GALAHAD EQP

PACKAGE SPECIFICATION GALAHAD Optimization Library version 2.2

1 SUMMARY

This package uses an iterative method to solve the **equality-constrained quadratic programming problem**

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

subject to the linear constraints

$$\mathbf{A}\mathbf{x} + \mathbf{c} = \mathbf{0},$$

where the n by n symmetric matrix $\mathbf{H}$, the m by n matrix $\mathbf{A}$, the vectors $\mathbf{g}$ and $\mathbf{c}$, and the scalar f are given. Full advantage is taken of any zero coefficients in the matrices $\mathbf{H}$ and $\mathbf{A}$.

ATTRIBUTES — Versions: GALAHAD_EQP_single, GALAHAD_EQP_double. **Uses:** GALAHAD_CPU_time, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_QPT, GALAHAD_SBLS, GALAHAD_GLTR, GALAHAD_SPECFILE. **Date:** March 2006. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

Access to the package requires a USE statement such as

Single precision version

```
USE GALAHAD_EQP_single
```

Double precision version

```
USE GALAHAD_EQP_double
```

If it is required to use both modules at the same time, the derived types QPT_problem_type, EQP_time_type, EQP_control_type, EQP_inform_type and EQP_data_type (Section 2.2) and the subroutines EQP_initialize, EQP_solve, EQP_terminate, (Section 2.3) and EQP_read_specfile (Section 2.5) must be renamed on one of the USE statements.

2.1 Matrix storage formats

Both the Hessian matrix $\mathbf{H}$ and the constraint Jacobian $\mathbf{A}$ 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 $1 \le j \le i \le n$) 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 $1 \le j \le i \le n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **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 **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 **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 **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 **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 **A**.

2.2 The derived data types

Six derived data types are accessible from the package.

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

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

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

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.2.2).

`val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_EQP_double`) and dimension at least `ne`, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix **H** is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

`row` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.2).

`col` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.1.2–2.1.3).

`ptr` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `m + 1`, that may holds the pointers to the first entry in each row (see §2.1.3).

2.2.2 The derived data type for holding the problem

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

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

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

`H` is scalar variable of type `SMT_TYPE` that holds the Hessian matrix $\mathbf{H}$. 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 `EQP_problem_type` and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( prob%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`H%ne` is a scalar variable of type default `INTEGER`, that holds the number of entries in the **lower triangular** part of $\mathbf{H}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.

`H%val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_EQP_double`), that holds the values of the entries of the **lower triangular** part of the Hessian matrix $\mathbf{H}$ in any of the storage schemes discussed in Section 2.1.

`H%row` is a rank-one allocatable array of type default `INTEGER`, that holds the row indices of the **lower triangular** part of $\mathbf{H}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

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

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

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

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

`A` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix $\mathbf{A}$. 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 EQP_problem_type and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

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

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

A%val is a rank-one allocatable array of type default REAL (double precision in GALAHAD_EQP_double), that holds the values of the entries of the Jacobian matrix **A** in any of the storage schemes discussed in Section 2.1.

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

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

A%ptr is a rank-one allocatable array of dimension m+1 and type default INTEGER, that holds the starting position of each row of **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 is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_EQP_double), that holds the values of the vector**c** of constant terms for the constraints. The i-th component of C, $i = 1,\ldots,m$, contains c_i.

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

Y is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_EQP_double), that holds the values **y** of estimates of the Lagrange multipliers corresponding to the linear constraints (see Section 4). The i-th component of Y, $i = 1,\ldots,m$, contains y_i.

2.2.3 The derived data type for holding control parameters

The derived data type EQP_control_type is used to hold controlling data. Default values may be obtained by calling EQP_initialize (see Section 2.3.1), while components may also be changed by calling GALAHAD_EQP_read_spec (see Section 2.5.1). The components of EQP_control_type are:

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

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

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

`preconditioner` is a scalar variable of type default `INTEGER`, that specifies which preconditioner to be used. When finding the required minimizer, the system matrix

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

will be replaced by a so-called preconditioner of the form

$$\mathbf{K}_G = \begin{pmatrix} \mathbf{G} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{0} \end{pmatrix} \equiv \mathbf{P} \begin{pmatrix} \mathbf{G}_{11} & \mathbf{G}_{21}^T & \mathbf{A}_1^T \\ \mathbf{G}_{21} & \mathbf{G}_{22} & \mathbf{A}_2^T \\ \mathbf{A}_1 & \mathbf{A}_2 & -\mathbf{C} \end{pmatrix} \mathbf{P}^T. \tag{2.1}$$

The leading-block matrix **G** will be a suitably-chosen approximation to **H** for which $\mathbf{K}_G$ is easier to invert/factorize than $\mathbf{K}_H$, while **P** is an appropriately-chosen permutation matrix. Possible values of `preconditioner` are:

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

`1` **G** is chosen to be the identity matrix.

