GALAHAD LPA

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package uses the **simplex method** to solve the **linear programming problem**

$$\text{minimize } \ell(\mathbf{x}) = \mathbf{g}^T\mathbf{x} + f \tag{1.1}$$

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 vectors $\mathbf{g}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the scalar f 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 $\mathbf{A}$ of vectors $\mathbf{a}_i$.

N.B. The package is simply a sophisticated interface to the HSL package `LA04`, and requires that a user has obtained the latter. **LA04 is not included in GALAHAD,** but is available without charge to recognised academics, see `http://www.hsl.rl.ac.uk/catalogue/la04.html`. If `LA04` is unavailable, the GALAHAD interior-point linear programming package `LPB` is an alternative.

ATTRIBUTES — Versions: `GALAHAD_LPA_single`, `GALAHAD_LPA_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_TOOLS`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_QPD`. **Date:** October 2018. **Origin:** N. I. M. Gould and J. K. Reid, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available using both single and double precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
        USE GALAHAD_LPA_single
```

with the obvious substitution `GALAHAD_LPA_double`, `GALAHAD_LPA_single_64` and `GALAHAD_LPA_double_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `LPA_time_type`, `LPA_control_type`, `LPA_inform_type` and `LPA_data_type` (Section 2.3) and the subroutines `LPA_initialize`, `LPA_solve`, `LPA_terminate`, (Section 2.4) and `LPA_read_specfile` (Section 2.6) 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 $\mathbf{a}_i^T$, $i = 1,\ldots,m$, 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 $i = 1,\ldots,m$, $j = 1,\ldots,n$.

2.1.2 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.

2.1.3 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 **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), ...,`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 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions and `DOUBLE PRECISION` for the double precision cases, and correspond to `rp_ = real32` and `rp_ = real64`, respectively, as supplied by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.3 The derived data types

Ten derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrix **A**. The components of `SMT_TYPE` used here are:

`m` is a scalar component of type `INTEGER(ip_)`, that holds the number of rows in the matrix.

`n` is a scalar component of type `INTEGER(ip_)`, that holds the number of columns in the matrix.

`ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.3.2).

`val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

`row` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold the row indices of the entries. (see §2.1.2).

`col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may the column indices of the entries (see §2.1.2–2.1.3).

`ptr` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `m + 1`, that may hold the pointers to the first entry in each row (see §2.1.3).

2.3.2 The derived data type for holding the problem

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

`n` is a scalar variable of type `INTEGER(ip_)`, that holds the number of optimization variables, n.

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

`gradient_kind` is a scalar variable of type `INTEGER(ip_)`, that is used to indicate whether the components of the gradient $\mathbf{g}$ have special or general values. Possible values for `gradient_kind` are:

- `0` In this case, $\mathbf{g} = 0$.
- `1` In this case, $g_j = 1$ for $j = 1, \ldots, n$.
- $\neq$ `0,1` In this case, general values of $\mathbf{g}$ will be used, and will be provided in the component `G` (see below).

`G` is a rank-one allocatable array type `REAL(rp_)`, that should be allocated to have length `n`, and its j-th component filled with the value g_j for $j = 1, \ldots, n$, whenever `gradient_kind` $\neq$ 0,1. If `gradient_kind` = 0, 1, `G` need not be allocated.

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

`A` is scalar variable of type `SMT_TYPE` that holds the Jacobian matrix $\mathbf{A}$ when it is available explicitly. 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`.

Just as for `H%type` above, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `A%type`. Once again, if `prob` is of derived type `LPA_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 `INTEGER(ip_)`, 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 appropriate schemes.

`A%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the Jacobian matrix $\mathbf{A}$ in any of the appropriate storage schemes discussed in Section 2.1.

`A%row` is a rank-one allocatable array of type `INTEGER(ip_)`, 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 appropriate schemes.

`A%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, 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 `INTEGER(ip_)`, 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 appropriate schemes are used.

`C_l` is a rank-one allocatable array of dimension `m` and type `REAL(rp_)`, that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i-th component of `C_l`, $i = 1, \ldots, m$, contains $\mathbf{c}_i^l$. 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 2.3.3).

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains $\mathbf{c}_i^u$. 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 2.3.3).

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}_j^l$. 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 2.3.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}_j^u$. 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 2.3.3).

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

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

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

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

2.3.3 The derived data type for holding control parameters

The derived data type LPA_control_type is used to hold controlling data. Default values may be obtained by calling LPA_initialize (see Section 2.4.1), while components may also be changed by calling LPA_read_specfile (see Section 2.6.1). The components of LPA_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in LPA_solve and LPA_terminate is suppressed if error ≤ 0. The default is error = 6.

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

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

start_print is a scalar variable of type INTEGER(ip_), that specifies the first iteration for which printing will occur in LPA_solve. If start_print is negative, printing will occur from the outset. The default is start_print = -1.

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

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

