



GALAHAD

SLS

1 SUMMARY

This package **solves dense or sparse symmetric systems of linear equations** using variants of Gaussian elimination. Given a sparse symmetric matrix $\mathbf{A} = \{a_{ij}\}_{n \times n}$, and an n -vector \mathbf{b} or a matrix $\mathbf{B} = \{b_{ij}\}_{n \times r}$, this subroutine solves the system $\mathbf{Ax} = \mathbf{b}$ or the system $\mathbf{AX} = \mathbf{B}$. The matrix \mathbf{A} need not be definite.

The method provides a common interface to a variety of well-known solvers from HSL and elsewhere. Currently supported solvers include MA27/SILS, HSL_MA57, HSL_MA77, HSL_MA86, HSL_MA87 and HSL_MA97 from HSL, SSIDS from SPRAL, MUMPS from Mumps Technologies, PARDISO both from the Pardiso Project and Intel's MKL, PaStiX from Inria, and WSMP from the IBM alpha Works, as well as POTR, SYTR and SBTR from LAPACK. Note that, with the exception of SSIDS and the Netlib reference LAPACK codes, **the solvers themselves do not form part of this package and must be obtained separately**. Dummy instances are provided for solvers that are unavailable. Also note that additional flexibility may be obtained by calling the solvers directly rather than via this package.

ATTRIBUTES — Versions: GALAHAD_SLS_single, GALAHAD_SLS_double. **Calls:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SORT, GALAHAD_SPACE, GALAHAD_SPECFILE, GALAHAD_STRING, GALAHAD_SMT, GALAHAD_SILS, GALAHAD_NODEND and optionally HSL_MA57, HSL_MA77, HSL_MA86, HSL_MA87, HSL_MA97, HSL_MC64, HSL_MC68, MC61, MC77, and METIS. **Date:** August 2009. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some solvers may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

