



GALAHAD

LPB

PACKAGE SPECIFICATION

GALAHAD Optimization Library version 2.1

1 SUMMARY

This package uses an primal-dual interior-point trust-region method to solve the **linear programming problem**

$$\text{minimize } \mathbf{g}^T \mathbf{x} + f$$

subject to the general linear constraints

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

and the simple bound constraints

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

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

ATTRIBUTES — Versions: GALAHAD_LPB_single, GALAHAD_LPB_double. **Uses:** GALAHAD_CPU_time, GALAHAD_GL-TR, GALAHAD_ROOTS, GALAHAD_LSQP, GALAHAD_SPECFILE, GALAHAD_SILS, GALAHAD_QPP, GALAHAD_QPT, GALAHAD_SMT, GALAHAD_FDC. **Date:** December 1999. **Origin:** C. Cartis and N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

Access to the package requires a USE statement such as

Single precision version

```
USE GALAHAD_LPB_single
```

Double precision version

```
USE GALAHAD_LPB_double
```

If it is required to use both modules at the same time, the derived types SMT_type, QPT_problem_type, LPB_time_type, LPB_control_type, LPB_inform_type and LPB_data_type (Section 2.2) and the subroutines LPB_initialize, LPB_solve, LPB_terminate, (Section 2.3) and LPB_read_specfile (Section 2.5) must be renamed on one of the USE statements.

2.1 Matrix storage formats

The constraint Jacobian \mathbf{A} , the matrix whose rows are the vectors \mathbf{a}_i^T , $i = 1, \dots, m$, may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix \mathbf{A} is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n * (i - 1) + j$ of the storage array A%val will hold the value a_{ij} for $i = 1, \dots, m$, $j = 1, \dots, n$.

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of \mathbf{A} , its row index i , column index j and value a_{ij} are stored in the l -th components of the integer arrays `A%row`, `A%col` and real array `A%val`. The order is unimportant, but the total number of entries `A%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{A} , the i -th component of a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr(m + 1)` holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i -th row are stored in components $l = \text{A\%ptr}(i), \dots, \text{A\%ptr}(i + 1) - 1$ of the integer array `A%col`, and real array `A%val`, respectively.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2 The derived data types

Six derived data types are accessible from the package.

2.2.1 The derived data type for holding matrices

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

- `m` is a scalar component of type default `INTEGER`, that holds the number of rows in the matrix.
- `n` is a scalar component of type default `INTEGER`, that holds the number of columns in the matrix.
- `ne` is a scalar variable of type default `INTEGER`, that holds the number of matrix entries.
- `type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.2.2).
- `val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_LPB_double`) and dimension at least `ne`, that holds the values of the entries. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.
- `row` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.2).
- `col` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `ne`, that may hold the column indices of the entries (see §2.1.2–2.1.3).
- `ptr` is a rank-one allocatable array of type default `INTEGER`, and dimension at least `m + 1`, that may holds the pointers to the first entry in each row (see §2.1.3).

2.2.2 The derived data type for holding the problem

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

- `new_problem_structure` is a scalar variable of type default `LOGICAL`, that is `.TRUE.` if this is the first (or only) problem in a sequence of problems with identical "structure" to be attempted, and `.FALSE.` if a previous problem with the same "structure" (but different numerical data) has been solved. Here, the term "structure" refers both to the sparsity patterns of the Jacobian matrices \mathbf{A} involved (but not their numerical values), to the

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

zero/nonzero/infinity patterns (a bound is either zero, \pm infinity, or a finite but arbitrary nonzero) of each of the constraint bounds, and to the variables and constraints that are fixed (both bounds are the same) or free (the lower and upper bounds are \pm infinity, respectively).

