GALAHAD BQP

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package uses a preconditioned, projected-gradient method to solve the **convex bound-constrained quadratic programming problem**

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

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-definite matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{x}^l$, $\mathbf{x}^u$ 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 $\mathbf{H}$; the matrix need not be provided as there are options to obtain matrix-vector products involving $\mathbf{H}$ either by reverse communication or from a user-provided subroutine.

ATTRIBUTES — Versions: `GALAHAD_BQP_single`, `GALAHAD_BQP_double`. **Uses:** `GALAHAD_CPU_time`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SBLS`, `GALAHAD_QPT`, `GALAHAD_SPECFILE`. **Date:** November 2009. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **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_BQP_single
```

with the obvious substitution `GALAHAD_BQP_double`, `GALAHAD_BQP_single_64` and `GALAHAD_BQP_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`, `BQP_time_type`, `BQP_control_type`, `BQP_inform_type` and `BQP_data_type` (Section 2.3) and the subroutines `BQP_initialize`, `BQP_solve`, `BQP_terminate`, (Section 2.4) and `BQP_read_specfile` (Section 2.8) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

When it is explicitly available, the Hessian matrix $\mathbf{H}$ may be stored in a variety of input formats.

2.1.1 Dense storage format

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. The lower-triangular part of $\mathbf{H}$ is stored as a compact dense matrix by rows, that is, the values of the entries that occur before or on the diagonal of each row in turn are stored in order within an appropriate real one-dimensional array. 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 lower triangular part of **H** is stored. For the l-th entry of the lower triangular part of **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. 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 nonzero entries of the lower triangular part of **H** is stored, but this time they are ordered so that those that occur before or on the diagonal in row i appear directly before those in row $i+1$. For the i-th row of **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 h_{ij} of the entries occurring before or on the diagonal in the i-th row are stored in components $l =$ `H%ptr`(i), ...,`H%ptr` $(i+1)-1$ of the integer array `H%col`, and real array `H%val`, respectively.

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.

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 The derived data types

Ten derived data types are accessible from the package.

2.3.1 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:

`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 §2.3.2).

`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).

2.3.2 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.

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H**. The following components are used:

H%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of H%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of H%type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of H%type must contain the string SPARSE_BY_ROWS, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of H%type must contain the string DIAGONAL.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into H%type. For example, if prob is of derived type BQP_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

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

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

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

H%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.1.

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

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

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

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

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

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 2.3.3).

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 2.3.3).

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.

2.3.3 The derived data type for holding control parameters

The derived data type BQP_control_type is used to hold controlling data. Default values may be obtained by calling BQP_initialize (see Section 2.4.1), while components may also be changed by calling BQP_read_specfile (see Section 2.8.1). The components of BQP_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 BQP_solve and BQP_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 BQP_solve is suppressed if out < 0. The default is out = 6.

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

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

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

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

cold_start is a scalar variable of type INTEGER(ip_), that should be set to 0 if a warm start is required (with variables assigned according to B_stat, see below), and to any other value if the values given in prob%X suffice. The default is cold_start = 1.

ratio_cg_vs_sd is a scalar variable of type INTEGER(ip_), that specifies the ratio of how many iterations use CG rather steepest descent will be attempted. The default is ratio_cg_vs_sd = 1.

change_max is a scalar variable of type INTEGER(ip_), that specifies the maximum number of per-iteration changes in the working set permitted when allowing subspace solution rather than steepest descent (see §4). The default is change_max = 2.

cg_maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of conjugate-gradient iterations which will be allowed per main iteration in GALAHAD_BQP_solve. The default is cg_maxit = 1000.

