GALAHAD RPD

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

Read and write data for the linear program (LP)

$$\text{minimize } \mathbf{g}^T\mathbf{x}+f \text{ subject to } \mathbf{c}_l \leq \mathbf{A}\mathbf{x} \leq \mathbf{c}_u \text{ and } \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u,$$

the linear program with quadratic constraints (QCP)

$$\text{minimize } \mathbf{g}^T\mathbf{x}+f \text{ subject to } \mathbf{c}_l \leq \mathbf{A}\mathbf{x} + \frac{1}{2}\text{vec}(\mathbf{x}.\mathbf{H}_c.\mathbf{x}) \leq \mathbf{c}_u \text{ and } \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u,$$

the bound-constrained quadratic program (BQP)

$$\text{minimize } \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f \text{ subject to } \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u,$$

the quadratic program (QP)

$$\text{minimize } \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f \text{ subject to } \mathbf{c}_l \leq \mathbf{A}\mathbf{x} \leq \mathbf{c}_u \text{ and } \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u,$$

or the quadratic program with quadratic constraints (QCQP)

$$\text{minimize } \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f \text{ subject to } \mathbf{c}_l \leq \mathbf{A}\mathbf{x} + \frac{1}{2}\text{vec}(\mathbf{x}.\mathbf{H}_c.\mathbf{x}) \leq \mathbf{c}_u \text{ and } \mathbf{x}_l \leq \mathbf{x} \leq \mathbf{x}_u,$$

involving the n by n symmetric matrices $\mathbf{H}$ and $(\mathbf{H}_c)_i$, $i = 1,\ldots,m$, the m by n matrix $\mathbf{A}$, the vectors $\mathbf{g}$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$, the scalar f, and where $\text{vec}(\mathbf{x}.\mathbf{H}_c.\mathbf{x})$ is the vector whose i-th component is $\mathbf{x}^T(\mathbf{H}_c)_i\mathbf{x}$ for the i-th constraint, **from and to a QPLIB-format data file**. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite. Full advantage is taken of any zero coefficients in the matrices $\mathbf{H}$, $(\mathbf{H}_c)_i$ and $\mathbf{A}$.

