# GALAHAD                                                                      PSLS

## 1  SUMMARY

Given a sparse symmetric matrix $\mathbf{A} = \{a_{ij}\}_{n \times n}$, this package **builds a suitable symmetric, positive definite—or diagonally dominant—preconditioner P** of **A** or a symmetric sub-matrix thereof. The matrix **A** need not be definite. Facilities are provided to apply the preconditioner to a given vector, and to remove rows and columns (symmetrically) from the initial preconditioner without a full re-factorization.

**ATTRIBUTES — Versions:** `GALAHAD_PSLS_single`, `GALAHAD_PSLS_double`. **Calls:** `GALAHAD_CLOCK`, `GALAHAD`-`_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SORT`, `GALAHAD_SPECFILE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_SLS`, `GALA`-`HAD_SCU`, `GALAHAD_EXTEND`, `GALAHAD_NORMS`, `LANCELOT_BAND` and optionally `MC61`, `HSL_MI28` and `ICFS`, **Date:** April 2008. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** GALAHAD_SLS may use OpenMP and its runtime library.

## 2  HOW TO USE THE PACKAGE

### 2.1  Calling sequences

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

*Single precision version*

        USE GALAHAD_PSLS_single

*Double precision version*

        USE GALAHAD_PSLS_double

If it is required to use both modules at the same time, the derived types `SMT_type`, `PSLS_control_type`, `PSLS_time`-`_type`, `PSLS_data_type`, and `PSLS_inform_type` (§2.6), and the subroutines `PSLS_initialize`, `PSLS_form_and_fact`-`orize`, `PSLS_update_factors`, `PSLS_apply`, `PSLS_terminate` (§2.7), must be renamed on one of the `USE` statements. There are five principal subroutines for user calls (see §2.9 for further features):

`PSLS_initialize` must be called to set default values for solver-specific components of the control structure. If non-default values are wanted for any of the control components, the corresponding components should be altered after the call to `PSLS_initialize`.

`PSLS_form_and_factorize` accepts the pattern of **A**, removes any unwanted rows and columns, sets up necessary data structures and then forms and factorizes the required preconditioner, **P**.

`PSLS_update_factors` modifies the preconditioner and its factorization if further rows and columns are removed.

`PSLS_apply` uses the factors generated by `PSLS_factorize` to solve a system of equations $\mathbf{Px} = \mathbf{b}$.

`PSLS_terminate` deallocates the arrays held inside the structure for the factors. It should be called when all the systems involving its matrix have been solved or before another external solver is to be used.

---

| solver | factorization | indefinite **A** | out-of-core | parallelised |
|---|---|---|---|---|
| SILS/MA27 | multifrontal | yes | no | no |
| HSL_MA57 | multifrontal | yes | no | no |
| HSL_MA77 | multifrontal | yes | yes | OpenMP core |
| HSL_MA86 | left-looking | yes | no | OpenMP fully |
| HSL_MA87 | left-looking | no | no | OpenMP fully |
| HSL_MA97 | multifrontal | yes | no | OpenMP core |
| SSIDS | multifrontal | yes | no | CUDA core |
| PARDISO | left-right-looking | yes | no | OpenMP fully |
| MKL_PARDISO | left-right-looking | yes | optionally | OpenMP fully |
| WSMP | left-right-looking | yes | no | OpenMP fully |
| POTR | dense | no | no | with parallel LAPACK |
| SYTR | dense | yes | no | with parallel LAPACK |
| PBTR | dense band | no | no | with parallel LAPACK |

Table 2.1: External solver characteristics.

## 2.2 Supported external solvers

In Table 2.1 we summarize key features of the external solvers supported by PSLS, and used by some preconditioning options. Further details are provided in the references cited in §4 of the documentation to the GALAHAD package SLS. Note that many of the solvers **are not part of the package, and must be obtained separately.**

## 2.3 Matrix storage formats

The matrix **A** may be stored in a variety of input formats.

