GALAHAD LSP

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package reorders to a standard form the variables and constraints for the **linearly-constrained linear least-squares problem**

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

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 o by n matrix $\mathbf{A}_o$, and the vectors $\mathbf{b}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ are given. Full advantage is taken of any zero coefficients in the matrix $\mathbf{A}_o$ or the vectors $\mathbf{a}_i$. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite.

The variables are rordered so that any free variables (ie, those without bounds) occur first, followed respectively by non-negativities (i.e., those for which the only bounds are that $x_j \geq 0$), lower-bounded variables (i.e., those for which the only bounds are that $x_j \geq x_j^l \neq 0$), range-bounded variables (i.e., those for which the bounds satisfy $-\infty < x_j^l < x_j^u < \infty$) upper-bounded variables (i.e., those for which the only bounds are that $x_j \leq x_j^u \neq 0$), and finally non-positivities (i.e., those for which the only bounds are that $x_j \leq 0$). Fixed variables will be removed.

The constraints are reordered so that equality constraints (i.e., those for which $c_i^l = c_i^u$) occur first, followed respectively by those which are lower-bounded (i.e., those for which the only bounds are that $\mathbf{a}_i^T\mathbf{x} \geq c_i^l$), those which have ranges (i.e., those for which the bounds satisfy $-\infty < c_j^l < c_j^u < \infty$), and finally those which are upper-bounded (i.e., those for which the only bounds are that $\mathbf{a}_i^T\mathbf{x} \leq c_i^u$). Free constraints, that is those for which $c_i^l = -\infty$ and $c_i^u = \infty$, are removed.

Procedures are provided to detetmine the required ordering, to reorder the problem to standard form, and to recover the problem, or perhaps just the values of the original variables, once it has been converted to standard form.

The derived type is also capable of supporting *parametric* problems, in which an additional objective term $\theta\delta\mathbf{b}$ is added to $\mathbf{b}$, and the trajectory of solution are required for all $0 \leq \theta \leq \theta_{\max}$ for which

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

and

$$x_j^l + \theta x_j^l \leq x_j \leq x_j^u + \delta x_j^u, \quad j = 1,\ldots,n.$$

It is anticipated that this module will principally be used as a pre- and post-processing tool for other GALAHAD packages.

ATTRIBUTES — Versions: `GALAHAD_LSP_single`, `GALAHAD_LSP_double`. **Uses:** `GALAHAD_SYMBOLS`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_SORT`. **Date:** August 2022. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