ATTRIBUTES — Versions: `GALAHAD_RPD_single`, `GALAHAD_RPD_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_NORMS`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_SPECFILE`, `GALAHAD_SORT`, `GALAHAD_LMS` **Date:** January 2006 **Origin:** N. I. M. Gould. **Language:** 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_RPD_single
```

with the obvious substitution `GALAHAD_RPD_double`, `GALAHAD_RPD_single_64` and `GALAHAD_RPD_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`, `RPD_control_type`, `RPD_inform_type`, `RPD_data_type`, (Section 2.3) and the subroutines `RPD_read_problem_data`, must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

The objective Hessian matrix **H**, the constraint Hessians $(H_c)_i$ and the constraint Jacobian **A** will be available in a sparse co-ordinate storage format.

2.1.1 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **A**, its row index i, column index j and value a_{ij} are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to **H** (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val` and an integer value `H%ne`), except that only the entries in the lower triangle need be stored. For the constraint Hessians, a third index giving the constraint involved is required for each entry, and is stored in the integer array `H%ptr`. Once again, only the lower traingle is 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 The derived data types

Five 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 matrices **H**, $(\mathbf{H}_c)_i$ and **A**. 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.

`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 the *symmetric* matrix **H** is represented as a single entry (see §2.1.1). 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 §2.1.1).

`col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.1.1).

`ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may holds the indices of the constraints involved when storing $(\mathbf{H}_c)_i$ (see §2.1.1). This component is not required when storing **H** or **A**.

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.

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

`H` is scalar variable of type `SMT_TYPE` that holds the Hessian matrix **H**, if required, in the sparse co-ordinate storage scheme (see Section 2.1.1). 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. Specifically, the first ten components of `H%type` will contain the string `COORDINATE`,
- `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 2.1.1).
- `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 the sparse co-ordinate storage scheme.
- `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.
- `H%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of the matrix $(\mathbf{H}_c)_i$ in the sparse co-ordinate scheme.

The components of `H` will only be set if the problem has a nonlinear objective function.

`G` is a rank-one allocatable array type `REAL(rp_)`, that will be allocated to have length `n`, and its j-th component filled with the value g_j for $j = 1,\ldots,n$.

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

`H_c` is scalar variable of type `SMT_TYPE` that holds the constraint Hessian matrices $(\mathbf{H}_c)_i$, if required, in the sparse co-ordinate storage scheme (see Section 2.1.1). The following components are used:

- `H_c%type` is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. Specifically, the first ten components of `H_c%type` will contain the string `COORDINATE`,
- `H_c%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the total number of entries in the **lower triangular** part of the collection of constraint Hessians $(\mathbf{H}_c)_i$ in the sparse co-ordinate storage scheme (see Section 2.1.1).
- `H_c%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 constraint Hessian matrices $(\mathbf{H}_c)_i$ in the sparse co-ordinate storage scheme.
- `H_c%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of $(\mathbf{H}_c)_i$ in the sparse co-ordinate storage scheme.
- `H_c%col` is a rank-one allocatable array variable of type default `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of $(\mathbf{H}_c)_i$ in the sparse co-ordinate scheme.
- `H_c%ptr` is a rank-one allocatable array of variable of type `INTEGER(ip_)`, that holds the constraint indices i of the constraint Hessians $(\mathbf{H}_c)_i$ in the sparse co-ordinate storage scheme.

The components of `H_c` will only be set if the problem has a nonlinear constraints.

`A` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix **A**, if required, in the sparse co-ordinate storage scheme (see Section 2.1.1). The following components are used:

A%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. Specifically, the first ten components of A%type will contain the string COORDINATE,

A%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in **A**, if any, in the sparse co-ordinate storage scheme (see Section 2.1.1).

A%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix **A** in the sparse co-ordinate storage scheme.

A%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of **A** in the sparse co-ordinate storage scheme.

A%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **A** in either the sparse co-ordinate scheme.

The components of A will only be set if the problem has general consraints.

infinity is a scalar variable of type REAL(rp_), that indicates when a variable or consraint bound is actually infinite. Any component of C_l or X_l (see below) that is smaller than -infinity should be viewed as $-\infty$, while those of of C_u or X_u (see below) that are larger than infinity should be viewed as ∞,

C_l is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i-th component of C_l, $i = 1,\ldots,m$, contains $\mathbf{c}^l_i$. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity.

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains $\mathbf{c}^u_i$. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than infinity.

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.

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.

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.

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

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

X_type is a rank-one allocatable array of dimension n and type INTEGER(ip_), that defines the types of variables. If X_type(i) = 0, variable x_i is allowed to take continuous values, if X_type(i) = 1, it may only take integer values, and if X_type(i) = 2, it is restricted to the binary choice, 0 or 1.

2.3.3 The derived data type for holding control parameters

The derived data type RPD_control_type is used to hold controlling data. Default values may be obtained by calling RPD_initialize (see Section 2.4.1). The components of RPD_control_type are:

qplib is a scalar variable of type INTEGER(ip_), that holds the stream number for input QPLIB file. The default is qplib = 21.

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in RPD_read_problem_data and RPD_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 RPD_read_problem_data 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.

space_critical is a scalar variable of type default LOGICAL, that may be set .TRUE. if the user wishes the package to allocate as little internal storage as possible, and .FALSE. otherwise. The package may be more efficient if space_critical is set .FALSE.. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that may be set .TRUE. if the user wishes the package to return to the user in the unlikely event that an internal array deallocation fails, and .FALSE. if the package should be allowed to try to continue. The default is deallocate_error_fatal = .FALSE..

2.3.4 The derived data type for holding informational parameters

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

status is a scalar variable of type INTEGER(ip_), that gives the current status of the algorithm. See Section 2.5 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last internal 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.

io_status is a scalar variable of type INTEGER(ip_), that gives the status of the last read attempt. This will be 0 if status = 0.

line is a scalar variable of type INTEGER(ip_), that gives the number of the last line read from the input file. This may be used to track an incorrectly-formated file.

p_type is a scalar variable of type default CHARACTER and length 3 that contains a key that describes the problem. The first character indicates the type of objective function used. It will be one of the following:

L a linear objective function.

D a convex quadratic objective function whose Hessian is a diagonal matrix.

C a convex quadratic objective function.

Q a quadratic objective function whose Hessian may be indefinite.

The second character indicates the types of variables that are present. It will be one of the following:

C all the variables are continuous.

B all the variables are binary (0-1).

M the variables are a mix of continuous and binary.

I all the variables are integer.

G the variables are a mix of continuous, binary and integer.

The third character indicates the type of the (most extreme) constraint function used; other constraints may be of a lesser type. It will be one of the following:

N there are no constraints.

B some of the variables lie between lower and upper bounds (box constraint).

L the constraint functions are linear.

D the constraint functions are convex quadratics with diagonal Hessians.

C the constraint functions are convex quadratics.

Q the constraint functions are quadratics whose Hessians may be indefinite.

Thus for continuous problems, we would have

LCL a linear program.

LCC or LCQ a linear program with quadratic constraints.

CCB or QCB a bound-constrained quadratic program.

CCL or QCL a quadratic program.

CCC or CCQ or QCC or QCQ a quadratic program with quadratic constraints.

For integer problems, the second character would be I rather than C, and for mixed integer problems, the second character would by M or G.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.6 for further features):

1. The subroutine RPD_initialize is used to set default values and initialize private data.
2. The subroutine RPD_read_problem_data is called to read the prolem from a specified QPLIB file into a QPT_problem_type structure.
3. The subroutine RPD_terminate is provided to allow the user to automatically deallocate array components of the problem structure set by RPD_read_problem_data once the input file has been proccessed.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL RPD_initialize( control, inform )
```