### 2.3.1 Sparse co-ordinate storage format

Only the nonzero entries of the lower-triangular part of **A** are stored. For the $l$-th entry of the lower-triangular portion of **A**, its row index $i$, column index $j$ and value $a_{ij}$ are stored in the $l$-th components of the integer arrays row, col and real array val, respectively. The order is unimportant, but the total number of entries ne is also required.

### 2.3.2 Sparse row-wise storage format

Again only the nonzero entries of the lower-triangular part are stored, but this time they are ordered so that those in row $i$ appear directly before those in row $i+1$. For the $i$-th row of **A**, the $i$-th component of an integer array ptr holds the position of the first entry in this row, while 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 $l = \text{ptr}(i), \ldots, \text{ptr}(i+1)-1$ of the integer array col, and real array val, respectively.

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

### 2.3.3 Dense storage format

The matrix **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. Since **A** is symmetric, only the lower triangular part (that is the part $a_{ij}$ for $1 \leq j \leq i \leq n$) need be held, and this part will be stored by rows, that is component $i*(i-1)/2+j$ of the storage array val will hold the value $a_{ij}$ (and, by symmetry, $a_{ji}$) for $1 \leq j \leq i \leq n$.

## 2.4 Integer kinds

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

## 2.5 OpenMP

OpenMP may be used by the `GALAHAD_PSLS` package to provide parallelism for some solvers in shared memory environments. 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`.

The code may be compiled and run in serial mode.

## 2.6 The derived data types

Five derived data types are used by the package.

### 2.6.1 The derived data type for holding the matrix

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

n  is a scalar variable of type default `INTEGER`, that holds the order *n* of the matrix **A**. **Restriction:** n $\geq$ 1.

type  is an allocatable array of rank one and type default `CHARACTER`, that indicates the storage scheme used. If the sparse co-ordinate scheme (see §2.3.1) is used the first ten components of `type` must contain the string `COORDINATE`. For the sparse row-wise storage scheme (see §2.3.2), the first fourteen components of `type` must contain the string `SPARSE_BY_ROWS`, and for dense storage scheme (see §2.3.3) the first five components of `type` must contain the string `DENSE`.

   For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `type`. For example, if **A** is to be stored in the structure `A` of derived type `SMT_type` and we wish to use the co-ordinate scheme, we may simply

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

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

ne  is a scalar variable of type default `INTEGER`, that holds the number of entries in the **lower triangular** part of **A** in the sparse co-ordinate storage scheme (see §2.3.1). It need not be set for any of the other three schemes.

val  is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that holds the values of the entries of the **lower triangular** part of the matrix **A** for each of the storage schemes discussed in §2.3. 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 default `INTEGER`, that holds the row indices of the **lower triangular** part of **A** in the sparse co-ordinate storage scheme (see §2.3.1). It need not be allocated for any of the other schemes. Any entry whose row index lies out of the range [1,n] will be ignored.

col  is a rank-one allocatable array variable of type default `INTEGER`, that holds the column indices of the **lower triangular** part of **A** in either the sparse co-ordinate (see §2.3.1), or the sparse row-wise (see §2.3.2) storage scheme. It need not be allocated when the dense storage scheme is used. Any entry whose column index lies out of the range [1,n] will be ignored, while the row and column indices of any entry from the **strict upper triangle** will implicitly be swapped.

ptr  is a rank-one allocatable array of size `n+1` and type default `INTEGER`, that holds the starting position of each row of the **lower triangular** part of **A**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see §2.3.2). It need not be allocated for the other schemes.

---

### 2.6.2 The derived data type for holding control parameters

The derived data type PSLS_control_type is used to hold controlling data. Default values specifically for the desired solver may be obtained by calling PSLS_initialize (see §2.7.1), while components may be changed at run time by calling PSLS_read_specfile (see §2.9.1). The components of PSLS_control_type are:

error  is a scalar variable of type default INTEGER, that holds the unit number for error messages. Printing of error messages is suppressed if error < 0. The default is error = 6.