max_iterative_refinements is a scalar variable of type default INTEGER(ip_), that holds the maximum number of iterative refinements per linear system solved that may be attempted. The default is max_iterative_refinements = 0.

min_real_factor_size is a scalar variable of type INTEGER(ip_), that specifies the amount of real storage that will initially be allocated for the factors and other data. The default is min_real_factor_size = 10000, and this default is used if min_real_factor_size < 1.

min_integer_factor_size is a scalar variable of type INTEGER(ip_), that specifies the amount of integer storage that will initially be allocated for the factors and other data. The default is min_integer_factor_size = 20000, and this default is used if min_integer_factor_size < 1.

random_number_seed is a scalar variable of type INTEGER(ip_), that holds the initial seed used by the random-number generator. The default is random_number_seed = 0.

infinity is a scalar variable of type REAL(rp_), that is used to specify which constraint bounds are infinite. Any bound larger than infinity in modulus will be regarded as infinite. The default is infinity = 10^{19}.

tol_data is a scalar variable of type REAL(rp_), that hold the maximum violation that a constraint is allowed and be still considered to be feasible. The default is tol_data = $u^{2/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPA_double).

feas_tol is a scalar variable of type REAL(rp_), that hold the maximum violation that a constraint is allowed and be still considered to be feasible. The default is feas_tol = $u^{2/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPA_double).

relative_pivot_tolerance is a scalar variable of type REAL(rp_), that holds the relative pivot tolerance that is used to control the stability of the factorizations of the basis matrices that arise as the iteration proceeds. The default is relative_pivot_tolerance = 0.1.

growth_limit is a scalar variable of type REAL(rp_), that specifies the maximum growth in the entries of the factors of a basis matrix that will be tolerated before a refactorization is required. The default is growth_limit = $1/u^{2/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPA_double).

zero_tolerance is a scalar variable of type REAL(rp_), that controls which small entries are to be ignored during the factorization of a basis matrix. Any entry smaller in absolute value than zero_tolerance will be treated as zero; as a consequence when zero_tolerance > 0, the factors produced will be of a perturbation of order zero_tolerance. The default is zero_tolerance = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPA_double).

change_tolerance is a scalar variable of type REAL(rp_), that provides a tolerance whose purpose is to assess when a change to a solution component may be neglected. Specifically any change that is smaller than this tolerence times the largest change may be considered to be zero. The default is change_tolerance = $u^{2/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LPA_double).

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

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

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

`scale` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the problem data is to be scaled prior to solution in an attempt to make the process faster and more accurate. The default is `scale = .FALSE.`.

`warm_start` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if an initial guess to the optimal status of the constraints and simple bounds is to be provided, and `.FALSE.` if the algorithm is to make its own initial choice. The default is `warm_start = .FALSE.`.

`dual` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the simplex method is to be applied to the dual of the linear program, and `.FALSE.` if the primal problem should be solved. The default is `dual = .FALSE.`.

`steepest_edge` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if steepest-edge weights are to be used to try to improve the choice of variable that will leave the basis at each iteration of the simplex method, and `.FALSE.` if weights of one are used. The default is `steepest_edge = .TRUE.`.

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

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

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

2.3.4 The derived data type for holding timing information

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

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

`preprocess` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent preprocess the problem prior to solution.

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

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

`clock_solve` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent computing the search direction.

2.3.5 The derived data type for holding informational parameters

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

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

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

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

`la04_job` is a scalar variable of type `INTEGER(ip_)`, that gives the value of the argument `job` returned from `LA04`. See the specification sheet, `http://www.hsl.rl.ac.uk/specs/la04.pdf`, for the package `LA04` for details.

`la04_job_info` is a scalar variable of type `INTEGER(ip_)`, that gives the value of the argument `RINFO(35)` returned from an unsuccessful call to `LA04`. See the specification sheet, `http://www.hsl.rl.ac.uk/specs/la04.pdf`, for the package `LA04` for details.

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

`primal_infeasibility` is a scalar variable of type `REAL(rp_)`, that holds the norm of the violation of primal optimality (see Section 2.3.4) 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 $\mathbf{x}$ satisfies the constraints, and the value `.FALSE.` otherwise.

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

2.3.6 The derived data type for holding problem data

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

2.4 Argument lists and calling sequences

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

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

We use square brackets `[ ]` to indicate `OPTIONAL` arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

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

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

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

`inform` is a scalar `INTENT(OUT)` argument of type `LPA_inform_type` (see Section 2.3.5). A successful call to `LPA_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.5.

2.4.2 The linear programming subroutine

The linear programming solution algorithm is called as follows:

```
CALL LPA_solve( prob, data, control, inform[, C_stat, X_stat] )
```

`prob` is a scalar `INTENT(INOUT)` argument of type `QPT_problem_type` (see Section 2.3.2). It is used to hold data about the problem being solved. The user must allocate all the array components, and set values for all components except `prob%X`, `prob%C`, `prob%Y` and `prob%Z`

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

`data` is a scalar `INTENT(INOUT)` argument of type `LPA_data_type` (see Section 2.3.6). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `LPA_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `LPA_control_type` (see Section 2.3.3). Default values may be assigned by calling `LPA_initialize` prior to the first call to `LPA_solve`.

`inform` is a scalar `INTENT(INOUT)` argument of type `LPA_inform_type` (see Section 2.3.5). A successful call to `LPA_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.5.

`C_stat` is an `OPTIONAL` rank-one `INTENT(OUT)` array argument of dimension `p%m` and type `INTEGER(ip_)`, that indicates which of the general linear constraints are in the optimal active set. Possible values for `C_stat(i)`, `i`$= 1,\ldots,$ `p%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.

If a warm start is to be performed (see `control%wam_start`), `C_stat` must be provided and set as above.

`X_stat` is an `OPTIONAL` rank-one `INTENT(OUT)` array argument of dimension `p%n` and type `INTEGER(ip_)`, that indicates which of the simple bound constraints are in the current active set. Possible values for `X_stat(j)`, `j`$= 1,\ldots,$ `p%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.

If a warm start is to be performed (see control%wam_start), X_stat must be provided and set as above.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type LPA_control_type exactly as for LPA_solve.

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

2.5 Warning and error messages

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

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

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

-3. One of the restrictions prob%n > 0 , prob%m ≥ 0 or the requirement that prob%A_type contain its relevant string 'DENSE', 'COORDINATE' or 'SPARSE_BY_ROWS', 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.

-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.

-29. The HSL solver LA04 is unavailable.

2.6 Further features

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

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

```
( .. lines ignored by LPA_read_specfile .. )
  BEGIN LPA
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by LPA_read_specfile .. )
```

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

and

