GALAHAD   CLLS

1 SUMMARY

This package uses a primal-dual interior-point method to solve the **polyhedrally-constrained regularized linear least-squares problem**

$$\text{minimize} \quad \tfrac{1}{2}\|\mathbf{A}_o\mathbf{x} - \mathbf{b}\|_W^2 + \tfrac{1}{2}\sigma\|\mathbf{x}\|^2 \tag{1.1}$$

subject to the general linear constraints

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

and the simple bound constraints

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

where the norms $\|r\|_W = \sqrt{\sum_{i=1}^o w_i r_i^2}$ and $\|x\| = \sqrt{\sum_{i=1}^n x_i^2}$, and the o by n matrix $\mathbf{A}_o$, the vectors $\mathbf{b}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the weights $w > 0$ and $\sigma \geq 0$ are given, 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 matrix $\mathbf{A}_o$ and the matrix $\mathbf{A}$ of vectors $\mathbf{a}_i$.

ATTRIBUTES — Versions: `GALAHAD_CLLS_single`, `GALAHAD_CLLS_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_LSP`, `GALAHAD_QPD`, `GALAHAD_CQP`, `GALAHAD_FIT`, `GALAHAD_ROOTS`, `GALAHAD_NORMS`, `GALAHAD_SLS`, `GALAHAD_CRO`, `GALAHAD_FDC`. **Date:** October 2023. **Origin:** N. I. M. Gould. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

Access to the package requires a `USE` statement such as

Single precision version

        USE GALAHAD_CLLS_single

Double precision version

        USE GALAHAD_CLLS_double

If it is required to use both modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `NLPT_userdata_type`, `CLLS_time_type`, `CLLS_control_type`, `CLLS_inform_type` and `CLLS_data_type` (Section 2.4) and the subroutines `CLLS_initialize`, `CLLS_solve`, `CLLS_terminate`, (Section 2.5) and `CLLS_read_specfile` (Section 2.7) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

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

2.1.1 Dense row-wise storage format

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

**All use is subject to the conditions of the GNU Lesser General Public License version 3.
See** `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

2.1.2 Dense column-wise storage format

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

2.1.3 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.

2.1.4 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 **A**, the i-th component of a integer array A%ptr holds the position of the first entry in this row, while A%ptr $(m+1)$ holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i-th row are stored in components $l = $ A%ptr$(i), \ldots,$A%ptr $(i+1)-1$ of the integer array A%col, and real array A%val, respectively.

2.1.5 Sparse column-wise storage format

Once again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j + 1$. For the j-th column of **A**, the j-th component of a integer array A%ptr holds the position of the first entry in this column, while A%ptr $(n+1)$ holds the total number of entries plus one. The row indices i and values a_{ij} of the entries in the j-th column are stored in components $l = $ A%ptr$(j), \ldots,$A%ptr $(j+1)-1$ of the integer array A%row, and real array A%val, respectively.

For sparse matrices, the latter pair schemes almost always require less storage than the co-ordinate one.

2.2 Integer kinds

We use the term long INTEGER to denote INTEGER(kind=long), where long = selected_int_kind(18)).

2.3 Parallel usage

OpenMP may be used by the GALAHAD_CLLS package to provide parallelism for some solvers in shared memory environments. See the documentation for the GALAHAD package SLS for more details. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of -openmp). The number of threads may be controlled at runtime by setting the environment variable OMP_NUM_THREADS.

MPI may also be used by the package to provide parallelism for some solvers in a distributed memory environment. To use this form of parallelism, MPI must be enabled at runtime by using the correct compiler flag (usually some variant of -lmpi). Although the MPI process will be started automatically when required, it should be stopped by the calling program once no further use of this form of parallelism is needed. Typically, this will be via statements of the form

```
CALL MPI_INITIALIZED( flag, ierr )
IF ( flag ) CALL MPI_FINALIZE( ierr )
```

The code may be compiled and run in serial mode.

2.4 The derived data types

Ten derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrices $\mathbf{A}_o$ and $\mathbf{A}$. 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 §2.4.2).

val   is a rank-one allocatable array of type default REAL (double precision in GALAHAD_CLLS_double) and dimension at least ne, that holds the values of the entries.