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_p is a scalar variable of type REAL(rp_), that holds the required accuracy for the primal infeasibility (see Section 4). The default is stop_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BQP_double).

stop_d is a scalar variable of type default REAL(rp_), that holds the required accuracy for the dual infeasibility (see Section 4). The default is stop_d = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BQP_double).

stop_c is a scalar variable of type default REAL(rp_), that holds the required accuracy for the violation of complementarity slackness (see Section 4). The default is stop_c = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BQP_double).

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

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

zero_curvature is a scalar variable of type REAL(rp_) that specifies the threshold below which any objective-function curvature encountered is regarded as zero. The default is zero_curvature = $10u$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_BQP_double).

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

exact_arcsearch is a scalar variable of type default LOGICAL, that should be set .TRUE. if the exact generalized Cauchy point, the first estimate of the minimizer of the objective in the Cauchy direction within the feasible box, is required, and .FALSE. if an approximation suffices. The default is exact_arcsearch = .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 = "".

SBLS_control is a scalar variable of type SBLS_control_type whose components are used to control the factorization and/or preconditioner used, performed by the package GALAHAD_SBLS. See the documentation for GALAHAD_SBLS for further details.

2.3.4 The derived data type for holding timing information

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

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

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

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 search direction.

2.3.5 The derived data type for holding informational parameters

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

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Section 2.7 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.

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.

time is a scalar variable of type BQP_time_type whose components are used to hold elapsed CPU times for the various parts of the calculation (see Section 2.3.4).

SBLS_inform is a scalar variable of type SBLS_inform_type whose components provide information about the progress and needs of the factorization/preconditioner performed by the package GALAHAD_SBLS. See the documentation for GALAHAD_SBLS for further details.

2.3.6 The derived data type for holding problem data

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

2.3.7 The derived data type for holding user data

The derived data type NLPT_userdata_type is available to allow the user to pass data to and from user-supplied matrix-vector product and preconditioning subroutines (see Section 2.5). Components of variables of type NLPT_userdata_type may be allocated as necessary. The following components are available:

integer is a rank-one allocatable array of type INTEGER(ip_).

real is a rank-one allocatable array of type default REAL(rp_)

complex is a rank-one allocatable array of type default COMPLEX (double precision complex in GALAHAD_BQP_double).

character is a rank-one allocatable array of type default CHARACTER.

logical is a rank-one allocatable array of type default LOGICAL.

integer_pointer is a rank-one pointer array of type INTEGER(ip_).

real_pointer is a rank-one pointer array of type default REAL(rp_)

complex_pointer is a rank-one pointer array of type default COMPLEX (double precision complex in GALAHAD_BQP_double).

character_pointer is a rank-one pointer array of type default CHARACTER.

logical_pointer is a rank-one pointer array of type default LOGICAL.

2.3.8 The derived data type for holding reverse-communication data

The derived data type BQP_reverse_type is used to hold data needed for reverse communication when this is required. The components of BQP_reverse_type are:

nz_v_start is a scalar variable of type INTEGER(ip_), that may be used to hold the starting position in NZ_v (see below) of the list of indices of nonzero components of **v**.

nz_v_end is a scalar variable of type INTEGER(ip_), that may be used to hold the finishing position in NZ_v (see below) of the list of indices of nonzero components of **v**.

NZ_v is a rank-one allocatable array of dimension n and type INTEGER(ip_), that may be used to hold the indices of the nonzero components of **v**. If used, components NZ_v(nz_v_start:nz_v_end) of V (see below) will be nonzero.

V is a rank-one allocatable array of dimension n and type REAL(rp_), that is used to hold the components of the output vector **v**.

PROD is a rank-one allocatable array of dimension n and type REAL(rp_), that is used to record the components of the resulting vector **Hv**.

nz_prod_end is a scalar variable of type INTEGER(ip_), that is used to record the finishing position in NZ_prod (see below) of the list of indices of nonzero components of **Hv** if required.

NZ_prod is a rank-one allocatable array of dimension n and type INTEGER(ip_), that is used to record the list of indices of nonzero components of **Hv** if required. Components NZ_prod(1:nz_prod_end) of PROD should then be nonzero.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.8 for further features):

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

We use square brackets [ ] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL BQP_initialize( data, control )
```

data is a scalar INTENT(INOUT) argument of type BQP_data_type (see Section 2.3.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type BQP_control_type (see Section 2.3.3). On exit, control contains default values for the components as described in Section 2.3.3. These values should only be changed after calling BQP_initialize.