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_SLS_single
```

with the obvious substitution `GALAHAD_SLS_double`, `GALAHAD_SLS_quadruple`, `GALAHAD_SLS_single_64`, `GALAHAD_SLS_double_64` and `GALAHAD_SLS_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`, `SLS_control_type`, `SLS_time_type`, `SLS_data_type`, and `SLS_inform_type` (§2.6), and the subroutines `SLS_initialize`, `SLS_analyse`, `SLS_factorize`, `SLS_solve`, `SLS_terminate` (§2.7), `SLS_enquire`, `SLS_alter_d`, and `SLS_part_solve` (§2.9) must be renamed on one of the `USE` statements.

There are five principal subroutines for user calls (see §2.9 for further features):

The subroutine `SLS_initialize` must be called to specify the external solver to be used. It may also 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 `SLS_initialize`.

`SLS_analyse` accepts the pattern of \mathbf{A} and optionally chooses pivots for Gaussian elimination using a selection criterion to preserve sparsity. It subsequently constructs subsidiary information for actual factorization by `SLS_factorize`. If the user provides the pivot sequence, only the necessary information for `SLS_factorize` will be generated.

`SLS_factorize` factorizes the matrix \mathbf{A} using the information from a previous call to `SLS_analyse`. The actual pivot sequence used may differ from that of `SLS_analyse` if \mathbf{A} is not definite.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`SLS_solve` uses the factors generated by `SLS_factorize` to solve a system of equations with one ($\mathbf{Ax} = \mathbf{b}$) or several ($\mathbf{AX} = \mathbf{B}$) right-hand sides. Iterative refinement may be used to improve a given solution or set of solutions.

`SLS_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.

2.2 Supported external solvers

In Table 2.1 we summarize key features of the external solvers supported by SLS. Further details are provided in the references cited in §4.

solver	factorization	indefinite \mathbf{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
MUMPS	multifrontal	yes	optionally	MPI
PARDISO	left-right-looking	yes	no	OpenMP fully
MKL-PARDISO	left-right-looking	yes	optionally	OpenMP fully
PaStiX	left-right-looking	yes	no	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.

2.3 Matrix storage formats

The matrix \mathbf{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 \mathbf{A} are stored. For the l -th entry of the lower-triangular portion 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 `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 \mathbf{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 $l = \text{ptr}(i), \dots, \text{ptr}(i + 1) - 1$ of the integer array `col`, and real array `val`, respectively.

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

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

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 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.

2.5 Parallel usage

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

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

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

The code may be compiled and run in serial mode.

2.6 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 `INTEGER(ip_)`, 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

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```
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 `INTEGER(ip_)`, 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 `REAL(rp_)`, 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 `INTEGER(ip_)`, 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 `INTEGER(ip_)`, 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 `INTEGER(ip_)`, 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.

Although any of the above-mentioned matrix storage formats may be used with each supported solver, MA27/SILS and HSL_MA57 from HSL are most efficient if co-ordinate input is provided.

2.6.2 The derived data type for holding control parameters

The derived data type `SLS_control_type` is used to hold controlling data. Default values specifically for the desired solver may be obtained by calling `SLS_initialize` (see §2.7.1), while components may be changed at run time by calling `SLS_read_specfile` (see §2.10.1). The components of `SLS_control_type` are:

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

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

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

`statistics` is a scalar variable of type `INTEGER(ip_)`, that holds the unit number for statistical output, if any. Printing of statistical messages is suppressed if `statistics < 0`. The default is `statistics = 0`.

`print_level` is a scalar variable of type `INTEGER(ip_)`, that is used to control the amount of informational output that 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 step of iterative refinement performed. The default is `print_level = 0`.

`print_level_solver` is a scalar variable of type `INTEGER(ip_)`, that is used to control the amount of informational output that is required by the external solver. No informational output will occur if `print_level ≤ 0`. If `print_level ≥ 1` the amount of output produced is solver dependent. The default is `print_level_solver = 0`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`bits` is a scalar variable of type `INTEGER(ip_)`, that indicates the machine architecture being used. It should be set to 32 on a 32-bit architecture and to 64 on a 64-bit architecture. The default value is 32, and this default is used if `bits` \neq 64.

`block_size_kernel` is a scalar variable of type `INTEGER(ip_)`, that gives the target block size for the kernel factorization, if any. The default value is 40, and this default is used if `block_size_kernel` $<$ 1.

`block_size_elimination` is a scalar variable of type `INTEGER(ip_)`, that gives the target block size for parallel factorization, if any. The default is `block_size_elimination` = 32, and this default is used if `block_size_elimination` $<$ 1.

`blas_block_size_factorize` is a scalar variable of type `INTEGER(ip_)`, that gives the block size for level-three basic linear algebra subprograms (BLAS) in the factorization phase. The default is `blas_block_size_factorize` = 16, and this default is used if `blas_block_size_factorize` $<$ 1.

`blas_block_size_solve` is a scalar variable of type `INTEGER(ip_)`, that gives the block size for level-two and -three basic linear algebra subprograms (BLAS) in the solution phase. The default is `blas_block_size_solve` = 16, and this default is used if `blas_block_size_solve` $<$ 1.

`node_amalgamation` is a scalar variable of type `INTEGER(ip_)`, that controls node amalgamation. Two neighbours in the elimination tree are merged if they both involve fewer than `node_amalgamation` eliminations. The default is `node_amalgamation`=32, and this default is used if `node_amalgamation` $<$ 1.

`initial_pool_size` is a scalar variable of type `INTEGER(ip_)`, that holds the initial size of the arrays that store the task pool for parallel factorization, if any. The default is `initial_pool_size` = 100000, and this default is used if `initial_pool_size` $<$ 1.

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

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

`max_real_factor_size` is a scalar variable of type `INTEGER(int64)`, that specifies the maximum amount of real storage that will be allocated for the factors and other data. The default is `max_real_factor_size` = `HUGE(0)`.

`max_integer_factor_size` is a scalar variable of type `INTEGER(int64)`, that specifies the maximum amount of integer storage that will be allocated for the factors and other data. The default is `max_integer_factor_size` = `HUGE(0)`.

`max_in_core_store` is a scalar variable of type `INTEGER(int64)`, that specifies the maximum amount of storage (measured in Fortran storage units) to be used if the user wants to use in-core arrays when possible in place of out-of-core direct-access files for solvers that support out-of-core factorization. If `max_in_core_store` = 0, factorization will occur entirely out-of-core. The default is `max_in_core_store` = `HUGE(0)/4` (`HUGE(0)/8` in `GALAHAD_SLS_double`) for 32-bit architectures and `max_in_core_store` = `HUGE(0_long)/4` (`HUGE(0_long)/8` in `GALAHAD_SLS_double`) in the 64-bit case.

`array_increase_factor` is a scalar variable of type `REAL(rp_)`, that holds the factor by which arrays sizes are to be increased if they are too small. The default is `array_increase_factor` = 2.0.

`array_decrease_factor` is a scalar variable of type `REAL(rp_)`, that holds a factor that is used to assess whether previously allocated internal workspace arrays are excessive. In particular, if current requirements are less than `array_decrease_factor` times the currently allocated space, the space will be re-allocated to current requirements. The default is `array_decrease_factor` = 2.0.

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

`pivot_control` is a scalar variable of type `INTEGER(ip_)`, that is used to control numerical pivoting by `SLS_factorize`. Possible values are:

- 1 Numerical pivoting will be performed, with relative pivot tolerance given by the component `relative_pivot_tolerance`.
- 2 No pivoting will be performed and an error exit will occur immediately a sign change is detected among the pivots. This is suitable for cases when **A** is thought to be definite and is likely to decrease the factorization time while still providing a stable decomposition.
- 3 No pivoting will be performed and an error exit will occur if a zero pivot is detected. This is likely to decrease the factorization time, but may be unstable if there is a sign change among the pivots or a tiny pivot is encountered.
- 4 No pivoting will be performed but the matrix will be altered to ensure that the result is “sufficiently” positive definite if a non-positive pivot is encountered.

The default is `pivot_control = 1`, and any value outside of `[1, 4]` will be reset to the default.

`ordering` is a scalar variable of type `INTEGER(ip_)`, that controls the initial order of the rows when performing the factorization. This will be ignored if an explicit permutation is specified (see the argument `PERM` for `SLS_analyse`). Possible values are:

- < 0 The ordering will be chosen by the specified solver with its own ordering-selection value – `ordering`.
- 0 The ordering will be chosen to be the default for the solver used. If the specified solver has no default, the rows will be unordered which is unlikely to be satisfactory.
- 1 The ordering will be chosen by the Approximate Minimum Degree method with provisions for “dense” rows/columns.
- 2 The ordering will be chosen by the Minimum Degree method.
- 3 The ordering will be chosen by the Nested Dissection method using the GALAHAD package `NODEND`.
- 4 An indefinite ordering involving a combination of 1x1 and 2x2 pivots will be used.
- 5 An ordering that aims to provide a small profile or wavefront will be used.
- 6 An ordering that aims to provide a small bandwidth will be used.
- 7 The ordering will be chosen by an earlier implementation of the Approximate Minimum Degree method with no provisions for “dense” rows/columns.
- > 7 The ordering will be chosen automatically depending on matrix characteristics (not yet implemented).

Values 1 to 4 are only supported if the HSL package `HSL_MC68` is available, while 5 and 6 are only possible when the HSL package `MC61` is provided. The default is `ordering = 0`, and currently any value `> 7` will be reset to this default.