`2` **G** is chosen to be **H**

`3` **G** is chosen to be the diagonal matrix whose diagonals are the larger of those of **H** and a positive constant (see `min_diagonal` below).

`4` **G** is chosen to be the band matrix of given semi-bandwidth whose entries coincide with those of **H** within the band. (see `semi_bandwidth` below).

`11` **G** is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$ and $\mathbf{G}_{22} = \mathbf{H}_{22}$.

`12` **G** is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = \mathbf{H}_{21}$ and $\mathbf{G}_{22} = \mathbf{H}_{22}$.

`-1` for the special case when $\mathbf{C} = 0$, **G** is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22}$ is the identity matrix, and the preconditioner is computed implicitly.

`-2` for the special case when $\mathbf{C} = 0$, **G** is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22} = \mathbf{H}_{22}$ and the preconditioner is computed implicitly.

Other values may be added in future. The default is `preconditioner = 0`.

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

`factorization` is a scalar variable of type default `INTEGER`, that specifies which factorization of the preconditioner should be used. Possible values are:

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

`1` a Schur-complement factorization, involving factors of **G** and $\mathbf{A}\mathbf{G}^{-1}\mathbf{A}^T$, will be used.

`2` an augmented-system factorization, involving factors of $\mathbf{K}_G$, will be used.

The default is `factorization = 0`.

`max_col` is a scalar variable of type default `INTEGER`, that specifies the maximum number of nonzeros in a column of **A** which is permitted by the Schur-complement factorization. The default is `max_col = 35`.

`indmin` is a scalar variable of type default `INTEGER`, that specifies an initial estimate as to the amount of integer workspace required by the factorization package `SILS`. The default is `indmin = 1000`.

`valmin` is a scalar variable of type default `INTEGER`, that specifies an initial estimate as to the amount of real workspace required by the factorization package `SILS`. The default is `valmin = 1000`.

`len_ulsmin` is a scalar variable of type default `INTEGER`, that specifies an initial estimate as to the amount of workspace required by the factorization package `ULS`. The default is `len_ulsmin = 1000`.

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

`new_h` is a scalar variable of type default `INTEGER`, that is used to indicate how **H** has changed (if at all) since the previous call to `SBLS_form_and_factorize`. Possible values are:

`0` **H** is unchanged

`1` the values in **H** have changed, but its nonzero structure is as before.

`2` both the values and structure of **H** have changed.

The default is `new_h = 2`.

`new_a` is a scalar variable of type default `INTEGER`, that is used to indicate how **A** has changed (if at all) since the previous call to `SBLS_form_and_factorize`. Possible values are:

`0` **A** is unchanged

`1` the values in **A** have changed, but its nonzero structure is as before.

`2` both the values and structure of **A** have changed.

The default is `new_a = 2`.

`cg_maxit` is a scalar variable of type default `INTEGER`, that is used to limit the number of conjugate-gradient iterations performed in the optimality phase. If `cg_maxit` is negative, no limit will be impossed. The default is `cg_maxit = 200`.

`pivot_tol` is a scalar variable of type default `REAL` (double precision in `GALAHAD_EQP_double`), that holds the threshold pivot tolerence used by the matrix factorization. See the documentation for the packages `SILS` and `ULS` for details. The default is `pivot_tol = 0.01`.

`pivot_tol_for_basis` is a scalar variable of type default `REAL` (double precision in `GALAHAD_EQP_double`), that holds the threshold pivot tolerence used by the package `ULS` when computing the non-singular basis matrix $\mathbf{A}_1$ for an implicit preconditioner. Since the calculation of a suitable basis is crucial, it is sensible to pick a larger value of `pivot_tol_for_basis` than of `pivot_tol`. The default is `pivot_tol_for_basis = 0.5`.