2.4.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

```
CALL BQP_solve( prob, B_stat, data, control, inform, userdata[, reverse, eval_HPROD] )
```

prob is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.3.2). 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 perhaps prob%H; if the effect of **H** is only available to form products via reverse communication (see reverse below) or with a user-supplied subroutine (see eval_HPROD below) prob%H and it is not needed, but otherwise prob%H should be set using whichever of the matrix formats described in Section 2.1 is appropriate for **H** for the user's application.

The components prob%X and prob%Z must be set to initial estimates of the primal variables, **x** 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 and (if **H** is provided) prob%H%ne ≥ -2. prob%H_type $\in$ { 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' }.

B_stat is a rank-one INTENT(INOUT) array argument of dimension prob%n and type INTEGER(ip_), that indicates which of the simple bound constraints are in the current working set. Possible values for B_stat(j), j= 1,..., prob%n, and their meanings are

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

Suitable values must be supplied if control%bqp_control%cold_start = 0 on entry, but need not be provided for other input values of control%cold_start. Inappropriate values will be ignored. On exit, B_stat will contain values appropriate for the ultimate working set.

data is a scalar INTENT(INOUT) argument of type BQP_data_type (see Section 2.3.6). 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 BQP_initialize.

control is a scalar INTENT(IN) argument of type BQP_control_type (see Section 2.3.3). Default values may be assigned by calling BQP_initialize prior to the first call to BQP_solve.

inform is a scalar INTENT(INOUT) argument of type BQP_inform_type (see Section 2.3.5). On initial entry, the component status must be set to 1, while other components need not be set. A successful call to BQP_solve is indicated when the component status has the value 0. For other return values of status, see Sections 2.6 and 2.7.

userdata is a scalar INTENT(INOUT) argument of type NLPT_userdata_type whose components may be used to communicate user-supplied data (see Section 2.3.7) to and from the OPTIONAL subroutine eval_HPROD (see below).

reverse is an OPTIONAL scalar INTENT(INOUT) argument of type BQP_reverse_type (see Section 2.3.8). It is used to communicate reverse-communication data between the subroutine and calling program. If reverse is PRESENT and eval%HPROD (see below) is absent, the user should monitor inform%status on exit (see Section 2.6).

eval_HPROD is an OPTIONAL user-supplied subroutine whose purpose is to evaluate the value of the product **Hv** of the Hessian of the objective function with a given vector **v**. See Section 2.5.1 for details. If eval_HPROD is present, it must be declared EXTERNAL in the calling program. If eval_HPROD is absent, GALAHAD_BQP_solve will use reverse communication (see Section 2.6) to obtain Hessian-vector products if reverse is PRESENT or otherwise require that the user has provided all relevant components of prob%H.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type BQP_control_type exactly as for BQP_solve.

inform is a scalar INTENT(OUT) argument of type BQP_inform_type exactly as for BQP_solve. Only the component status will be set on exit, and a successful call to BQP_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.7.

2.5 Matrix-vector operations

2.5.1 Hessian-vector products via internal evaluation

If the argument eval_HPROD is present when calling GALAHAD_BQP_solve, the user is expected to provide a subroutine of that name to evaluate the product of the Hessian of the objective function **H** with a given vector **v**. The routine must be specified as

```
SUBROUTINE eval_HPROD( status, userdata, V, PROD[, NZ_v, nz_v_start,  nz_v_end,     &
                       NZ_prod, nz_prod_end] )
```

whose arguments are as follows:

status is a scalar INTENT(OUT) argument of type INTEGER(ip_), that should be set to 0 if the routine has been able to evaluate the product **Hv** and to a non-zero value if the evaluation has not been possible.

userdata is a scalar INTENT(INOUT) argument of type NLPT_userdata_type whose components may be used to communicate user-supplied data to and from the subroutine (see Section 2.3.7).