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_LSP_single
```

with the obvious substitution GALAHAD_LSP_double, GALAHAD_LSP_single_64 and GALAHAD_LSP_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, QPT_dimensions_type, LSP_control_type, LSP_inform_type and LSP_map_type (Section 2.3) and the subroutines LSP_initialize, LSP_reorder, LSP_apply, LSP_get_values, LSP_restore and LSP_terminate (Section 2.4) must be renamed on one of the USE statements.

2.1 Matrix storage formats

Both the objective Jacobian $\mathbf{A}_o$ 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. Here we refer to $\mathbf{A}$, but identical formats apply to $\mathbf{A}_o$ (with the index o replacing m as necessary).

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 $i = 1,\ldots,m$, $j = 1,\ldots,n$.

2.1.2 Dense row-wise storage format

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

Here only the nonzero entries are stored, but by contrast they are ordered so that those in column j appear directly before those in column $j+1$. For the j-th column of $\mathbf{A}$, 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), ...,A%ptr $(j+1)-1$ of the integer array A%row, and real array A%val, respectively.

For sparse matrices, the later two schemes almost always requires less storage than its co-ordinate one.

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

Six derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrices $\mathbf{A}_o$ and $\mathbf{A}$. The components of `SMT_TYPE` used here for $\mathbf{A}$ 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 either holds the number of matrix entries or is used to flag the storage scheme used.

`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. Any duplicated entries that appear in the sparse co-ordinate, row-wise or column-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 §2.1.3 and 2.1.5).

`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 §2.1.3 and 2.1.4).

`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 in a row-wise scheme (see §2.1.4). For a column-wise scheme (see §2.1.5) it must be of dimension at least `m + 1`, that may hold the pointers to the first entry in each column.

For the matrix $\mathbf{A}_o$, `m` will hold the row dimension o, and `ptr` will be of length $o+1$ in a row-wise storage scheme.

2.3.2 The derived data type for holding linearly-constrained linear least-squares problems

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

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

`o` is a scalar variable of type `INTEGER(ip_)`, that holds the number of observations (and rows of $\mathbf{A}_o$), o.

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

`Ao` is scalar variable of type `SMT_TYPE` that holds the 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 or the first thirteen components must contain the string DENSE_BY_ROWS.

By contrast, if the dense column-wise storage scheme (see Section 2.1.2) is used, the first sixteen components of Ao%type must contain the 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. Just as for H%type above, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into Ao%type. If prob is of derived type LSP_problem_type and involves a Jacobian $\mathbf{A}_o$ we wish to store using the sparse column-wise storage scheme, we may simply

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

Ao%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in $\mathbf{A}_o$ in the sparse co-ordinate storage scheme (see Section 2.1.3). It need not be set for either of the other four schemes.

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

Ao%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of $\mathbf{A}_o$ in the sparse co-ordinate or column-wise storage schemes (see §2.1.3 and 2.1.5). It need not be allocated for the other three schemes.

Ao%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of $\mathbf{A}_o$ in either the sparse co-ordinate (see Section 2.1.3), or the sparse row-wise (see 2.1.4) storage scheme. It need not be allocated for the other three schemes.

Ao%ptr is a rank-one allocatable array of dimension o+1 and type INTEGER(ip_), that holds the starting position of each row of $\mathbf{A}_o$, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.4). By contrast, if the spares column-wise scheme is used, it should be of length n+1, and hold starting position of each column of $\mathbf{A}_o$, as well as the total number of entries plus one.

B is a rank-one allocatable array type REAL(rp_), that should be allocated to have length o, and its j-th component filled with the value b_i for $i = 1,\ldots,o$,

DB is a rank-one allocatable array of dimension o and type REAL(rp_), that may hold the perturbation $\delta\mathbf{b}$ of the observation vector $\mathbf{b}$. The j-th component of DB, $i = 1,\ldots,o$, contains δb_i.

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

A%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense row-wise storage scheme (see Section 2.1.1) is used, the first five components of A%type must contain the string DENSE or the first thirteen components must contain the string DENSE_BY_ROWS. By contrast, if the dense column-wise storage scheme (see Section 2.1.2) is used, the first sixteen components of A%type must contain the 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. Once again, if prob is of derived type LSP_problem_type and involves a Jacobian $\mathbf{A}$ we wish to store using the sparse row-wise storage scheme, we may simply

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

A%ne is a scalar variable of type INTEGER(ip_), 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 either of the other four schemes.

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

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

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 **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, if the spares column-wise scheme is used, it should be of length n+1, and hold starting position of each column of **A**, as well as the total number of entries plus one.

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 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 solver-dependent value that will be recognised as infinity.

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

DC_l is a rank-one allocatable array of dimension m and type REAL(rp_), that may hold the vector of parametric lower bounds $\delta\mathbf{c}^l$ on the general constraints. The i-th component of DC_l, $i = 1,\ldots,m$, contains δc_i^l. Only components corresponding to finite lower bounds c_i^l need be set.

DC_u is a rank-one allocatable array of dimension m and type REAL(rp_), that may hold the vector of parametric upper bounds $\delta\mathbf{c}^u$ on the general constraints. The i-th component of DC_u, $i = 1,\ldots,m$, contains δc_i^u. Only components corresponding to finite upper bounds c_i^u need be set.

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 variables. The j-th component of X_l, $j = 1,\ldots,n$, contains x_j^l. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a solver-dependent value that will be recognised as infinity.

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

DX_l is a rank-one allocatable array of dimension n and type REAL(rp_), that may hold the vector of parametric lower bounds $\delta\mathbf{x}^l$ on the variables. The j-th component of DX_l, $j = 1,\ldots,n$, contains δx_j^l. Only components corresponding to finite lower bounds x_j^l need be set.

DX_u is a rank-one allocatable array of dimension n and type REAL(rp_), that may hold the vector of parametric upper bounds $\delta\mathbf{x}^u$ on the variables. The j-th component of DX_u, $j = 1,\ldots,n$, contains δx_j^u. Only components corresponding to finite upper bounds x_j^u need be set.

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

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

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

Z_u is a rank-one allocatable array of dimension n and type default REAL(rp_), that holds the values $\mathbf{z}^u$ of estimates of the dual variables corresponding to the upper simple bound constraints $\mathbf{x} \leq \mathbf{x}^u$ (see Section 4). The j-th component of Z_l, $j = 1,\ldots,n$, contains z^l_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 $\mathbf{A}^T_i\mathbf{x} \equiv (\mathbf{Ax})_i$.

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.

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

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

2.3.3 The derived data type for holding the problem dimensions

The derived data type QPT_dimensions_type is used to hold scalar data that defines the problem partitioning for the reordered problem. The components of QPT_dimensions_type are:

x_free is a scalar variable of type INTEGER(ip_), that holds the number of free variables.

x_l_start is a scalar variable of type INTEGER(ip_), that holds the index of the first variable with a nonzero lower (or lower range) bound.

x_l_end is a scalar variable of type INTEGER(ip_), that holds the index of the last variable with a nonzero lower (or lower range) bound.

x_u_start is a scalar variable of type INTEGER(ip_), that holds the index of the first variable with a nonzero upper (or upper range) bound.

x_u_end is a scalar variable of type INTEGER(ip_), that holds the index of the last variable with a nonzero upper (or upper range) bound.

c_equality is a scalar variable of type INTEGER(ip_), that holds the number of equality constraints.

c_l_start is a scalar variable of type INTEGER(ip_), that holds the index of the first inequality constraint with a lower (or lower range) bound.

c_l_end is a scalar variable of type INTEGER(ip_), that holds the index of the last inequality constraint with a lower (or lower range) bound.

c_u_start is a scalar variable of type INTEGER(ip_), that holds the index of the first inequality constraint with an upper (or upper range) bound.

c_u_end is a scalar variable of type INTEGER(ip_), that holds the index of the last inequality constraint with an upper (or upper range) bound.

2.3.4 The derived data type for holding control parameters

The derived data type LSP_control_type is used to hold controlling data. Default values may be obtained by calling LSP_initialize (see Section 2.4.1). The components of LSP_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 LSP_reorder and LSP_terminate is suppressed if error ≤ 0. The default is error = 6.

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}.

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_LSP 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..

2.3.5 The derived data type for holding informational parameters

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

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Sections 2.5 and 2.6 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation.

2.3.6 The derived data type for holding reordering data

The derived data type LSP_map_type is used to hold all the reordering and workspace data for a particular problem, or sequences of problems with the same structure, between calls of LSP procedures. This data should be preserved, untouched, from the initial call to LSP_initialize to the final call to LSP_terminate.

2.4 Argument lists and calling sequences

There are six procedures for user calls:

1. The subroutine LSP_initialize is used to set default values, and initialize private data, before reordered one or more problems with the same sparsity and bound structure. Here, the term "structure" refers both to the sparsity patterns of the Jacobian matrices $\mathbf{A}_o$ and $\mathbf{A}$ involved (but not their numerical values), to the zero/nonzero/infinity patterns (a bound is either zero, $\pm$ infinity, or a finite but arbitrary nonzero) of each of the constraint bounds, and

to the variables and constraints that are fixed (both bounds are the same) or free (the lower and upper bounds are $\pm$ infinity, respectively).

2. The subroutine LSP_reorder is called to reorder a problem, or the first of a sequence of structurally identical problems.

3. The subroutine LSP_apply may be called to reorder real data for subsequent structurally identical problems.

4. The subroutine LSP_get_values may be used to obtain the values of the original primal and dual variables and Lagrange multipliers from those for the reordered problem.

5. The subroutine LSP_restore may be used to recover the original problem from the data for the reordered one.

6. The subroutine LSP_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by LSP_reorder, at the end of the reordering process. It is important to do this if the data object is re-used for another problem **with a different structure** since LSP_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL LSP_initialize( map, control )
```