`full_row_threshold` is a scalar variable of type `INTEGER(ip_)`, that controls the threshold for detecting rows with a large percentage (relative to the matrix order) of nonzeros by `SLS_analyse`. Such rows will normally be ordered last when the factorization occurs. If `full_row_threshold = 100`, only fully dense rows will be detected. The default is `full_row_threshold = 100`, and any value outside of `[0, 100]` will be reset to the default.

`row_search_indefinite` is a scalar variable of type `INTEGER(ip_)`, that controls the maximum number of rows searched for a pivot when using the indefinite ordering (see `ordering = 4` above). The default is `row_search_indefinite = 10`, and this default is used if `row_search_indefinite < 1`.

`scaling` is a scalar variable of type `INTEGER(ip_)`, that may control scaling of the matrix. Possible values are:

- < 0 The scaling will be chosen by the specified solver with its own scaling-selection value – `scaling`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- 0 No scaling is used.
- 1 Scale the matrix so that the diagonal and off diagonal entries of the result are one and smaller than one, respectively, in absolute value using the package HSL_MC64.
- 2 Scale the matrix so that ℓ_1 -norms of each column of the result are approximately one using the package MC77.
- 3 Scale the matrix so that ℓ_∞ -norms of each column of the result are approximately one using the package MC77.
- 4 Use the default for the specified solver, or 0 (no scaling) if there is no default.

This option is not currently available for all solvers; -1 is only available for HSL_MA57, -2 for HSL_MA77 and -3 to -1 for HSL_MA97. The default is `scaling = 0`, and any value larger than 4 will be reset to the default.

`scale_maxit` is a scalar variable of type `INTEGER(ip_)`, that controls the maximum number of iterations performed by MC77 when scaling is requested (see `scale` above). The default is `scale_maxit = 10`, and this default is used if `scale_maxit < 1`.

`scale_thresh` is a scalar variable of type `REAL(rp_)`, that is used by MC77 to terminate the scaling iteration. The iteration stops as soon as the difference between every column norm and one is smaller than `scale_thresh` in magnitude. The default is `scale_thresh = 0.1`.

`relative_pivot_tolerance` is a scalar variable of type `REAL(rp_)`, that holds the relative pivot tolerance that is used to control the stability of the factorization of indefinite **A**. The default is `relative_pivot_tolerance = 0.01`. For problems requiring greater than average numerical care a higher value than the default would be advisable. Values greater than 0.5 are treated as 0.5 for all solvers except HSL_MA77 (where an upper bound 1.0 is enforced) and less than 0.0 as 0.0.

`minimum_pivot_tolerance` is a scalar variable of type `REAL(rp_)`, that holds the minimum permitted value of the relative pivot tolerance. If, at any stage of the computation, fewer than the expected number of stable pivots have been found using the current tolerance but the largest candidate pivot would be acceptable with tolerance `minimum_pivot_tolerance`, the pivot is accepted, and all subsequent pivots will be assessed relative to `minimum_pivot_tolerance` rather than `relative_pivot_tolerance`. The default is `minimum_pivot_tolerance=0.01`. Values of `minimum_pivot_tolerance` greater than `relative_pivot_tolerance` are treated as `relative_pivot_tolerance`, while values less than 0 are treated as 0.

`absolute_pivot_tolerance` is a scalar variable of type `REAL(rp_)`, that holds the absolute pivot tolerance that is used to control the stability of the factorization of indefinite **A**. No pivot smaller than `absolute_pivot_tolerance` in absolute value will be accepted. The default is `absolute_pivot_tolerance = EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_SLS_double)`.

`zero_tolerance` is a scalar variable of type `REAL(rp_)`, that controls which small entries are to be ignored during the factorization of **A**. Any entry smaller in absolute value than `zero_tolerance` will be treated as zero; as a consequence when `zero_tolerance > 0`, the factors produced will be of a perturbation of order `zero_tolerance` of **A**. The default is `zero_tolerance = 0.0`.

`static_pivot_tolerance` and `static_level_switch` are scalar variables of type `REAL(rp_)`, that are used to set the static pivot level when the solvers HSL_MA57 or HSL_MA77 are used. If `static_pivot_tolerance > 0.0` and if, at any stage of the computation, relatively fewer than `static_level_switch` pivots can be found with relative pivot tolerance greater than `minimum_pivot_tolerance`, diagonal entries are accepted as pivots. If a candidate diagonal entry has absolute value at least `static_pivot_tolerance`, it is selected as a pivot; otherwise, the pivot is given the value that has the same sign but absolute value `static_pivot_tolerance`. The defaults are `static_pivot_tolerance = 0.0` and `static_level_switch = 0.0`. If `static_pivot_tolerance` is larger than zero, but smaller than `zero_tolerance`, the value `zero_tolerance` will be used.

`consistency_tolerance` is a scalar variable of type `REAL(rp_)`, that holds the tolerance used to access whether a singular system is consistent or not. The default is `consistency_tolerance = EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_SLS_double)`.

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

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

`acceptable_residual_relative` and `acceptable_residual_absolute` are scalar variables of type `REAL(rp_)`, that specify an acceptable level for the residual $\mathbf{Ax} - \mathbf{b}$ or residuals $\mathbf{Ax}_i - \mathbf{b}_i$, $i = 1, \dots, r$, when there are more than one. In particular, iterative refinement will cease as soon as $\|\mathbf{Ax} - \mathbf{b}\|_\infty$ falls below $\max(\|\mathbf{b}\|_\infty * \text{acceptable_residual_relative}, \text{acceptable_residual_absolute})$; for the multiple residual case, we require that $\|\mathbf{Ax}_i - \mathbf{b}_i\|_\infty$ falls below $\max(\|\mathbf{b}_i\|_\infty * \text{acceptable_residual_relative}, \text{acceptable_residual_absolute})$, for each $i = 1, \dots, r$. The defaults are `acceptable_residual_relative = acceptable_residual_absolute = 10u`, where u is `EPSILON(1.0)` (`EPSILON(1.0D0)` in `GALAHAD_SLS_double`).

`out_of_core_directory` is a scalar variable of type default `CHARACTER` and length 400, that identifies the full path name of the directory in which direct-access files will be written if out-of-core factorization is performed; the full path must end with a `/`. Exceptionally, the “empty” string `""` refers to the current directory. The default is `out_of_core_directory = ""`.

`out_of_core_integer_factor_file` is a scalar variable of type default `CHARACTER` and length 400, that identifies the base files name for direct-access files that hold integer components of the matrix factors when an out-of-core factorization is obtained. Files with names `out_of_core_integer_factor_file` appended with 1, 2, ... will be created as needed. There should be no other files with this base file name in the directory specified for the task. The default is `out_of_core_integer_factor_file = "factor_integer.ooc"`.

`out_of_core_real_factor_file` is a scalar variable of type default `CHARACTER` and length 400, that identifies the base files name for direct-access files that hold real components of the matrix factors when an out-of-core factorization is obtained. Files with names `out_of_core_real_factor_file` appended with 1, 2, ... will be created as needed. There should be no other files with this base file name in the directory specified for the task. The default is `out_of_core_real_factor_file = "factor_real.ooc"`.

`out_of_core_real_work_file` is a scalar variable of type default `CHARACTER` and length 400, that identifies the base files name for direct-access files that hold real workspace components when an out-of-core factorization is obtained. Files with names `out_of_core_real_work_file` appended with 1, 2, ... will be created as needed. There should be no other files with this base file name in the directory specified for the task. The default is `out_of_core_real_work_file = "work_real.ooc"`.

`out_of_core_indefinite_file` is a scalar variable of type default `CHARACTER` and length 400, that identifies the base files name for direct-access files that hold additional real workspace components in the indefinite case when an out-of-core factorization is obtained. Files with names `out_of_core_indefinite_file` appended with 1, 2, ... will be created as needed. There should be no other files with this base file name in the directory specified for the task. The default is `out_of_core_indefinite_file = "work_indefinite.ooc"`.

`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.

`NODEND_control` is a scalar variable of type `NODEND_control_type` whose components are used to control nested-dissection ordering, when required by `control%ordering = 3` as performed by the package `GALAHAD_NODEND`. See the specification sheet for the package `GALAHAD_NODEND` for details, and appropriate default values.

2.6.3 The derived data type for holding timing information

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

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

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

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

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

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

`order_external` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent by the external solvers in the ordering phase.

`analyse_external` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent by the external solvers in the analysis phase.

`factorize_external` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent by the external solvers in the factorization phase.

`solve_external` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent by the external solvers in the solve phases.

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

`clock_analyse` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent in the analysis phase.

`clock_factorize` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent in the factorization phase.

`clock_solve` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent in the solve phases.

`clock_order_external` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent by the external solvers in the ordering phase.

`clock_analyse_external` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent by the external solvers in the analysis phase.

`clock_factorize_external` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent by the external solvers in the factorization phase.

`clock_solve_external` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent by the external solvers in the solve phases.

2.6.4 The derived data type for holding informational parameters

The derived data type `SLS_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `SLS_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 `INTEGER(ip_)`, that gives the exit status of the algorithm. See §2.8 for details.

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

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

`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.