- `n` is a scalar variable of type default `INTEGER`, that holds the number of optimization variables, n .
- `m` is a scalar variable of type default `INTEGER`, that holds the number of general linear constraints, m .
- `G` is a rank-one allocatable array of dimension `n` and type default `REAL` (double precision in `GALAHAD_LPB_double`), that holds the gradient \mathbf{g} of the linear term of the quadratic objective function. The j -th component of `G`, $j = 1, \dots, n$, contains \mathbf{g}_j .
- `f` is a scalar variable of type default `REAL` (double precision in `GALAHAD_LPB_double`), that holds the constant term, f , in the objective function.
- `A` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix \mathbf{A} . The following components are used:

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

The procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `A%type`. If `prob` is of derived type `LPB_problem_type` and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

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

- `A%ne` is a scalar variable of type default `INTEGER`, that holds the number of entries in \mathbf{A} in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for either of the other two schemes.
- `A%val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_LPB_double`), that holds the values of the entries of the Jacobian matrix \mathbf{A} in any of the storage schemes discussed in Section 2.1.
- `A%row` is a rank-one allocatable array of type default `INTEGER`, that holds the row indices of \mathbf{A} in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.
- `A%col` is a rank-one allocatable array variable of type default `INTEGER`, that holds the column indices of \mathbf{A} in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense storage scheme is used.
- `A%ptr` is a rank-one allocatable array of dimension `m+1` and type default `INTEGER`, that holds the starting position of each row of \mathbf{A} , as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.
- `C_l` is a rank-one allocatable array of dimension `m` and type default `REAL` (double precision in `GALAHAD_LPB_double`), that holds the vector of lower bounds \mathbf{c}^l on the general constraints. The i -th component of `C_l`, $i = 1, \dots, m$, contains \mathbf{c}_i^l . Infinite bounds are allowed by setting the corresponding components of `C_l` to any value smaller than `-infinity`, where `infinity` is a component of the control array `control` (see Section 2.2.3).
- `C_u` is a rank-one allocatable array of dimension `m` and type default `REAL` (double precision in `GALAHAD_LPB_double`), that holds the vector of upper bounds \mathbf{c}^u on the general constraints. The i -th component of `C_u`, $i = 1, \dots, m$, contains \mathbf{c}_i^u . Infinite bounds are allowed by setting the corresponding components of `C_u` to any value larger than `infinity`, where `infinity` is a component of the control array `control` (see Section 2.2.3).

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

- `X_l` is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_LPB_double), that holds the vector of lower bounds \mathbf{x}^l on the variables. The j -th component of `X_l`, $j = 1, \dots, n$, contains x_j^l . Infinite bounds are allowed by setting the corresponding components of `X_l` to any value smaller than `-infinity`, where `infinity` is a component of the control array `control` (see Section 2.2.3).
- `X_u` is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_LPB_double), that holds the vector of upper bounds \mathbf{x}^u on the variables. The j -th component of `X_u`, $j = 1, \dots, n$, contains x_j^u . Infinite bounds are allowed by setting the corresponding components of `X_u` to any value larger than that `infinity`, where `infinity` is a component of the control array `control` (see Section 2.2.3).
- `X` is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_LPB_double), that holds the values \mathbf{x} of the optimization variables. The j -th component of `X`, $j = 1, \dots, n$, contains x_j .
- `Z` is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_LPB_double), 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`, $j = 1, \dots, n$, contains z_j .
- `C` is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_LPB_double), that holds the values \mathbf{Ax} of the constraints. The i -th component of `C`, $i = 1, \dots, m$, contains $\mathbf{a}_i^T \mathbf{x} \equiv (\mathbf{Ax})_i$.
- `Y` is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_LPB_double), 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`, $i = 1, \dots, m$, contains y_i .

2.2.3 The derived data type for holding control parameters

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

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

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

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

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

`start_print` is a scalar variable of type default INTEGER, that specifies the first iteration for which printing will occur in `QPB_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 default INTEGER, that specifies the last iteration for which printing will occur in `QPB_solve`. If `stop_print` is negative, printing will occur once it has been started by `start_print`. The default is `stop_print` = -1.

`factor` is a scalar variable of type default INTEGER, that indicates the type of factorization of the preconditioner to be used. Possible values are:

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

- 0 the type is chosen automatically on the basis of which option looks likely to be the most efficient.
- 1 a Schur-complement factorization will be used.
- 2 an augmented-system factorization will be used.

The default is `factor = 0`.

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

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

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

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

`infeas_max` is a scalar variable of type default `INTEGER`, 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`.

`cg_maxit` is a scalar variable of type default `INTEGER`, that holds the maximum number of conjugate-gradient inner iterations that may be performed during the computation of each search direction in `QPB_solve`. If `cg_maxit` is set to a negative number, it will be reset by `QPB_solve` to the dimension of the relevant linear system +1. The default is `cg_maxit = 200`.