map is a scalar INTENT(OUT) argument of type LSP_map_type (see Section 2.3.6). It is used to hold all the reordering and workspace data for the problem.

control is a scalar INTENT(OUT) argument of type LSP_control_type (see Section 2.3.4). On exit, control contains default values for the components as described in Section 2.3.4. These values should only be changed after calling LSP_initialize.

2.4.2 The initial reordering subroutine

The initial reordering algorithm is applied as follows:

```
CALL LSP_reorder( map, control, info, dims, prob, &
                  get_x, get_y, get_z [, parametric ] )
```

map is a scalar INTENT(INOUT) argument of type LSP_map_type. It is used to hold reordering and workspace data for the problem. It must not have been altered **by the user** since the last call to LSP_initialize.

control is a scalar INTENT(IN) argument of type LSP_control_type (see Section 2.3.4). Default values may be assigned by calling LSP_initialize prior to the first call to LSP_reorder.

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

dims is a scalar INTENT(OUT) argument of type QPT_dimensions_type that is used to hold scalar data that defines the reordered problem. On successful exit, all components will have been set to values that define the reordered problem (see Section 2.3.3).

prob is a scalar INTENT(INOUT) argument of type QPT_problem_type that is used to hold data that defines the original and reordered problem. On entry, components f, gradient_kind, G, A, C_l, C_u, X_l and X_u must be appropriately allocated and set (see Section 2.3.2). The same is true of component H in the quadratic programming case and components Hessian_kind, target_kind, WEIGHT and X0 in the least-distance case. In addition, for parametric problems DG, DC_l, DC_u, DX_l and DX_u must be allocated appropriately and set. If the user wishes to provide suitable starting values for $\mathbf{x}$, $\mathbf{y}$ (or alternatively $\mathbf{y}^l$ and $\mathbf{y}^u$) and $\mathbf{z}$ (or alternatively $\mathbf{z}^l$ and $\mathbf{z}^u$), they should be placed in X, Y (or Y_l and Y_u) and Z (or Z_l and Z_u) respectively, and the arguments get_x, get_y and get_z set appropriately (see below).

On successful exit, all provided components will have been set to values that define the reordered problem (see Section 2.2.1). The reordered arrays $\mathbf{A}_o$ and $\mathbf{A}$ will be stored using the sparse column-wise and row-wise schemes, respectively. In addition the components X, Y, Z and C will contain values of $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ and $\mathbf{Ax}$ for the reordered problem. The user should be aware that fixed variables and free constraints will have been removed, and thus that the components prob%n and prob%m may be smaller than their values on entry.

get_x is a scalar INTENT(IN) argument of type default LOGICAL, that must be set .FALSE. if the user wishes to provide suitable values for the primal variables in X, and .TRUE., if appropriate values should be calculated by the subroutine.

get_y is a scalar INTENT(IN) argument of type default LOGICAL, that must be set .FALSE. if the user wishes to provide suitable values for the Lagrange multiplies for the general linear constraints in Y (or alternatively in Y_l and Y_u), and .TRUE., if appropriate values should be calculated by the subroutine. In the latter case, the array Y (or the arrays Y_l and Y_u) must have been allocated.

get_z is a scalar INTENT(IN) argument of type default LOGICAL, that must be set .FALSE. if the user wishes to provide suitable values for the dual variables for the simple bound constraints in Z (or alternatively in Z_l and Z_u), and .TRUE., if appropriate values should be calculated by the subroutine. In the latter case, the array Z (or the arrays Z_l and Z_u) must have been allocated.

parametric is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If parametric is present, the problem will be assumed to include parametric data $\delta\mathbf{g}$, $\delta\mathbf{x}_l$, $\delta\mathbf{x}_u$, $\delta\mathbf{x}_l$ and $\delta\mathbf{x}_u$ as part of prob, and this data will be reordered. If If parametric is absent, no parametric data will be processed.