```
END LPA SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN LPA" 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 LPA_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 LPA_read_specfile.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL LPA_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type LPA_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling LPA_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of control that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
maximum-number-of-iterations	%maxit	integer
max-iterative-refinements	%max_iterative_refinements	integer
minimum-real-factor-size	%min_real_factor_size	integer
minimum-integer-factor-size	%min_integer_factor_size	integer
random-number-seed	%random_number_seed	integer
infinity-value	%infinity	real
tolerable-relative-data-perturbation	% tol_data	real
feasibility-tolerance	%feas_tol	real
relative-pivot-tolerance	%relative_pivot_tolerance	real
growth-limit-tolerance	% growth_limit	real
zero-basis-entry-tolerance	%zero_tolerance	real
change-tolerance	%change_tolerance	real
identical-bounds-tolerance	%identical_bounds_tol	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
scale-problem-data	%remove_dependencies	logical
warm-startl	%warm_start	logical
solve-dual	%dual	logical
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

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

2.7 Information printed

If control%print_level = 1, a summary of the return status of the algorithm will be printed on unit control%out. If control%print_level $\geq$ 2, information about the progress of the algorithm will be given. At each iteration two lines of output are produced showing :

(i) The iteration number.

(ii) The length LENL of the **L** part of the factorization of the basis matrix.

(iii) The length LENU of the **U** part of the factorization of the basis matrix.

(iv) The variable, `JIN`, entering the basis.

(v) The variable, `JOUT`, leaving the basis.

(vi) The number, `NINF`, of infeasibilities.

(vii) The value of the objective function (or the sum of the infeasibilities if the phase-one problem is being solved).

(viii) The growth parameter, `G`.

(ix) The number, `NCP`, of compresses performed when updating the factors of the basis matrix.

(x) The reduced cost, `Z`, of the incoming variable.

(xi) The recurred approximation, `APPROX Z`, to this reduced cost.

(xii) The steepest-edge weight `GAMMA` used to choose the incoming variable.

(xiii) The recurred approximation, `APPROX GAMMA`, to this weight.

(xiv) The time taken so far.

(xv) The value of the pivot.

When the run terminates, a message to this effect is printed and the value of the objective function is output. If `control%print_level` ≥ 100 this output will be increased to provide detailed debuging information.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `LPA_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT` and `GALAHAD_QPD`, and the HSL package `LA04`.

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` $\in$ {`'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

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

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

$$\mathbf{g} = \mathbf{A}^T\mathbf{y} + \mathbf{z} \tag{4.3}$$

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 vectors **y** and **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.

The so-called dual to this problem is another linear program

$$-\text{minimize } \mathbf{c}^{lT} \mathbf{y}^l + \mathbf{c}^{uT} \mathbf{y}^u + \mathbf{x}^{lT} \mathbf{z}^l + \mathbf{x}^{uT} \mathbf{z}^u + f \text{ subject to the constraints (4.3) and (4.4)}$$

that uses the same data. The solution to the two problems, it is exists, is the same, but if one is infeasible, the other is unbounded. It can be more efficient to solve the dual, particularly if m is much larger than n.

The bulk of the work is peformed by the HSL package `LA04`. The main subbroutine from this package requires that the input problem be transformed into the "standard form"

$$\begin{array}{rl} \text{minimize} & \mathbf{g}'^T \mathbf{x}' \\ \text{subject to} & \mathbf{A}'\mathbf{x}' = \mathbf{b} \\ & l_i \leq x'_i \leq u_i, \;\; (i \leq k) \\ \text{and} & x'_l \geq 0, \;\; (i \geq l) \end{array} \tag{4.6}$$

by introducing slack an surpulus variables, reordering and removing fixed variables and free constraints. The resulting problem involves n' unknowns and m' general constraints. In order to deal with the possibility that the general constraints are inconsistent or not of full rank, `LA04` introduces additional "artifical" variables **v**, replaces the constraints of (4.6) by

$$\mathbf{A}'\mathbf{x}' + \mathbf{v} = \mathbf{b} \tag{4.7}$$

and gradually encourages **v** to zero as a first solution phase.