`zero_pivot` is a scalar variable of type default `REAL` (double precision in `GALAHAD_EQP_double`), that is used to detect singularity. Any pivot encountered during the factorization whose absolute value is less than or equal to `zero_pivot` will be regarded as zero, and the matrix as singular. The default is `zero_pivot` $= u^{0.75}$ where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_EQP_double`).

`radius` is a scalar variable of type default `REAL` (double precision in `GALAHAD_EQP_double`), that may be used to specify an upper bound on the norm of the allowed solution (a “trust-region” constraint) during the iterative solution of the optimality phase of the problem. This is particularly useful if the problem is unbounded from below. If `radius` is set too small, there is a possibility that this will preclude the package from finding the actual solution. If `initial_radius` is not positive, it will be reset to the default value, `initial_radius = SQRT(0.1*HUGE(1.0))` (`SQRT(0.1*HUGE(1.0D0))` in `GALAHAD_EQP_double`).

`inner_fraction_opt` is a scalar variable of type default `REAL` (double precision in `GALAHAD_EQP_double`), that specifies the fraction of the optimal value which is acceptable for the iterative solution of the optimality phase of the problem using the package `GALAHAD_GLTR`, and correspond to the value `control%inner_fraction` in that package. A negative value is considered to be zero, and a value of larger than one is considered to be one. Reducing `fraction_opt` below one will result in a reduction of the computation performed at the expense of an inferior approximation to the optimal value. The default is `inner_fraction_opt = 0.1`.

`inner_stop_relative` and `inner_stop_absolute` are scalar variables of type default `REAL` (double precision in `GALAHAD_EQP_double`), that hold the relative and absolute convergence tolerances for the iterative solution of the optimality phase of the problem using the package `GALAHAD_GLTR`, and correspond to the values `control%stop_relative` and `control%stop_absolute` in that package. The defaults are `inner_stop_relative = 0.01` and `inner_stop_absolute =` $\sqrt{u}$, where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_EQP_double`).

`max_infeasibility_relative` and `max_infeasibility_absolute` are scalar variables of type default `REAL` (double precision in `GALAHAD_EQP_double`), that hold the relative and absolute tolerances for assessing infeasibility in the feasibility phase. If the constraints are believed to be rank deficient and the norm of the residual $\mathbf{A}\mathbf{x}_T + \mathbf{c}$ at a "typical" feasiblke point is larger than max(`max_infeasibility_relative`$*\|A\|$, `max_infeasibility_absolute`), the problem will be marked as infeasible, The defaults are `max_infeasibility_relative = max_infeasibility_absolute =` $u^{0.75}$ where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_EQP_double`).

`remove_dependencies` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if linear dependent constraints $\mathbf{A}\mathbf{x}+\mathbf{c}=\mathbf{0}$ should be removed and `.FALSE.` otherwise. The default is `remove_dependencies = .TRUE.`.

`find_basis_by_transpose` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the invertible subblock $\mathbf{A}_1$ of the columns of $\mathbf{A}$ is computed by analysing the transpose of $\mathbf{A}$ and `.FALSE.` if the analysis is based on $\mathbf{A}$ itself. Generally an analysis based on the transpose is faster. The default is `find_basis_by_transpose = .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.`.

`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 ignoreing 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 = ""`.

2.2.4 The derived data type for holding timing information

The derived data type `EQP_time_type` is used to hold elapsed CPU times for the various parts of the calculation. The components of `EQP_time_type` are:

`total` is a scalar variable of type default `REAL`, that gives the total time spent in the package.

`factorize` is a scalar variable of type default `REAL`, that gives the time spent factorizing the required matrices.

`solve` is a scalar variable of type default `REAL`, that gives the time spent computing the solution given the factorization(s).

2.2.5 The derived data type for holding informational parameters

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

status is a scalar variable of type default INTEGER, that gives the exit status of the algorithm. See Section 2.4 for details.

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

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

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

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

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

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

time is a scalar variable of type EQP_time_type whose components are used to hold elapsed CPU times for the various parts of the calculation (see Section 2.2.4).

SBLS_inform is a scalar variable of type SBLS_inform_type whose components are used to hold information relating to the formation and factorization of the preconditioner. See the documentation for the GALAHAD package SBLS for further details.

GLTR_inform is a scalar variable of type GLTR_inform_type whose components are used to hold information relating to the computation of the solution via the conjugate-gradient method. See the documentation for the GALAHAD package GLTR for further details.

