{z}^l + \mathbf{z}^u]^T)H^{-1}[A^T(\mathbf{y}^l + \mathbf{y}^u) + \mathbf{z}^l + \mathbf{z}^u] \\ & & -[(\mathbf{y}^l + \mathbf{y}^u)^T A + (\mathbf{z}^l + \mathbf{z}^u]^T)H^{-1}g - (\mathbf{c}^{lT}\mathbf{y}^l + \mathbf{c}^{uT}\mathbf{y}^u + \mathbf{x}^{lT}\mathbf{z}^l + \mathbf{x}^{uT}\mathbf{z}^u) \\ \text{subject to} & & (\mathbf{y}^l, \mathbf{z}^l) \geq 0 \ \text{and} \ (\mathbf{y}^u, \mathbf{z}^u) \leq 0, \end{array} \tag{4.6}$$

and then recovering the required solution from the linear system

$$\mathbf{H}\mathbf{x} = -\mathbf{g} + A^T(\mathbf{y}^l + \mathbf{y}^u) + \mathbf{z}^l + \mathbf{z}^u.$$

The dual problem (4.6) is solved by an accelerated gradient-projection method comprising of alternating phases in which (i) the current projected dual gradient is traced downhill (the 'arc search') as far as possible and (ii) the dual variables that are currently on their bounds are temporarily fixed and the unconstrained minimizer of $q^D(\mathbf{y}^l, \mathbf{y}^u, \mathbf{z}^l, \mathbf{z}^u)$ with respect to the remaining variables is sought; the minimizer in the second phase may itself need to be projected back into the dual feasible region (either using a brute-force backtrack or a second arc search).

Both phases require the solution of sparse systems of symmetric linear equations, and these are handled by the GALAHAD matrix factorization package GALAHAD_SBLS or the conjugate-gradient package GALAHAD_GLTR. The systems are commonly singular, and this leads to a requirement to find the Fredholm Alternative for the given matrix and its right-hand side. In the non-singular case, there is an option to update existing factorizations using the "Schur-complement" approach given by the package GALAHAD_SCU.

Optionally, the problem may be pre-processed temporarily to eliminate dependent constraints using the package GALAHAD_FDC. This may improve the performance of the subsequent iteration.

References:

The basic algorithm is described in

N. I. M. Gould and D. P. Robinson, "A dual gradient-projection method for large-scale strictly-convex quadratic problems", Computational Optimization and Applications **67(1)** (2017) 1-38.

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

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

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

```
! THIS VERSION: GALAHAD 2.5 - 01/08/2012 AT 08:00 GMT.
   PROGRAM GALAHAD_DQP_EXAMPLE
   USE GALAHAD_DQP_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 ( DQP_data_type ) :: data
   TYPE ( DQP_control_type ) :: control
   TYPE ( DQP_inform_type ) :: inform
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: C_stat, X_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( X_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, 3, 3 /)                    ! NB lower triangle
   p%H%col = (/ 1, 2, 2, 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 DQP_initialize( data, control, inform ) ! Initialize control parameters
!  control%print_level = 1
   control%infinity = infinity                  ! Set infinity
   CALL DQP_solve( p, data, control, inform, C_stat, X_stat ) ! Solve
   IF ( inform%status == 0 ) THEN               !  Successful return
     WRITE( 6, "( ' DQP: ', I0, ' iterations  ', /,                          &
    &     ' Optimal objective value =', ES12.4, /,                           &
    &     ' Optimal solution = ', ( 5ES12.4 ) )" ) inform%iter, inform%obj, p%X
   ELSE                                         !  Error returns
     WRITE( 6, "( ' DQP_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL DQP_terminate( data, control, inform )  !  delete internal workspace
   END PROGRAM GALAHAD_DQP_EXAMPLE
```

This produces the following output:

```
 DQP: 20 iterations
 Optimal objective value =  6.3461E+00
 Optimal solution =   1.5385E-01  6.9230E-01  1.3077E+00
```

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

```
!  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 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, 1.0_wp, 3.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 2, 3 /)                    ! NB lower triangle
   p%H%ptr = (/ 1, 2, 3, 5 /)
   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 /)
! 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, 0.0_wp, 1.0_wp, 3.0_wp /) ! Hessian H
   p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
! 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', s )  ! Specify dense storage for H
   ALLOCATE( p%H%val( n ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp /) ! Hessian values
```