control is a scalar INTENT(OUT) argument of type RPD_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 RPD_initialize.

inform is a scalar INTENT(OUT) argument of type RPD_inform_type (see Section 2.3.4). A successful call to RPD_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

2.4.2 Subroutine to extract the data from a QPLIB format file

Extract the data from a QPLIB format file as follows:

```
CALL RPD_read_problem_data( problem, control, inform )
```

problem is a scalar INTENT(INOUT) argument of type qpt_problem_type (see Section 2.3.2) whose components will be filled with problem data extracted from the QPLIB file.

control is a scalar INTENT(IN) argument of type RPD_control_type (see Section 2.3.3). Default values may be assigned by calling RPD_initialize prior to the first call to RPD_read_problem_data. Of particular note, the component control%qplib specifies the stream number for input QPLIB file.

inform is a scalar INTENT(INOUT) argument of type RPD_inform_type (see Section 2.3.4) whose components need not be set on entry. A successful call to RPD_read_problem_data is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type RPD_control_type exactly as for RPD_read_problem_data.

inform is a scalar INTENT(OUT) argument of type RPD_inform_type exactly as for RPD_read_problem_data. Only the component status will be set on exit, and a successful call to RPD_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.5.

2.5 Warning and error messages

A negative value of inform%status on exit from RPD_read_problem_data or RPD_terminate indicates that an error might have 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.

-22. An input/output error occurred.

-25. The end of the input file was encountered before the problem specification was complete.

-29. The problem type was not recognised.

2.6 Further features

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

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

```
( .. lines ignored by RPD_read_specfile .. )
  BEGIN RPD
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by RPD_read_specfile .. )
```

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

and

```
END RPD SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN RPD" 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 RPD_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 RPD_read_specfile.

Control parameters corresponding to the components SLS_control and IR_control may be changed by including additional sections enclosed by "BEGIN SLS" and "END SLS", and "BEGIN IR" and "END IR", respectively. See the specification sheets for the packages GALAHAD_SLS and GALAHAD_IR for further details.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL RPD_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type RPD_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling RPD_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.

command	component of control	value type
qplib-file-device	%qplib	integer
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical

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.

2.7 Information printed

If control%print_level is positive, information about the progress of the algorithm may be printed on unit control%out.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: RPD_read_problem_data calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SPECFILE, GALAHAD_SORT and GALAHAD_LMS.

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

Portability: ISO Fortran 2003. The package is thread-safe.

4 METHOD

The QPBLIB format is defined in

F. Furini, E. Traversi, P. Belotti, A. Frangioni, A. Gleixner, N. Gould, L. Liberti, A. Lodi, R. Misener, H. Mittelmann, N. V. Sahinidis, S. Vigerske and A. Wiegele (2019). QPLIB: a library of quadratic programming instances, Mathematical Programming Computation **11** 237–265.

5 EXAMPLE OF USE

Suppose we wish to read the data encoded in the QPLIB file `ALLINIT.qplib` that may be found in the directory `examples` of the GALAHAD distribution. Then we may use the following code:

```
! THIS VERSION: GALAHAD 4.1 - 2022-11-27 AT 14:00 GMT.
   PROGRAM GALAHAD_RPD_example
   USE GALAHAD_RPD_double                       ! double precision version
   IMPLICIT NONE
   TYPE ( RPD_control_type ) :: control
   TYPE ( RPD_inform_type ) :: inform
   TYPE ( QPT_problem_type ) :: prob
   INTEGER :: i, length
   INTEGER :: qplib_unit = 21
   CHARACTER ( LEN = 8 ) :: galahad_var = 'GALAHAD'
   CHARACTER( LEN = : ), ALLOCATABLE :: galahad
!  open the QPLIB file ALLINIT.qplib for reading on unit 21
   CALL GET_ENVIRONMENT_VARIABLE( galahad_var, length = length )
   ALLOCATE( CHARACTER( LEN = length ):: galahad )
   CALL GET_ENVIRONMENT_VARIABLE( galahad_var, value = galahad )
   OPEN( qplib_unit, file = galahad // "/examples/ALLINIT.qplib",              &
         FORM = 'FORMATTED', STATUS = 'OLD' )
   CALL RPD_initialize( control, inform )
   control%qplib = qplib_unit
!  collect the problem statistics
   CALL RPD_read_problem_data( prob, control, inform )
   WRITE( 6, "( ' read status = ', I0 )" ) inform%status
   WRITE( 6, "( ' qplib example ALLINIT type = ', A )" ) inform%p_type
   WRITE( 6, "( ' n, m, h_ne, a_ne, h_c_ne =', 5I3 )" )                        &
     prob%n, prob%m, prob%H%ne, prob%A%ne, prob%H_c%ne
!  close the QPLIB file after reading
   CLOSE( qplib_unit )
   WRITE( 6, "( ' G =', 5F5.1 )" ) prob%G
   WRITE( 6, "( ' f =', F5.1 )" ) prob%f
   WRITE( 6, "( ' X_l =', 5F4.1 )" ) prob%X_l
   WRITE( 6, "( ' X_u =', 5F4.1 )" ) prob%X_u
   WRITE( 6, "( ' C_l =', 2F4.1 )" ) prob%C_l
   WRITE( 6, "( ' C_u =', 2ES8.1 )" ) prob%C_u
   IF ( ALLOCATED( prob%H%row ) .AND. ALLOCATED( prob%H%col ) .AND.            &
        ALLOCATED( prob%H%val ) ) THEN
     DO i = 1, prob%H%ne
       WRITE( 6, "( ' H(row, col, val) =', 2I3, F5.1 )" )                      &
         prob%H%row( i ), prob%H%col( i ), prob%H%val( i )
     END DO
   END IF
   IF ( ALLOCATED( prob%A%row ) .AND. ALLOCATED( prob%A%col ) .AND.            &
        ALLOCATED( prob%A%val ) ) THEN
     DO i = 1, prob%A%ne
       WRITE( 6, "( ' A(row, col, val) =', 2I3, F5.1 )" )                      &
         prob%A%row( i ), prob%A%col( i ), prob%A%val( i )
     END DO
   END IF
   IF ( ALLOCATED( prob%H_c%ptr ) .AND. ALLOCATED( prob%H_c%row ) .AND.        &
        ALLOCATED( prob%H_c%col ) .AND. ALLOCATED( prob%H_c%val ) ) THEN
     DO i = 1, prob%H_c%ne
```

```
      WRITE( 6, "( ' H_c(ptr, row, col, val) =', 3I3, F5.1 )" )                &
        prob%H_c%ptr( i ), prob%H_c%row( i ), prob%H_c%col( i ),               &
        prob%H_c%val( i )
    END DO
  END IF
  WRITE( 6, "( ' X_type =', 5I2 )") prob%X_type
  WRITE( 6, "( ' X =', 5F4.1 )" ) prob%X
  WRITE( 6, "( ' Y =', 2F4.1 )" ) prob%Y
  WRITE( 6, "( ' Z =', 5F4.1 )" ) prob%Z
!  deallocate internal array space
  CALL RPD_terminate( prob, control, inform )
  END PROGRAM GALAHAD_RPD_example
```

This produces the following output:

```
read status = 0
qplib example ALLINIT type = QGQ
n, m, h_ne, a_ne, h_c_ne =  5  2  9  4  1
G = -0.2 -0.4 -0.6 -0.8 -1.0
f =  0.0
X_l = 0.0 0.0 0.0 0.0 0.0
X_u = 2.0 2.0 2.0 2.0 2.0
C_l = 1.0 1.0
C_u = 1.0E+20 1.0E+20
H(row, col, val) =  1  1  2.0
H(row, col, val) =  2  1 -1.0
H(row, col, val) =  2  2  2.0
H(row, col, val) =  3  2 -1.0
H(row, col, val) =  3  3  2.0
H(row, col, val) =  4  3 -1.0
H(row, col, val) =  4  4  2.0
H(row, col, val) =  5  4 -1.0
H(row, col, val) =  5  5  2.0
A(row, col, val) =  1  1  1.0
A(row, col, val) =  1  3  1.0
A(row, col, val) =  2  2  1.0
A(row, col, val) =  2  4  1.0
H_c(ptr, row, col, val) =  1  1  1  2.0
X_type = 0 0 0 1 2
X = 0.0 0.0 0.0 0.0 0.0
Y = 0.0 0.0
Z = 0.0 0.0 0.0 0.0 0.0
```