V is a rank-one INTENT(IN) array argument of type REAL(rp_) whose components contain the vector **v**. If components nz_v_start, nz_v_end and NZ_v (see below) are PRESENT, only components NZ_v(nz_v_start:nz_v_end) of V will be nonzero and the remaining components of V should be ignored. Otherwise, all components of V should be presumed to be nonzero.

PROD is a rank-one INTENT(OUT) array argument of type REAL(rp_) whose components on output contain the required components of **Hv**. If components nz_prod_end and NZ_prod (see below) are PRESENT, only the nonzero components NZ_prod(1:nz_prod_end) of PROD need be assigned. Otherwise, all components of PROD must be set.

nz_v_start is an OPTIONAL scalar variable of type INTEGER(ip_), that, if PRESENT, holds the starting position in NZ_v of the list of indices of nonzero components of **v**.

nz_v_end is an OPTIONAL scalar variable of type INTEGER(ip_), that, if PRESENT, holds the finishing position in NZ_v of the list of indices of nonzero components of **v**.

NZ_v is an OPTIONAL rank-one allocatable array of dimension *n* and type INTEGER(ip_) that, if PRESENT, holds the indices of the nonzero components of **v**. If any of nz_v_start, nz_v_end and NZ_v are absent, all components of V are assumed to be nonzero.

nz_prod_end is an OPTIONAL scalar variable of type INTEGER(ip_), that, if PRESENT, must be set to record the number of non-zeros in **Hv**.

NZ_prod is an OPTIONAL rank-one allocatable array of dimension *n* and type INTEGER(ip_) that, if PRESENT, must be set to record the list of indices of nonzero components of **Hv**. If either of nz_prod_end and NZ_prod are absent, all components of PROD should be set even if they are zero.

2.6 Reverse Communication Information

A positive value of inform%status on exit from BQP_solve indicates that GALAHAD_BQP_solve is seeking further information—this will happen if the user has chosen to evaluate matrix-vector products by reverse communication. The user should compute the required information and re-enter GALAHAD_BQP_solve with inform%status and all other arguments (except those specifically mentioned below) unchanged.

Possible values of inform%status and the information required are

2. The user should compute the matrix-vector product **Hv** using the vector **v** whose components are stored in reverse%V and store the required product in reverse%PROD.

3. The user should compute the matrix-vector product **Hv** using the vector **v** whose nonzero components are stored in positions reverse%NZ_v(reverse%nz_v_start:reverse%nz_v_end) of reverse%V. The remaining components of reverse%V should be ignored.

4. The user should compute the nonzero components of the matrix-vector product **Hv** using the vector **v** whose nonzero components are stored in positions reverse%NZ_v(reverse%nz_v_start:reverse%nz_v_end) of reverse%V. The remaining components of reverse%V should be ignored. The nonzero components must occupy positions reverse%NZ_prod(1:reverse%nz_prod_end) of reverse%PROD, and the components reverse%NZ_prod and reverse%nz_prod_end must be set. This return can only happen if control%exact_arcsearch is .TRUE..

2.7 Warning and error messages

A negative value of inform%status on exit from BQP_solve or BQP_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 the requirement that prob%H_type contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'DIAGONAL' when **H** is available, has been violated.

-4. The bound constraints are inconsistent.

-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 CPU time limit has been reached. This may happen if control%cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem.

-20. The Hessian matrix **H** of the objective function appears to be indefinite.

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

2.8 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type BQP_control_type (see Section 2.3.3), by reading an appropriate data specification file using the subroutine BQP_read_specfile. This facility is useful as it allows a user to change BQP control parameters without editing and recompiling programs that call BQP.

