



Science and
Technology
Facilities Council



GALAHAD

LLSR

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

Given a real m by n matrix $\mathbf{A}$, a real n by n symmetric diagonally-dominant matrix $\mathbf{S}$, a real m vector $\mathbf{b}$, and scalars $\sigma > 0$ and $p \geq 2$, this package finds a **minimizer of the regularized linear least-squares objective function**

$$\frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \frac{\sigma}{p} \|\mathbf{x}\|_S^p,$$

where the $\mathbf{S}$ -norm of $\mathbf{x}$ is $\|\mathbf{x}\|_S = \sqrt{\mathbf{x}^T \mathbf{S} \mathbf{x}}$. This problem commonly occurs as a subproblem in nonlinear least-squares calculations. The matrix $\mathbf{S}$ need not be provided in the commonly-occurring ℓ_2 -regularization case for which $\mathbf{S} = \mathbf{I}$, the n by n identity matrix.

Factorization of matrices of the form

$$\begin{pmatrix} \lambda \mathbf{S} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{I} \end{pmatrix} \quad (1.1)$$

for a succession of scalars λ will be required, so this package is most suited for the case where such a factorization may be found efficiently. If this is not the case, the package GALAHAD_LSRT may be preferred.

ATTRIBUTES — Versions: GALAHAD_LLSR_single, GALAHAD_LLSR_double. **Uses:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_RAND, GALAHAD_NORMS, GALAHAD_ROOTS, GALAHAD_SPECFILE, GALAHAD_SBLs, GALAHAD_SLS, GALAHAD_IR, GALAHAD_MOP **Date:** June 2023. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple 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_LLSR_single
```

with the obvious substitution GALAHAD_LLSR_double, GALAHAD_LLSR_quadruple, GALAHAD_LLSR_single_64, GALAHAD_LLSR_double_64 and GALAHAD_LLSR_quadruple_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, LLSR_control_type, LLSR_history_type, LLSR_inform_type, LLSR_data_type, (Section 2.4) and the subroutines LLSR_initialize, LLSR_solve, LLSR_terminate (Section 2.5) and LLSR_read_specfile (Section 2.7) must be renamed on one of the USE statements.

2.1 Matrix storage formats

The matrices $\mathbf{A}$ and (if required) $\mathbf{S}$ may be stored in a variety of input formats.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.1.1 Dense storage format

The matrix $\mathbf{A}$ is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n * (i - 1) + j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1, \dots, m$, $j = 1, \dots, n$. Since $\mathbf{S}$ is symmetric, only the lower triangular part (that is the part s_{ij} for $1 \leq j \leq i \leq n$) need be held. In this case the lower triangle will be stored by rows, that is component $i * (i - 1) / 2 + j$ of the storage array `S%val` will hold the value s_{ij} (and, by symmetry, s_{ji}) for $1 \leq j \leq i \leq n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of $\mathbf{A}$, its row index i , column index j and value a_{ij} are stored in the l -th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to $\mathbf{S}$ (thus requiring integer arrays `S%row`, `S%col`, a real array `S%val` and an integer value `S%ne`), except that only the entries in the lower triangle need be stored.

2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i + 1$. For the i -th row of $\mathbf{A}$, the i -th component of a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr(m + 1)` holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i -th row are stored in components $l = \text{A\%ptr}(i), \dots, \text{A\%ptr}(i + 1) - 1$ of the integer array `A%col`, and real array `A%val`, respectively. The same scheme is applicable to $\mathbf{S}$ (thus requiring integer arrays `S%ptr`, `S%col`, and a real array `S%val`), except that only the entries in the lower triangle need be stored.

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

2.1.4 Diagonal storage format

If $\mathbf{S}$ is diagonal (i.e., $s_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonals entries s_{ii} , $1 \leq i \leq n$, need be stored, and the first n components of the array `S%val` may be used for the purpose. This scheme is inappropriate and thus unavailable for $\mathbf{A}$.

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, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined 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 Parallel usage

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

MPI may also be used by the package to provide parallelism for some solvers in a distributed memory environment. To use this form of parallelism, MPI must be enabled at runtime by using the correct compiler flag (usually some

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

variant of `-lmpi`). Although the MPI process will be started automatically when required, it should be stopped by the calling program once no further use of this form of parallelism is needed. Typically, this will be via statements of the form

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

The code may be compiled and run in serial mode.

2.4 The derived data types

Six derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrices $\mathbf{A}$ and perhaps $\mathbf{S}$. The components of `SMT_TYPE` used here are:

- `m` is a scalar component of type `INTEGER(ip_)`, that holds the number of rows in the matrix.
- `n` is a scalar component of type `INTEGER(ip_)`, that holds the number of columns in the matrix.
- `ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.
- `type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored.
- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. Each pair of off-diagonal entries $s_{ij} = s_{ji}$ of the *symmetric* matrix $\mathbf{S}$ is represented as a single entry (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 hold 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 `n + 1`, that may holds the pointers to the first entry in each row (see §2.1.3).

2.4.2 The derived data type for holding control parameters

The derived data type `LLSR_control_type` is used to hold controlling data. Default values may be obtained by calling `LLSR_initialize` (see Section 2.5.1). The components of `LLSR_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 `LLSR_solve` and `LLSR_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 `LLSR_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.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`new_a` is a scalar variable of type `INTEGER(ip_)`, that is used to indicate how $\mathbf{A}$ has changed (if at all) since the previous call to `LLSR_solve`. Possible values are:

- 0 $\mathbf{A}$ is unchanged.
- 1 the values in $\mathbf{A}$ have changed, but its nonzero structure is as before.
- 2 both the values and structure of $\mathbf{A}$ have changed.

The default is `new_a = 2`.

`new_s` is a scalar variable of type `INTEGER(ip_)`, that is used to indicate how $\mathbf{S}$ (if required) has changed (if at all) since the previous call to `LLSR_solve`. Possible values are:

- 0 $\mathbf{S}$ is unchanged.
- 1 the values in $\mathbf{S}$ have changed, but its nonzero structure is as before.
- 2 both the values and structure of $\mathbf{S}$ have changed.

The default is `new_s = 2`.

`max_factorizations` is a scalar variable of type `INTEGER(ip_)`, that holds the maximum number of factorizations which will be permitted. If `max_factorizations` is set to a negative number, there will be no limit on the number of factorizations allowed. The default is `max_factorizations = -1`.

`taylor_max_degree` is a scalar variable of type `INTEGER(ip_)`, that specifies the maximum degree of Taylor approximant that will be used to approximate the secular function when trying to improve λ ; a first-degree approximant results in Newton's method. The higher the degree, the better in general the improvement, but the larger the cost. Thus there is a balance between many cheap low-degree approximants and a few more expensive higher-degree ones. Our experience favours higher-degree approximants. The default is `taylor_max_degree = 3`, which is the highest degree currently supported.

`initial_multiplier` is a scalar variables of type `REAL(rp_)`, that should be set to an initial estimate of the required multiplier λ_* (see Section 4). The algorithm will only use this value if `%use_initial_multiplier` is set `.TRUE.` (see below), and otherwise will be reset by `LLSR_solve`. A good initial estimate may sometimes dramatically improve the performance of the package. The default is `initial_multiplier = 0.0`.

`lower` is a scalar variables of type `REAL(rp_)`, that holds the value of any known lower bound on the required multiplier λ_* . A good lower bound may sometimes dramatically improve the performance of the package, but an incorrect value might cause the method to fail. Thus resetting `lower` from its default should be used with caution. The default is `lower = - HUGE(1.0) (-HUGE(1.0D0) in GALAHAD_LLSR_double)`.

`upper` is a scalar variables of type `REAL(rp_)`, that holds the value of any known upper bound on the required multiplier λ_* . A good upper bound may sometimes dramatically improve the performance of the package, but an incorrect value might cause the method to fail. Thus resetting `upper` from its default should be used with caution. The default is `upper = HUGE(1.0) (HUGE(1.0D0) in GALAHAD_LLSR_double)`.

`stop_normal` is a scalar variable of type `REAL(rp_)`, that hold values for the standard convergence tolerances of the method (see Section 4). In particular, the method is deemed to have converged when the computed solution $\mathbf{x}$ and its multiplier λ satisfy $|||\mathbf{x}||_S - (\lambda/\sigma)^{1/(p-2)}| \leq \text{stop_normal} * \max(1, ||\mathbf{x}||)$. The default is `stop_normal =  $u^{0.75}$` , where u is `EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_LLSR_double)`.

`use_initial_multiplier` is a scalar variable of type `default LOGICAL`, that may be set `.TRUE.` if the user wishes to use the value of initial multiplier supplied in `%initial_multiplier`, and `.FALSE.` if the initial value will be chosen automatically. The default is `use_initial_multiplier = .FALSE.`

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

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

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

`definite_linear_solver` is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any symmetric positive-definite linear system that might arise. Current possible choices are `'sils'`, `'ma27'`, `'ma57'`, `'ma77'`, `'ma86'`, `'ma87'`, `'ma97'`, `'ssids'`, `'pardiso'` and `'wsmp'`. See the documentation for the GALAHAD package SLS for further details. The default is `definite_linear_solver = 'sils'`.

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

`SBLs_control` is a scalar variable of type `SBLs_control_type` that is used to control various aspects of the factorization package SBLs. See the documentation for GALAHAD_SBLs for more details.

`SLS_control` is a scalar variable of type `SLS_control_type` that is used to control various aspects of the factorization package SLS. See the documentation for GALAHAD_SLS for more details.

`IR_control` is a scalar variable of type `IR_control_type` that is used to control various aspects of the iterative refinement package IR. See the documentation for GALAHAD_IR for more details.