2.4.3 The subsequent reordering subroutine

The reordering calculated by a previous call to LSP_reorder may be applied to a structurally identical problem with different real data as follows:

```
CALL LSP_apply( map, info, dims, prob [, get_all, get_all_parametric, &
                get_g, get_dg, get_x, get_y, get_z,                   &
                get_x_bounds, get_dx_bounds, get_c, get_c_bounds,     &
                get_dc_bounds, get_Ao, get_A ] )
```

The arguments map, and info are exactly as for LSP_reorder. The values of the integers prob%n, prob%m, prob%o, prob%ao%ne, prob%A%ne, integer arrays prob%Ao%row, prob%Ao%ptr, prob%A%col, prob%A%ptr, and the remaining (integer) components of dims must have been preserved exactly as they were on exit from the most recent call to LSP_reorder or LSP_restore, and are not altered by the subroutine.

New REAL(rp_), values may be assigned to the arguments prob%Ao%val, prob%B, prob%A%val, prob%C_l, prob%C_l, prob%X_l, prob%X_u, prob%X, prob%Y (or alternatively prob%Y_l and prob%Y_u), and prob%Z (or alternatively prob%Z_l and prob%Z_u), (and optionally prob%DB, prob%DC, prob%DC_l, prob%DC_l, prob%DX_l and prob%DX_u for parametric problems), but the components of these arrays must be in exactly the same order as originally presented to LSP_reorder. The exit values of all of these real values depend on the following, remaining arguments:

get_all is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_all is present, the entire non-parametric problem input in prob will be reordered according to the mappings generated by the last successful call to LSP_reorder. Any parametric data will be ignored.

get_all_parametric is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_all_parametric is present, the entire parametric problem input in prob will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_f is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_f is present, the constant objective term f, input in prob%f will be adjusted for the the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_g is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_g is present, the gradient $\mathbf{g}$, input in prob%G will be adjusted for the the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_dg is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dg is present, the parametric gradient $\delta\mathbf{g}$, input in prob%DG), will be adjusted for the the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_x is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_x is present, the vector of primal variables $\mathbf{x}$, input in prob%X, will be adjusted for the the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_y is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_y is present, the Lagrange multipliers $\mathbf{y}$, input in prob%Y (or alternatively in prob%Y_l and prob%Y_u), will be adjusted for the the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_z is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_z is present, the vector of dual variables $\mathbf{z}$, input in prob%Z (or alternatively in prob%Z_l and prob%Z_u), will be adjusted for the reordered problem according to the mappings generated by the last successful call to LSP_reorder.

get_x_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_x_bounds is present, the vectors of variable bounds $\mathbf{x}_l$ and $\mathbf{x}_u$, input in prob%X_l and prob%X_u respectively will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_dx_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dx_bounds is present, the vectors of parametric variable bounds $\delta\mathbf{x}_l$ and $\delta\mathbf{x}_u$, input in prob%DX_l and prob%DX_u respectively, will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_c is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_c is present, the vector $\mathbf{Ax}$ for the reordered problem will be returned in prob%C.

get_c_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_c_bounds is present, the vectors of constraint bounds $\mathbf{c}_l$ and $\mathbf{c}_u$, input in prob%C_l and prob%C_u respectively will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_dc_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dc_bounds is present, the vectors of parametric constraint bounds $\delta\mathbf{c}_l$ and $\delta\mathbf{c}_u$, input in prob%DC_l and prob%DC_u respectively, will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_A is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_A is present, the entries of the matrix $\mathbf{A}_o$, input in prob%Ao%val, will be reordered according to the mappings generated by the last successful call to LSP_reorder.

get_L is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_L is present, the entries of the matrix $\mathbf{A}$, input in prob%A%val, will be reordered according to the mappings generated by the last successful call to LSP_reorder.

2.4.4 The variable recovery reordering subroutine

The values of minimization variables that have been determined for the reordered problem may be recovered for the original problem as follows:

```
CALL LSP_get_values( map, info, prob [, X_val, Y_val, Z_val ] )
```

The arguments map and info are exactly as for LSP_reorder. The INTENT(IN) argument prob must contain reordered problem data from a previous call to LSP_reorder or LSP_apply.

X_val is an OPTIONAL rank-one INTENT(OUT) array argument of type REAL(rp_). If present, it will be filled with the values of the primal variables **x** for the original problem, corresponding to those for the the reordered problem input in X.

Y_val is an OPTIONAL rank-one INTENT(OUT) array argument of type REAL(rp_). If present, it will be filled with the values of the Lagrange multipliers **y** for the original problem, corresponding to those for the the reordered problem input in Y (or Y_l + Y_u). .

Z_val is an OPTIONAL rank-one INTENT(OUT) array argument of type REAL(rp_). If present, it will be filled with the values of the primal variables **z** for the original problem, corresponding to those for the the reordered problem input in Z (or Z_l + Z_u).

2.4.5 The problem restoration subroutine

The data for the original problem may be recovered from its reordered variant as follows:

```
CALL LSP_restore( map, info, dims, prob [, get_all, get_all_parametric, &
                  get_g, get_dg, get_x, get_y, get_z,                   &
                  get_x_bounds, get_dx_bounds, get_c, get_c_bounds,     &
                  get_dc_bounds, get_Ao, get_A ] )
```

The arguments map, info, dims and prob are exactly as described as output from LSP_reorder or LSP_apply, and correspond to data for the reordered problem. They may be restored to data for the original problem by appropriate settings for the remaining arguments:

get_all is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_all is present, the entire non-parametric problem input in prob and dims will be restored using the mappings generated by the last successful call to LSP_reorder. Any parametric data will be ignored.

get_all_parametric is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_all_parametric is present, the entire parametric problem input in prob and dims will be recovered from the mappings generated by the last successful call to LSP_reorder.