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

print_level  is a scalar variable of type default INTEGER, that is used to control the amount of informational output that is required. No informational output will occur if print_level ≤ 0. Every increasing values produce more information. The default is print_level = 0.

preconditioner  is a scalar variable of type default INTEGER, that indicates the preconditioner required. Possible values are

    <0  no preconditioning occurs, $\mathbf{P} = \mathbf{I}$

    0  the preconditioner is chosen automatically (forthcoming, and currently defaults to 1).

    1  $\mathbf{A}$ is replaced by the diagonal, $\mathbf{P}$ = diag( max( $\mathbf{A}$, min_diagonal ) ) (see below).

    2  $\mathbf{A}$ is replaced by the band $\mathbf{P}$ = band( $\mathbf{A}$ ) with semi-bandwidth semi_bandwidth (see below).

    3  $\mathbf{A}$ is replaced by the reordered band $\mathbf{P}$ = band( order( $\mathbf{A}$ ) ) with semi-bandwidth semi_bandwidth, where order is chosen by the HSL package MC61 (see §4) to move entries closer to the diagonal.

    4  $\mathbf{P}$ is a full factorization of $\mathbf{A}$ using Schnabel-Eskow modifications, in which small or negative diagonals are made sensibly positive during the factorization.

    5  $\mathbf{P}$ is a full factorization of $\mathbf{A}$ due to Gill, Murray, Ponceléon and Saunders, in which an indefinite factorization is altered to give a positive definite one.

    6  $\mathbf{P}$ is an incomplete Cholesky factorization of $\mathbf{A}$ using the package ICFS due to Lin and Moré (see §4).

    7  $\mathbf{P}$ is an incomplete factorization of $\mathbf{A}$ implemented as HSL_MI28 from HSL (see §4).

    8  $\mathbf{P}$ is an incomplete factorization of $\mathbf{A}$ due to Munskgaard (forthcoming).

    >8  treated as 0.

    **N.B.** Options 3–8 may require additional external software that is not part of the package, and that must be obtained separately. The default is preconditioner = 0.

semi_bandwidth  is a scalar variable of type default INTEGER, that holds the semi-bandwidth used if a band preconditioner (%preconditioner = 2,3) is selected. Any negative value will be regarded as 0. The default is semi_bandwidth = 5.

max_col  is a scalar variable of type default INTEGER, that holds the maximum number of nonzeros in a column of $\mathbf{A}$ that are allowed for a Schur-complement factorization, rather than a refactorization, to accommodate newly deleted columns. The default is max_col = 100.

icfs_vectors  is a scalar variable of type default INTEGER, that holds the number of extra vectors of length $n$ required by the Lin–Moré incomplete Cholesky preconditioner (%preconditioner = 6). Usually, the larger the number, the better the preconditioner, but the more space and effort required to use it. Any negative value will be regarded as 0. The default is icfs_vectors = 10.

---

mi28_lsize  is a scalar variable of type default `INTEGER`, that holds the maximum number of fill entries within each column of the incomplete factor **L** computed by `HSL_MI28` (`%preconditioner = 7`). In general, increasing `mi28_lsize` improves the quality of the preconditioner but increases the time to compute and then apply the preconditioner. Values less than 0 are treated as 0. The default is `mi28_lsize = 10`.

mi28_rsize  is a scalar variable of type default `INTEGER`, that holds the maximum number of entries within each column of the strictly lower triangular matrix **R** used in the computation of the preconditioner by `HSL_MI28` (`%preconditioner = 7`). Rank-1 arrays of size `mi28_rsize` times *n* are allocated internally to hold **R**. Thus the amount of memory used, as well as the amount of work involved in computing the preconditioner, depends on `mi28_rsize`. Setting `mi28_rsize` > 0 generally leads to a higher quality preconditioner than using `mi28_rsize` = 0, and choosing `mi28_rsize` ≥ `mi28_lsize` is generally recommended. The default is `mi28_rsize = 10`.

max_col  is a scalar variable of type default `INTEGER`, that specifies Any non-positive value will be regarded as 1. The default is `max_col = 100`.