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 §2.1.3 & 2.1.5).

col   is a rank-one allocatable array of type default INTEGER, and dimension at least ne, that may the column indices of the entries (see §2.1.3–2.1.4).

ptr   is a rank-one allocatable array of type default INTEGER, and dimension at least m + 1, that may hold the pointers to the first entry in each row (see §2.1.4), or dimension at least n + 1, that may hold the pointers to the first entry in each column (see §2.1.5),

2.4.2 The derived data type for holding the problem

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

n   is a scalar variable of type default INTEGER, that holds the number of optimization variables, *n*.

o   is a scalar variable of type default INTEGER, that holds the number of observations, *o*.

m   is a scalar variable of type default INTEGER, that holds the number of general linear constraints, *m*.

Ao   is scalar variable of type SMT_TYPE that holds the objective Jacobian matrix $\mathbf{A}_o$. The following components are used:

     Ao%type   is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense row-wise storage scheme (see Section 2.1.1) is used, the first five components of Ao%type must contain the string DENSE. By contrast, if the dense column-wise storage scheme (see Section 2.1.2) is used, the first fifteen components of Ao%type must contain the string DENSE_BY_COLUMNS. For the sparse co-ordinate scheme (see Section 2.1.3), the first ten components of Ao%type must contain the string COORDINATE, while for the sparse row-wise storage scheme (see Section 2.1.4), the first fourteen components of Ao%type must contain the string SPARSE_BY_ROWS, and for the sparse column-wise storage scheme (see Section 2.1.5), the first seventeen components of Ao%type must contain the string SPARSE_BY_COLUMNS,

         The procedure SMT_put may be used to allocate sufficient space and insert the required keyword into Ao%type. For example, if prob is of derived type CLLS_problem_type and involves an objective Jacobian we wish to store using the co-ordinate storage scheme, we may simply

---

All use is subject to the conditions of the GNU Lesser General Public License version 3.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

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

Ao%ne   is a scalar variable of type default `INTEGER`, that holds the number of entries in **A** in the sparse co-ordinate storage scheme (see Section 2.1.3). It need not be set for any of the other appropriate schemes.

Ao%val   is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that holds the values of the entries of the Jacobian matrix **A** in any of the appropriate storage schemes discussed in Section 2.1.

Ao%row   is a rank-one allocatable array of type default `INTEGER`, that holds the row indices of **A** in the sparse co-ordinate storage or sparse column-wise schemes (see Sections 2.1.3 and 2.1.5). It need not be allocated for any of the other schemes.

Ao%col   is a rank-one allocatable array variable of type default `INTEGER`, that holds the column indices of **A** in either the sparse co-ordinate (see Section 2.1.3), or the sparse row-wise (see Section 2.1.4) storage scheme. It need not be allocated for any of the other schemes.

Ao%ptr   is a rank-one allocatable array of dimension `o+1` and type default `INTEGER`, that holds the starting position of each row of **A**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.4). By contrast, for the sparse column-wise storage scheme (see Section 2.1.5), it should be of dimension `n+1` and holds the starting position of each column of **A**, as well as the total number of entries plus one, It need not be allocated when the other appropriate schemes are used.

B   is a rank-one allocatable array type default `REAL` (double precision in `GALAHAD_CLLS_double`), that should be allocated to have length `o`, and its *i*-th component filled with the value b_i for $i = 1, \ldots, o$.

A   is scalar variable of type `SMT_TYPE` that holds the constraint Jacobian matrix **A**. The following components are used:

A%type   is an allocatable array of rank one and type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense row-wise storage scheme (see Section 2.1.1) is used, the first five components of `A%type` must contain the string `DENSE`. By contrast, if the dense column-wise storage scheme (see Section 2.1.2) is used, the first fifteen components of `A%type` must contain the string `DENSE_BY_COLUMNS`. For the sparse co-ordinate scheme (see Section 2.1.3), the first ten components of `A%type` must contain the string `COORDINATE`, while for the sparse row-wise storage scheme (see Section 2.1.4), the first fourteen components of `A%type` must contain the string `SPARSE_BY_ROWS`, and for the sparse column-wise storage scheme (see Section 2.1.5), the first seventeen components of `A%type` must contain the string `SPARSE_BY_COLUMNS`,