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

```
( .. lines ignored by BQP_read_specfile .. )
  BEGIN BQP
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by BQP_read_specfile .. )
```

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

and

```
END BQP SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN BQP" 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 BQP_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 BQP_read_specfile.

2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL BQP_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type BQP_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling BQP_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 2.3.3) of control that each affects are given in Table 2.1.

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

2.9 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

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
iterations-between-printing	%print_gap	integer
maximum-number-of-iterations	%maxit	integer
cold-start	%cold_start	integer
ratio-of-cg-iterations-to-steepest-descent	%ratio_cg_vs_sd	integer
max-change-to-working-set-for-subspace-solution	%change_max	integer
maximum-number-of-cg-iterations-per-iteration	%cg_maxit	integer
infinity-value	%infinity	real
primal-accuracy-required	%stop_p	real
dual-accuracy-required	%stop_d	real
complementary-slackness-accuracy-required	%stop_c	real
identical-bounds-tolerance	%identical_bounds_tol	real
cg-relative-accuracy-required	%stop_cg_relative	real
cg-absolute-accuracy-required	%stop_cg_absolute	real
zero-curvature-threshold	%zero_curvature	real
maximum-cpu-time-limit	%cpu_time_limit	real
exact-arcsearch-used	%exact_arcsearch	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.

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: BQP_solve calls the GALAHAD packages GALAHAD_CPU_time, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SBLS, GALAHAD_QPT 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%H_type $\in$ { 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' } (if **H** is explicit).

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

4 METHOD

The required solution **x** necessarily satisfies the primal optimality conditions

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

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{z} \text{ and } \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \tag{4.2}$$

and

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

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 components of the vector **z** are known as the dual variables for the bounds, and where the vector inequalities hold componentwise. Projected-gradient methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (4.2), while ensuring that (4.1), and (4.3) and (4.4) are satisfied at each stage. Appropriate norms of the amounts by which (4.1), (4.2) and (4.4) fail to be satisfied are known as the primal and dual infeasibility, and the violation of complementary slackness, respectively.

The method is iterative. Each iteration proceeds in two stages. Firstly, the so-called generalized Cauchy point for the quadratic objective is found. (The purpose of this point is to ensure that the algorithm converges and that the set of bounds which are satisfied as equations at the solution is rapidly identified.) Thereafter an improvement to the objective is sought using either a direct-matrix or truncated conjugate-gradient algorithm.

References:

This is a specialised version of the method presented in

A. R. Conn, N. I. M. Gould and Ph. L. Toint (1988). Global convergence of a class of trust region algorithms for optimization with simple bounds. SIAM Journal on Numerical Analysis **25** 433-460,

5 EXAMPLE OF USE

Suppose we wish to minimize $\frac{1}{2}x_1^2 + x_2^2 + x_1x_2 + \frac{3}{2}x_3^2 + 2x_2 + 1$ 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

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

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

```
! THIS VERSION: GALAHAD 4.2 - 2023-12-22 AT 14:45 GMT.
   PROGRAM GALAHAD_BQP_EXAMPLE
   USE GALAHAD_BQP_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 ( BQP_data_type ) :: data
   TYPE ( BQP_control_type ) :: control
   TYPE ( BQP_inform_type ) :: inform
   TYPE ( GALAHAD_userdata_type ) :: userdata
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, h_ne = 4
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: B_stat
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%X( n ), p%Z( n ) )
   ALLOCATE( B_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 BQP_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = infinity                  ! Set infinity
!  control%print_level = 1                      ! print one line/iteration
   inform%status = 1
   CALL BQP_solve( p,  B_stat, data, control, inform, userdata )
   IF ( inform%status == 0 ) THEN             !  Successful return
     WRITE( 6, "( ' BQP: ', I0, ' iterations  ', /,                          &
    &     ' Optimal objective value =',                                      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )              &
     inform%iter, inform%obj, p%X
   ELSE                                       ! Error returns
     WRITE( 6, "( ' BQP_solve exit status = ', I0 ) " ) inform%status
     WRITE( 6, * ) inform%alloc_status, inform%bad_alloc
   END IF
   CALL BQP_terminate( data, control, inform )  !  delete workspace
   DEALLOCATE( p%G, p%X, p%X_l, p%X_u, p%Z, B_stat )
   DEALLOCATE( p%H%val, p%H%row, p%H%col, p%H%type )
   END PROGRAM GALAHAD_BQP_EXAMPLE
```

This produces the following output:

```
 BQP: 2 iterations
 Optimal objective value = -1.0000E+00
 Optimal solution =   2.0000E+00 -2.0000E+00  0.0000E+00
```

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

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

by

```
! sparse row-wise storage format
   CALL SMT_put( p%H%type, 'SPARSE_BY_ROWS', s )  ! Specify sparse-by-row
   ALLOCATE( p%H%val( h_ne ), p%H%col( h_ne ), p%H%ptr( n + 1 ) )
   ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 1.0_wp, 3.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 3, 3 /)                     ! NB lower triangular
   p%H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( p%H%type, 'DENSE', s )  ! Specify dense
   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
! 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
```

The same problem may be solved using reverse communication with the following code:

```
! THIS VERSION: GALAHAD 3.3 - 03/06/2021 AT 08:15 GMT.
   PROGRAM GALAHAD_BQP_SECOND_EXAMPLE
   USE GALAHAD_BQP_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 ( BQP_reverse_type ) :: reverse
   TYPE ( BQP_data_type ) :: data
   TYPE ( BQP_control_type ) :: control
   TYPE ( BQP_inform_type ) :: inform
   TYPE ( GALAHAD_userdata_type ) :: userdata
   INTEGER :: nflag, i, j, k, l
   REAL ( KIND = wp ) :: v_j
   INTEGER, PARAMETER :: n = 3, h_ne = 4, h_all = 5
!  INTEGER, PARAMETER :: n = 3, h_ne = 3, h_all = 3
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: B_stat, FLAG, ROW, PTR
   REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : ) :: VAL
```

```
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%X( n ), p%Z( n ) )
   ALLOCATE( B_stat( n ), FLAG( n ) )
   ALLOCATE( VAL( h_all ), ROW( h_all ), PTR( n + 1 ) )
   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
   PTR = (/ 1, 3, 5, 6 /)                      ! whole Hessian by rows
   ROW = (/ 1, 2, 1, 2, 3 /)                   ! for matrix-vector products
   VAL = (/ 1.0_wp, 1.0_wp, 1.0_wp, 2.0_wp, 3.0_wp /)
! problem data complete
   CALL BQP_initialize( data, control, inform )  ! Initialize control parameters
   control%infinity = infinity                ! Set infinity
!  control%print_level = 1                    ! print one line/iteration
   control%maxit = 40                         ! limit the # iterations
!  control%print_gap = 100                    ! print every 100 terations
!  control%exact_gcp = .FALSE.
   nflag = 0 ; FLAG = 0
   inform%status = 1
10 CONTINUE            ! Solve problem - reverse commmunication loop
     CALL BQP_solve( p,  B_stat, data, control, inform, userdata, reverse )
     SELECT CASE ( inform%status )
     CASE ( 0 )          !  Successful return
       WRITE( 6, "( ' BQP: ', I0, ' iterations  ', /,                          &
      &     ' Optimal objective value =',                                      &
      &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )              &
       inform%iter, inform%obj, p%X
     CASE ( 2 )          ! compute H * v
       reverse%PROD = 0.0_wp
       DO j = 1, p%n
         v_j = reverse%V( j )
         DO k = PTR( j ), PTR( j + 1 ) - 1
           i = ROW( k )
           reverse%PROD( i ) = reverse%PROD( i ) + VAL( k ) * v_j
         END DO
       END DO
       GO TO 10
     CASE ( 3 )          ! compute H * v for sparse v
       reverse%PROD = 0.0_wp
       DO l = reverse%nz_v_start, reverse%nz_v_end
         j = reverse%NZ_v( l ) ; v_j = reverse%V( j )
         DO k = PTR( j ), PTR( j + 1 ) - 1
           i = ROW( k )
           reverse%PROD( i ) = reverse%PROD( i ) + VAL( k ) * v_j
         END DO
       END DO
       GO TO 10
     CASE ( 4 )          ! compute H * v for very sparse v and record nonzeros
       nflag = nflag + 1
       reverse%nz_prod = 0
       DO l = reverse%nz_v_start, reverse%nz_v_end
```