`entries` is a scalar variable of type INTEGER(ip_), that is set to the total number of entries of **A** supplied.

`out_of_range` is a scalar variable of type INTEGER(ip_), 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 INTEGER(ip_), that is set to the number of duplicate off-diagonal entries of **A** supplied.

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

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

`max_depth_assembly_tree` is a scalar variable of type INTEGER(ip_), that gives the maximum depth of the assembly tree.

`nodes_assembly_tree` is a scalar variable of type INTEGER(ip_), that holds the number of nodes in the assembly tree.

`real_size_desirable` is a scalar variable of type INTEGER(int64), that give the total number of real words required for a successful factorization without the need for data compression, provided no numerical pivoting is performed.

`integer_size_desirable` is a scalar variable of type INTEGER(int64), that give the total number of integer words required for a successful factorization without the need for data compression, provided no numerical pivoting is performed.

`real_size_necessary` is a scalar variable of type INTEGER(int64), that give the total number of real words required for a successful factorization allowing for data compression, provided no numerical pivoting is performed.

`integer_size_necessary` is a scalar variable of type INTEGER(int64), that give the total number of integer words required for a successful factorization allowing for data compression, provided no numerical pivoting is performed.

`real_size_factors` is a scalar variable of type INTEGER(int64), that gives the predicted or actual number of real words to hold factors of **A**.

`integer_size_factors` is a scalar variable of type INTEGER(int64), that gives the predicted or actual number of integer words to hold factors of **A**.

`entries_in_factors` is a scalar variable of type INTEGER(int64), that gives the number of entries in the factors of **A**.

`max_task_pool_size` is a scalar variable of type INTEGER(ip_), that gives the maximum number of tasks in the parallel factorization task pool.

`max_front_size` is a scalar variable of type INTEGER(ip_), that gives the forecast or actual size of the largest front encountered during the factorization.

`compresses_real` is a scalar variable of type INTEGER(ip_), that gives the number of compresses of real data required.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`compresses_integer` is a scalar variable of type `INTEGER(ip_)`, that gives the number of compresses of integer data required.

`two_by_two_pivots` is a scalar variable of type `INTEGER(ip_)`, that gives the number of two-by-two pivots used in the factorization.

`semi_bandwidths` is a scalar variable of type `INTEGER(ip_)`, that gives the semi-bandwidth of the input matrix following profile or bandwidth reduction.

`delayed_pivots` is a scalar variable of type `INTEGER(ip_)`, that gives the total number of delayed pivots in the factorization.

`pivot_sign_changes` is a scalar variable of type `INTEGER(ip_)`, that gives the number of pivot sign changes encountered if the no pivoting option (`control%pivot_control = 3`) is used successfully.

`static_pivots` is a scalar variable of type `INTEGER(ip_)`, that gives the number of modified pivots during static pivoting are chosen.

`first_modified_pivot` is a scalar variable of type `INTEGER(ip_)`, that gives the step at which the first pivot is modified when performing static pivoting (i.e., if `control%static_pivot_tolerance > 0.0`) or when modifying the matrix to ensure it is sufficiently positive definite (i.e., if `control%pivot_control = 4`).

`rank` is a scalar variable of type `INTEGER(ip_)`, that gives an estimate of the rank of **A**.

`negative_eigenvalues` is a scalar variable of type `INTEGER(ip_)`, that gives an estimate of the number of negative eigenvalues of **A**.

`iterative_refinements` is a scalar variable of type `INTEGER(ip_)`, that gives the number of iterative refinements performed.

`alternative` is a scalar variable of type default `LOGICAL`, that will be set `.FALSE.` on exit from `SLS_fredholm_alternative` (see §2.9.5) if a solution \mathbf{x} to $\mathbf{Ax} = \mathbf{b}$ has been found, and `.TRUE.` if instead the alternative vector \mathbf{x} for which $\mathbf{Ax} = \mathbf{0}$ and $\mathbf{b}^T \mathbf{x} > 0$ has been determined.

`flops_assembly` is a scalar variable of type `INTEGER(int64)`, that gives the anticipated or actual number of floating-point operations performed when manipulating the matrix prior to factorization.

`flops_elimination` is a scalar variable of type `INTEGER(int64)`, that gives the anticipated or actual number of floating-point operations performed during the factorization.

`flops_blas` is a scalar variable of type `INTEGER(int64)`, that gives the number of additional floating-point operations that result from using the basic linear algebra subprograms (BLAS) in the factorization phase.

`largest_modified_pivot` is a scalar variable of type `REAL(rp_)`, that gives the value of the largest pivot modification when performing static pivoting (i.e., if `control%static_pivot_tolerance > 0.0`) or when modifying the matrix to ensure it is sufficiently positive definite (i.e., if `control%pivot_control = 4`).

`minimum_scaling_factor` is a scalar variable of type `REAL(rp_)`, that gives the minimum scaling factor used.

`maximum_scaling_factor` is a scalar variable of type `REAL(rp_)`, that gives the maximum scaling factor used.

`backward_error_1` and `backward_error_2` are scalar variables of type `REAL(rp_)`, that gives estimates of the scaled backward errors (i.e., the residuals of the equations suitably normalised by the sizes of **A** and **b/B**) for category-1 and -2 equations, respectively; category-2 equations are exceptional ones for which the residuals are nonzero but the normalising factors are large.

`condition_number_1` and `condition_number_2` are scalar variables of type `REAL(rp_)`, that gives estimates of the condition numbers of the matrix for category-1 and -2 equations, respectively.

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

`forward_error` is a scalar variable of type `REAL(rp_)`, that gives an estimate of the forward error (i.e., the error in the solution).

`solver` is a scalar variable of type default `CHARACTER` and length 20, that gives the name of the actual solver used.

`time` is a scalar variable of type `SLS_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).

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

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

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

`ma57_ainfo` is a scalar variable of type `ma57_ainfo`, that corresponds to the output value `ma57_ainfo` from HSL_MA57. See the documentation for HSL_MA57 for further details.

`ma57_finfo` is a scalar variable of type `ma57_finfo`, that corresponds to the output value `ma57_finfo` from HSL_MA57. See the documentation for HSL_MA57 for further details.

`ma57_sinfo` is a scalar variable of type `ma57_sinfo`, that corresponds to the output value `ma57_sinfo` from HSL_MA57. See the documentation for HSL_MA57 for further details.

`ma77_info` is a scalar variable of type `ma77_info`, that corresponds to the output value `ma77_info` from HSL_MA77. See the documentation for HSL_MA77 for further details.

`ma86_info` is a scalar variable of type `ma86_info`, that corresponds to the output value `ma86_info` from HSL_MA86. See the documentation for HSL_MA86 for further details.

`ma87_info` is a scalar variable of type `ma87_info`, that corresponds to the output value `ma87_info` from HSL_MA87. See the documentation for HSL_MA87 for further details.

`ma97_info` is a scalar variable of type `ma97_info`, that corresponds to the output value `ma97_info` from HSL_MA97. See the documentation for HSL_MA97 for further details.

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

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

`mc64_info` is a scalar variable of type `mc64_info`, that corresponds to the output value `mc64_info` from HSL_MC64. See the documentation for HSL_MC64 for further details.

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

`mc68_info` is a scalar variable of type `mc68_info`, that corresponds to the output value `mc68_info` from HSL_MC68. See the documentation for HSL_MC68 for further details.

`mc77_info` is an array of rank one, of dimension 10 and of type `INTEGER(ip_)`, that corresponds to the output array `INFO` from the primary subroutine in MC77. See the documentation for MC77 for further details.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`mc77_rinfo` is an array of rank one, of dimension 10 and of type `REAL(rp_)`, that corresponds to the output array `RINFO` from the primary subroutine in `MC77`. See the documentation for `MC77` for further details.

`MUMPS_error` is a scalar variable of type `INTEGER(ip_)`, that corresponds to the component `INFOG(1)` in the solver derived type from `MUMPS`. See the documentation for `MUMPS` for further details.

`MUMPS_info` is an array of size 80 and type `INTEGER(ip_)`, whose components correspond to those in the array `INFOG` in the solver derived type from `MUMPS`. See the documentation for `MUMPS` for further details.

`MUMPS_rinfo` is an array of size 40 and type `REAL(rp_)`, whose components correspond to those in the array `RINFOG` in the solver derived type from `MUMPS`. See the documentation for `MUMPS` for further details.

`PARDISO_error` is a scalar variable of type `INTEGER(ip_)`, that corresponds to the output value `error` from `PARDISO`. See the documentation for `PARDISO` for further details.

`PARDISO_iparm` is an array of size 64 and type `INTEGER(ip_)`, whose components correspond to those in the output array `IPARM` from `PARDISO`. See the documentation for `PARDISO` for further details.

`PARDISO_dparm` is an array of size 64 and type `REAL(rp_)`, whose components correspond to those in the output array `DPARM` from `PARDISO`. See the documentation for `PARDISO` for further details. This is only available for the Pardiso Project version of the package.

`pastix_info` is a scalar variable of type `INTEGER(ip_)`, that corresponds to the output value `info` returned from the `PaStix`. See the documentation for `PaStiX` for further details.

`WSMP_error` is a scalar variable of type `INTEGER(ip_)`, that corresponds to the output value `error` from `WSMP`. See the documentation for `WSMP` for further details.

`WSMP_iparm` is an array of size 64 and type `INTEGER(ip_)`, whose components correspond to those in the output array `IPARM` from `WSMP`. See the documentation for `WSMP` for further details.

`WSMP_dparm` is an array of size 64 and type `REAL(rp_)`, whose components correspond to those in the output array `DPARM` from `WSMP`. See the documentation for `WSMP` for further details.

`lapack_error` is a scalar variable of type `INTEGER(ip_)`, that corresponds to the output value `info` returned from the LAPACK routines `S/DPOTRF/S`, `S/DSYTRF/S` and `S/DPBTRF/S`. See the documentation for LAPACK for further details.

2.6.5 The derived data type for holding problem data

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

2.7 Argument lists and calling sequences

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

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 SLS_initialize( solver, data, control, inform[, check] )
```