    The procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `A%type`. For example, if `prob` is of derived type `CLLS_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', istat )
```

A%ne   is a scalar variable of type default `INTEGER`, that holds the number of entries in **A** in the sparse co-ordinate storage scheme (see Section 2.1.3). It need not be set for any of the other appropriate schemes.

A%val   is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that holds the values of the entries of the Jacobian matrix **A** in any of the appropriate 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 **A** in the sparse co-ordinate storage or sparse column-wise schemes (see Sections 2.1.3 and 2.1.5). It need not be allocated for any of the other schemes.

A%col   is a rank-one allocatable array variable of type default `INTEGER`, that holds the column indices of **A** in either the sparse co-ordinate (see Section 2.1.3), or the sparse row-wise (see Section 2.1.4) storage scheme. It need not be allocated for any of the other schemes.

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 **A**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.4). By contrast, for the sparse column-wise storage scheme (see Section 2.1.5), it should be of dimension n+1 and holds the starting position of each column of **A**, as well as the total number of entries plus one, It need not be allocated when the other appropriate schemes are used.

C_l is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_CLLS_-double), 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 $\mathbf{c}_i^l$. 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 2.4.3).

C_u is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_CLLS_-double), 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 $\mathbf{c}_i^u$. 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 2.4.3).

X_l is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_CLLS_-double), 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 $\mathbf{x}_j^l$. 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 2.4.3).

X_u is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_CLLS_-double), 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 $\mathbf{x}_j^u$. 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 2.4.3).

X is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_CLLS_-double), that holds the values **x** of the optimization variables. The j-th component of X, $j = 1, \ldots, n$, contains x_j.

C is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_CLLS_-double), that holds the values **Ax** of the constraints. The i-th component of C, $i = 1, \ldots, m$, contains $\mathbf{a}_i^T \mathbf{x} \equiv (\mathbf{A}\mathbf{x})_i$.

Y is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_CLLS_-double), that holds the values **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 (double precision in GALAHAD_CLLS_-double), that holds the values **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.

C_status is a rank-one allocatable array argument of dimension m and type default INTEGER, that indicates which of the general linear constraints are in the optimal active set. Possible values for C_status(i), i= 1,..., m, and their meanings are

    <0 the i-th general constraint is in the active set, on its lower bound,

    >0 the i-th general constraint is in the active set, on its upper bound, and

    0 the i-th general constraint is not in the active set.

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

    -1 the i-th general constraint is both independent and active on its lower bound,

    -2 the i-th general constraint is on its lower bound but linearly dependent on others,

    1 the i-th general constraint is both independent and active on its upper bound,

---

All use is subject to the conditions of the GNU Lesser General Public License version 3.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

  2   the i-th general constraint is on its upper bound but linearly dependent on others, and

  0   the i-th general constraint is not in the active set.

X_status   is a rank-one allocatable array argument of dimension n and type default INTEGER, that indicates which of the simple bound constraints are in the current active set. Possible values for X_status(j), j= 1,..., n, and their meanings are

  <0   the j-th simple bound constraint is in the active set, on its lower bound,

  >0   the j-th simple bound constraint is in the active set, on its upper bound, and

  0   the j-th simple bound constraint is not in the active set.

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

  -1   the j-th simple bound constraint is both independent and active on its lower bound,

  -2   the j-th simple bound constraint is on its lower bound but linearly dependent on others,

  1   the j-th simple bound constraint is both independent and active on its upper bound,

  2   the j-th simple bound constraint is on its upper bound but linearly dependent on others, and

  0   the j-th simple bound constraint is not in the active set.

2.4.3 The derived data type for holding control parameters

The derived data type CLLS_control_type is used to hold controlling data. Default values may be obtained by calling CLLS_initialize (see Section 2.5.1), while components may also be changed by calling CLLS_read_specfile (see Section 2.7.1). The components of CLLS_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 CLLS_solve and CLLS_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 CLLS_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 CLLS_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 CLLS_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 CLLS_solve. If stop_print is negative, printing will occur once it has been started by start_print. The default is stop_print = -1.

infeas_max   is a scalar variable of type 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.

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

---

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

0 nothing is restored.

1 the vector data **w**, **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 Jacobian matrix **A**, will be restored.

The default is restore_problem = 2.

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

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

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

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

The default is indicator_type = 3.

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

1 the residual trajectory proposed by Zhang will be used.

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

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

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

The default is arc = 1.

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

infinity is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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 = 10^{19}.

stop_abs_p and stop_rel_p are scalar variables of type default REAL (double precision in GALAHAD_CLLS_double), that hold the required absolute and relative accuracy for the primal infeasibility (see Section 4). The absolute value of each component of the primal infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_p = stop_rel_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_CLLS_double).

stop_abs_d and stop_rel_d are scalar variables of type default REAL (double precision in GALAHAD_CLLS_double), that hold the required absolute and relative accuracy for the dual infeasibility (see Section 4). The absolute value of each component of the dual infeasibility on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_d = stop_rel_d = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_CLLS_double).

stop_abs_c and stop_rel_c are scalar variables of type default REAL (double precision in GALAHAD_CLLS_double), that hold the required absolute and relative accuracy for the violation of complementary slackness (see Section 4). The absolute value of each component of the complementary slackness on exit is required to be smaller than the larger of stop_abs_p and stop_rel_p times a "typical value" for this component. The defaults are stop_abs_c = stop_rel_c = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_CLLS_double).