2.2.6 The derived data type for holding problem data

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

2.3 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.5 for further features):

1. The subroutine EQP_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 EQP_solve is called to solve the problem.
3. The subroutine EQP_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by EQP_solve, at the end of the solution process.

2.3.1 The initialization subroutine

Default values are provided as follows:

```
CALL EQP_initialize( data, control )
```

`data` is a scalar `INTENT(OUT)` argument of type `EQP_data_type` (see Section 2.2.6). It is used to hold data about the problem being solved. `EQP_initialize` will ensure that all components that are allocatable arrays are disassociated.

`control` is a scalar `INTENT(OUT)` argument of type `EQP_control_type` (see Section 2.2.3). On exit, `control` contains default values for the components as described in Section 2.2.3. These values should only be changed after calling `EQP_initialize`.

2.3.2 The equality-constrained-quadratic programming subroutine

The equality-constrained quadratic programming algorithm is called as follows:

```
CALL EQP_solve( p, data, control, info )
```

`p` is a scalar `INTENT(INOUT)` argument of type `QPT_problem_type` (see Section 2.2.2). It is used to hold data about the problem being solved. The user must have allocated all array components, and set appropriate values for all components. 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.

The components `p%X` and `p%Y` must be set to initial estimates, $\mathbf{x}^0$, of the solution variables, **x**, and Lagrange multipliers for the constraints, **y**. 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` and `p%Y=0.0`.

On exit, the components `p%X` and `p%Y` will contain the best estimates of the solution variables **x**, and Lagrange multipliers for the constraints **y**. **Restrictions:** `p%n` > 0 and, `p%m` ≥ 0, and `prob%H_type` and `prob%A_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }.

`data` is a scalar `INTENT(INOUT)` argument of type `EQP_data_type` (see Section 2.2.6). 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 `EQP_initialize`.

`control` is a scalar `INTENT(INOUT)` argument of type `EQP_control_type` (see Section 2.2.3). Default values may be assigned by calling `EQP_initialize` prior to the first call to `EQP_solve`.

`info` is a scalar `INTENT(OUT)` argument of type `EQP_inform_type` (see Section 2.2.5). A successful call to `EQP_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.3.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

`info` is a scalar `INTENT(OUT)` argument of type `EQP_inform_type` exactly as for `EQP_solve`. Only the component `status` will be set on exit, and a successful call to `EQP_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.4 Warning and error messages

A negative value of `info%status` on exit from `EQP_solve` or `EQP_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 be inconsistent.

`-9.` An error was reported by the subroutine `SILS_analyse` called by `SBLS`. The return status from `SILS_analyse` is given in `inform%sils_analyse_status`. See the documentation for the GALAHAD package `SILS` for further details.

`-10.` An error was reported by the subroutine `SILS_factorize`called by `SBLS`. The return status from `SILS_factorize` is given in `inform%sils_factorize_status`. See the documentation for the GALAHAD package `SILS` for further details.

`-11.` An error was reported by the subroutine `SILS_solve` called by `SBLS`. The return status from `SILS_solve` is given in `inform%sils_solve_status`. See the documentation for the GALAHAD package `SILS` for further details.

`-12.` An error was reported by the subroutine `ULS_analyse` called by `SBLS`. The return status from `ULS_analyse` is given in `inform%uls_analyse_status`. See the documentation for the GALAHAD package `ULS` for further details.

`-14.` An error was reported by the subroutine `ULS_solve` called by `SBLS`. The return status from `ULS_solve` is given in `inform%uls_solve_status`. See the documentation for the GALAHAD package `ULS` for further details.

`-15.` The computed preconditioner has the wrong inertia and is thus unsuitable.

`-16.` The residuals from the preconditioning step are large, indicating that the factorization may be unsatisfactory.

2.5 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `GALAHAD_EQP_control_type` (see Section 2.2.3), by reading an appropriate data specification file using the subroutine `GALAHAD_EQP_read_specfile`. This facility is useful as it allows a user to change `EQP` control parameters without editing and recompiling programs that call `EQP`.