2.4.3 The derived data type for holding history information

The derived data type `LLSR_history_type` is used to hold the value of $\|\mathbf{x}(\lambda)\|_S$, where $\mathbf{x}(\lambda)$ satisfies $(\mathbf{H} + \lambda\mathbf{S})\mathbf{x}(\lambda) = -\mathbf{c}$ and $\mathbf{A}\mathbf{x}(\lambda) = \mathbf{0}$ for a specific λ arising during the computation. The components of `LLSR_history_type` are:

`lambda` is a scalar variable of type `REAL(rp_)`, that gives the value λ .

`x_norm` is a scalar variable of type default REAL, that gives the corresponding value $\|\mathbf{x}(\lambda)\|_S$.

2.4.4 The derived data type for holding timing information

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

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

`assemble` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent assembling the matrix (1.1) from its constituent parts.

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

`factorize` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent factorizing the required matrices.

`solve` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent using the factors to solve relevant linear equations.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

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

`clock_assemble` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent assembling the matrix (1.1) from its constituent parts.

`clock_analyse` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent analysing required matrices prior to factorization.

`clock_factorize` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent factorizing the required matrices.

`clock_solve` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent using the factors to solve relevant linear equations.

2.4.5 The derived data type for holding informational parameters

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

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the current status of the algorithm. See Section 2.6 for details.

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

`bad_alloc` is a scalar variable of type default `CHARACTER` and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if `status = 0`.

`factorizations` is a scalar variable of type `INTEGER(ip_)`, that gives the number of factorizations of the matrix (1.1) for different λ , performed during the calculation.

`max_entries_factors` is a scalar variable of type `INTEGER(ip_)`, that gives the maximum number of entries in any of the matrix factorizations performed during the calculation.

`len_history` is a scalar variable of type `INTEGER(ip_)`, that gives the number of $(\lambda, \|\mathbf{x}(\lambda)\|_S)$ pairs encountered during the calculation.

`r_normj` is a scalar variable of type `REAL(rp_)`, that holds the value of the norm of the residual $\|\mathbf{Ax} - \mathbf{b}\|_2$.

`x_norm` is a scalar variable of type `REAL(rp_)`, that holds the value of $\|\mathbf{x}\|_S$.

`multiplier` is a scalar variable of type `REAL(rp_)`, that holds the value of the multiplier λ associated with the regularization term.

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

`history` is an array argument of dimension `len_history` and type `LLSR_history_type` that contains a list of pairs $(\lambda, \|\mathbf{x}(\lambda)\|_S)$ encountered during the calculation (see Section 2.4.3).

`SBLS_inform` is a scalar variable of type `SBLS_inform_type`, that holds informational parameters concerning the analysis, factorization and solution phases performed by the GALAHAD sparse matrix factorization package SBLS. See the documentation for the package SBLS for details of the derived type `SBLS_inform_type`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`SLS_inform` is a scalar variable of type `SLS_inform_type`, that holds informational parameters concerning the analysis, factorization and solution phases performed by the GALAHAD sparse matrix factorization package `SLS`. See the documentation for the package `SLS` for details of the derived type `SLS_inform_type`.

`IR_inform` is a scalar variable of type `IR_inform_type`, that holds informational parameters concerning the iterative refinement subroutine contained in the GALAHAD refinement package `IR`. See the documentation for the package `IR` for details of the derived type `IR_inform_type`.

2.4.6 The derived data type for holding problem data

The derived data type `LLSR_data_type` is used to hold all the data for a particular problem between calls of `LLSR` procedures. This data should be preserved, untouched, from the initial call to `LLSR_initialize` to the final call to `LLSR_terminate`.

2.5 Argument lists and calling sequences

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

1. The subroutine `LLSR_initialize` is used to set default values and initialize private data.
2. The subroutine `LLSR_solve` is called to solve the problem.
3. The subroutine `LLSR_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `LLSR_solve`, at the end of the solution process.

We use square brackets [] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

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

`control` is a scalar INTENT(OUT) argument of type `LLSR_control_type` (see Section 2.4.2). On exit, `control` contains default values for the components as described in Section 2.4.2. These values should only be changed after calling `LLSR_initialize`.

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

2.5.2 The optimization problem solution subroutine

The optimization problem solution algorithm is called as follows:

```
CALL LLSR_solve( m, n, power, weight, A, B, X, data, control, inform[, S] )
```

`m` is a scalar INTENT(IN) argument of type `INTEGER(ip_)`, that must be set to the number of residuals, m . **Restriction:** $m > 0$.

`n` is a scalar INTENT(IN) argument of type `INTEGER(ip_)`, that must be set to the number of unknowns, n . **Restriction:** $n > 0$.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`power` is a scalar `INTENT(IN)` variable of type `REAL(rp_)`, that must be set on initial entry to the value of the regularization power, p . **Restriction:** $\text{power} \geq 2$.

`weight` is a scalar `INTENT(IN)` variable of type `REAL(rp_)`, that must be set on initial entry to the value of the regularization weight, σ . **Restriction:** $\text{weight} > 0$.

`A` is a scalar `INTENT(IN)` argument of type `SMT_TYPE` that holds the constraint matrix $\mathbf{A}$. The following components are used:

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

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `A%type`. For example, if we wish to store $\mathbf{A}$ using the co-ordinate scheme, we may simply

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

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

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

`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 schemes.

`A%val` is a rank-one allocatable array of type `REAL(rp_)`, that holds the values of the entries of the constraint matrix $\mathbf{A}$ 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 $\mathbf{A}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other 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 $A\%m+1$ 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 schemes are used.

`B` is an array `INTENT(IN)` argument of dimension m and type `REAL(rp_)`, whose i -th entry holds the component b_i of the vector $\mathbf{b}$ in the residual.

`X` is an array `INTENT(OUT)` argument of dimension n and type `REAL(rp_)`, that holds an estimate of the solution $\mathbf{x}$ of the problem on exit.

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

`control` is a scalar `INTENT(IN)` argument of type `LLSR_control_type`. (see Section 2.4.2). Default values may be assigned by calling `LLSR_initialize` prior to the first call to `LLSR_solve`.

`inform` is a scalar `INTENT(INOUT)` argument of type `LLSR_inform_type` (see Section 2.4.5). On initial entry, the component `status` must be set to 1. The remaining components need not be set. A successful call to `LLSR_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.6.

`S` is an OPTIONAL scalar `INTENT(IN)` argument of type `SMT_TYPE` that holds the diagonally dominant scaling matrix $\mathbf{S}$. It need only be set if $\mathbf{S} \neq \mathbf{I}$ and in this case the following components are used:

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`S%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 `S%type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of `S%type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of `S%type` must contain the string `SPARSE_BY_ROWS`, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of `S%type` must contain the string `DIAGONAL`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `S%type`. For example, if we wish to store $\mathbf{S}$ using the co-ordinate scheme, we may simply