prfeas is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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 = 10^4.

dufeas is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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 = 10^4.

muzero is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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.

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

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

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

reduce_infeas is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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.

identical_bounds_tol is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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_CLLS_double).

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

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

indicator_tol_tapia is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double) that provides the indicator tolerance associated with the test indicator_type = 3. The default is indicator_tol-_tapia = 0.9.

cpu_time_limit is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_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.

---

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

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_CLLS is able to take special advantage of non-negativities/non-positivities, so if a single problem, or if a sequence of problems whose bound structure is known not to change, is/are to be solved, it will pay to set the variable to .FALSE.. The default is treat_zero_bounds_as_general = .FALSE..

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

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

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

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

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

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

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

symmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any symmetric linear system that might arise. Current possible choices are 'sils', 'ma27', 'ma57', 'ma77', 'ma86', 'ma97', ssids, 'pardiso' and 'wsmp', although only 'sils' and, for OMP 4.0-compliant compilers, 'ssids' are installed by default. See the documentation for the GALAHAD package SLS for further details. The default is symmetric_linear_solver = 'sils'.

---

prefix is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

`FDC_control` is a scalar variable of type `FDC_control_type` whose components are used to control any detection of linear dependencies performed by the package `GALAHAD_FDC`. See the specification sheet for the package `GALAHAD_FDC` for details, and appropriate default values.

`SLS_control` and SLS_pounce_control are scalar variables of type `SLS_control_type` whose components are used to control factorizations performed by the package `GALAHAD_SLS` for generic and pounce-specific factorizations. See the specification sheet for the package `GALAHAD_SLS` for details, and appropriate default values.

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

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

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

2.4.4 The derived data type for holding timing information

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

total is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the total CPU time spent in the package.

preprocess is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the CPU time spent preprocess the problem prior to solution.

find_dependent is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the CPU time spent detecting and removing dependent constraints prior to solution.

analyse is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the CPU time spent analysing the required matrices prior to factorization.

factorize is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the CPU time spent computing the search direction.

clock_total is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the total elapsed system clock time spent in the package.

clock_preprocess is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the elapsed system clock time spent preprocess the problem prior to solution.

clock_find_dependent is a scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that gives the elapsed system clock time spent detecting and removing dependent constraints prior to solution.

clock_analyse is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that gives the elapsed system clock time spent analysing the required matrices prior to factorization.

clock_factorize is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that gives the elapsed system clock time spent computing the search direction.

2.4.5   The derived data type for holding informational parameters

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

status is a scalar variable of type default INTEGER, that gives the exit status of the algorithm. See Section 2.6 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.

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 default INTEGER, that gives the return status from the matrix factorization.

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

factorization_real is a scalar variable of type long 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 linesearches.

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

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

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

primal_infeasibility is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that holds the norm of the violation of primal optimality (see Section 2.4.4) at the best estimate of the solution found.

dual_infeasibility is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that holds the norm of the violation of dual optimality (see Section 2.4.4) at the best estimate of the solution found.

complementary_slackness is a scalar variable of type default REAL (double precision in GALAHAD_CLLS_double), that holds the norm of the violation of complementary slackness (see Section 2.4.4) at the best estimate of the solution found.

feasible is a scalar variable of type default LOGICAL, that has the value .TRUE. if the output value of **x** satisfies the constraints, and the value .FALSE. otherwise.

time is a scalar variable of type CLLS_time_type whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

FDC_inform is a scalar variable of type FDC_inform_type whose components are used to provide information about any detection of linear dependencies performed by the package GALAHAD_FDC. See the specification sheet for the package GALAHAD_FDC for details.

SLS_inform and SLS_pounce_inform are scalar variables of type SLS_inform_type whose components are used to provide information about generic and pounce-specific factorizations performed by the package GALAHAD_SLS. See the specification sheet for the package GALAHAD_SLS for details.

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

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

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

2.4.6 The derived data type for holding problem data

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

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.7 for further features):

1. The subroutine CLLS_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 CLLS_solve is called to solve the problem.

3. The subroutine CLLS_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by CLLS_solve, at the end of the solution process.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

data is a scalar INTENT(INOUT) argument of type CLLS_data_type (see Section 2.4.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type CLLS_control_type (see Section 2.4.3). On exit, control contains default values for the components as described in Section 2.4.3. These values should only be changed after calling CLLS_initialize.

All use is subject to the conditions of the GNU Lesser General Public License version 3.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

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

2.5.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