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, the default `prefix = ""` should be used.

min_diagonal  is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that specifies the smallest permitted diagonal in **P** for some of the preconditioners provided. See `preconditioner` above. The default is `min_diagonal = 0.00001`.

new_structure  is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the storage structure for the input matrix has changed, and `.FALSE.` if only the values have changed. The default is `new_structure = .TRUE.`.

get_semi_bandwidth  is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes the package to calculate the semi-bandwidth of the preconditioner, **P** and `.FALSE.` otherwise. The default is `get_semi_bandwidth = .TRUE.`.

get_norm_residual  is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes the package to return the value of the norm of the residuals for the computed solution when applying the preconditioner and `.FALSE.` otherwise. The default is `get_norm_residual = .FALSE.`.

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

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

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 when `%precondit-ioner = 4,5`. Possible choices are `'sils'`, `'ma27'`, `'ma57'`, `'ma77'`, `'ma86'`, `'ma87'`, `'ma97'`, `'ssids'`, `'pardiso'` `'mkl_pardiso'`, `'wsmp'`, `'potr'` and `'pbtr'`, although only `'sils'`, `'potr'`, `'pbtr'` and, for OMP 4.0-compliant compilers, `'ssids'` are installed by default. See Table 2.1 and 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 = ""`.

---

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

### 2.6.3   The derived data type for holding timing information

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

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

`analyse` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the CPU time spent analysing the matrix structure prior to building and factorizing the preconditioner.

`factorize` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the CPU time spent building and factorizing the preconditioner.

`solve` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the CPU time spent applying the preconditioner.

`update` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the CPU time spent updating existing factorizations.

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

`clock_analyse` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the elapsed system clock time spent analysing the matrix structure prior to building and factorizing the preconditioner.

`clock_factorize` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the elapsed system clock time spent building and factorizing the preconditioner.

`clock_solve` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the elapsed system clock time spent applying the preconditioner.

`clock_update` is a scalar variable of type default `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the elapsed system clock time spent updating existing factorizations.

### 2.6.4   The derived data type for holding informational parameters

The derived data type `PSLS_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `PSLS_inform_type` are as follows—any component that is not relevant to the solver being used will have the value -1 or -1.0 as appropriate:

`status` is a scalar variable of type default `INTEGER`, that gives the exit status of the algorithm. See §2.8 for details.

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

`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 there have been no allocation or deallocation errors.

`preconditioner` is a scalar variable of type default `INTEGER`, that indicates the preconditioner method used. The range of values returned corresponds to those requested in `control%preconditioner`, excepting that the requested value may have been altered to a more appropriate one during the factorization. In particular, if the automatic choice `control%preconditioner = 0` is requested, `preconditioner` reports the actual choice made.

---

`semi_bandwidth` is a scalar variable of type default `INTEGER`, that indicates the actual semi-bandwith.

`reordered_semi_bandwidth` is a scalar variable of type default `INTEGER`, that indicates the semi-bandwith used after reordering.

`semi_bandwidth_used` is a scalar variable of type default `INTEGER`, that indicates the actual semi-bandwith used.

`out_of_range` is a scalar variable of type default `INTEGER`, that is set to the number of entries of **A** supplied with one or both indices out of range.

`duplicates` is a scalar variable of type default `INTEGER`, that is set to the number of duplicate off-diagonal entries of **A** supplied.

`upper` is a scalar variable of type default `INTEGER`, that is set to the number of input entries from the strict upper triangle of **A**.

`missing_diagonals` is a scalar variable of type default `INTEGER`, that gives the number of diagonal entries entries missing for an allegedly-definite matrix **A**.

`neg1` is a scalar variable of type default `INTEGER`, that gives the number of 1 by 1 pivots present in the factorization of **A**.

`neg2` is a scalar variable of type default `INTEGER`, that gives the number of 2 by 2 pivots present in the factorization of **A**.

`perturbed` is a scalar variable of type default `LOGICAL`, that is /true/ if the factorization has been perturbed when building the preconditioner.