`precon` is a scalar variable of type default `INTEGER`, that specifies which preconditioner to be used to accelerate the conjugate-gradient inner iteration. Possible values are:

- 0 the type is chosen automatically on the basis of which option looks likely to be the most efficient at any given stage of the solution process. Different preconditioners may be used at different stages.
- 1 the Hessian matrix is replaced by the identity matrix.
- 2 a full factorization using the Hessian, which is equivalent to replacing the conjugate gradient inner iteration by a direct method. The Hessian may be perturbed to ensure that the resultant matrix is a preconditioner.
- 3 the Hessian matrix is replaced by a band of given semi-bandwidth (see `nsemib` below).
- 4 the Hessian matrix is replaced by its barrier terms.

The default is `precon = 0`.

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

`restore_problem` is a scalar variable of type default `INTEGER`, that specifies how much of the input problem is to be retored on output. Possible values are:

- 0 nothing is restored.
- 1 the vector data \mathbf{g} , \mathbf{c}^l , \mathbf{c}^u , \mathbf{x}^l , and \mathbf{x}^u will be restored to their input values.
- 2 the entire problem, that is the above vector data along with the Hessian matrix **H** and the Jacobian matrix **A**, will be restored.

The default is `restore_problem = 2`.

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

`infinity` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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 = 1019`.

`stop_p` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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_LPB_double).

`stop_d` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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_LPB_double).

`stop_c` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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_LPB_double).

`prfeas` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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 = 1.0`.

`dufeas` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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 = 1.0`.

`muzero` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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`.

`reduce_infeas` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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 (double precision in GALAHAD_LPB_double), 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 `potential_unbounded = $-u^{-2}$` , where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPB_double).

`pivot_tol` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), that holds the threshold pivot tolerance used by the matrix factorization. See the documentation for the package SILS for details. The default is `pivot_tol = $u^{3/4}$` , where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPB_double).

`pivot_tol_for_dependencies` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), that holds the threshold pivot tolerance used by the matrix factorization when attempting to detect linearly dependent constraints. A value larger than `pivot_tol` is appropriate. See the documentation for the package SILS for details. The default is `pivot_tol_for_dependencies = 0.5`.

`zero_pivot` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double). Any pivots smaller than `zero_pivot` in absolute value will be regarded to be zero when attempting to detect linearly dependent constraints. The default is `zero_pivot = $u^{3/4}$` , where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPB_double).

**All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html> .
For any commercial application, a separate license must be signed.**

`identical_bounds_tol` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double). 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_LPB_double).

`initial_radius` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), that specifies the initial trust-region radius. If `initial_radius` is not positive, it will be reset automatically to an appropriate value. The default is `initial_radius = - 1.0`.

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

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

`cpu_time_limit` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), 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`.

`remove_dependencies` is a scalar variable of type default LOGICAL, that must be set `.TRUE.` if the algorithm is to attempt to remove any linearly dependent constraints before solving the problem, and `.FALSE.` otherwise. We recommend removing linearly dependencies. The default is `remove_dependencies = .TRUE..`

`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_LPB 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..`

`center` is a scalar variable of type default LOGICAL, that must be set `.TRUE.` if the algorithm will should use the analytic center of the feasible set as its initial feasible point, and `.FALSE.` otherwise. We recommend using the analytic center. The default is `center = .TRUE..`

`primal` is a scalar variable of type default LOGICAL, that must be set `.TRUE.` if a primal-barrier Hessian will be used and `.FALSE.` if the primal-dual Hessian is preferred. We recommend using the primal-dual Hessian. The default is `primal = .FALSE..`

`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..`

`array_syntax_worse_than_do_loop` is a scalar variable of type default LOGICAL, that should be set `.TRUE.` if the compiler is better able to optimize Fortran 77-style do-loops than to exploit Fortran 95-style array syntax when

**All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html> .
For any commercial application, a separate license must be signed.**

performing vector operations, and `.FALSE.` otherwise. The default is `array_syntax_worse_than_do_loop = .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 = ""`.

`LSQP_control` is a scalar variable of type `LSQP_control_type` whose components are used to control the initial feasible point calculation, performed by the package `GALAHAD_LSQP`. See the specification sheet for the package `GALAHAD_LSQP` for details, and appropriate default values (but note that default value for `LSQP_control%feasol` is changed to `.FALSE..`

2.2.4 The derived data type for holding timing information

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

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

`preprocess` is a scalar variable of type default `REAL`, that gives the time spent reordering the problem to standard form prior to solution.