Once a selection of m' independent (*non-basic*) variables is made, the constraints (4.7) determine the remaining m' dependent (*basic*) variables. The **simplex method** is a scheme for systematically adjusting the choice of basic and non-basic variables until a set which defines an optimal solution of (4.6) is obtained. Each iteration of the simplex method requires the solution of a number of sets of linear equations whose coefficient matrix is the *basis* matrix **B**, made up of the columns of $[\mathbf{A}' \;\; \mathbf{I}]$ corresponding to the basic variables, or its transpose $\mathbf{B}^T$. As the basis matrices for consecutive iterations are closely related, it is normally advantageous to update (rather than recompute) their factorizations as the computation proceeds. If an initial basis is not provided by the user, a set of basic variables which provide a (permuted) triangular basis matrix is found by the simple crash algorithm of Gould and Reid (1989), and initial steepest-edge weights are calculated.

Phases one (finding a feasible solution) and two (solving (4.6)) of the simplex method are applied, as appropriate, with the choice of entering variable as described by Goldfarb and Reid (1977) and the choice of leaving variable as proposed by Harris (1973). Refactorizations of the basis matrix are performed whenever doing so will reduce the average iteration time or there is insufficient memory for its factors. The reduced cost for the entering variable is computed afresh. If it is found to be of a different sign from the recurred value or more than 10% different in magnitude, a fresh computation of all the reduced costs is performed. Details of the factorization and updating procedures are given by Reid (1982). Iterative refinement is encouraged for the basic solution and for the reduced costs after each factorization of the basis matrix and when they are recomputed at the end of phase 1.

References:

D. Goldfarb and J. K. Reid (1977). "A practicable steepest-edge simplex algorithm". Mathematical Programming **12** 361-371.

N. I. M. Gould and J. K. Reid (1989). "New crash procedures for large systems of linear constraints". Mathematical

Programming **45** 475-501.

P. M. J. Harris (1973). “Pivot selection methods of the Devex LP code”. Mathematical Programming **5** 1-28.

J. K. Reid (1982). “A sparsity-exploiting variant of the Bartels-Golub decomposition for linear-programming bases”. Mathematical Programming **24** 55-69.

5 EXAMPLE OF USE

Suppose we wish to minimize x_1+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 $-1 \leq x_1 \leq 1$, $x_2 \geq 3$ and $x_3 \leq 2$. Then, on writing the data for this problem as $f=1$,

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

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 3.1 - 07/10/2018 AT 12:05 GMT.
   PROGRAM GALAHAD_LPA_EXAMPLE
   USE GALAHAD_LPA_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 ( LPA_data_type ) :: data
   TYPE ( LPA_control_type ) :: control
   TYPE ( LPA_inform_type ) :: inform
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: C_stat, X_stat
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, m = 2, a_ne = 4
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ), p%X( n ), p%Z( n ), X_stat( n ) )
   ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ), p%Y( m ), C_stat( m ) )
   p%new_problem_structure = .TRUE.           ! new structure
   p%n = n ; p%m = m ; p%f = 1.0_wp           ! dimensions & objective constant
   p%G = (/ 1.0_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, 3.0_wp, - 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 LPA_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = infinity                ! Set infinity
! Solve the problem
```

```
   CALL LPA_solve( p, data, control, inform, C_stat = C_stat, X_stat = X_stat )
   IF ( inform%status == 0 ) THEN              !  Successful return
     WRITE( 6, "( ' LPA: ', I0, ' iterations. Optimal objective value =',      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, p%X
   ELSE                                       !  Error returns
     WRITE( 6, "( ' LPA_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL LPA_terminate( data, control, inform )  !  delete internal workspace
   DEALLOCATE( p%G, p%X_l, p%X_u, p%X, p%Z, X_stat )
   DEALLOCATE( p%C, p%C_l, p%C_u, p%Y, C_stat )
   END PROGRAM GALAHAD_LPA_EXAMPLE
```

This produces the following output:

```
 LPA: 2 iterations. Optimal objective value =  6.0000E+00
 Optimal solution =  -1.0000E+00  3.0000E+00 -1.0000E+00
```

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

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

by

```
! sparse row-wise storage format
   CALL SMT_put( p%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.