GALAHAD BQPB

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package uses an primal-dual interior-point method to solve the **bound-constrained convex quadratic programming problem**

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

or the **bound-constrained shifted least-distance problem**

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

subject to the simple bound constraints

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

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

If general linear constraints are present, the package `GALAHAD_CQP` should be used instead.

ATTRIBUTES — Versions: `GALAHAD_BQPB_single, GALAHAD_BQPB_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPD`, `GALAHAD_FIT`, `GALAHAD_ROOTS`, `GALAHAD_CQP`, `GALAHAD_SBLS`, `GALAHAD_CRO`. **Date:** July 2021. **Origin:** N. I. M. Gould and D. P. Robinson, 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_BQPB_single
```

with the obvious substitution `GALAHAD_BQPB_double`, `GALAHAD_BQPB_single_64` and `GALAHAD_BQPB_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`, `BQPB_time_type`, `BQPB_control_type`, `BQPB_inform_type` and `BQPB_data_type` (Section 2.4) and the subroutines `BQPB_initialize`, `BQPB_solve`, `BQPB_terminate`, (Section 2.5) and `BQPB_read_specfile` (Section 2.7) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

When they are explicitly available, The Hessian matrix $\mathbf{H}$ may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix $\mathbf{H}$ 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. 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 $\mathbf{H}$, its row index i, column index j and value h_{ij} are stored in the l-th components of the integer arrays `H%row`, `H%col` and real array `H%val`, respectively. Only the entries in the lower triangle need be stored. The order is unimportant, but the total number of entries `H%ne` is also required.

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 $\mathbf{H}$, the i-th component of a integer array `H%ptr` holds the position of the first entry in this row, while `H%ptr` $(n+1)$ 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 $l =$ `H%ptr`(i), $\ldots$,`H%ptr` $(i+1)-1$ of the integer array `H%col`, and real array `H%val`, respectively. 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.

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.

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 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_BQPB 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 matrix **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.

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 BQPB_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 $\mathbf{g}$ 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 $\mathbf{g}$ 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.

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}^l_j$. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.4.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}^u_j$. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.4.3).

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

Z is a rank-one allocatable array of dimension n and type default REAL(rp_), that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the simple bound constraints (see Section 4). The j-th component of Z, $j = 1,\ldots,n$, contains z_j.

2.4.3 The derived data type for holding control parameters

The derived data type BQPB_control_type is used to hold controlling data. Default values may be obtained by calling BQPB_initialize (see Section 2.5.1), while components may also be changed by calling BQPB_read_specfile (see Section 2.7.1). The components of BQPB_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 BQPB_solve and BQPB_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 BQPB_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.

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

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

infeas_max is a scalar variable of type INTEGER(ip_), that specifies the number of iterations for which the overall infeasibility of the problem is not reduced by at least a factor reduce_infeas before the problem is flagged as infeasible (see reduce_infeas). The default is infeas_max = 200.

muzero_fixed is a scalar variable of type INTEGER(ip_), that specifies the number of iterations before the initial barrier parameter (see muzero) may be altered. The default is muzero_fixed = 1.

indicator_type is a scalar variable of type INTEGER(ip_), that specifies the type of indicator used to assess when a variable or constraint is active. Possible values are:

1 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_p (see below).

2 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_pd (see below) times the magnitude of its corresponding dual variable.

3 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_tapia (see below) times the distance to the same bound at the previous iteration.

The default is indicator_type = 3.

arc is a scalar variable of type INTEGER(ip_), that specifies the type of residual trajectory used to define the path to the solution. Possible values are:

1 the residual trajectory proposed by Zhang will be used.

2 the residual trajectory proposed by Zhao and Sun will be used; note this trajectory does not necessarily ensure convergence, so should be used with caution.

3 a combination in which Zhang's trajectory is used until the method determines that Zhou and Sun's will be better.

4 a mixed linear-quadratic variant of Zhang's proposal will be used.

The default is arc = 1.

series_order is a scalar variable of type INTEGER(ip_), that specifies the order of (Puiseux or Taylor) series to approximate the residual trajectory. The default is series_order = 2.

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_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_BQPB_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_BQPB_double).

perturb_h is a scalar variable of type REAL(rp_), that specifies any perturbation that is to be added to the diagonal of $\mathbf{H}$. The default is perturb_h = 0.0.

prfeas is a scalar variable of type REAL(rp_), that aims to specify the closest that any initial variable may be to infeasibility. Any variable closer to infeasibility than prfeas will be moved to prfeas from the offending bound. However, if a variable is range bounded, and its bounds are closer than prfeas apart, it will be moved to the mid-point of the two bounds. The default is prfeas = 10^4.

dufeas is a scalar variable of type REAL(rp_), that aims to specify the closest that any initial dual variable or Lagrange multiplier may be to infeasibility. Any variable closer to infeasibility than prfeas will be moved to dufeas from the offending bound. However, if a dual variable is range bounded, and its bounds are closer than dufeas apart, it will be moved to the mid-point of the two bounds. The default is dufeas = 10^4.

muzero is a scalar variable of type REAL(rp_), that holds the initial value of the barrier parameter. If muzero is not positive, it will be reset automatically to an appropriate value. The default is muzero = -1.0.

tau is a scalar variable of type REAL(rp_), that holds the weight attached to dual infeasibility compared to complementarity when assessing step acceptance. The default is tau = 1.0.

gamma_c is a scalar variable of type REAL(rp_), that holds the smallest value that individual complementarity pairs are allowed to be relative to the average as the iteration proceeds. The default is gamma_c = 10^{-5}.

gamma_f is a scalar variable of type REAL(rp_), that holds the smallest value the average complementarity is allowed to be relative to the dual infeasibility as the iteration proceeds. The default is gamma_c = 10^{-5}.

reduce_infeas is a scalar variable of type default REAL(rp_), that specifies the least factor by which the overall infeasibility of the problem must be reduced, over infeas_max consecutive iterations, for it not be declared infeasible (see infeas_max). The default is reduce_infeas = 0.99.

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

