GALAHAD QPT

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package defines a derived type capable of supporting a variety of **quadratic programming problem storage schemes.** Quadratic programming aims to minimize or maximize either a general objective function

$$\tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f, \tag{1.1}$$

or sometimes a (shifted) squared-least-distance objective function,

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

subject to the general linear constraints

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

and the simple bound constraints

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

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

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

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

and

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

The principal use of the package is to allow exchange of data between GALAHAD subprograms and other codes.

ATTRIBUTES — Versions: `GALAHAD_QPT_single, GALAHAD_QPT_double`. **Uses:** `GALAHAD_SMT`. **Date:** April 2001. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

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_QPT_single
```

with the obvious substitution `GALAHAD_QPT_double`, `GALAHAD_QPT_single_64` and `GALAHAD_QPT_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` and `QPT_problem_type`, (Section 2.4) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Both the Hessian matrix $\mathbf{H}$ and the constraint Jacobian $\mathbf{A}$, the matrix whose rows are the vectors $\mathbf{a}_i^T$, $i = 1,\ldots,m$, 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 $\mathbf{H} = \mathbf{0}$ (i.e., $h_{ij} = 0$ for all $1 \leq i, j \leq n$), no explicit entries needs be stored.

2.2 Optimality conditions

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

$$\mathbf{A}\mathbf{x} = \mathbf{c}, \ \mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \ \text{and} \ \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u,$$

the dual optimality conditions

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

where

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

and the complementary slackness conditions

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

where 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 componentwise.

2.3 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.

2.4 The derived data type

Two derived data types, `SMT_TYPE` and `QPT_problem_type`, are accessible from the package. It is intended that, for any particular application, only those components which are needed will be set.

2.4.1 The derived data type for holding matrices

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

`type` is an allocatable array of rank one and type default `CHARACTER`, that holds a string which indicates the storage scheme used.

`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 may hold the number of matrix entries (see §2.1.2).

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

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

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

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

2.4.2 The derived data type for holding quadratic programs

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

name is a rank-one allocatable array of type default CHARACTER that may be used to hold the name of the problem.

new_problem_structure is a scalar variable of type default LOGICAL, that is .TRUE. if this is the first (or only) problem in a sequence of problems with identical "structure" to be attempted, and .FALSE. if a previous problem with the same "structure" (but different numerical data) has been solved. Here, the term "structure" refers both to the sparsity patterns of the Jacobian matrices **A** involved (but not their numerical values), to the zero/nonzero/infinity patterns (a bound is either zero, ± infinity, or a finite but arbitrary nonzero) of each of the constraint bounds, and to the variables and constraints that are fixed (both bounds are the same) or free (the lower and upper bounds are ± infinity, respectively).

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

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

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 contains 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, 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 QPT_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

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

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

H%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **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.

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

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

A is scalar variable of type SMT_TYPE that holds the Jacobian matrix **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 QPT_problem_type and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

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

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

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

A%ptr is a rank-one allocatable array of dimension m+1 and type INTEGER(ip_), that holds the starting position of each row of $\mathbf{A}$, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other 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 c_i^l. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity, where infinity is a solver-dependent value that will be recognised as infinity.

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 c_i^u. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than infinity, where infinity is a solver-dependent value that will be recognised as infinity.

DC_l is a rank-one allocatable array of dimension m and type REAL(rp_), that may hold the vector of parametric lower bounds $\delta\mathbf{c}^l$ on the general constraints. The i-th component of DC_l, $i = 1,\ldots,m$, contains δc_i^l. Only components corresponding to finite lower bounds c_i^l need be set.

DC_u is a rank-one allocatable array of dimension m and type REAL(rp_), that may hold the vector of parametric upper bounds $\delta\mathbf{c}^u$ on the general constraints. The i-th component of DC_u, $i = 1,\ldots,m$, contains δc_i^u. Only components corresponding to finite upper bounds c_i^u need be set.

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains x_j^l. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a solver-dependent value that will be recognised as infinity.

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains x_j^u. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a solver-dependent value that will be recognised as infinity.

DX_l is a rank-one allocatable array of dimension n and type REAL(rp_), that may hold the vector of parametric lower bounds $\delta\mathbf{x}^l$ on the variables. The j-th component of DX_l, $j = 1,\ldots,n$, contains δx^l_j. Only components corresponding to finite lower bounds x^l_j need be set.

DX_u is a rank-one allocatable array of dimension n and type REAL(rp_), that may hold the vector of parametric upper bounds $\delta\mathbf{x}^u$ on the variables. The j-th component of DX_u, $j = 1,\ldots,n$, contains δx^u_j. Only components corresponding to finite upper bounds x^u_j need be set.

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

X_status is a rank-one allocatable array of dimension m and type INTEGER(ip_), that holds the status of the problem variables (active or inactive). Variable j is said to be inactive if its value is fixed to the current value of X(j), in which case it can be interpreted as a parameter of the problem.

Z is a rank-one allocatable array of dimension n and type default REAL(rp_), that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the simple bound constraints (see Section 2.2). The j-th component of Z, $j = 1,\ldots,n$, contains z_j.

Z_l is a rank-one allocatable array of dimension n and type REAL(rp_), that may be used to hold a vector of lower bounds $\mathbf{z}^l$ on the dual variables. The j-th component of Z_l, $j = 1,\ldots,n$, contains z^l_j. Infinite bounds are allowed by setting the corresponding components of Z_l to any value smaller than -infinity, where infinity is a solver-dependent value that will be recognised as infinity.

Z_u is a rank-one allocatable array of dimension n and type REAL(rp_), that may be used to hold a vector of upper bounds $\mathbf{z}^u$ on the dual variables. The j-th component of Z_u, $j = 1,\ldots,n$, contains z^u_j. Infinite bounds are allowed by setting the corresponding components of Z_u to any value larger than that infinity, where infinity is a solver-dependent value that will be recognised as infinity.

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 $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$.

C_status is a rank-one allocatable array of dimension m and type INTEGER(ip_), that holds the status of the problem constraints (active or inactive). A constraint is said to be inactive if it is not included in the formulation of the problem under consideration.

Y is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the values $\mathbf{y}$ of estimates of the Lagrange multipliers corresponding to the general linear constraints (see Section 2.2). The i-th component of Y, $i = 1,\ldots,m$, contains y_i.

Y_l is a rank-one allocatable array of dimension n and type REAL(rp_), that may be used to hold a vector of lower bounds $\mathbf{y}^l$ on the Lagrange multipliers The i-th component of Y_l, $i = 1,\ldots,m$, contains y^l_i. Infinite bounds are allowed by setting the corresponding components of Y_l to any value smaller than -infinity, where infinity is a solver-dependent value that will be recognised as infinity.

Y_u is a rank-one allocatable array of dimension n and type REAL(rp_), that may be used to hold a vector of upper bounds $\mathbf{y}^u$ on the Lagrange multipliers The i-th component of Y_u, $i = 1,\ldots,m$, contains y^u_i. Infinite bounds are allowed by setting the corresponding components of Y_u to any value larger than that infinity, where infinity is a solver-dependent value that will be recognised as infinity.

3 GENERAL INFORMATION

Other modules used directly: GALAHAD_SMT.

Input/output: None.

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

4 EXAMPLE OF USE

Suppose we wish to present the data for the problem, "QPprob", of minimizing the objective function $\frac{1}{2}x_1^2+x_2^2+\frac{3}{2}x_3^2+4x_1x_3+2x_2+1$ subject to the general linear constraints $1 \leq 2x_1+x_2 \leq 2$, $x_2+x_3=2$, and simple bounds $-1 \leq x_1 \leq 1$ and $x_3 \leq 2$ to a minimizer in sparse co-ordinate format. Then, on writing the data for this problem as

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

and

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

we may use the following code segment:

```
   PROGRAM GALAHAD_QPT_EXAMPLE
   USE GALAHAD_QPT_double                        ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20 ! solver-dependent
   TYPE ( QPT_problem_type ) :: p
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
! start problem data
   ALLOCATE( p%name( 6 ) )
   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 ) )
   p%name = TRANSFER( 'QPprob', p%name )      ! name
   p%new_problem_structure = .TRUE.           ! new structure
   p%Hessian_kind = - 1 ; p%gradient_kind = - 1 ! generic quadratic program
   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' )  ! Specify co-ordinate
   CALL SMT_put( p%A%type, 'COORDINATE' )  ! storage for H and A
   ALLOCATE( p%H%val( h_ne ), p%H%row( h_ne ), p%H%col( h_ne ) )
   ALLOCATE( p%A%val( a_ne ), p%A%row( a_ne ), p%A%col( a_ne ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 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
! now call minimizer ....
```

```
! ...
! ... minimization call completed. Deallocate arrays
   DEALLOCATE( p%name, p%G, p%X_l, p%X_u, p%C, p%C_l, p%C_u, p%X, p%Y, p%Z )
   DEALLOCATE( p%H%val, p%H%row, p%H%col, p%A%val, p%A%row, p%A%col )
   END PROGRAM GALAHAD_QPT_EXAMPLE
```

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

```
!  sparse co-ordinate storage format
...
! problem data complete
```

and

```
  DEALLOCATE( p%H%val, p%H%row, p%H%col, p%A%val, p%A%row, p%A%col )
```

by

```
!  sparse row-wise storage format
   CALL SMT_put( p%H%type, 'SPARSE_BY_ROWS' )  ! Specify sparse row-wise
   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 /)
   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
```

and

```
   DEALLOCATE( p%H%val, p%H%col, p%H%ptr, p%A%val, p%A%col, p%A%ptr )
```

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 ), 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
```

and

```
! dense storage format: real components
   DEALLOCATE( p%H%val, p%A%val )
! real components 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( p%H%type, 'DIAGONAL' )  ! Specify dense storage for H
ALLOCATE( p%H%val( n ) )
p%H%val = (/ 1.0_wp, 0.0_wp, 3.0_wp /) ! Hessian values
```

Notice here that zero diagonal entries are stored.

For examples of how the derived data type `packagename_problem_type` may be used in conjunction with GALAHAD quadratic and least-distance programming codes, see the specification sheets for the packages `GALAHAD_QPA`, `GALAHAD_QPB`, `GALAHAD_LSQP` and `GALAHAD_PRESOLVE`.