```
CALL SMT_put( M%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package `SMT` for further details on the use of `SMT_put`.

`S%ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of entries in the **lower triangular** part of $\mathbf{S}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.

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

`S%row` is a rank-one allocatable array of type `INTEGER(ip_)`, that holds the row indices of the **lower triangular** part of $\mathbf{S}$ in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

`S%col` is a rank-one allocatable array variable of type `INTEGER(ip_)`, that holds the column indices of the **lower triangular** part of $\mathbf{S}$ 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 or diagonal storage schemes are used.

`S%ptr` is a rank-one allocatable array of dimension $n+1$ and type `INTEGER(ip_)`, that holds the starting position of each row of the **lower triangular** part of $\mathbf{S}$, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

If $\mathbf{S}$ is absent, the ℓ_2 -norm, $\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T \mathbf{x}}$, will be employed.

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

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

2.6 Warning and error messages

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

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

- 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. (LLSR_solve only) One of the restrictions $n > 0, m > 0, \text{power} \geq 2$ or $\text{weight} > 0$ has been violated.
- 9. (LLSR_solve only) The analysis phase of the factorization of the matrix (1.1) failed.
- 10. (LLSR_solve only) The factorization of the matrix (1.1) failed.
- 15. (LLSR_solve only) The matrix $\mathbf{S}$ appears not to be diagonally dominant.
- 16. (LLSR_solve only) The problem is so ill-conditioned that further progress is impossible.
- 18. (LLSR_solve only) Too many factorizations have been required. This may happen if `control%max_factorizations` is too small, but may also be symptomatic of a badly scaled problem.

2.7 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `LLSR_control_type` (see Section 2.4.2), by reading an appropriate data specification file using the subroutine `LLSR_read_specfile`. This facility is useful as it allows a user to change LLSR control parameters without editing and recompiling programs that call LLSR.

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

```
( .. lines ignored by LLSR_read_specfile .. )
BEGIN LLSR
    keyword    value
    .....
    keyword    value
END
( .. lines ignored by LLSR_read_specfile .. )
```

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

and

```
END LLSR SPECIFICATION
```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

are acceptable. Furthermore, between the “BEGIN LLSR” 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 `LLSR_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 `LLSR_read_specfile`.

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

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL LLSR_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `LLSR_control_type` (see Section 2.4.2). Default values should have already been set, perhaps by calling `LLSR_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.4.2) 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
use-dense-factorization	%dense_factorization	integer
has-a-changed	%new_a	integer
has-s-changed	%new_s	integer
factorization-limit	%max_factorizations	integer
max-degree-taylor-approximant	%taylor_max_degree	integer
initial-multiplier	%initial_multiplier	real
lower-bound-on-multiplier	%lower	real
upper-bound-on-multiplier	%upper	real
stop-normal-case	%stop_normal	real
use-initial-multiplier	%use_initial_multiplier	logical
initialize-approximate-eigenvector	%initialize_approx_eigenvector	real
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
definite-linear-equation-solver	%definite_linear_solver	character
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of `control`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`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`.