potential_unbounded is a scalar variable of type default REAL(rp_), that specifies smallest value of the potential function divided by the number of one-sided variable and constraint bounds that will be tolerated before the analytic center is declared to be unbounded. The default is potential_unbounded = -10.0.

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

mu_pounce is a scalar variable of type REAL(rp_) that specifies the value of the barrier parameter that must be attained before an extrapolating "pounce" may be attempted. The default is mu_pounce = 10^{-5}.

indicator_tol_p is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 1. The default is indicator_tol_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BQPB_double).

indicator_tol_pd is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 2. The default is indicator_tol_pd = 1.0.

indicator_tol_tapia is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 3. The default is indicator_tol_tapia = 0.9.

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

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

treat_zero_bounds_as_general is a scalar variable of type default LOGICAL. If it is set to .FALSE., variables which are only bounded on one side, and whose bound is zero, will be recognised as non-negativities/non-positivities rather than simply as lower- or upper-bounded variables. If it is set to .TRUE., any variable bound x_j^l or x_j^u which has the value 0.0 will be treated as if it had a general value. Setting treat_zero_bounds_as_general to .TRUE. has the advantage that if a sequence of problems are reordered, then bounds which are "accidentally" zero will be considered to have the same structure as those which are nonzero. However, GALAHAD_BQPB 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..

just_feasible is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm should stop as soon as a feasible point of the constraint set is found, and .FALSE. otherwise. The default is just_feasible = .FALSE..

getdua is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user-provided estimates of the dual variables should be replaced by estimates whose aim is to try to balance the requirements of dual feasibility and complementary slackness, and .FALSE. if users estimates are to be used. The default is getdua = .FALSE..

puiseux is a scalar variable of type default LOGICAL, that must be set .TRUE. if a Puiseux series will be used when extrapolating along the central path and .FALSE. if a Taylor series is preferred. We recommend using the Puiseux series unless the solution is known to be non-degenerate. The default is puiseux = .TRUE..

every_order is a scalar variable of type default LOGICAL, that must be set .TRUE. if every order of approximation up to series_order will be tried and the best taken, and .FALSE. if only the exact order specified by series_order will be used. The default is every_order = .TRUE..

feasol is a scalar variable of type default LOGICAL, that should be set .TRUE. if the final solution obtained will be perturbed so that variables close to their bounds are moved onto these bounds, and .FALSE. otherwise. The default is feasol = .FALSE..

balance_initial_complentarity is a scalar variable of type default LOGICAL, that should be set .TRUE. if the initial complementarity is required to be balanced, and .FALSE. otherwise. The default is balance_initial_complentarity = .FALSE..

crossover is a scalar variable of type default LOGICAL, that must be set .TRUE. if the solution is to be defined in terms of linearly-independent constraints, and .FALSE. otherwise. The default is crossover = .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 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.

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.

FIT_control is a scalar variable of type FIT_control_type whose components are used to control fitting of data to polynomials performed by the package GALAHAD_FIT. See the specification sheet for the package GALAHAD_FIT for details, and appropriate default values.

ROOTS_control is a scalar variable of type ROOTS_control_type whose components are used to control the polynomial root finding performed by the package GALAHAD_ROOTS. See the specification sheet for the package GALAHAD_ROOTS for details, and appropriate default values.

CRO_control is a scalar variable of type CRO_control_type whose components are used to control crossover performed by the package GALAHAD_CRO. See the specification sheet for the package GALAHAD_CRO for details, and appropriate default values.

2.4.4 The derived data type for holding timing information

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

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.

2.4.5 The derived data type for holding informational parameters

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

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.

nbacts is a scalar variable of type INTEGER(ip_), that gives the total number of backtracks performed during the sequence of linesearches.

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

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

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.

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

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 BQPB_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.

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.

FIT_inform is a scalar variable of type FIT_inform_type whose components are used to provide information about the fitting of data to polynomials performed by the package GALAHAD_FIT. See the specification sheet for the package GALAHAD_FIT for details.

ROOTS_inform is a scalar variable of type ROOTS_inform_type whose components are used to provide information about the polynomial root finding performed by the package GALAHAD_ROOTS. See the specification sheet for the package GALAHAD_ROOTS for details.

CRO_inform is a scalar variable of type CRO_inform_type whose components are used to provide information about the crossover performed by the package GALAHAD_CRO. See the specification sheet for the package GALAHAD_CRO for details.

2.4.6 The derived data type for holding problem data

The derived data type BQPB_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of BQPB procedures. This data should be preserved, untouched, from the initial call to BQPB_initialize to the final call to BQPB_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 BQPB_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 BQPB_solve is called to solve the problem.
3. The subroutine BQPB_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by BQPB_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 BQPB_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type BQPB_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 BQPB_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 BQPB_initialize.

inform is a scalar INTENT(OUT) argument of type BQPB_inform_type (see Section 2.4.5). A successful call to BQPB_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 BQPB_solve( prob, data, control, inform[, 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 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 and prob%Z=0.0.

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

data is a scalar INTENT(INOUT) argument of type BQPB_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 BQPB_initialize.

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

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

X_stat is an OPTIONAL rank-one INTENT(OUT) array argument of dimension prob%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$=1,\ldots,$ prob%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.

When control%crossover is .TRUE., more specific values are

- -1 the j-th simple bound constraint is both independent and active on its lower bound,
- -2 the j-th simple bound constraint is on its lower bound but linearly dependent on others,
- 1 the j-th simple bound constraint is both independent and active on its upper bound,
- 2 the j-th simple bound constraint is on its upper bound but linearly dependent on others, 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 BQPB_terminate( data, control, inform )
```