`solver` is scalar, of `INTENT(IN)`, of type `CHARACTER`, and of variable length that specifies which solver to use. Possible values are

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`sils` if the GALAHAD solver `SILS` is desired.

`ma27` is an alias for `sils` that reflects the fact that the GALAHAD solver `SILS` is a Fortran-90 encapsulation of the Fortran-77 package `MA27` from HSL.

`ma57` if the HSL solver `HSL_MA57` is desired. This is a more advanced version of `SILS/MA27`.

`ma77` if the HSL solver `HSL_MA77` is desired. This is particularly appropriate when the matrix factors are too large to fit in main memory, and offers the option of parallel execution of core computations.

`ma86` if the HSL solver `HSL_MA86` is desired. This option offers the opportunity for general parallel solution, but may be less efficient than `ma87` below when A is positive definite.

`ma87` if the HSL solver `HSL_MA87` is desired. This option should only be selected when A is positive definite, but offers the opportunity for general parallel solution.

`ma97` if the HSL solver `HSL_MA97` is desired. This option offers the functionality of `HSL_MA57` but offers the option of parallel execution of core computations.

`ssids` if the HSL solver `HSL_SSIDS` is desired. This option offers the functionality of `HSL_MA97` but additionally performs core computations on a GPU if available. N.B., `SSIDS` is only supported for compilers that are OMP 4.0 compliant, and only available in double precision.

`mumps` if the Mumps Technologies solver `MUMPS` (version 5.5.1 or above) is desired. Distributed parallel solution is offered with this choice.

`pardiso` if the Pardiso Project solver `PARDISO` (version 4.0.0 or above) is desired. Again parallel solution is offered with this choice.

`mkl_pardiso` if the Intel MKL solver `PARDISO` that forked from the Pardiso Project version in 2006, and supports parallel solution.

`pastix` if the Inria solver `PaStiX` (version 6.2.1 or above) is desired. The solver supports parallel solution.

`wsmp` if the IBM alpha Works solver `WSMP` (version 10.9 or above) is desired. Parallel solution is also offered with this choice.

`potr` if the LAPACK dense Cholesky factorization package `(S/D)POTR(F/S)` is desired.

`sytr` if the LAPACK dense symmetric indefinite factorization package `S/DSYTRF/S` is desired.

`pbtr` if the LAPACK banded Cholesky factorization package `S/DPBTRF/S` is desired.

Other solvers may be added in the future.

`data` is a scalar `INTENT(OUT)` argument of type `SLS_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 `SLS_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 `SLS_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `SLS_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.

`check` is an OPTIONAL scalar `LOGICAL INTENT(IN)` argument that if `PRESENT` and set `.TRUE.` will check to see if the requested solver is available, and if not will replace this by a suitable equivalent; the equivalent will be recorded in `control%solver`. In addition, the output default values of `control%ordering` and `control%scaling` may be adjusted to ensure that these options are available. No checks will be performed if `check` is not `PRESENT` or if it is set to `.FALSE.`

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.7.2 The sparsity pattern analysis subroutine

The sparsity pattern of **A** may be analysed as follows:

```
CALL SLS_analyse( matrix, data, control, inform[, PERM] )
```

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) except `matrix%val`. Incorrectly-set components will result in errors flagged in `inform%status`, see §2.8.

data is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_initialize`.

control is scalar `INTENT(IN)` argument of type `SLS_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 `SLS_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.

PERM is an `OPTIONAL INTENT(IN)` rank-one `INTEGER(ip_)` argument of length `matrix%n` that may be used to provide a permutation/pivot order. If present, `PERM(i)`, $i = 1, \dots, \text{matrix}\%n$, should be set to the desired position of the input row i in the permuted matrix.

2.7.3 The numerical factorization subroutine

Once it has been analysed, the matrix **A** may be factorized as follows:

```
CALL SLS_factorize(matrix, data, control, inform )
```

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). Those components set for `SLS_analyse` must not have been altered in the interim.

data is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_initialize` and not altered by the user in the interim.

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

inform is a scalar `INTENT(INOUT)` argument of type `SLS_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 solution subroutine

Given the factorization, a set of equations may be solved as follows:

```
CALL SLS_solve( matrix, X, data, control, inform )
```

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). Those components set for `SLS_factorize` must not have been altered in the interim.

X is an `INTENT(INOUT)` assumed-shape array argument of rank 1 or 2 and of type `REAL(rp_)`. On entry, **X** must be set to the vector **b** or the matrix **B** and on successful return it holds the solution **x** or **X**. For the single right-hand side case, the i -th component of **b** and the resulting i -th component of the solution **x** occupy the i -th component of **X**. When there are multiple right-hand sides, the i -th component of the j -th right-hand side **b_j** and the resulting solution **x_j** occupy the i, j -th component of **X**.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

`data` is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_initialize` and not altered by the user in the interim.