get_g is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_g is present, the gradient **g** will be recovered from the reordered problem and placed in prob%G using the mappings generated by the last successful call to LSP_reorder.

will be recovered from the reordered problem using the mappings generated by the last successful call to LSP_reorder.

get_dg is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dg is present, the parametric gradient δ**g** will be recovered from the reordered problem and placed in prob%DG using the mappings generated by the last successful call to LSP_reorder.

get_x is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_x is present, the vector of primal variables **x** will be recovered from the reordered problem and placed in prob%X using the mappings generated by the last successful call to LSP_reorder.

get_y is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_y is present and prob%Y is allocated, the Lagrange multipliers **y** will be recovered from the reordered problem and placed in prob%Y using the mappings generated by the last successful call to LSP_reorder. If prob%Y_l and prob%Y_u are allocated, the Lagrange multipliers $\mathbf{y}^l$ and $\mathbf{y}^u$ will be recovered from the reordered problem and placed in prob%Y_l and prob%Y_u respectively.

get_z is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_z is present and prob%Z is allocated, the vector of dual variables **z** will be recovered from the reordered problem and placed in prob%Z using the mappings generated by the last successful call to LSP_reorder. If prob%Z_l and prob%Z_u are allocated, the dual variables $\mathbf{z}^l$ and $\mathbf{z}^u$ will be recovered from the reordered problem and placed in prob%Z_l and prob%Z_u respectively.

get_x_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_x_bounds is present, the vectors of variable bounds $\mathbf{x}_l$ and $\mathbf{x}_u$ will be recovered from the reordered problem and placed in prob%X_l and prob%X_u using the mappings generated by the last successful call to LSP_reorder.

get_dx_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dx_bounds is present, the vectors of parametric variable bounds $\delta\mathbf{x}_l$ and $\delta\mathbf{x}_u$ will be recovered from the reordered problem and placed in prob%DX_l and prob%DX_u using the mappings generated by the last successful call to LSP_reorder.

get_c is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_c is present, the vector **Ax** will be recovered from the reordered problem and placed in prob%C using the mappings generated by the last successful call to LSP_reorder.

get_c_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_c_bounds is present, the vectors of constraint bounds $\mathbf{c}_l$ and $\mathbf{c}_u$ will be recovered from the reordered problem and placed in prob%C_l and prob%C_u using the mappings generated by the last successful call to LSP_reorder.

get_dc_bounds is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_dc_bounds is present, the vectors of parametric constraint bounds $\delta\mathbf{c}_l$ and $\delta\mathbf{c}_u$ will be recovered from the reordered problem and placed in prob%DC_l and prob%DC_u using the mappings generated by the last successful call to LSP_reorder.

get_Ao is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_Ao is present, the matrix $\mathbf{A}_o$ will be recovered from the reordered problem and placed in prob%Ao using the mappings generated by the last successful call to LSP_reorder.

get_A is an OPTIONAL scalar INTENT(IN) argument of type default LOGICAL. If get_L is present, the matrix **A** will be recovered from the reordered problem and placed in prob%A using the mappings generated by the last successful call to LSP_reorder.

2.4.6 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL LSP_terminate( map, control, info )
```

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

control is a scalar INTENT(IN) argument of type LSP_control_type exactly as for LSP_reorder.

info is a scalar INTENT(OUT) argument of type LSP_inform_type exactly as for LSP_reorder. Only the component status will be set on exit, and a successful call to LSP_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 info%status on exit from LSP_solve or LSP_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 is given by the value inform%alloc_status.

-3. One of the restrictions prob%n > 0, prob%o ≥ 0 or prob%m ≥ 0 or requirements that prob%Ao%type and prob%A%type contain its relevant string 'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS' or 'SPARSE_BY_COLUMNS' has been violated.

-5. The constraints are inconsistent.

-31. An attempt to use LSP_apply, LSP_get_values or LSP_restore has been made before a successful call to LSP_reorder.

-52. An attempt to change a matrix storage format has been made without first recalling LSP_reorder.

-53. At least one of the matrices $\mathbf{A}_o$ or $\mathbf{A}$ has not been reordered, while the current subroutine call requires it to have been.

-54. Neither the array prob%Y nor the pair prob%Y_l and prob%Y_u have been allocated.

-55. Neither the array prob%Z nor the pair prob%Z_l and prob%Z_u have been allocated.

2.6 Information printed

The only information printed will be error messages, corresponding to nonzero values of info%status, on unit control%error.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: GALAHAD_SYMBOLS, GALAHAD_SMT, GALAHAD_QPT, and GALAHAD_SORT.

Input/output: Output is under control of the argument control%error.

Restrictions: prob%n > 0, prob%m ≥ 0, prob%o ≥ 0, prob%Ao%type and prob%A%type $\in$ {'DENSE', 'DENSE_BY_ROWS', '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

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

where

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

the dual optimality conditions

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

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 $\mathbf{y}$ and $\mathbf{z}$ are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise.

Two passes are made through the sets of bounds on the variables. In the first, the number belonging to each of the required categories (free, non-negativities, lower-bounded, range-bounded, upper-bounded, non-positivities and fixed) is computed. On the second pass, a permutation of the variables to rearange them into the required standard form is obtained. A mapping array of the original Jacobian entries into their permuted form is then obtained, and the permutations applied in place (ie, without resorting to further storage) to $\mathbf{A}_o$, $\mathbf{A}$ $\mathbf{x}$, $\mathbf{z}$, $\mathbf{b}$, $\mathbf{x}^l$ and $\mathbf{x}^u$, suitable values of $\mathbf{x}$ and $\mathbf{z}$ satisfying (4.2) and (4.4) having optionally been computed.

Next, two passes are made through the sets of constraint bounds. In the first, the number belonging to each of the required categories (equality, lower-bounded, range-bounded, upper-bounded, and free) is computed, while in the second the required permutation of the constraints into the required standard form is obtained. A mapping array of the original Jacobian entries into their permuted form is then obtained, and the permutations applied in place to $\mathbf{A}_o$, $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{y}$, $\mathbf{c}^l$ and $\mathbf{c}^u$, suitable values of $\mathbf{c}$ and $\mathbf{y}$, satisfying (4.4), having, as before, optionally been computed. Both sets of permutations, and the matrix mapping arrays are saved for possible later use.

Any fixed variables and free constraints are removed. Fixing variables results in changes to the values of f, $\mathbf{g}$, $\mathbf{c}^l$ and $\mathbf{c}^u$. Subsequent reorderings for structurally similar problems, or restorations of data from reordered problems, are easily obtained from the permuation and mapping arrays, and their inverses.

5 EXAMPLE OF USE

Suppose we wish to minimize

$$\tfrac{1}{2}\left\| \begin{pmatrix} x_1 - 1 \\ x_1 + 2x_2 - 2 \\ x_1 + x_2 + 3x_3 - 3 \\ x_1 + x_2 + x_3 + 4x_4 - 4 \\ 5x_2 + x_3 + x_4 - 5 \\ 6x_3 + x_4 - 6 \\ 7x_7 - 7 \end{pmatrix} \right\|^2$$

subject to the general linear constraints $1 \leq 2x_1 + x_2 \leq 2$, $x_2 + x_3 + x_4 = 2$, and simple bounds $-1 \leq x_1 \leq 1$, $x_3 = 1$ and $x_4 \leq 2$, but first wish to convert the problem to our standard form. Then, on writing the data for this problem as

$$\mathbf{A}_o = \begin{pmatrix} 1 & & & \\ 1 & 2 & & \\ 1 & 1 & 3 & \\ 1 & 1 & 1 & 4 \\ & 5 & 1 & 1 \\ & & 6 & 1 \\ & & & 7 \end{pmatrix}, \ \mathbf{b} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{pmatrix}, \ \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ 1 \\ -\infty \end{pmatrix} \text{ and } \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 1 \\ 2 \end{pmatrix},$$

and

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

we may use the following code:

```
! THIS VERSION: GALAHAD 4.1 - 2022-09-07 AT 10:25 GMT.
   PROGRAM GALAHAD_LSP_EXAMPLE
   USE GALAHAD_LSP_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_dimensions_type ) :: d
   TYPE ( LSP_map_type ) :: map
   TYPE ( LSP_control_type ) :: control
   TYPE ( LSP_inform_type ) :: info
   TYPE ( QPT_problem_type ) :: p
   INTEGER :: i, j, s
   INTEGER, PARAMETER :: coordinate = 1, sparse_by_rows = 2
   INTEGER, PARAMETER :: sparse_by_columns = 3, dense = 4
   INTEGER, PARAMETER :: type = 1
   INTEGER, PARAMETER :: n = 4, m = 2, o = 7, a_ne = 16, l_ne = 5
   REAL ( KIND = wp ) :: X_orig( n )
! sparse co-ordinate storage format
   IF ( type == coordinate ) THEN
     WRITE( 6, "( ' co-ordinate storage' )" )
     CALL SMT_put( p%Ao%type, 'COORDINATE', s )  ! Specify co-ordinate
     CALL SMT_put( p%A%type, 'COORDINATE', s )  ! storage for A and L
     ALLOCATE( p%Ao%val( a_ne ), p%Ao%row( a_ne ), p%Ao%col( a_ne ) )
     ALLOCATE( p%A%val( l_ne ), p%A%row( l_ne ), p%A%col( l_ne ) )
     p%Ao%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp,                              &
                   2.0_wp, 1.0_wp, 1.0_wp, 5.0_wp,                              &
                   3.0_wp, 1.0_wp, 1.0_wp, 6.0_wp,                              &
                   3.0_wp, 1.0_wp, 1.0_wp, 6.0_wp /)  ! Jacobian A_o
     p%Ao%row = (/ 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, 4, 5, 6, 7 /)
     p%Ao%col = (/ 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4 /)
     p%Ao%ne = a_ne
     p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)  ! Jacobian L
     p%A%row = (/ 1, 1, 2, 2, 2 /)
     p%A%col = (/ 1, 2, 2, 3, 4 /)
     p%A%ne = l_ne
! sparse row-wise storage format
   ELSE IF ( type == sparse_by_rows ) THEN
     WRITE( 6, "( ' sparse by rows storage' )" )
     CALL SMT_put( p%Ao%type, 'SPARSE_BY_ROWS', s )  ! Specify sparse-by-rows
```