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

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

`inform` is a scalar `INTENT(OUT)` argument of type `BQPB_inform_type` exactly as for `BQPB_solve`. Only the component `status` will be set on exit, and a successful call to `BQPB_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 `BQPB_solve` or `BQPB_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 or the requirement that `prob%H_type` contain its relevant string `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL' SCALED_IDENTITY`, `IDENTITY` or `ZERO`, when **H** is available, has been violated.

`-4`. The bound constraints are inconsistent.

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

`-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.

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

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

```
( .. lines ignored by BQPB_read_specfile .. )
  BEGIN BQPB
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by BQPB_read_specfile .. )
```

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

and

```
END BQPB SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN BQPB" 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 BQPB_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 BQPB_read_specfile.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL BQPB_read_specfile( control, device )
```

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

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
maximum-number-of-iterations	%maxit	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
indicator-type-used	%indicator_type	integer
arc-used	%arc	integer
series-order	%series_order	integer
infinity-value	%infinity	real
identical-bounds-tolerance	%identical_bounds_tol	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
perturb-hessian-by	%perturb_h	real
initial-barrier-parameter	%muzero	real
feasibility-vs-complementarity-weight	%tao	real
balance-complementarity-factor	%gamma_c	real
balance-feasibility-factor	%gamma_f	real
poor-iteration-tolerance	%reduce_infeas	real
minimum-objective-before-unbounded	%obj_unbounded	real
minimum-potential-before-unbounded	%potential_unbounded	real
identical-bounds-tolerance	%identical_bounds_tol	real
required-barrier-value-before-pounce	mu_pounce	real
primal-indicator-tolerance	%indicator_tol_p	real
primal-dual-indicator-tolerance	%indicator_tol_pd	real
tapia-indicator-tolerance	%indicator_tol_tapia	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
treat-zero-bounds-as-general	%treat_zero_bounds_as_general	logical
just-find-feasible-point	%just_feasible	logical
puiseux-series	%puiseux	logical
try-every-order-of-series	%every_order	logical
cross-over-solution	%crossover	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