`control` is scalar `INTENT(IN)` argument of type `SLS_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 `SLS_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 SLS_terminate( data, control, inform )
```

`data` is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_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 `SLS_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 `SLS_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.
- 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.
- 26 The requested solver is not available.
- 29 This option is not available with this solver.
- 32 More than `control%max_integer_factor_size` words of internal integer storage are required for in-core factorization.
- 34 The package PARDISO failed; check the solver-specific information components `inform%pardiso_iparm` and `inform%pardiso_dparm` along with PARDISO's documentation for more details.
- 35 The package WSMP failed; check the solver-specific information components `inform%wsmp_iparm` and `inform%wsmp_dparm` along with WSMP's documentation for more details.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

- 36 The scaling package HSL_MC64 failed; check the solver-specific information component `inform%mc64_info` along with HSL_MC64's documentation for more details.
- 37 The scaling package MC77 failed; check the solver-specific information components `inform%mc77_info` and `inform%mc77_rinfo` along with MC77's documentation for more details.
- 39 The input permutation/pivot order is not a permutation or is faulty in some other way.
- 40 It is not possible to alter the block diagonal factors for this solver.
- 41 There is no information about the permutation used or the values of the pivots used for this solver (see §2.9.1).
- 42 There is no information about the values of diagonal perturbations made for this solver (see §2.9.1).
- 43 A direct-access file error occurred. See the value of `inform%ma77_info%flag` for more details.
- 50 A solver-specific error occurred; check the solver-specific information component of `inform` along with the solver's documentation for more details.

2.9 Further features

In this section, we describe features for enquiring about and manipulating the parts of the factorization constructed. These features will not be needed by a user who wants simply to solve systems of equations with matrix **A**.

The solvers used each produce an \mathbf{LDL}^T factorization of **A** or a perturbation thereof, where **L** is a permuted lower triangular matrix and **D** is a block diagonal matrix with blocks of order 1 and 2. It is convenient to write this factorization in the form

$$\mathbf{A} + \mathbf{E} = \mathbf{P}\mathbf{L}\mathbf{D}\mathbf{L}^T\mathbf{P}^T,$$

where **P** is a permutation matrix and **E** is any diagonal perturbation introduced. The following subroutines are provided:

`SLS_enquire` returns any of **P**, **D**, **E**, or the pivot permutations implicit in **D**.

`SLS_alter_d` alters **D**. Note that this means that we no longer have a factorization of the given matrix **A**.

`SLS_part_solve` solves one of the systems of equations $\mathbf{P}\mathbf{L}\mathbf{x} = \mathbf{b}$, $\mathbf{D}\mathbf{x} = \mathbf{b}$, $\mathbf{L}^T\mathbf{P}^T\mathbf{x} = \mathbf{b}$, or when **A** is positive definite, $\mathbf{P}\mathbf{L}\sqrt{\mathbf{D}}\mathbf{x} = \mathbf{b}$, for one or more right-hand sides.

`SLS_sparse_forward_solve` solves $\mathbf{P}\mathbf{L}\mathbf{x} = \mathbf{b}$ when **b** is sparse and aims to return a sparse **x**.

`SLS_fredholm_alternative` computes the Fredholm alternative for the data (\mathbf{A}, \mathbf{b}) , i.e., returns either **x** satisfying $\mathbf{A}\mathbf{x} = \mathbf{b}$ or **x** satisfying $\mathbf{A}\mathbf{x} = \mathbf{0}$ and $\mathbf{b}^T\mathbf{x} > 0$.

Support for these features from the solvers available with SLS is summarised in Table 2.2.

2.9.1 To return **P** or **D** or both

```
CALL SLS_enquire( data, inform[, PERM][, PIVOTS][, D][, PERTURBATION] )
```

`data` is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_initialize` and not altered by the user in the interim.

`inform` is a scalar `INTENT(INOUT)` argument of type `SLS_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.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

solver	SLS_enquire				SLS_alter_d	SLS_part_solve	SLS_sparse- _forward_solve	SLS_fredholm- _alterantive
	P	pivots	D	E				
SILS/MA27	✓	✓	✓	✓	✓	✓	× ¹	×
HSL_MA57	✓	✓	✓	✓	✓	✓	✓	✓
HSL_MA77	✓	✓	✓	×	✓	✓	× ¹	✓
HSL_MA86	✓	×	×	×	×	✓	× ¹	×
HSL_MA87	✓	×	×	×	×	✓	✓	×
HSL_MA97	✓	✓	✓	✓	✓	✓	✓	✓
SSIDS	✓	✓	✓	✓	✓	✓	× ¹	×
MUMPS	✓	×	×	×	×	×	×	×
PARDISO	×	×	×	×	×	✓	× ¹	×
MKL_PARDISO	×	×	×	×	×	✓	× ¹	×
PaStiX	✓	×	×	×	×	×	×	×
WSMP	×	×	×	×	×	✓	× ¹	×
POTR	×	×	✓	×	✓	✓	× ¹	×
SYTR	×	✓	✓	×	✓	✓	× ¹	×
PBTR	×	×	✓	×	✓	✓	× ¹	×

Table 2.2: Options supported. ×¹ indicates a feature that is not available directly from the solver, but for which an (inefficient) simulation is provided.

PERM is an OPTIONAL rank-one INTEGER(ip_) array argument of INTENT(OUT) and length n . If present, PERM will be set to the pivot permutation selected by SLS_analyse.

PIVOTS is an OPTIONAL rank-one INTEGER(ip_) array argument of INTENT(OUT) and length n . If present, the index of pivot i will be placed in PIVOTS(i), $i = 1, \dots, n$, with its sign negative if it is the index of a 2 x 2 block.

D is an OPTIONAL rank-two REAL(rp_) array argument of INTENT(OUT) and shape $(2, n)$. If present, the diagonal entries of \mathbf{D}^{-1} will be placed in D(1, i), $i = 1, \dots, n$ and the off-diagonal entries of \mathbf{D}^{-1} will be placed in D(2, i), $i = 1, \dots, n-1$.

PERTURBATION is an OPTIONAL rank-one REAL(rp_) array argument of INTENT(OUT) and length n . If present, PERTURBATION will be set to a vector of diagonal perturbations chosen by SLS_factorize. This array can only be nonzero if SLS_factorize was last called with control%pivoting = 4 or if the solver chose its own pivot sequence (control%pivoting ≤ 0).

2.9.2 To alter D

```
CALL SLS_alter_d( data, D, inform )
```

data is a scalar INTENT(INOUT) argument of type SLS_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 SLS_initialize and not altered by the user in the interim.

D is an INTENT(INOUT) REAL(rp_) array argument of shape $(2, n)$. The diagonal entries of \mathbf{D}^{-1} will be altered to D(1, i), $i = 1, \dots, n$ and the off-diagonal entries of \mathbf{D}^{-1} will be altered to D(2, i), $i = 1, \dots, n-1$.

inform is a scalar INTENT(INOUT) argument of type SLS_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.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.9.3 To perform a partial solution