```
     CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS', s )  ! storage for A and L
     ALLOCATE( p%Ao%val( a_ne ), p%Ao%col( a_ne ), p%Ao%ptr( o + 1 ) )
     ALLOCATE( p%A%val( l_ne ), p%A%col( l_ne ), p%A%ptr( m + 1 ) )
     p%Ao%val = (/ 1.0_wp, 1.0_wp, 2.0_wp, 1.0_wp, 1.0_wp, 3.0_wp,             &
                   1.0_wp, 1.0_wp, 1.0_wp, 4.0_wp, 5.0_wp, 1.0_wp,             &
                   1.0_wp, 6.0_wp, 1.0_wp, 7.0_wp /)          ! Jacobian A_o
     p%Ao%col = (/ 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 3, 4, 4 /)
     p%Ao%ptr = (/ 1, 2, 4, 7, 11, 14, 16, 17 /)              ! Set row pointers
     p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /)  ! Jacobian L
     p%A%col = (/ 1, 2, 2, 3, 4 /)
     p%A%ptr = (/ 1, 3, 6 /)                                 ! Set row pointers
! sparse colum-wise storage format
   ELSE IF ( type == sparse_by_columns ) THEN
     WRITE( 6, "( ' sparse by columns storage' )" )
     CALL SMT_put( p%Ao%type, 'SPARSE_BY_COLUMNS', s ) !Specify sparse-by-column
     CALL SMT_put( p%A%type, 'SPARSE_BY_COLUMNS', s ) ! storage for A and L
     ALLOCATE( p%Ao%val( a_ne ), p%Ao%row( a_ne ), p%Ao%ptr( n + 1 ) )
     ALLOCATE( p%A%val( l_ne ), p%A%row( l_ne ), p%A%ptr( n + 1 ) )
     p%Ao%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 2.0_wp, 1.0_wp,             &
                   1.0_wp, 5.0_wp, 3.0_wp, 1.0_wp, 1.0_wp, 6.0_wp,             &
                   4.0_wp, 1.0_wp, 1.0_wp, 7.0_wp /)        ! Jacobian A_o
     p%Ao%row = (/ 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, 4, 5, 6, 7 /)
     p%Ao%ptr = (/ 1, 5, 9, 13, 17 /)                        ! Set column pointer
     p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian L
     p%A%row = (/ 1, 1, 2, 2, 2 /)
     p%A%ptr = (/ 1, 2, 4, 5, 6 /)                           ! Set column pointer
! dense storage format
   ELSE IF ( type == dense ) THEN
     WRITE( 6, "( ' dense (by rows) storage' )" )
     CALL SMT_put( p%Ao%type, 'DENSE', s )  ! Specify dense (by rows)
     CALL SMT_put( p%A%type, 'DENSE', s )  ! storage for A and L
     ALLOCATE( p%Ao%val( n * o ) )
     ALLOCATE( p%A%val( n * m ) )
     p%Ao%val = (/ 1.0_wp, 0.0_wp, 0.0_wp, 0.0_wp,                             &
                   1.0_wp, 2.0_wp, 0.0_wp, 0.0_wp,                             &
                   1.0_wp, 1.0_wp, 3.0_wp, 0.0_wp,                             &
                   1.0_wp, 1.0_wp, 1.0_wp, 4.0_wp,                             &
                   0.0_wp, 5.0_wp, 1.0_wp, 1.0_wp,                             &
                   0.0_wp, 0.0_wp, 6.0_wp, 1.0_wp,                             &
                   0.0_wp, 0.0_wp, 0.0_wp, 7.0_wp /)         ! Jacobian A_o
     p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 0.0_wp, 1.0_wp,              &
                  1.0_wp, 1.0_wp /)                          ! Jacobian L
   END IF
! matrices complete, initialize arrays
   ALLOCATE( p%B( o ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ) )
   ALLOCATE( p%X( n ), p%Y( m ), p%Z( n ) )
   p%n = n ; p%m = m ; p%o = o                ! dimensions
   p%B = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp, 5.0_wp, 6.0_wp, 7.0_wp /) ! obs
   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, 1.0_wp, - infinity /) ! variable lower bound
   p%X_u = (/ 1.0_wp, infinity, 1.0_wp, 2.0_wp /)         ! variable upper bound
!  WRITE( 6, "( /, 5X,'i', 7X, 'l', 7X, 'u' )" )
!  DO i = 1, p%m
```

```
!   WRITE( 6, "( I6, 2ES9.1 )" ) i, p%C_l( i ), p%C_u( i )
!  END DO
   CALL LSP_initialize( map, control )          ! Initialize control parameters
   control%infinity = infinity                  ! Set infinity
! reorder problem
   CALL LSP_reorder( map, control, info, d, p, .TRUE., .TRUE., .TRUE. )
   IF ( info%status /= 0 ) & !  Error returns
     WRITE( 6, "( ' LSP_solve exit status = ', I6 ) " ) info%status
   WRITE( 6, "( ' reordered problem now involves ', I0, ' variables and ',     &
  &       I0, ' constraints' )" ) p%n, p%m
! re-ordered variables
   WRITE( 6, "( /, 5X, 'i', 6x, 'v', 6X, 'l', 8X, 'u', 8X, 'z', 5X, 'type' )" )
   DO i = 1, d%x_free                       ! free variables
     WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), '  '
   END DO
   DO i = d%x_free + 1, d%x_l_start - 1     ! non-negativities
    WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), '0< '
   END DO
   DO i = d%x_l_start, d%x_u_start - 1      ! lower-bounded variables
    WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), 'l< '
   END DO
   DO i = d%x_u_start, d%x_l_end            ! range-bounded variables
    WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), 'l<u'
   END DO
   DO i = d%x_l_end + 1, d%x_u_end          ! upper-bounded variables
    WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), ' <u'
   END DO
   DO i = d%x_u_end + 1, p%n                ! non-positivities
    WRITE( 6, 10 ) i, p%X( i ), p%X_l( i ), p%X_u( i ), p%Z( i ), ' <0'
   END DO
! re-ordered constraints
   WRITE( 6, "( /, 5X,'i', 5x, 'L*v', 7X, 'l', 8X, 'u', 8X, 'y', 3X, 'type' )" )
   DO i = 1, d%c_l_start - 1                ! equality constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l=u'
   END DO
   DO i = d%c_l_start, d%c_u_start - 1      ! lower-bounded constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l< '
   END DO
   DO i = d%c_u_start, d%c_l_end            ! range-bounded constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l<u'
   END DO
   DO i = d%c_l_end + 1, d%c_u_end          ! upper-bounded constraints
     WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), ' <u'
   END DO
! re-ordered matrices
   WRITE( 6, "( /, ' Observations B', /, 7ES8.1 )" ) p%B( : p%o )
   WRITE( 6, 20 ) 'Objective Jacobian A', ( ( 'A', i, p%Ao%row( j ),           &
     p%Ao%val( j ), j = p%Ao%ptr( i ), p%Ao%ptr( i + 1 ) - 1 ), i = 1, p%n )
   WRITE( 6, 20 ) 'Constraint Jacobian L', ( ( 'L', i, p%A%col( j ),           &
     p%A%val( j ), j = p%A%ptr( i ), p%A%ptr( i + 1 ) - 1 ), i = 1, p%m )
   p%X( : 3 ) = (/ 1.6_wp, 0.2_wp, -0.6_wp /)
   CALL LSP_get_values( map, info, p, X_val = X_orig )
   WRITE( 6, "( /, ' solution = ', ( 4ES9.1 ) )" ) X_orig( : n )
! recover observations and constraint bounds
   CALL LSP_restore( map, info, p, get_b = .TRUE., get_c_bounds = .TRUE. )
```