```
        j = reverse%NZ_v( l ) ; v_j = reverse%V( j )
        DO k = PTR( j ), PTR( j + 1 ) - 1
          i = ROW( k )
          IF ( FLAG( i ) < nflag ) THEN
            FLAG( i ) = nflag
            reverse%PROD( i ) = VAL( k ) * v_j
            reverse%nz_prod_end = reverse%nz_prod_end + 1
            reverse%NZ_prod( reverse%nz_prod_end ) = i
          ELSE
            reverse%PROD( i ) = reverse%PROD( i ) + VAL( k ) * v_j
          END IF
        END DO
      END DO
      GO TO 10
    CASE DEFAULT        ! Error returns
      WRITE( 6, "( ' BQP_solve exit status = ', I6 ) " ) inform%status
    END SELECT
  CALL BQP_terminate( data, control, inform, reverse )  !  delete workspace
  DEALLOCATE( p%G, p%X, p%X_l, p%X_u, p%Z, B_stat, FLAG, PTR, ROW, VAL )
  END PROGRAM GALAHAD_BQP_SECOND_EXAMPLE
```

Notice that storage for the Hessian is now not needed. This produces the same output.

The same problem may also be solved by user-provided matrix-vector products as follows:

```
! THIS VERSION: GALAHAD 3.3 - 03/06/2021 AT 08:15 GMT.
  PROGRAM GALAHAD_BQP_THIRD_EXAMPLE
  USE GALAHAD_BQP_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 ( BQP_reverse_type ) :: reverse
  TYPE ( BQP_data_type ) :: data
  TYPE ( BQP_control_type ) :: control
  TYPE ( BQP_inform_type ) :: inform
  TYPE ( GALAHAD_userdata_type ) :: userdata
  INTEGER, PARAMETER :: n = 3, h_ne = 4, h_all = 5
  INTEGER, PARAMETER :: len_integer = 2 * n + 3 + h_all, len_real = h_all
  INTEGER, PARAMETER :: nflag = 2, st_flag = 2, st_ptr = st_flag + n
  INTEGER, PARAMETER :: st_row = st_ptr + n + 1, st_val = 0
  INTEGER, ALLOCATABLE, DIMENSION( : ) :: B_stat
  EXTERNAL :: HPROD
! start problem data
  ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
  ALLOCATE( p%X( n ), p%Z( n ) )
  ALLOCATE( userdata%integer( len_integer ), userdata%real( len_real ) )
  ALLOCATE( B_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
! whole Hessian by rows for efficient matrix-vector products
  userdata%integer( st_ptr + 1 : st_ptr + n + 1 ) = (/ 1, 3, 5, 6 /)
  userdata%integer( st_row + 1 : st_row + h_all ) = (/ 1, 2, 1, 2, 3 /)
```

```
   userdata%real( st_val + 1 : st_val + h_all )                              &
     = (/ 1.0_wp, 1.0_wp, 1.0_wp, 2.0_wp, 3.0_wp /)
! problem data complete
   CALL BQP_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = infinity                ! Set infinity
!  control%print_level = 1                    ! print one line/iteration
   control%maxit = 40                         ! limit the # iterations
!  control%print_gap = 100                    ! print every 100 terations
!  control%exact_gcp = .FALSE.
   userdata%integer( 1 ) = n
   userdata%integer( nflag ) = 0
   userdata%integer( st_flag + 1 : st_flag + n ) = 0
   inform%status = 1
   CALL BQP_solve( p,  B_stat, data, control, inform, userdata,              &
                   eval_HPROD = HPROD )
   IF ( inform%status == 0 ) THEN             !  Successful return
     WRITE( 6, "( ' BQP: ', I0, ' iterations  ', /,                          &
    &     ' Optimal objective value =',                                      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )              &
     inform%iter, inform%obj, p%X
   ELSE
     WRITE( 6, "( ' BQP_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL BQP_terminate( data, control, inform )  !  delete workspace
   DEALLOCATE( p%G, p%X, p%X_l, p%X_u, p%Z, B_stat )
   DEALLOCATE( userdata%integer, userdata%real )
   END PROGRAM GALAHAD_BQP_THIRD_EXAMPLE