2.8 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level = 1`, a single line of output will be produced for each iteration of the process. In the first phase of the algorithm, this will include the current estimate of the multiplier and known brackets on its optimal value. In the second phase, the residual $\|\mathbf{x}\|_S - \Delta$, the current estimate of the multiplier and the size of the correction will be printed. If `control%print_level` ≥ 2 , this output will be increased to provide significant detail of each iteration. This extra output includes times for various phases.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `LLSR_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_RAND`, `GALAHAD_NORMS`, `GALAHAD_ROOTS`, `GALAHAD_SPECFILE`, `GALAHAD_SBLS`, `GALAHAD_SLS`, `GALAHAD_IR` and `GALAHAD_MOP`.

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

Restrictions: $n > 0$, $\Delta > 0$.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

The required solution $\mathbf{x}_*$ necessarily satisfies the optimality condition $\mathbf{A}^T \mathbf{A} \mathbf{x}_* + \lambda_* \mathbf{S} \mathbf{x}_* = \mathbf{A}^T \mathbf{b}$, where $\lambda_* = \sigma \|\mathbf{x}_*\|^{p-2}$. The method is iterative, and proceeds in two phases. Firstly, lower and upper bounds, λ_L and λ_U , on λ_* are computed using Gershgorin's theorems and other eigenvalue bounds, including those that may involve the Cholesky factorization of $\mathbf{S}$. The first phase of the computation proceeds by progressively shrinking the bound interval $[\lambda_L, \lambda_U]$ until a value λ for which $\|\mathbf{x}(\lambda)\|_{\mathbf{M}} \geq \sigma \|\mathbf{x}(\lambda)\|_S^{p-2}$ is found. Here $\mathbf{x}(\lambda)$ and its companion $\mathbf{y}(\lambda)$ are defined to be a solution of

$$(\mathbf{A}^T \mathbf{A} + \lambda \mathbf{S}) \mathbf{x}(\lambda) = \mathbf{A}^T \mathbf{b}. \quad (4.1)$$

Once the terminating λ from the first phase has been discovered, the second phase consists of applying Newton or higher-order iterations to the nonlinear ‘secular’ equation $\lambda = \sigma \|\mathbf{x}(\lambda)\|_S^{p-2}$ with the knowledge that such iterations are both globally and ultimately rapidly convergent.

The dominant cost is the requirement that we solve a sequence of linear systems (4.1). This may be rewritten as

$$\begin{pmatrix} \lambda \mathbf{S} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{x}(\lambda) \\ \mathbf{y}(\lambda) \end{pmatrix} = \begin{pmatrix} \mathbf{A}^T \mathbf{b} \\ \mathbf{0} \end{pmatrix}, \quad (4.2)$$

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

for some auxiliary vector $\mathbf{y}(\lambda)$. In general a sparse symmetric, indefinite factorization of the coefficient matrix of (4.2) is often preferred to a Cholesky factorization of that of (4.1).

Reference: The method is the obvious adaptation to the linear least-squares problem of that described in detail in H. S. Dollar, N. I. M. Gould and D. P. Robinson. On solving trust-region and other regularised subproblems in optimization. *Mathematical Programming Computation* **2(1)** (2010) 21–57.