```
!  WRITE( 6, "( /, 5X,'i', 7X, 'l', 7X, 'u' )" )
!  DO i = 1, p%m
!   WRITE( 6, "( I6, 2ES9.1 )" ) i, p%C_l( i ), p%C_u( i )
!  END DO
!  WRITE( 6, "( /, 5X,'i', 7X, 'b' )" )
!  DO i = 1, p%o
!   WRITE( 6, "( I6, ES9.1 )" ) i, p%B( i )
!  END DO

   WRITE( 6, "( /, ' modified problem now involves ', I0, ' variables and ',   &
  &       I0, ' constraints' )" ) p%n, p%m
! change upper bound
   p%C_u( 1 ) = 3.0_wp
! reorder new problem
   CALL LSP_apply( map, info, p, get_c_bounds = .TRUE. )
! re-ordered new constraints
   WRITE( 6, "( /, 5X,'i', 5x, 'A*v', 7X, 'l', 8X, 'u', 8X, 'y', 3X, 'type' )" )
   DO i = 1, d%c_l_start - 1               ! equality constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l=u'
   END DO
   DO i = d%c_l_start, d%c_u_start - 1     ! lower-bounded constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l< '
   END DO
   DO i = d%c_u_start, d%c_l_end           ! range-bounded constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), 'l<u'
   END DO
   DO i = d%c_l_end + 1, d%c_u_end         ! upper-bounded constraints
    WRITE( 6, 10 ) i, p%C( i ), p%C_l( i ), p%C_u( i ), p%Y( i ), ' <u'
   END DO
   CALL LSP_terminate( map, control, info ) !  delete internal workspace
10 FORMAT( I6, 4ES9.1, 2X, A3 )
20 FORMAT( /, 1X, A, /, ( :, 3 ( 1X, A1, '(', 2I2, ') =', ES8.1, : ) ) )
   END PROGRAM GALAHAD_LSP_EXAMPLE
```

This produces the following output:

```
 co-ordinate storage
 reordered problem now involves 3 variables and 2 constraints

     i      v      l        u        z     type
     1  0.0E+00 -1.0E+20  1.0E+20  0.0E+00
     2  0.0E+00 -1.0E+00  1.0E+00  0.0E+00  l<u
     3  1.0E+00 -1.0E+20  2.0E+00 -1.0E+00   <u

     i     L*v       l        u        y   type
     1  1.0E+00  1.0E+00  1.0E+00  1.0E+00  l=u
     2  0.0E+00  1.0E+00  2.0E+00  1.0E+00  l<u

 Observations B
 1.0E+00 2.0E+00 0.0E+00 3.0E+00 4.0E+00 0.0E+00 7.0E+00

 Objective Jacobian A
 A( 1 2) = 2.0E+00 A( 1 3) = 1.0E+00 A( 1 4) = 1.0E+00
 A( 1 5) = 5.0E+00 A( 2 1) = 1.0E+00 A( 2 2) = 1.0E+00
 A( 2 3) = 1.0E+00 A( 2 4) = 1.0E+00 A( 3 4) = 3.0E+00
```

```
A( 3 5) = 1.0E+00 A( 3 6) = 1.0E+00 A( 3 7) = 6.0E+00

Constraint Jacobian L
L( 1 1) = 1.0E+00 L( 1 3) = 1.0E+00 L( 2 1) = 1.0E+00
L( 2 2) = 2.0E+00

solution =   2.0E-01  1.6E+00  1.0E+00 -6.0E-01

modified problem now involves 4 variables and 2 constraints

    i     A*v       l         u         y   type
    1  1.0E+00  1.0E+00  1.0E+00  1.0E+00  l=u
    2  0.0E+00  1.0E+00  3.0E+00  1.0E+00  l<u
```

The same problem may be solved holding the data in a sparse row-wise, a sparse column-wise, or a dense (row-wise) format by changing the parameter `type` in the program above to `2`, `3` or `4` respectively.