`find_dependent` is a scalar variable of type default `REAL`, that gives the time spent detecting and removing linearly-dependent equality constraints

`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.

`phase1_total` is a scalar variable of type default `REAL`, that gives the total time spent in the initial-feasible-point phase of the package.

`phase1_analyse` is a scalar variable of type default `REAL`, that gives the time spent analysing the required matrices prior to factorization in the initial-feasible-point phase.

`phase1_factorize` is a scalar variable of type default `REAL`, that gives the time spent factorizing the required matrices in the initial-feasible-point phase.

`phase1_solve` is a scalar variable of type default `REAL`, that gives the time spent computing the search direction in the initial-feasible-point phase.

2.2.5 The derived data type for holding informational parameters

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

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

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

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

`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`.

`iter` is a scalar variable of type default INTEGER, that gives the total number of iterations required.

`factorization_status` is a scalar variable of type default INTEGER, that gives the return status from the matrix factorization.

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

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

`nfacts` is a scalar variable of type default INTEGER, that gives the total number of factorizations performed.

`nbacts` is a scalar variable of type default INTEGER, that gives the total number of backtracks performed during the sequence of line searches.

`nmods` is a scalar variable of type default INTEGER, that gives the total number of factorizations which were modified to ensure that the matrix is an appropriate preconditioner.

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

`non_negligible_pivot` is a scalar variable of type default REAL (double precision in GALAHAD_LPB_double), that holds the value of the smallest pivot larger than `control%zero_pivot` when searching for dependent linear constraints. If `non_negligible_pivot` is close to `control%zero_pivot`, this may indicate that there are further dependent constraints, and it may be worth increasing `control%zero_pivot` above `non_negligible_pivot` and solving again.

`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 LPB_time_type whose components are used to hold elapsed CPU times for the various parts of the calculation (see Section 2.2.4).

2.2.6 The derived data type for holding problem data

The derived data type `LPB_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of LPB procedures. This data should be preserved, untouched, from the initial call to `LPB_initialize` to the final call to `LPB_terminate`.

2.3 Argument lists and calling sequences

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

1. The subroutine `LPB_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 `LPB_solve` is called to solve the problem.
3. The subroutine `LPB_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `LPB_solve`, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since `LPB_initialize` cannot test for this situation, and any existing associated targets will subsequently become unreachable.

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

2.3.1 The initialization subroutine

Default values are provided as follows:

```
CALL LPB_initialize( data, control )
```

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

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

2.3.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

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

`p` is a scalar INTENT(INOUT) argument of type `QPT_problem_type` (see Section 2.2.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for all components except `p%C`. `p%new_problem_structure` must be set `.TRUE.`, but will have been reset to `.FALSE.` on exit from `LPB_solve`. Users are free to choose whichever of the matrix formats described in Section 2.1 is appropriate for **A** for their application—different formats may be used for the two matrices.

For a problem with the same structure as one that has just been solved, the user may set `p%new_problem_structure` to `.FALSE.`, so long as `LPB_terminate` has not been called in the interim. The INTEGER components must be unaltered since the previous call to `LPB_solve`, but the REAL (double precision in `GALAHAD_LPB_double`) may be altered to reflect the new problem.

The components `p%X`, `p%Y` and `p%Z` must be set to initial estimates, \mathbf{x}^0 , of the primal variables, **x**, Lagrange multipliers for the general constraints, **y**, 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 `p%X=0.0`, `p%Y=0.0` and `p%Z=0.0`. The component `p%C` need not be set on entry.

On exit, the components `p%X`, `p%Y`, `p%Z` and `p%C` will contain the best estimates of the primal variables **x**, Lagrange multipliers for the general constraints **y**, dual variables for the bound constraints **z**, and values of the constraints **Ax** respectively. What of the remaining problem data has been restored depends upon the input value of the control parameter `control%restore_problem`. The return format for a restored array component will be the same as its input format. **Restrictions:** `p%n > 0`, `p%m ≥ 0` and `p%A%ne ≥ -2`.