```
CALL SLS_part_solve( part, X, data, control, inform )
```

`part` is scalar, of `INTENT(IN)` and of type `CHARACTER`. It must have one of the values

- L for solving $\mathbf{PLx} = \mathbf{b}$ or $\mathbf{PLX} = \mathbf{B}$,
- D for solving $\mathbf{Dx} = \mathbf{b}$ or $\mathbf{DX} = \mathbf{B}$, or
- U for solving $\mathbf{L}^T \mathbf{P}^T \mathbf{x} = \mathbf{b}$ or $\mathbf{L}^T \mathbf{P}^T \mathbf{X} = \mathbf{B}$.
- S for solving $\mathbf{PL}\sqrt{\mathbf{D}}\mathbf{x} = \mathbf{b}$ or $\mathbf{PL}\sqrt{\mathbf{D}}\mathbf{X} = \mathbf{B}$ where $\sqrt{\mathbf{D}}$ is the diagonal matrix whose entries are the square roots of those of \mathbf{D} when \mathbf{A} is positive definite.

`data` is a scalar `INTENT(INOUT)` argument of type `SLS_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 `SLS_initialize` and not altered by the user in the interim.

`X` is an `INTENT(INOUT)` `REAL(rp_)` assumed-shape array argument of rank 1 or 2. On entry, `X` must be set to the vector \mathbf{b} or the matrix \mathbf{B} and on successful return it holds the solution \mathbf{x} or \mathbf{X} . For the single right-hand side case, the i -th component of \mathbf{b} and the resulting i -th component of the solution \mathbf{x} occupy the i -th component of `X`. When there are multiple right-hand sides, the i -th component of the j -th right-hand side \mathbf{b}_j and the resulting solution \mathbf{x}_j occupy the i, j -th component of `X`.

`control` is scalar `INTENT(IN)` argument of type `SLS_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 `SLS_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.

2.9.4 To solve $\mathbf{PLx} = \mathbf{b}$ for sparse \mathbf{b}

```
CALL SLS_sparse_forward_solve( nnz_b, INDEX_b, B, nnz_x, INDEX_x, X, data, &
                               control, inform )
```

`nnz_b` is an `INTENT(IN)` scalar of type `INTEGER` that must hold the number of nonzero entries in the right-hand side.
Restriction: $1 \leq \text{nnz_b} \leq n$.

`INDEX_b` is an `INTENT(IN)` rank-1 array of type `INTEGER`. The first `nnz_b` entries must hold the indices of the nonzero entries in the right-hand side.

`B` is an `INTENT(IN)` rank-one assumed-shape array argument of length at least n and of type `REAL(rp_)`. If `INDEX_b(i)=k`, `B(k)` must hold the i -th nonzero component of the right-hand side; other entries of `B` are not accessed.

`nnz_x` is an `INTENT(OUT)` scalar of type `INTEGER`. On exit, `nnz_x` holds the number of nonzero entries in the solution.

`INDEX_x` is an `INTENT(OUT)` rank-1 array of type `INTEGER` and size `nnz_x` (that is at most n). On exit, the first `nnz_x` entries hold the indices of the nonzero entries in the solution.

`X` is an `INTENT(INOUT)` rank-one assumed-shape array argument of length at least n and of type `REAL(rp_)`. On entry, its first n entries must be set by the user to zero. On exit, if `INDEX_x(i)=k`, `X(k)` holds the i -th nonzero component of the solution; all other entries of `X` are zero.

`data`, `control` and `inform`: see Section 2.7.4.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

2.9.5 To find the Fredholm alternative for the data (A,b)

```
CALL SLS_fredholm_alternative( matrix, X, data, control, inform )
```

matrix, data, control and inform: see Section 2.7.4.

X is an INTENT(INOUT) rank-one assumed-shape array argument of length at least n and of type `REAL(rp_)`. On entry, its first n entries must be set by the user to the vector **b**. On exit, X will contain a solution **x** to $\mathbf{Ax} = \mathbf{b}$ if `inform%alternative` is `.FALSE.` or a vector **x** for which $\mathbf{Ax} = \mathbf{0}$ and $\mathbf{b}^T \mathbf{x} > 0$ if `inform%alternative` is `.TRUE.`.

2.10 Setting control parameters

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

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 `SLS_read_specfile` must start with a “BEGIN SLS” 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 SLS” 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
```

are acceptable. Furthermore, between the “BEGIN SLS” 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).

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

The specification file must be open for input when `SLS_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 `SLS_read_specfile`.

2.10.1 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 `SLS_control_type` (see §2.6.2). Default values should have already been set, perhaps by calling `SLS_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 2.3.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, 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_SORT_single/double`, `GALAHAD_SPACE_single/double`, `GALAHAD_SPECFILE_single/double`, `GALAHAD_STRING_single/double`, `GALAHAD_SMT_single/double`, `GALAHAD_SILS_single/double`, `GALAHAD_NODEND_single/double` and optionally `HSL_MA57_single/double`, `HSL_MA77_single/double`, `HSL_MA86_single/double`, `HSL_MA87_single/double`, `HSL_MA97_single/double`, `HSL_MC64_single/double` and `HSL_MC68_single/double`.

Other routines called directly: Optionally `MC61`, `MC77` and `METIS`.

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

Restrictions: $\text{matrix}\%n \geq 1, \text{matrix}\%ne \geq 0$ if `matrix%type = 'COORDINATE'`, `matrix%type` 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

Variants of sparse Gaussian elimination are used.

The solver `SILS` is available as part of `GALAHAD` and relies on the HSL Archive package `MA27`. To obtain HSL Archive packages, see

<http://hsl.rl.ac.uk/archive/> .

The solvers `HSL_MA57`, `HSL_MA77`, `HSL_MA86`, `HSL_MA87` and `HSL_MA97`, the ordering packages `MC61` and `HSL_MC68`, and the scaling packages `HSL_MC64` and `MC77` are all part of HSL 2011. To obtain HSL 2011 packages, see

<http://hsl.rl.ac.uk> .

The solver `SSIDS` is from the `SPRAL` sparse-matrix collection, and is available as part of `GALAHAD`.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

command	component of control	value type
error-printout-device	%error	integer
warning-printout-device	%warning	integer
printout-device	%out	integer
statistics-printout-device	%statistics	integer
print-level	%print_level	integer
print-level-solver	%print_level_solver	integer
architecture-bits	%bits	integer
block-size-for-kernel	%block_size_kernel	integer
block-size-for--elimination	%block_size_elimination	integer
blas-block-for-size-factorize	%blas_block_size_factorize	integer
blas-block-size-for-solve	%blas_block_size_solve	integer
node-amalgamation-tolerance	%node_amalgamation	integer
initial-pool-size	%initial_pool_size	integer
minimum-real-factor-size	%min_real_factor_size	integer
minimum-integer-factor-size	%min_integer_factor_size	integer
maximum-real-factor-size	%max_real_factor_size	integer (long)
maximum-integer-factor-size	%max_integer_factor_size	integer (long)
maximum-in-core-store	%max_in_core_store	integer (long)
pivot-control	%pivot_control	integer
ordering	%ordering	integer
full-row-threshold	%full_row_threshold	integer
pivot-row-search-when-indefinite	%row_search_indefinite	integer
scaling	%scaling	integer
scale-maxit	%scale_maxit	integer
scale-thresh	%scale_thresh	real
max-iterative-refinements	%max_iterative_refinements	integer
array-increase-factor	%array_increase_factor	real
array-decrease-factor	%array_decrease_factor	real
relative-pivot-tolerance	%relative_pivot_tolerance	real
absolute-pivot-tolerance	%absolute_pivot_tolerance	real
zero-tolerance	%zero_tolerance	real
static-pivot-tolerance	%static_pivot_tolerance	real
static-level-switch	%static_level_switch	real
consistency-tolerance	%consistency_tolerance	real
acceptable-residual-relative	%acceptable_residual_relative	real
acceptable-residual-absolute	%acceptable_residual_absolute	real
out-of-core-directory	%out_of_core_directory	character
out-of-core-integer-factor-file	%out_of_core_integer_factor_file	character
out-of-core-real-factor-file	%out_of_core_real_factor_file	character
out-of-core-real-work-file	%out_of_core_real_work_file	character
out-of-core-indefinite-file	%out_of_core_indefinite_file	character
output-line-prefix	%prefix	character