`fill_in_ratio` is a scalar variable of type `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the ratio of the fill-ins during factorization to the original numbers of non-zeros of **A**.

`norm_residual` is a scalar variable of type `REAL` (double precision in `GALAHAD_PSLS_double`), that gives the norm of the residual $\|\mathbf{Px} - \mathbf{b}\|$ when solving $\mathbf{Px} = \mathbf{b}$.

`mc61_info` is an array of size 10 and type `INTEGER`, that corresponds to the output array `INFO` from `MC61`. See the HSL documentation for `MC61` for further details.

`mc61_rinfo` is an array of size 15 and type `REAL` (double precision in `GALAHAD_PSLS_double`), that corresponds to the output array `RINFO` from `MC61`. See the HSL documentation for `MC61` for further details.

`time` is a scalar variable of type `PSLS_time_type` whose components are used to hold elapsed CPU and system clock times (in seconds) for the various parts of the calculation (see Section 2.6.3).

`sls_inform` is a scalar variable of type `sls_inform_type`, that corresponds to the output value `sls_inform` from the GALAHAD package `SLS`. See the documentation for `SLS` for further details.

`mi28_info` is a scalar variable of type `mi28_info`, that corresponds to the output value `mi28_info` from `HSL_MI28`. See the documentation for `HSL_MI28` for further details.

### 2.6.5 The derived data type for holding problem data

The derived data type `PSLS_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls to `PSLS` procedures. All components are private.

### 2.7 Argument lists and calling sequences

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

---

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

### 2.7.1 The initialization subroutine

The initialization subroutine must be called for each solver used to initialize data and solver-specific control parameters.

        CALL PSLS_initialize( data, control, inform )

data is a scalar INTENT(OUT) argument of type PSLS_data_type (see §2.6.5). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type PSLS_control_type (see §2.6.2). On exit, control contains solver-specific default values for the components as described in §2.6.2. These values should only be changed after calling PSLS_initialize.

inform is a scalar INTENT(OUT) argument of type PSLS_inform_type (see §2.6.4). A successful call is indicated when the component status has the value 0. For other return values of status, see §2.8.

### 2.7.2 The subroutine for constructing the preconditioner

This subroutine assembles and factorizes the required preconditioner from the input matrix **A** as follows:

        CALL PSLS_form_and_factorize( matrix, data, control, inform[, SUB] )

matrix is scalar INTENT(IN) argument of type SMT_type that is used to specify **A**. The user must set all of the relevant components of matrix according to the storage scheme desired (see §2.6.1). Incorrectly-set components will result in errors flagged in inform%status, see §2.8.

data is a scalar INTENT(INOUT) argument of type PSLS_data_type (see §2.6.5). It is used to hold the factors of the preconditioner and other data concerning the matrix used. It must have been initialized by a call to PSLS_initialize.

control is scalar INTENT(IN) argument of type PSLS_control_type. Its components control the action of the analysis phase, as explained in §2.6.2.

inform is a scalar INTENT(INOUT) argument of type PSLS_inform_type (see §2.6.4). A successful call is indicated when the component status has the value 0. For other return values of status, see §2.8.

SUB is an OPTIONAL INTENT(IN) rank-one default assumed-size INTEGER argument that may be used to provide a set of distinct indices that specify the rows (and, by symmetry, columns) of **A** that are to be considered. Each component of SUB must lie between 1 and n, and they should be input in increasing order. If SUB is not supplied, all indices $1 \leq i \leq n$ will be used.

### 2.7.3 The subroutine for updating the preconditioner

This subroutine updates the factorization of the preconditioner when a subset of the rows (and, by symmetry, columns) are removed.

        CALL PSLS_update_factors( DEL, data, control, inform )

DEL is an INTENT(IN) rank-one default assumed-size INTEGER argument whose indices are those of rows (and columns) of **A** that are to be deleted. Each component of DEL must lie between 1 and n.

data is a scalar INTENT(INOUT) argument of type PSLS_data_type (see §2.6.5). It is used to hold the factors of the updated preconditioner and other data concerning the matrix used. It must have been previously set by a call to PSLS_form_and_factorize.