     SUBROUTINE HPROD( status, userdata, V, PROD, NZ_v, nz_v_start, nz_v_end,  &
                       NZ_prod, nz_prod_end )
! compute the matrix-vector product H * v
     USE GALAHAD_USERDATA_double
     INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
     INTEGER, INTENT( OUT ) :: status
     TYPE ( GALAHAD_userdata_type ), INTENT( INOUT ) :: userdata
     REAL ( KIND = wp ), DIMENSION( : ), INTENT( IN ) :: V
     REAL ( KIND = wp ), DIMENSION( : ), INTENT( OUT ) :: PROD
     INTEGER, OPTIONAL, INTENT( IN ) :: nz_v_start, nz_v_end
     INTEGER, OPTIONAL, INTENT( INOUT ) :: nz_prod_end
     INTEGER, DIMENSION( : ), OPTIONAL, INTENT( INOUT ) :: NZ_v
     INTEGER, DIMENSION( : ), OPTIONAL, INTENT( INOUT ) :: NZ_prod
     INTEGER :: i, j, k, l, n, nflag, st_flag, st_ptr, st_row, st_val
     REAL ( KIND = wp ) :: v_j
     n = userdata%integer( 1 )
     nflag = 2
     st_flag = 2
     st_ptr = st_flag + n
     st_row = st_ptr + n + 1
     st_val = 0
! compute H * v for very sparse v and record nonzeros
     IF ( PRESENT( NZ_prod ) .AND. PRESENT( nz_prod_end ) ) THEN
       userdata%integer( nflag ) = userdata%integer( nflag ) + 1
       nz_prod = 0
       DO l = nz_v_start, nz_v_end
         j = NZ_v( l ) ; v_j = V( j )
```

```
        DO k = userdata%integer( st_ptr + j ),                                &
               userdata%integer( st_ptr + j + 1 ) - 1
          i = userdata%integer( st_row + k )
          IF ( userdata%integer( st_flag + i ) <                              &
               userdata%integer( nflag ) ) THEN
            userdata%integer( st_flag + i ) = userdata%integer( nflag )
            PROD( i ) = userdata%real( st_val + k ) * v_j
            nz_prod_end = nz_prod_end + 1
            NZ_prod( nz_prod_end ) = i
          ELSE
            PROD( i ) = PROD( i ) + userdata%real( st_val + k ) * v_j
          END IF
        END DO
      END DO
! compute H * v for sparse v
    ELSE IF ( PRESENT( NZ_v ) .AND. PRESENT( nz_v_start ) .AND.               &
              PRESENT( nz_v_end ) ) THEN
      PROD = 0.0_wp
      DO l = nz_v_start, nz_v_end
        j = NZ_v( l ) ; v_j = V( j )
        DO k = userdata%integer( st_ptr + j ),                                &
               userdata%integer( st_ptr + j + 1 ) - 1
          i = userdata%integer( st_row + k )
          PROD( i ) = PROD( i ) + userdata%real( st_val + k ) * v_j
        END DO
      END DO
! compute H * v
    ELSE
      PROD = 0.0_wp
      DO j = 1, n
        v_j = V( j )
        DO k = userdata%integer( st_ptr + j ),                                &
               userdata%integer( st_ptr + j + 1 ) - 1
          i = userdata%integer( st_row + k )
          PROD( i ) = PROD( i ) + userdata%real( st_val + k ) * v_j
        END DO
      END DO
    END IF
    status = 0
    END SUBROUTINE HPROD
```

Now notice how the matrix **H** is passed to the matrix-vector product evaluation routine via the `integer` and `real` components of the derived type `userdata`.