`data` is a scalar INTENT(INOUT) argument of type `LPB_data_type` (see Section 2.2.6). It is used to hold data about the problem being solved. It must not have been altered by the user since the last call to `LPB_initialize`.

`control` is a scalar INTENT(IN) argument of type `LPB_control_type` (see Section 2.2.3). Default values may be assigned by calling `LPB_initialize` prior to the first call to `LPB_solve`.

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

2.3.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

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

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

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

2.4 Warning and error messages

A negative value of `info%status` on exit from `LPB_solve` or `LPB_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

- 1. One of the restrictions `prob%n > 0` or `prob%m ≥ 0` or requirements that `prob%A_type` and `prob%H_type` contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'DIAGONAL' has been violated.
- 2. 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.
- 3. 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.
- 5. The bound constraints are inconsistent.
- 6. The constraints appear to have no feasible point.
- 7. The factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.
- 8. The problem is so ill-conditioned that further progress is impossible.
- 9. The step is too small to make further impact.
- 10. 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.
- 11. The objective function appears to be unbounded from below on the feasible set.
- 12. The CPU time limit has been reached. This may happen if `cpu_time_limit` is too small, but may also be symptomatic of a badly scaled problem.

2.5 Further features

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

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

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `GALAHAD_LPB_read_specfile` must start with a "BEGIN GALAHAD_LPB" command and ends with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by LPB_read_specfile .. )
BEGIN LPB
  keyword      value
  .....      .....
  keyword      value
END
( .. lines ignored by LPB_read_specfile .. )
```

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

and

```
END LPB SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN GALAHAD_LPB" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when `GALAHAD_LPB_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 `GALAHAD_LPB_read_specfile`.

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL LPB_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `GALAHAD_LPB_control_type` (see Section 2.2.3). Default values should have already been set, perhaps by calling `GALAHAD_LPB_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.2.3) of `control` that each affects are given in Table 2.1.

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

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

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
factorization-used	%factor	integer
maximum-column-nonzeros-in-schur-complement	%max_col	integer
initial-integer-workspace	%indmin	integer
initial-real-workspace	%valmin	integer
maximum-refinements	%itref_max	integer
maximum-poor-iterations-before-infeasible	%infeas_max	integer
maximum-number-of-cg-iterations	%cg_maxit	integer
preconditioner-used	%precon	integer
semi-bandwidth-for-band-preconditioner	%nsemib	integer
restore-problem-on-output	%restore_problem	integer
infinity-value	%infinity	real
primal-accuracy-required	%stop_p	real
dual-accuracy-required	%stop_d	real
complementary-slackness-accuracy-required	%stop_c	real
mininum-initial-primal-feasibility	%prfeas	real
mininum-initial-dual-feasibility	%dufeas	real
initial-barrier-parameter	%muzero	real
poor-iteration-tolerance	%reduce_infeas	real
minimum-objective-before-unbounded	%obj_unbounded	real
pivot-tolerance-used	%pivot_tol	real
pivot-tolerance-used-for-dependencies	%pivot_tol_for_dependencies	real
zero-pivot-tolerance	%zero_pivot	real
identical-bounds-tolerance	%identical_bounds_tol	real
initial-trust-region-radius	%initial_radius	real
inner-iteration-fraction-optimality-required	%inner_fraction_opt	real
inner-iteration-relative-accuracy-required	%inner_stop_relative	real
inner-iteration-absolute-accuracy-required	%inner_stop_absolute	real
maximum-cpu-time-limit	%cpu_time_limit	real
remove-linear-dependencies	%remove_dependencies	logical
treat-zero-bounds-as-general	%treat_zero_bounds_as_general	logical
start-at-analytic-center	%center	logical
primal-barrier-used	%primal	logical
move-final-solution-onto-bound	%feasol	logical
array-syntax-worse-than-do-loop	%array_syntax_worse_than_do_loop	logical

Table 2.1: Specfile commands and associated components of control.

**All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html> .
For any commercial application, a separate license must be signed.**

2.6 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level = 1`, 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: `LPB_solve` calls the GALAHAD packages `GALAHAD_LSQP`, `GALAHAD_GLTR`, `GALAHAD_ROOTS`, `GALAHAD_QPP`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_SPECFILE`, `GALAHAD_SILS` and `GALAHAD_FDC`.