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 `GALAHAD_EQP_read_specfile` must start with a ”`BEGIN GALAHAD_EQP`” command and ends with an ”`END`” command. The syntax of the specfile is thus defined as follows:

```
  ( .. lines ignored by EQP_read_specfile .. )
    BEGIN EQP
       keyword    value
       .......    .....
       keyword    value
    END
  ( .. lines ignored by EQP_read_specfile .. )
```

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

and

```
    END EQP SPECIFICATION
```

are acceptable. Furthermore, between the "`BEGIN GALAHAD_EQP`" 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 `GALAHAD_EQP_read_specfile` is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is `REWIND`ed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `GALAHAD_EQP_read_specfile`.

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
     CALL EQP_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)`argument of type `GALAHAD_EQP_control_type` (see Section 2.2.3). Default values should have already have been set, perhaps by calling `GALAHAD_EQP_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 2.2.3) of `control` that each affects are given in Table 2.1.

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

2.6 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` $= 1$, the norm of the constraint violation and the value of the objective function for both the feasibility and optimality phases are reported. Additionally, if `control%print_level` $= 2$, `print_level = 1` output from both `SBLS` and `GLTR` occurs, summarising the factorization and iteration phases, as well as timing statistics from the two phases. If `control%print_level` ≥ 3 detailed output from `SBLS` and `GLTR` occurs which is unlikely to be useful to general users.

command	component of `control`	value type
`error-printout-device`	`%error`	integer
`printout-device`	`%out`	integer
`print-level`	`%print_level`	integer
`preconditioner-used`	`%preconditioner`	integer
`semi-bandwidth-for-band-preconditioner`	`%semi_bandwidth`	integer
`factorization-used`	`%factorization`	integer
`maximum-column-nonzeros-in-schur-complement`	`%max_col`	integer
`initial-workspace-for-unsymmetric-solver`	`%len_ulsmin`	integer
`initial-integer-workspace`	`%indmin`	integer
`initial-real-workspace`	`%valmin`	integer
`maximum-refinements`	`%itref_max`	integer
`maximum-number-of-cg-iterations`	`%cg_maxit`	integer
`trust-region-radius`	`%radius`	real
`max-relative-infeasibility-allowed`	`%max_infeasibility_relative`	real
`max-absolute-infeasibility-allowed`	`%max_infeasibility_absolute`	real
`pivot-tolerance-used`	`%pivot_tol`	real
`pivot-tolerance-used-for-basis`	`%pivot_tol_for_basis`	real
`inner-iteration-fraction-optimality-required`	`%inner_fraction_opt`	real
`inner-iteration-relative-accuracy-required`	`%inner_stop_relative`	real
`inner-iteration-absolute-accuracy-required`	`%inner_stop_absolute`	real
`find-basis-by-transpose`	`%find_basis_by_transpose`	logical
`remove-linear-dependencies`	`%remove_dependencies`	logical
`space-critical`	`%space_critical`	logical
`deallocate-error-fatal`	`%deallocate_error_fatal`	logical

Table 2.1: Specfile commands and associated components of `control`.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `EQP_solve` calls the GALAHAD packages `GALAHAD_CPU_time`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_QPT`, `GALAHAD_SBPS`, `GALAHAD_GLTR` and `GALAHAD_SPECFILE`.

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

Any finite solution $\mathbf{x}$ to the problem necessarily satisfies the primal optimality conditions

$$\mathbf{A}\mathbf{x} + \mathbf{c} = \mathbf{0} \tag{4.1}$$

and the dual optimality conditions

$$\mathbf{H}\mathbf{x}+\mathbf{g}-\mathbf{A}^T\mathbf{y}=\mathbf{0}, \quad (4.2)$$

where the components of the vector $\mathbf{y}$ are known as the Lagrange multipliers for the constraints.

A solution to the problem is found in two phases. In the first, a point $\mathbf{x}_F$ satisfying (4.1) is found. In the second, the required solution $\mathbf{x}=\mathbf{x}_F+\mathbf{s}$ is determined by finding $\mathbf{s}$ to minimize $q(\mathbf{s})=\frac{1}{2}\mathbf{s}^T\mathbf{H}\mathbf{s}+\mathbf{g}_F^T\mathbf{s}+f_F$ subject to the homogeneous constraints $\mathbf{A}\mathbf{s}=\mathbf{0}$, where $\mathbf{g}_F=\mathbf{H}\mathbf{x}_F+\mathbf{g}$ and $f_F=\frac{1}{2}\mathbf{x}_F^T\mathbf{H}\mathbf{x}_F+\mathbf{g}^T\mathbf{x}_F+f$. The required constrained minimizer of $q(\mathbf{s})$ is obtained by implictly applying the preconditioned conjugate-gradient method in the null space of $\mathbf{A}$. Any preconditioner of the form (2.1) is suitable, and the GALAHAD package SBLS provides a number of possibilities. In order to ensure that the minimizer obtained is finite, an additional, precautionary trust-region constraint $\|s\|\leq\Delta$ for some suitable positive radius Δ is imposed, and the GALAHAD package GLTR is used to solve this additionally-constrained problem.