5 EXAMPLE OF USE

Suppose we wish to solve a problem with $m = 5,000$ residuals and $n = 10,001$ unknowns, whose data is

$$\mathbf{A} = \begin{pmatrix} 1 & & & & 1 & & & & 1 \\ & 1 & . & & & 2 & . & & 1 \\ & . & . & . & & . & . & . & 1 \\ & & . & 1 & & . & m-1 & & 1 \\ & & & & 1 & & & m & 1 \end{pmatrix}, \mathbf{S} = \begin{pmatrix} 1 & & & & \\ & 4 & & & \\ & & . & & \\ & & & (n-1)^2 & \\ & & & & n^2 \end{pmatrix}, \text{ and } \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ . \\ 1 \\ 1 \end{pmatrix},$$

with a weight $\sigma = 0.1$ and regularization power $p = 3$. Then we may use the following code:

```
PROGRAM GALAHAD_LLSR_EXAMPLE ! GALAHAD 4.1 - 2023-06-05 AT 13:15 GMT
USE GALAHAD_LLSR_DOUBLE ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: working = KIND( 1.0D+0 ) ! set precision
REAL ( KIND = working ), PARAMETER :: one = 1.0_working, zero = 0.0_working
INTEGER, PARAMETER :: m = 1000, n = 2 * m + 1 ! problem dimensions
INTEGER :: i, l
REAL ( KIND = working ) :: power = 3.0_working ! cubic regularization
REAL ( KIND = working ) :: weight = 0.1_working ! weight of 1/10th
REAL ( KIND = working ), DIMENSION( n ) :: X
REAL ( KIND = working ), DIMENSION( m ) :: B
TYPE ( SMT_type ) :: A, S
TYPE ( LLSR_data_type ) :: data
TYPE ( LLSR_control_type ) :: control
TYPE ( LLSR_inform_type ) :: inform
CALL LLSR_initialize( data, control, inform ) ! Initialize control parameters
control%sbils_control%symmetric_linear_solver = "sytr "
control%sbils_control%definite_linear_solver = "sytr "
B = one ! The term b is a vector e of ones
A%m = m ; A%n = n ; A%ne = 3 * m ! A = ( I : Diag(1:n) : e )
CALL SMT_put( A%type, 'COORDINATE', i )
ALLOCATE( A%row( A%ne ), A%col( A%ne ), A%val( A%ne ) )
DO i = 1, m
  A%row( i ) = i ; A%col( i ) = i ; A%val( i ) = one
  A%row( m + i ) = i ; A%col( m + i ) = m + i
  A%val( m + i ) = REAL( i, working )
  A%row( 2 * m + i ) = i ; A%col( 2 * m + i ) = n
  A%val( 2 * m + i ) = one
END DO
S%m = n ; S%n = n ; S%ne = n ! S = diag(1:n)**2
CALL SMT_put( S%type, 'DIAGONAL', i )
ALLOCATE( S%val( n ) )
DO i = 1, n
  S%val( i ) = REAL( i * i, working )
```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```

END DO
CALL LLSR_solve( m, n, power, weight, A, B, X, data, control, inform, S = S )
IF ( inform%status == 0 ) THEN ! Successful return
  DO l = 1, A%ne
    i = A%row( l )
    B( i ) = B( i ) - A%val( l ) * X( A%col( l ) )
  END DO
  WRITE( 6, "( ' ||x||_S recurred and calculated = ', 2ES16.8 )" ) &
    inform%x_norm, SQRT( DOT_PRODUCT( X, S%val * X ) )
  WRITE( 6, "( ' ||Ax-b||_2 recurred and calculated = ', 2ES16.8 )" ) &
    inform%r_norm, SQRT( DOT_PRODUCT( B, B ) )
ELSE
  ! Error returns
  WRITE( 6, "( ' LLSR_solve exit status = ', I0 ) " ) inform%status
END IF
CALL LLSR_terminate( data, control, inform ) ! delete workspace
DEALLOCATE( A%row, A%col, A%val, S%val, A%type, S%type )
END PROGRAM GALAHAD_LLSR_EXAMPLE

```

This produces the following output:

```

||x||_S recurred and calculated = 9.57563551E+00 9.57563551E+00
||Ax-b||_2 recurred and calculated = 2.84505117E+01 2.84505117E+01

```