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

Restrictions: `prob%n > 0`, `prob%m ≥ 0`, `prob%A_type` and `prob%H_type ∈ { 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL' }`.

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

$$\mathbf{Ax} = \mathbf{c} \quad (4.1)$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \quad \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \quad (4.2)$$

the dual optimality conditions

$$\mathbf{g} = \mathbf{A}^T \mathbf{y} + \mathbf{z}, \quad \mathbf{y} = \mathbf{y}^l + \mathbf{y}^u \quad \text{and} \quad \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \quad (4.3)$$

and

$$\mathbf{y}^l \geq 0, \quad \mathbf{y}^u \leq 0, \quad \mathbf{z}^l \geq 0 \quad \text{and} \quad \mathbf{z}^u \leq 0, \quad (4.4)$$

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

and the complementary slackness conditions

$$(\mathbf{Ax} - \mathbf{c}^l)^T \mathbf{y}^l = 0, (\mathbf{Ax} - \mathbf{c}^u)^T \mathbf{y}^u = 0, (\mathbf{x} - \mathbf{x}^l)^T \mathbf{z}^l = 0 \text{ and } (\mathbf{x} - \mathbf{x}^u)^T \mathbf{z}^u = 0, \quad (4.5)$$

where the vectors \mathbf{y} and \mathbf{z} are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise. Primal-dual interior point methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (4.1), (4.3) and (4.5), while ensuring that (4.2) and (4.4) are satisfied as strict inequalities at each stage. Appropriate norms of the amounts by which (4.1), (4.3) and (4.5) fail to be satisfied are known as the primal and dual infeasibility, and the violation of complementary slackness, respectively. The fact that (4.2) and (4.4) are satisfied as strict inequalities gives such methods their other title, namely interior-point methods.

The problem is solved in two phases. The goal of the first "initial feasible point" phase is to find a strictly interior point which is primal feasible, that is that (4.1) is satisfied. The GALAHAD package `GALAHAD_LSQP` is used for this purpose, and offers the options of either accepting the first strictly feasible point found, or preferably of aiming for the so-called "analytic center" of the feasible region. Having found such a suitable initial feasible point, the second "optimality" phase ensures that (4.1) remains satisfied while iterating to satisfy dual feasibility (4.3) and complementary slackness (4.5). The optimality phase proceeds by approximately minimizing a sequence of barrier functions

$$\mathbf{g}^T \mathbf{x} + f - \mu \left[\sum_{i=1}^m \log(c_i - c_i^l) + \sum_{i=1}^m \log(c_i^u - c_i) + \sum_{j=1}^n \log(x_j - x_j^l) + \sum_{j=1}^n \log(x_j^u - x_j) \right],$$

for an appropriate sequence of positive barrier parameters μ converging to zero while ensuring that (4.1) remain satisfied and that \mathbf{x} and \mathbf{c} are strictly interior points for (4.2). Note that terms in the above summations corresponding to infinite bounds are ignored, and that equality constraints are treated specially.

Each of the barrier subproblems is solved using a trust-region method. Such a method generates a trial correction step $\Delta(\mathbf{x}, \mathbf{c})$ to the current iterate (\mathbf{x}, \mathbf{c}) by replacing the nonlinear barrier function locally by a suitable quadratic model, and approximately minimizing this model in the intersection of (4.1) and a trust region $\|\Delta(\mathbf{x}, \mathbf{c})\| \leq \Delta$ for some appropriate strictly positive trust-region radius Δ and norm $\|\cdot\|$. The step is accepted/rejected and the radius adjusted on the basis of how accurately the model reproduces the value of barrier function at the trial step. If the step proves to be unacceptable, a linesearch is performed along the step to obtain an acceptable new iterate. In practice, the natural primal "Newton" model of the barrier function is frequently less successful than an alternative primal-dual model, and consequently the primal-dual model is usually to be preferred.

The trust-region subproblem is approximately solved using the combined conjugate-gradient/Lanczos method implemented in the GALAHAD package `GALAHAD_GLTR`. Such a method requires a suitable preconditioner, and in our case, the only flexibility we have is in approximating the model of the Hessian. Although using a fixed form of preconditioning is sometimes effective, we have provided the option of an automatic choice, that aims to balance the cost of applying the preconditioner against the needs for an accurate solution of the trust-region subproblem. The preconditioner is applied using the GALAHAD matrix factorization package `GALAHAD_SILS`, but options at this stage are to factorize the preconditioner as a whole (the so-called "augmented system" approach), or to perform a block elimination first (the "Schur-complement" approach). The latter is usually to be preferred when a (non-singular) diagonal preconditioner is used, but may be inefficient if any of the columns of \mathbf{A} is too dense.