---

control is scalar `INTENT(IN)` argument of type `PSLS_control_type`. Its components control the action of the analysis phase, as explained in §2.6.2.

inform is a scalar `INTENT(INOUT)` argument of type `PSLS_inform_type` (see §2.6.4). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

### 2.7.4 The subroutine for applying the preconditioner

Given the preconditioner **P**, a set of equations **Px** = **b** may be solved as follows:

> CALL PSLS_apply( X, data, control, inform )

X is an `INTENT(INOUT)` assumed-shape array argument of rank 1 and of type `REAL` (double precision in GALAH-AD_PSLS_double). On entry, X must be set to the vector **b**, and on successful return it holds the solution **x**. The i-th component of **b** and the resulting i-th component of the solution **x** occupy the i-th component of X. Any component corresponding to rows/columns not in the initial subset recorded by `PSLS_form_and_factorize`, or in those subsequently deleted by `PSLS_update_factors`, will not be altered.

data is a scalar `INTENT(INOUT)` argument of type `PSLS_data_type` (see §2.6.5). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to `PSLS_initialize` and not altered by the user in the interim.

control is scalar `INTENT(IN)` argument of type `PSLS_control_type`. Its components control the action of the solve phase, as explained in §2.6.2.

inform is a scalar `INTENT(INOUT)` argument of type `PSLS_inform_type` (see §2.6.4). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

### 2.7.5 The termination subroutine

All previously allocated internal arrays are deallocated and OpenMP locks destroyed as follows:

> CALL PSLS_terminate( data, control, inform )

data is a scalar `INTENT(INOUT)` argument of type `PSLS_data_type` (see §2.6.5). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to `PSLS_initialize` and not altered by the user in the interim. On exit, its allocatable array components will have been deallocated.

control is scalar `INTENT(IN)` argument of type `PSLS_control_type`. Its components control the action of the termination phase, as explained in §2.6.2.

inform is a scalar `INTENT(INOUT)` argument of type `PSLS_inform_type` (see §2.6.4). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

### 2.8 Warning and error messages

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

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

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

---

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

-3 One of the restrictions `matrix%n` > 0 or `matrix%ne` < 0, for co-ordinate entry, or requirements that `matrix%type` contain its relevant string `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'DENSE'` has been violated.

-20 The matrix is not positive definite while the solver used expected it to be.

-9 The solver required by `SLS` reported an error during its analysis phase. See `inform%SLS_inform%status` for more details.

-10 The solver required by `SLS` reported an error during its factorization phase. See `inform%SLS_inform%status` for more details.

-26 The requested factorization solver is not available.

-29 A specified option is not available with this solver.

-45 The requested preconditioner is not available.

-80 An error occurred when calling `HSL_MI28`. See `inform%mi28_info%stat` for more details.

## 2.9 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `PSLS_control_type` (see §2.6.2), by reading an appropriate data specification file using the subroutine `PSLS_read_specfile`. This facility is useful as it allows a user to change `PSLS` control parameters without editing and recompiling programs that call `PSLS`.

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", that 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 specification file is limited to 80 characters, including the blanks separating keyword and value.

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

```
( .. lines ignored by SLS_read_specfile .. )
  BEGIN SLS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by SLS_read_specfile .. )
```

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

and

```
  END SLS SPECIFICATION
```

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

are acceptable. Furthermore, between the "`BEGIN PSLS`" 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 way to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameter may be of three different types, namely integer, character or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively).

The specification file must be open for input when `PSLS_read_specfile` is called, and the associated unit number passed to the routine in `device` (see below). Note that the corresponding file is rewound, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `PSLS_read_specfile`.