```
CALL CLLS_solve( prob, data, control, inform[, w, regularization_weight] )
```

prob is a scalar `INTENT(INOUT)` argument of type `QPT_problem_type` (see Section 2.4.2). It is used to hold data about the problem being solved. The user must allocate all the array components, and set values for all components except `prob%C`.

The components `prob%X`, `prob%Y` and `prob%Z` must be set to initial estimates of the primal variables, **x**, Lagrange multipliers, **y**, for the general constraints and dual variables for the bound constraints, **z**, respectively. Inappropriate initial values will be altered, so the user should not be overly concerned if suitable values are not apparent, and may be content with merely setting `prob%X=0.0`, `prob%Y=0.0` and `prob%Z=0.0`.

On exit, the components `prob%X` , `prob%C`, `prob%Y`, and `prob%Z` will contain the best estimates of the primal variables **x**, the linear constraints **Ax**, Lagrange multipliers, **y**, for the general constraints and dual variables for the bound constraints **z**, respectively. **Restrictions:** $prob\%n > 0$, $prob\%m \geq 0$, `prob%Ao_type` and `prob%A_type` $\in \{$'DENSE', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS' $\}$, 'SPARSE_BY_COLUMNS' $\}$.

data is a scalar `INTENT(INOUT)` argument of type `CLLS_data_type` (see Section 2.4.6). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `CLLS_initialize`.

control is a scalar `INTENT(IN)` argument of type `CLLS_control_type` (see Section 2.4.3). Default values may be assigned by calling `CLLS_initialize` prior to the first call to `CLLS_solve`.

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

W is an `OPTIONAL` rank-one allocatable array of dimension o and type default `REAL` (double precision in `GALAHAD_CLLS_double`), that holds the positive residual weights, *w*. If it is `PRESENT`, its *i*-th component must contain the value w_i for $i = 1, \ldots, o$. If all the weights are one, `W` need not be `PRESENT`.

regularization_weight is an `OPTIONAL` scalar variable of type default `REAL` (double precision in `GALAHAD_CLLS_double`), that holds the regularization weight, $\sigma \geq 0$. It need not be `PRESENT` if $\sigma = 0$.

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

inform is a scalar `INTENT(OUT)` argument of type `CLLS_inform_type` exactly as for `CLLS_solve`. Only the component `status` will be set on exit, and a successful call to `CLLS_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.6.

2.6   Warning and error messages

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

-1. An allocation error occurred. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-3. One of the restrictions `prob%n` > 0 , `prob%o` > 0, `prob%m` ≥ 0 the requirement that `prob%A_type` contain its relevant string `'DENSE'`, `'DENSE_BY_COLUMNS'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'SPARSE_BY_COLUMNS'`, has been violated.

-4. The bound constraints are inconsistent.

-5. The constraints appear to have no feasible point.

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

-9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.

-10. The factorization failed; the return status from the factorization package is given in the component `inform%factor_status`.

-16. The problem is so ill-conditioned that further progress is impossible.

-17. The step is too small to make further impact.

-18. Too many iterations have been performed. This may happen if `control%maxit` is too small, but may also be symptomatic of a badly scaled problem.

-19. The elapsed CPU or system clock time limit has been reached. This may happen if either `control%cpu_time_limit` or `control%clock_time_limit` is too small, but may also be symptomatic of a badly scaled problem.

2.7   Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `CLLS_control_type` (see Section 2.4.3), by reading an appropriate data specification file using the subroutine `CLLS_read_specfile`. This facility is useful as it allows a user to change `CLLS` control parameters without editing and recompiling programs that call `CLLS`.

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

All use is subject to the conditions of the GNU Lesser General Public License version 3.
See `http://galahad.rl.ac.uk/galahad-www/cou.html` **for full details.**