In order to make the solution as efficient as possible, the variables and constraints are reordered internally by the GALAHAD package `GALAHAD_QPP` prior to solution. In particular, fixed variables, and free (unbounded on both sides) constraints are temporarily removed.

References:

The method is described in detail in

A. R. Conn, N. I. M. Gould, D. Orban and Ph. L. Toint (1999). A primal-dual trust-region algorithm for minimizing a non-convex function subject to general inequality and linear equality constraints. *Mathematical Programming* **87** 215-249. ????

**All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html> .
For any commercial application, a separate license must be signed.**

5 EXAMPLE OF USE

Suppose we wish to minimize $\frac{3}{2}2x_2 + 1$ subject to the the general linear constraints $1 \leq 2x_1 + x_2 \leq 2$ and $x_2 + x_3 = 2$, and simple bounds $-1 \leq x_1 \leq 1$ and $x_3 \leq 2$. Then, on writing the data for this problem as

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

and

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

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

```
! THIS VERSION: GALAHAD 2.1 - 22/03/2007 AT 09:00 GMT.
  USE GALAHAD_LPB_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 ( LPB_data_type ) :: data
  TYPE ( LPB_control_type ) :: control
  TYPE ( LPB_inform_type ) :: info
  INTEGER :: s
  INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
! start problem data
  ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
  ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ) )
  ALLOCATE( p%X( n ), p%Y( m ), p%Z( n ) )
  p%new_problem_structure = .TRUE.          ! new structure
  p%n = n ; p%m = m ; p%f = 1.0_wp         ! dimensions & objective constant
  p%G = ( / 0.0_wp, 2.0_wp, 0.0_wp / )     ! objective gradient
  p%C_l = ( / 1.0_wp, 2.0_wp / )           ! constraint lower bound
  p%C_u = ( / 2.0_wp, 2.0_wp / )           ! constraint upper bound
  p%X_l = ( / - 1.0_wp, - infinity, - infinity / ) ! variable lower bound
  p%X_u = ( / 1.0_wp, infinity, 2.0_wp / ) ! variable upper bound
  p%X = 0.0_wp ; p%Y = 0.0_wp ; p%Z = 0.0_wp ! start from zero
! sparse co-ordinate storage format
  CALL SMT_put( p%A%type, 'COORDINATE', s ) ! Specify co-ordinate storage for A
  ALLOCATE( p%A%val( a_ne ), p%A%row( a_ne ), p%A%col( a_ne ) )
  p%A%val = ( / 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp / ) ! Jacobian A
  p%A%row = ( / 1, 1, 2, 2 / )
  p%A%col = ( / 1, 2, 2, 3 / ) ; p%A%ne = a_ne
! problem data complete
  CALL LPB_initialize( data, control )      ! Initialize control parameters
  control%infinity = infinity              ! Set infinity
  CALL LPB_solve( p, data, control, info ) ! Solve problem
  IF ( info%status == 0 ) THEN             ! Successful return
    WRITE( 6, "( ' LPB: ', I0, ' iterations. Optimal objective value =',      &
      &      ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )           &
      info%iter, info%obj, p%X
  ELSE
    ! Error returns
    WRITE( 6, "( ' LPB_solve exit status = ', I6 ) " ) info%status
  END IF
  CALL LPB_terminate( data, control, info ) ! delete internal workspace
```

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.

END PROGRAM GALAHAD_LPB_EXAMPLE

This produces the following output:

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%A$type, 'SPARSE_BY_ROWS' ) ! storage for A
ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )
p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
p%A%col = (/ 1, 2, 2, 3 /)
p%A%ptr = (/ 1, 3, 5 /) ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
CALL SMT_put( p%A$type, 'DENSE' ) ! storage for A
ALLOCATE( p%A%val( n * m ) )
p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian
! problem data complete
```

respectively.

All use is subject to licence. See <http://galahad.rl.ac.uk/galahad-www/cou.html>.
For any commercial application, a separate license must be signed.