### 2.9.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

        CALL SLS_read_specfile( control, device )

`control` is a scalar `INTENT(INOUT)` argument of type `PSLS_control_type` (see §2.6.2). Default values should have already been set, perhaps by calling `PSLS_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 §2.6.2) of `control` that each affects are given in Table 2.2.

| command | component of `control` | value type |
| --- | --- | --- |
| error-printout-device | %error | integer |
| printout-device | %out | integer |
| print-level | %print_level | integer |
| preconditioner-used | %preconditioner | integer |
| semi-bandwidth-for-band-preconditioner | %semi_bandwidth | integer |
| maximum-column-nonzeros-in-schur-complement | %max_col | integer |
| number-of-lin-more-vectors | %icfs_vectors | integer |
| mi28-l-fill-size | %mi28_lsize | integer |
| mi28-r-entry-size | %mi28_rsize | integer |
| minimum-diagonal | %min_diagonal | real |
| new-structure | %new_structure | logical |
| get-semi-bandwidth | %get_semi_bandwidth | logical |
| get-norm-residual | %get_norm_residual | logical |
| 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 2.2: Specfile commands and associated components of `control`.

`device` is a scalar `INTENT(IN)` argument of type default `INTEGER`, that must be set to the unit number on which the specification file 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`.

## 3 GENERAL INFORMATION

**Workspace:** Provided automatically by the module.

**Other modules used directly:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE_single/double, GALAHAD_SORT_single/double, GALAHAD_SPECFILE_single/double, GALAHAD_SMT_single/double, GALAHAD_QPT_single/double, GALAHAD_SLS_single/double, GALAHAD_SCU_single/double, GALAHAD_EXTEND_single/double, GALAHAD_NORMS_single/double, LANCELOT_BAND_single/double.

**Other routines called directly:** Optionally MC61, HSL_MI28 and ICFS, plus those called by GALAHAD_SLS.

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

**Restrictions:** matrix%n $\geq 1$, matrix%ne $\geq 0$ if matrix%type = 'COORDINATE', and matrix%type is one of 'COORDINATE', 'SPARSE_BY_ROWS' or 'DENSE'.

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

## 4 METHOD

The basic preconditioners are described in detail in Section 3.3.10 of

A. R. Conn, N. I. M. Gould and Ph. L. Toint (1992). LANCELOT. A fortran package for large-scale nonlinear optimization (release A). Springer Verlag Series in Computational Mathematics 17, Berlin,

along with the more modern versions implements in ICFS due to

C.-J. Lin and J. J. Moré (1999). Incomplete Cholesky factorizations with limited memory. SIAM Journal on Scientific Computing **21** 21-45,

and in HSL_MI28 described by

J. A. Scott and M. Tůma (2013). HSL MI28: an efficient and robust limited-memory incomplete Cholesky factorization code. ACM Transactions on Mathematical Software **40(4)** (2014), Article 24.

The factorization methods used by the GALAHAD package SLS in conjunction with some preconditioners are described in the documentation to that package. Orderings to reduce the bandwidth, as implemented in HSL's MC61, are due to

J. K. Reid and J. A. Scott (1999) Ordering symmetric sparse matrices for small profile and wavefront International Journal for Numerical Methods in Engineering **45** 1737-1755.

If a subset of the rows and columns are specified, the remaining rows/columns are removed before processing. Any subsequent removal of rows and columns is achieved using the GALAHAD Schur-complement updating package SCU unless a complete re-factorization is likely more efficient.

## 5 EXAMPLE OF USE

We illustrate the use of the package on the (indefinite) sparse matrix

$$\begin{pmatrix} 2 & 3 & & & \\ 3 & & 4 & & 6 \\ & 4 & 1 & 5 & \\ & & 5 & & \\ 6 & & & & 1 \end{pmatrix}$$

(Note that this example does not illustrate all the facilities). Then, choosing the solver SILS, we may use the following code to find a tridiagonal (banded, with a semi-bandwidth of 1) preconditioner and apply it to a given vector:

```
PROGRAM PSLS_EXAMPLE   !  GALAHAD 4.0 - 2022-01-24 AT 09:15 GMT.
USE GALAHAD_PSLS_double
IMPLICIT NONE
```