Table 2.3: Specfile commands and associated components of control.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

The solver MUMPS is available from Mumps Technologies in France, and version 5.5.1 or above is sufficient. To obtain MUMPS, see

<https://mumps-solver.org> .

The solver PARDISO is available from the Pardiso Project; version 4.0.0 or above is required. To obtain PARDISO, see

<http://www.pardiso-project.org/> .

The solver MKL PARDISO is available as part of Intel's oneAPI Math Kernel Library (oneMKL). To obtain this version of PARDISO, see

<https://software.intel.com/content/www/us/en/develop/tools/oneapi.html> .

The solver PaStix is available from Inria in France, and version 6.2 or above is sufficient. To obtain PaStiX, see

<https://solverstack.gitlabpages.inria.fr/pastix> .

The solver WSMP is available from the IBM alpha Works; version 10.9 or above is required. To obtain WSMP, see

<http://www.alphaworks.ibm.com/tech/wsmp> .

The solvers POTR, SYTR and PBTR, are available as $S/DPOTRF/S$, $S/DSYTRF/S$ and $S/DPBTRF/S$ as part of LAPACK. Reference versions are provided by GALAHAD, but for good performance machined-tuned versions should be used.

Explicit sparsity re-orderings are obtained by calling the HSL package HSL_MC68 and GALAHAD's NODEND. Both these, HSL_MA57 and PARDISO rely optionally on the ordering package METIS from the Karypis Lab. METIS versions 5.1 and 5.2 are provided, but to obtain METIS version 4.0, see

<http://glaros.dtc.umn.edu/gkhome/views/metis/> .

Bandwidth, Profile and wavefront reduction is supported by calling HSL's MC61.

References:

The methods used are described in the user-documentation for

HSL 2011, A collection of Fortran codes for large-scale scientific computation (2011).

<http://www.hsl.rl.ac.uk>

and papers

E. Agullo, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, A. Guermouche and F.-H. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations, *SIAM Journal on Scientific Computing*, **38(3)** (2016), C256–C279,

P. R. Amestoy, I. S. Duff, J. Koster and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling, *SIAM Journal on Matrix Analysis and Applications* **23(1)** (2001) 15–41.

A. Gupta, “WSMP: Watson Sparse Matrix Package Part I - direct solution of symmetric sparse systems”. IBM Research Report RC 21886, IBM T. J. Watson Research Center. NY 10598, USA (2010),

J.D. Hogg, E. Ovtchinnikov and J.A. Scott. A sparse symmetric indefinite direct solver for GPU architectures. *ACM Transactions on Mathematical Software* **42(1)** (2014), Article 1,

P. Hénon, P. Ramet and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, **28(2)** (2002) 301–321,

O. Schenk and K. Gärtner, “Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO”. *Journal of Future Generation Computer Systems*, **20(3)** (2004) 475–487, and

O. Schenk and K. Gärtner, “On fast factorization pivoting methods for symmetric indefinite systems”. *Electronic Transactions on Numerical Analysis* **23** (2006) 158–179.

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

5 EXAMPLE OF USE

We illustrate the use of the package on the solution of the single set of equations

$$\begin{pmatrix} 2 & 3 & & & \\ 3 & & 4 & & 6 \\ & 4 & 1 & 5 & \\ & & 5 & & \\ 6 & & & & 1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 8 \\ 45 \\ 31 \\ 15 \\ 17 \end{pmatrix}$$

(Note that this example does not illustrate all the facilities). Then, choosing the solver SLS, we may use the following code:

```
PROGRAM SLS_EXAMPLE ! GALAHAD 4.1 - 2022-11-27 AT 15:15 GMT.
USE GALAHAD_SLS_double
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
TYPE ( SMT_type ) :: matrix
TYPE ( SLS_data_type ) :: data
TYPE ( SLS_control_type ) control
TYPE ( SLS_inform_type ) :: inform
INTEGER, PARAMETER :: n = 5
INTEGER, PARAMETER :: ne = 7
REAL ( KIND = wp ) :: B( n ), X( n )
INTEGER :: 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( : ne ) = (/ 1, 2, 3, 3, 4, 5, 5 /)
matrix%col( : ne ) = (/ 1, 1, 2, 3, 3, 2, 5 /)
matrix%val( : ne ) = (/ 2.0_wp, 3.0_wp, 4.0_wp, 1.0_wp, 5.0_wp,
                        6.0_wp, 1.0_wp /)
! problem setup complete
! set right-hand side
B( : n ) = (/ 8.0_wp, 45.0_wp, 31.0_wp, 15.0_wp, 17.0_wp /)
! specify the solver (in this case ssids)
CALL SLS_initialize( 'ssids', data, control, inform, check = .TRUE. )
WRITE( 6, "( ' solver ', A, ' used' )" ) TRIM( inform%solver )
! analyse
CALL SLS_analyse( matrix, data, control, inform )
IF ( inform%status < 0 ) THEN
  WRITE( 6, '( A, I0 )' )
  ' Failure of SLS_analyse with status = ', inform%status
  STOP
END IF
! factorize
CALL SLS_factorize( matrix, data, control, inform )
IF ( inform%status < 0 ) THEN
  WRITE( 6, '( A, I0 )' )
  ' Failure of SLS_factorize with status = ', inform%status
  STOP
END IF
! solve using iterative refinement and ask for high relative accuracy
X = B
```

All use is subject to the conditions of a BSD-3-Clause License.

See <http://galahad.rl.ac.uk/galahad-www/cou.html> for full details.

```

control%max_iterative_refinements = 1
control%acceptable_residual_relative = 0.0_wp
CALL SLS_solve( matrix, X, data, control, inform )
IF ( inform%status == 0 ) WRITE( 6, ' ( A, ( 5F5.2 ) )' )      &
    ' Solution is', X
! clean up
CALL SLS_terminate( data, control, inform )
DEALLOCATE( matrix%type, matrix%val, matrix%row, matrix%col )
STOP
END PROGRAM SLS_EXAMPLE

```

This produces the following output:

```

solver ssids used
Solution is 1.00 2.00 3.00 4.00 5.00

```

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.