```
( .. lines ignored by CLLS_read_specfile .. )
  BEGIN CLLS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by CLLS_read_specfile .. )
```

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

and

```
  END CLLS SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN CLLS" 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 CLLS_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 CLLS_read_specfile.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
  CALL CLLS_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type CLLS_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling CLLS_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.4.3) of control that each affects are given in Table 2.1.

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.

2.8 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control%out. If control%print_level = 1, a single line of output will be produced for each iteration of the process. For the initial-feasible-point phase, this will include values of the current primal and dual infeasibility, and violation of complementary slackness, the feasibility-phase objective value, the current steplength, the value of the barrier

---

All use is subject to the conditions of the GNU Lesser General Public License version 3.
See http://galahad.rl.ac.uk/galahad-www/cou.html **for full details.**

command	component of `control`	value type
`error-printout-device`	`%error`	integer
`printout-device`	`%out`	integer
`print-level`	`%print_level`	integer
`maximum-number-of-iterations`	`%maxit`	integer
`start-print`	`%start_print`	integer
`stop-print`	`%stop_print`	integer
`indicator-type-used`	`%indicator_type`	integer
`arc-used`	`%arc`	integer
`series-order`	`%series_order`	integer
`infinity-value`	`%infinity`	real
`identical-bounds-tolerance`	`%identical_bounds_tol`	real
`absolute-primal-accuracy`	`%stop_abs_p`	real
`relative-primal-accuracy`	`%stop_rel_p`	real
`absolute-dual-accuracy`	`%stop_abs_d`	real
`relative-dual-accuracy`	`%stop_rel_d`	real
`absolute-complementary-slackness-accuracy`	`%stop_abs_c`	real
`relative-complementary-slackness-accuracy`	`%stop_rel_c`	real
`perturb-hessian-by`	`%perturb_h`	real
`initial-barrier-parameter`	`%muzero`	real
`feasibility-vs-complementarity-weight`	`%tao`	real
`balance-complementarity-factor`	`%gamma_c`	real
`balance-feasibility-factor`	`%gamma_f`	real
`poor-iteration-tolerance`	`%reduce_infeas`	real
`minimum-objective-before-unbounded`	`%obj_unbounded`	real
`minimum-potential-before-unbounded`	`%potential_unbounded`	real
`identical-bounds-tolerance`	`%identical_bounds_tol`	real
`primal-indicator-tolerance`	`%indicator_tol_p`	real
`primal-dual-indicator-tolerance`	`%indicator_tol_pd`	real
`tapia-indicator-tolerance`	`%indicator_tol_tapia`	real
`maximum-cpu-time-limit`	`%cpu_time_limit`	real
`maximum-clock-time-limit`	`%clock_time_limit`	real
`remove-linear-dependencies`	`%remove_dependencies`	logical
`treat-zero-bounds-as-general`	`%treat_zero_bounds_as_general`	logical
`just-find-feasible-point`	`%just_feasible`	logical
`puiseux-series`	`%puiseux`	logical
`try-every-order-of-series`	`%every_order`	logical
`cross-over-solution`	`%crossover`	logical
`space-critical`	`%space_critical`	logical
`deallocate-error-fatal`	`%deallocate_error_fatal`	logical
`symmetric-linear-equation-solver`	`%symmetric_linear_solver`	character
`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: `CLLS_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_TOOLS`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_LSP`, `GALAHAD_QPD`, `GALAHAD_FIT`, `GALAHAD_ROOTS`, `GALAHAD_NORMS`, `GALAHAD_SLS`, `GALAHAD_CRO` 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%o > 0, prob%m ≥ 0, prob%Ao_type` and `prob%A_type` $\in \{'DENSE', 'DENSE\_BY\_COLUMNS',$ $'COORDINATE', 'SPARSE\_BY\_ROWS', 'SPARSE\_BY\_COLUMNS', \}$.