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

2.8 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control%out. If control%print_level $= 1$, a single line of output will be produced for each iteration of the process. For the initial-feasible-point phase, this will include values of the current primal and dual infeasibility, and violation of complementary slackness, the feasibility-phase objective value, the current steplength, the value of the barrier parameter, the number of backtracks in the linesearch and the elapsed CPU time in seconds. Once a suitable feasible point has been found, the iteration is divided into major iterations, at which the barrier parameter is reduced, and minor iterations, and which the barrier function is approximately minimized for the current value of the barrier parameter. For the major iterations, the value of the barrier parameter, the required values of dual feasibility and violation of complementary slackness, and the current constraint infeasibility are reported. Each minor iteration of the optimality phase results in a line giving the current dual feasibility and violation of complementary slackness, the objective function value, the ratio of predicted to achieved reduction of the objective function, the trust-region radius, the number of backtracks in the linesearch, the number of conjugate-gradient iterations taken, 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 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: BQPB_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_TOOLS, GALAHAD_SPECFILE, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_QPP, GALAHAD_QPD, GALAHAD_FIT, GALAHAD_ROOTS, GALAHAD_CQP, GALAHAD_SBLS and GALAHAD_CRO

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

Restrictions: prob%n > 0, prob%H_type $\in$ { 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL', 'SCALED_IDENTITY', 'IDENTITY', 'ZERO' }. (if **H** is 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

$$\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \tag{4.1}$$

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{z} \ \text{(or } \mathbf{W}^2(\mathbf{x} - \mathbf{x}^0) + \mathbf{g} = \mathbf{z} \ \text{for the least-distance type objective)} \tag{4.2}$$

where

$$\mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \ \mathbf{z}^l \geq 0 \ \text{and} \ \mathbf{z}^u \leq 0, \tag{4.3}$$

and the complementary slackness conditions

$$(\mathbf{x} - \mathbf{x}^l)^T \mathbf{z}^l = 0 \ \text{and} \ (\mathbf{x} - \mathbf{x}^u)^T \mathbf{z}^u = 0, \tag{4.4}$$

where the diagonal matrix $\mathbf{W}^2$ has diagonal entries w_j^2, $j = 1, \ldots, n$, where the vector $\mathbf{z}$ are the dual variables for the bounds, and where the vector inequalities hold component-wise.

Primal-dual interior point methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (4.2) and (4.4), while ensuring that (4.1) and (4.3) are satisfied as strict inequalities at each stage. Appropriate norms of the amounts by which (4.2) and (4.4) fail to be satisfied are known as the dual infeasibility, and the violation of complementary slackness, respectively. The fact that (4.1) and (4.3) are satisfied as strict inequalities gives such methods their other title, namely interior-point methods.

The method aims at each stage to reduce the overall violation of (4.2) and (4.4), rather than reducing each of the terms individually. Given an estimate $\mathbf{v} = (\mathbf{x}, \mathbf{z}, \mathbf{z}^l, \mathbf{z}^u)$ of the primal-dual variables, a correction $\Delta\mathbf{v} = \Delta(\mathbf{x}, \mathbf{z}, \mathbf{z}^l, \mathbf{z}^u)$ is obtained by solving a suitable linear system of Newton equations for the nonlinear systems (4.2) and a parameterized "residual trajectory" perturbation of (4.4); residual trajectories proposed by Zhang (1994) and Zhao and Sun (1999) are possibilities. An improved estimate $\mathbf{v} + \alpha\Delta\mathbf{v}$ is then used, where the step-size α is chosen as close to 1.0 as possible while ensuring both that (4.1) and (4.3) continue to hold and that the individual components which make up the complementary slackness (4.4) do not deviate too significantly from their average value. The parameter that controls the perturbation of (4.4) is ultimately driven to zero.

If the algorithm believes that it is close to the solution, it may take a speculative "pounce" extrapolation, based on an estimate of the ultimate active set, to avoid further costly iterations. If the pounce is unsuccessful, the iteration continues, but further pounces may be attempted later.