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

```
      INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
      TYPE ( SMT_type ) :: matrix
      TYPE ( PSLS_data_type ) :: data
      TYPE ( PSLS_control_type ) control
      TYPE ( PSLS_inform_type ) :: inform
      INTEGER, PARAMETER :: n = 5
      INTEGER, PARAMETER :: ne = 7
      REAL ( KIND = wp ) :: X( n )
      INTEGER :: i, s
! allocate and set lower triangle of matrix in co-ordinate form
      CALL SMT_put( matrix%type, 'COORDINATE', s )
      matrix%n = n ; matrix%ne = ne
      ALLOCATE( matrix%val( ne ), matrix%row( ne ), matrix%col( ne ) )
      matrix%row = (/ 1, 2, 3, 3, 4, 5, 5 /)
      matrix%col = (/ 1, 1, 2, 3, 3, 2, 5 /)
      matrix%val = (/ 2.0_wp, 3.0_wp, 4.0_wp, 1.0_wp, 5.0_wp, 6.0_wp, 1.0_wp /)
! problem setup complete
! specify the solver used by SLS (in this case sils)
      CALL PSLS_initialize( data, control, inform )
      control%preconditioner = 2  ! band preconditioner
      control%semi_bandwidth = 1  ! semi-bandwidth of one
      control%definite_linear_solver = 'sils'
! form and factorize the preconditioner, P
      CALL PSLS_form_and_factorize( matrix, data, control, inform )
      IF ( inform%status < 0 ) THEN
        WRITE( 6, '( A, I0 )' )                                              &
            ' Failure of PSLS_form_and_factorize with status = ', inform%status
        STOP
      END IF
! use the factors to solve P x = b, with b input in x
      X( : n ) = (/ 8.0_wp, 45.0_wp, 31.0_wp, 15.0_wp, 17.0_wp /)
      CALL PSLS_apply( X, data, control, inform )
      IF ( inform%status == 0 ) THEN
        WRITE( 6, "( ' PSLS - Preconditioned solution is ', 5F6.2 )" ) X
      ELSE
        WRITE( 6, "( ' PSLS - exit status = ', I0 )" ) inform%status
      END IF
! clean up
      CALL PSLS_terminate( data, control, inform )
      DEALLOCATE( matrix%type, matrix%val, matrix%row, matrix%col )
      STOP
      END PROGRAM PSLS_EXAMPLE
```

This produces the following output:

```
 PSLS - Preconditioned solution is  -6.03  8.03 -4.66  4.20 17.00
```

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

```
! allocate and set lower triangle of matrix in co-ordinate form
...
! problem setup complete
```

by

```
! allocate and set lower triangle of matrix in spares row form
      CALL SMT_put( matrix%type, 'SPARSE_BY_ROWS', s )
```

---

```
    matrix%n = n
    ALLOCATE( matrix%val( ne ), matrix%col( ne ), matrix%ptr( n + 1 ) )
    matrix%ptr = (/ 1, 2, 3, 5, 6, 8 /)
    matrix%col = (/ 1, 1, 2, 3, 3, 2, 5 /)
    matrix%val = (/ 2.0_wp, 3.0_wp, 4.0_wp, 1.0_wp, 5.0_wp, 6.0_wp, 1.0_wp /)
! problem setup complete
```

or using a dense storage format with the replacement lines

```
! allocate and set lower triangle of matrix in dense form
    CALL SMT_put( matrix%type, 'DENSE', s )
    matrix%n = n
    ALLOCATE( matrix%val( n * ( n + 1 ) / 2 ) )
    matrix%val = (/ 2.0_wp, 3.0_wp, 0.0_wp, 0.0_wp, 4.0_wp, 1.0_wp,          &
                    0.0_wp, 0.0_wp, 5.0_wp, 0.0_wp, 0.0_wp, 6.0_wp,          &
                    0.0_wp, 0.0_wp, 1.0_wp /)
! problem setup complete
```

respectively.