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

$$\mathbf{A}\mathbf{x} = \mathbf{c} \tag{4.1}$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \ \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \tag{4.2}$$

the dual optimality conditions

$$\mathbf{A}_o^T W (\mathbf{A}_o \mathbf{x} - \mathbf{b}) + \sigma \mathbf{x} = \mathbf{A}^T \mathbf{y} + \mathbf{z}, \tag{4.3}$$

where

$$\mathbf{y} = \mathbf{y}^l + \mathbf{y}^u, \ \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \ \mathbf{y}^l \geq 0, \ \mathbf{y}^u \leq 0, \ \mathbf{z}^l \geq 0 \ \text{and} \ \mathbf{z}^u \leq 0, \tag{4.4}$$

and the complementary slackness conditions

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

where the diagonal matrix $\mathbf{W}$ has diagonal entries $w_i, i = 1, \ldots, o$, 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 component-wise.

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 method aims at each stage to reduce the overall violation of (4.1), (4.3) and (4.5), rather than reducing each of the terms individually. Given an estimate $\mathbf{v} = (\mathbf{x}, \mathbf{c}, \mathbf{y}, \mathbf{y}^l, \mathbf{y}^u, \mathbf{z}, \mathbf{z}^l, \mathbf{z}^u)$ of the primal-dual variables, a correction $\Delta \mathbf{v} = \Delta(\mathbf{x}, \mathbf{c}, \mathbf{y}, \mathbf{y}^l, \mathbf{y}^u, \mathbf{z}, \mathbf{z}^l, \mathbf{z}^u)$ is obtained by solving a suitable linear system of Newton equations for the nonlinear systems (4.1), (4.3) and a parameterized "residual trajectory" perturbation of (4.5); residual trajectories proposed by Zhang (1994) and Zhao and Sun (1999) are possibilities. An improved estimate $\mathbf{v} + \alpha \Delta \mathbf{v}$ is then used, where the step-size α is chosen as close to 1.0 as possible while ensuring both that (4.2) and (4.4) continue to hold and that the individual components which make up the complementary slackness (4.5) do not deviate too significantly from their average value. The parameter that controls the perturbation of (4.5) is ultimately driven to zero.

The Newton equations are solved by applying the GALAHAD matrix factorization package `GALAHAD_SLS`. Optionally, the problem may be pre-processed temporarily to eliminate dependent constraints using the package `GALAHAD_FDC`. This may improve the performance of the subsequent iteration.

References:

The basic algorithm is a generalisation of those of

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

and

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

with many enhancements described by

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

5 EXAMPLE OF USE

Suppose we wish to minimize

$$\tfrac{1}{2} \left\| \begin{pmatrix} x_1 + x_2 - 2 \\ x_2 + x_3 - 2 \\ x_1 + x_3 - 3 \\ 2(x_2 - 1) \end{pmatrix} \right\|^2$$

subject to the general linear constraints $1 \le 2x_1 + x_2 \le 2$ and $x_2 + x_3 = 2$, and simple bounds $-1 \le x_1 \le 1$ and $x_3 \le 2$. Then, on writing the data for this problem as

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & \\ & 1 & 1 \\ 1 & & 1 \\ & 1 & \end{pmatrix}, \ \mathbf{b} = \begin{pmatrix} 2 \\ 2 \\ 3 \\ 1 \end{pmatrix}, \ \mathbf{w} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \end{pmatrix}, \ \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ -\infty \end{pmatrix}, \ \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 2 \end{pmatrix}$$

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 4.2 - 2023-08-31 AT 10:15 GMT.
  PROGRAM GALAHAD_CLLS_EXAMPLE
```