References:

The preconditioning aspcets are described in detail in

H. S. Dollar, N. I. M. Gould and A. J. Wathen. “On implicit-factorization constraint preconditioners”. In Large Scale Nonlinear Optimization (G. Di Pillo and M. Roma, eds.) Springer Series on Nonconvex Optimization and Its Applications, Vol. 83, Springer Verlag (2006) 61–82

and

H. S. Dollar, N. I. M. Gould, W. H. A. Schilders and A. J. Wathen “On iterative methods and implicit-factorization preconditioners for regularized saddle-point systems”. SIAM Journal on Matrix Analysis and Applications, **28(1)** (2006) 170–189,

while the constrained conjugate-gradient method is discussed in

N. I. M. Gould, S. Lucidi, M. Roma and Ph. L. Toint, Solving the trust-region subproblem using the Lanczos method. SIAM Journal on Optimization **9:2** (1999), 504-525.

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 $2x_1+x_2-2=0$ and $x_2+x_3-2=0$. Then, on writing the data for this problem as

$$\mathbf{H}=\begin{pmatrix}1 & & 4\\ & 2 & \\ 4 & & 3\end{pmatrix},\ \mathbf{g}=\begin{pmatrix}0\\2\\0\end{pmatrix},\ \mathbf{A}=\begin{pmatrix}2 & 1 & \\ & 1 & 1\end{pmatrix} \text{ and } \mathbf{c}=\begin{pmatrix}-2\\-2\end{pmatrix}$$

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

```
! THIS VERSION: GALAHAD 2.1 - 22/03/2007 AT 09:00 GMT.
   PROGRAM GALAHAD_EQP_EXAMPLE
   USE GALAHAD_EQP_double         ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   TYPE ( QPT_problem_type ) :: p
   TYPE ( EQP_data_type ) :: data
   TYPE ( EQP_control_type ) :: control
   TYPE ( EQP_inform_type ) :: info
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
! start problem data
   ALLOCATE( p%G( n ), p%C( m ), p%X( n ), p%Y( 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 = (/ - 2.0_wp, - 2.0_wp /)             ! constraint constants
   p%X = 0.0_wp ; p%Y = 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 EQP_initialize( data, control )       !  Initialize control parameters
   CALL EQP_solve( p, data, control, info )   !  Solve problem
   IF ( info%status == 0 ) THEN               !  Successful return
     WRITE( 6, "( ' EQP: ', I0, ' CG iteration(s). Optimal objective value =', &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                 &
     info%cg_iter, info%obj, p%X
   ELSE                                       !  Error returns
     WRITE( 6, "( ' EQP_solve exit status = ', I6 ) " ) info%status
   END IF
   CALL EQP_terminate( data, control, info )  !  delete internal workspace
   DEALLOCATE( p%G, p%C, p%X, p%Y )           !  deallocate problem arrays
   DEALLOCATE( p%H%val, p%H%row, p%H%col, p%A%val, p%A%row, p%A%col )
   DEALLOCATE( p%H%type, p%A%type )
   END PROGRAM GALAHAD_EQP_EXAMPLE
```

This produces the following output:

```
 EQP: 1 CG iteration(s). Optimal objective value =  7.0541E+00
 Optimal solution =   3.2432E-01  1.3514E+00  6.4865E-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' )  ! Specify sparse-by-row
   CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS' )  ! storage for H and A
   ALLOCATE( p%H%val( h_ne ), p%H%col( h_ne ), p%H%ptr( n + 1 ) )
   ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 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' )  ! Specify dense
   CALL SMT_put( p%A%type, 'DENSE' )  ! storage for H and A
   ALLOCATE( p%H%val( n * ( n + 1 ) / 2 ) )
   ALLOCATE( p%A%val( n * m ) )
   p%H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 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' )  ! 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.