The Newton equations are solved by applying the GALAHAD matrix factorization package `GALAHAD_SBLS`, but there are options to factorize the matrix as a whole (the so-called "augmented system" approach), to perform a block elimination first (the "Schur-complement" approach), or to let the method itself decide which of the two previous options is more appropriate.

The package is actually just a front-end to the more-sophisticated GALAHAD package `CQP` that saves users from setting unnecessary arguments.

References:

The basic algorithm is a generalisation of those of

Y. Zhang (1994). On the convergence of a class of infeasible interior-point methods for the horizontal linear complementarity problem. SIAM J. Optimization **4** (1) 208-227,

and

G. Zhao and J. Sun (1999). On the rate of local convergence of high-order infeasible path-following algorithms for P_* linear complementarity problems. Computational Optimization and Applications **14** (1) 293-307,

with many enhancements described by

N. I. M. Gould, D. Orban and D. P. Robinson (2013). Trajectory-following methods for large-scale degenerate convex quadratic programming, *Mathematical Programming Computation* **5(2)** 113-142.

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 simple bounds $-1 \leq x_1 \leq 1$ and $x_3 \leq 2$. Then, on writing the data for this problem as

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

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

```
! THIS VERSION: GALAHAD 3.3 - 20/07/2021 AT 15:45 GMT.
   PROGRAM GALAHAD_BQPB_EXAMPLE
   USE GALAHAD_BQPB_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 ( BQPB_data_type ) :: data
   TYPE ( BQPB_control_type ) :: control
   TYPE ( BQPB_inform_type ) :: inform
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, h_ne = 4
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: X_stat
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%X( n ), p%Z( n ) )
   ALLOCATE( X_stat( n ) )
   p%new_problem_structure = .TRUE.           ! new structure
   p%n = n ; p%f = 1.0_wp                     ! dimensions & objective constant
   p%G = (/ 0.0_wp, 2.0_wp, 1.0_wp /)         ! objective gradient
   p%X_l = (/ - 1.0_wp, - infinity, 0.0_wp /) ! variable lower bound
   p%X_u = (/ infinity, 1.0_wp, 2.0_wp /)     ! variable upper bound
   p%X = 0.0_wp ; p%Z = 0.0_wp ! start from zero
!  sparse co-ordinate storage format
   CALL SMT_put( p%H%type, 'COORDINATE', s )     ! Co-ordinate  storage for H
   ALLOCATE( p%H%val( h_ne ), p%H%row( h_ne ), p%H%col( h_ne ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 1.0_wp, 3.0_wp /) ! Hessian H
   p%H%row = (/ 1, 2, 2, 3 /)                      ! NB lower triangle
   p%H%col = (/ 1, 2, 1, 3 /) ; p%H%ne = h_ne
! problem data complete
   CALL BQPB_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = 0.1_wp * infinity       ! Set infinity
!  control%print_level = 1                    ! print one line/iteration
   control%maxit = 100                        ! limit the # iterations
   inform%status = 1
   CALL BQPB_solve( p,  data, control, inform, X_stat )
   IF ( inform%status == 0 ) THEN             !  Successful return
     WRITE( 6, "( ' BQPB: ', I0, ' iterations  ', /,                          &
    &     ' Optimal objective value =',                                       &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )               &
     inform%iter, inform%obj, p%X
   ELSE                                       ! Error returns
     WRITE( 6, "( ' BQPB_solve exit status = ', I0 ) " ) inform%status
     WRITE( 6, * ) inform%alloc_status, inform%bad_alloc
   END IF
   CALL BQPB_terminate( data, control, inform )  !  delete workspace
   DEALLOCATE( p%G, p%X, p%X_l, p%X_u, p%Z, X_stat )
   DEALLOCATE( p%H%val, p%H%row, p%H%col, p%H%type )
   END PROGRAM GALAHAD_BQPB_EXAMPLE
```

This produces the following output:

```
 BQPB: 8 iterations
 Optimal objective value = -1.0000E+00
 Optimal solution =   2.0000E+00 -2.0000E+00  3.9754E-08
```

The same problem may be solved holding the 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 )     ! sparse row storage for H
   ALLOCATE( p%H%val( h_ne ), p%H%col( h_ne ), p%H%ptr( n + 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 /)
! problem data complete
```

or using a dense storage format with the replacement lines

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