```
    USE GALAHAD_CLLS_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 ( CLLS_data_type ) :: data
    TYPE ( CLLS_control_type ) :: control
    TYPE ( CLLS_inform_type ) :: inform
    INTEGER :: s
    INTEGER, PARAMETER :: n = 3, o = 4, m = 2, ao_ne = 7, a_ne = 4
    REAL ( KIND = wp ), DIMENSION( o ) :: W
! start problem data
    ALLOCATE( p%B( o ), p%R( o ) )
    ALLOCATE( 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 ) )
    ALLOCATE( p%C_status( m ), p%X_status( n ) )
    p%new_problem_structure = .TRUE.       ! new structure
    p%n = n ; p%o = o ; p%m = m ;          ! dimensions
    p%B = (/ 2.0_wp, 2.0_wp, 3.0_wp, 1.0_wp /) ! observations
    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
    W = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)   ! weights
!  sparse co-ordinate storage format
    CALL SMT_put( p%Ao%type, 'COORDINATE', s ) ! Specify co-ordinate
    CALL SMT_put( p%A%type, 'COORDINATE', s )  ! storage for Ao and A
    ALLOCATE( p%Ao%row( ao_ne ), p%Ao%col( ao_ne ), p%Ao%val( ao_ne ) )
    ALLOCATE( p%A%col( a_ne ), p%A%val( a_ne ), p%A%row( a_ne ) )
    p%Ao%row = (/ 1, 1, 2, 2, 3, 3, 4 /) ; p%Ao%ne = ao_ne  ! Ao
    p%Ao%col = (/ 1, 2, 2, 3, 1, 3, 2 /)
    p%Ao%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)
    p%A%row = (/ 1, 1, 2, 2 /) ; p%A%ne = a_ne  ! A
    p%A%col = (/ 1, 2, 2, 3 /)
    p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)
! problem data complete
    CALL CLLS_initialize( data, control, inform ) ! Initialize control parameters
    control%symmetric_linear_solver = 'sytr '
    control%FDC_control%symmetric_linear_solver = 'sytr '
    control%infinity = infinity            ! Set infinity
    CALL CLLS_solve( p, data, control, inform,                    &
                W = W )                    ! Solve
    IF ( inform%status == 0 ) THEN         ! Successful return
      WRITE( 6, "( ' CLLS: ', I0, ' iterations ', /,              &
     &    ' Optimal objective value =',                           &
     &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )  &
      inform%iter, inform%obj, p%X
    ELSE                                   ! Error returns
      WRITE( 6, "( ' CLLS_solve exit status = ', I6 ) " ) inform%status
    END IF
    CALL CLLS_terminate( data, control, inform )  !  delete internal workspace
    END PROGRAM GALAHAD_CLLS_EXAMPLE
```

This produces the following output:

```
 CLLS: 9 iterations
 Optimal objective value =  6.7857E-01
 Optimal solution =   7.8571E-01  4.2857E-01  1.5714E+00
```

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

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

by

```
! sparse row-wise storage format
  CALL SMT_put( p%Ao%type, 'SPARSE_BY_ROWS', s ) ! Specify sparse row
  CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS', s )  ! storage for Ao and A
  ALLOCATE( p%Ao%col( ao_ne ), p%Ao%ptr( o + 1 ), p%Ao%val( ao_ne ) )
  ALLOCATE( p%A%col( a_ne ), p%A%ptr( m + 1 ), p%A%val( a_ne ) )
  p%Ao%col = (/ 1, 2, 2, 3, 1, 3, 2 /)  ! Jacobian Ao
  p%Ao%ptr = (/ 1, 3, 5, 7, 8 /)
  p%Ao%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)
  p%A%col = (/ 1, 2, 2, 3 /) ! Jacobian A
  p%A%ptr = (/ 1, 3, 5 /)
  p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)
! problem data complete
```

or using a dense storage format with the replacement lines

```
!  dense storage format
  CALL SMT_put( p%Ao%type, 'DENSE', s ) ! Specify dense
  CALL SMT_put( p%A%type, 'DENSE', s )  ! storage for Ao and A
  ALLOCATE( p%Ao%val( o * n ), p%A%val( n * m ) )
  p%Ao%val = (/ 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp,  &
                1.0_wp, 0.0_wp, 1.0_wp, 0.0_wp, 1.0_wp, 0.0_wp /) ! Ao
  p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! A
! problem data complete
```

respectively.