GALAHAD SSLS

USER DOCUMENTATION GALAHAD Optimization Library version 5.3

1 SUMMARY

Given a matrix $\mathbf{A}$ and symmetric matrices $\mathbf{H}$ and $\mathbf{C}$, **form and factorize the block, symmetric matrix**

$$\mathbf{K} = \begin{pmatrix} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{C} \end{pmatrix},$$

and subsequently **solve systems**

$$\begin{pmatrix} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{C} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix}, \tag{1.1}$$

using the GALAHAD symmetric-indefinite factorization package `SLS`. Full advantage is taken of any zero coefficients in the matrices $\mathbf{H}$, $\mathbf{A}$ and $\mathbf{C}$.

ATTRIBUTES — Versions: `GALAHAD_SSLS_single`, `GALAHAD_SSLS_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_SLS`, `GALAHAD_SPECFILE`, **Date:** July 2025. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_SSLS_single
```

with the obvious substitution `GALAHAD_SSLS_double`, `GALAHAD_SSLS_quadruple`, `GALAHAD_SSLS_single_64`, `GALAHAD_SSLS_double_64` and `GALAHAD_SSLS_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`, `QPT_problem_type`, `SSLS_time_type`, `SSLS_control_type`, `SSLS_inform_type` and `SSLS_data_type` (§2.4) and the subroutines `SSLS_initialize`, `SSLS_analyse`, `SSLS_factorize`, `SSLS_solve`, `SSLS_terminate`, (§2.5) and `SSLS_read_specfile` (§2.7) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Each of the input matrices $\mathbf{H}$, $\mathbf{A}$ and $\mathbf{C}$ may be stored in a variety of input formats.

2.1.1 Dense storage format

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

2.1.2 Sparse co-ordinate storage format

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

2.1.3 Sparse row-wise storage format

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

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

2.1.4 Diagonal storage format

If $\mathbf{H}$ is diagonal (i.e., $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonals entries h_{ii}, $1 \leq i \leq n$, need be stored, and the first n components of the array H%val may be used for the purpose. The same applies to $\mathbf{C}$, but there is no sensible equivalent for the non-square $\mathbf{A}$.

2.1.5 Scaled-identity storage format

If $\mathbf{H}$ is a scalar multiple h of the identity matrix, $h\mathbf{I}$, only the value h needs be stored, and the first component of the array H%val may be used for the purpose. The same applies to $\mathbf{C}$, but as before there is no sensible equivalent for the non-square $\mathbf{A}$.

2.1.6 Identity storage format

If $\mathbf{H}$ is the identity matrix, $\mathbf{I}$, no numerical data need be stored. The same applies to $\mathbf{C}$, but once again there is no sensible equivalent for the non-square $\mathbf{A}$.

2.1.7 Zero storage format

Finally, if $\mathbf{H}$ is the zero matrix, $\mathbf{0}$, no numerical data need be stored. The same applies to both $\mathbf{A}$ and $\mathbf{C}$.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords REAL(rp_) and INTEGER(ip_), where rp_ and ip_ are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default REAL for the single precision versions, DOUBLE PRECISION for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to rp_ = real32, rp_ = real64 and rp_ = real128 respectively as defined by the fortran iso_fortran_env module. The latter are default (32-bit) and long (64-bit) integers, and correspond to ip_ = int32 and ip_ = int64, respectively, again from the iso_fortran_env module.

2.3 Parallel usage

OpenMP may be used by the `GALAHAD_SSLS` 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.

2.4 The derived data types

Six derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type `SMT_TYPE` is used to hold the matrices **H**, **A** and **C**. The components of `SMT_TYPE` used here are:

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

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

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see §2.1.1), is used, the first five components of `type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see §2.1.2), the first ten components of `type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see §2.1.3), the first fourteen components of `type` must contain the string `SPARSE_BY_ROWS`, for the diagonal storage scheme (see §2.1.4), the first eight components of `type` must contain the string `DIAGONAL`. for the scaled-identity storage scheme (see §2.1.5), the first fifteen components of `type` must contain the string `SCALED_IDENTITY`. and for the identity storage scheme (see §2.1.6), the first eight components of `type` must contain the string `IDENTITY`. It is also permissible to set the first four components of `type` to the either of the strings `ZERO` or `NONE` in the case of matrix **C** to indicate that $\mathbf{C} = 0$.

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

```
CALL SMT_put( H%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 matrix entries.

`val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix **H** is represented as a single entry (see §2.1.1–2.1.3); the same applies to **C**. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed. If the matrix is stored using the diagonal scheme (see §2.1.4), `val` should be of length `n`, and the value of the `i`-th diagonal stored in `val(i)`. If the matrix is stored using the scaled-identity scheme (see §2.1.6), `val(1)` should be set to h.

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

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

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

2.4.2 The derived data type for holding control parameters

The derived data type SSLS_control_type is used to hold controlling data. Default values may be obtained by calling SSLS_initialize (see §2.5.1), while components may also be changed by calling GALAHAD_SSLS_read_spec (see §2.7.1). The components of SSLS_control_type are:

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

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

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

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..

symmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any general symmetric linear systems that might arise. Possible choices are 'sils', 'ma27', 'ma57', 'ma77', 'ma86', 'ma97', 'ssids', 'pardiso', 'wsmp', 'sytr', although only 'sils', 'sytr' and, for OMP 4.0-compliant compilers, 'ssids' are installed by default. See the documentation for the GALAHAD package SLS for further details. The default is symmetric_linear_solver = 'ssids'.

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 = "".

SLS_control is a scalar variable argument of type SLS_control_type that is used to pass control options to external packages used to solve any symmetric linear systems that might arise. See the documentation for the GALAHAD package SLS for further details. In particular, default values are as for SLS.

2.4.3 The derived data type for holding timing information

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

`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 forming and analysing **K**.

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

`solve` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent solving the system (1.1).

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

`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 forming analysing **K**.

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

`clock_solve` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent solving the system (1.1).

2.4.4 The derived data type for holding informational parameters

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

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

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

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

`factorization_integer` is a scalar variable of type `INTEGER(int64)`, that reports the number of integers required to hold the factorization.

`factorization_real` is a scalar variable of type `INTEGER(int64)`, that reports the number of reals required to hold the factorization.

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

`rank_def` is a scalar variable of type default `LOGICAL`, that has the value `.TRUE.` if `SSLS_factorize` believes that **A** is rank defficient, and `.FALSE.` otherwise

`time` is a scalar variable of type `SSLS_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 2.4.3).

`SLS_inform` is a scalar variable argument of type `SLS_inform_type` that is used to pass information concerning the progress of the external packages used to solve any symmetric linear systems that might arise. See the documentation for the GALAHAD package `SLS` for further details.

2.4.5 The derived data type for holding problem data

The derived data type `SSLS_data_type` are used to hold all the data for the problem and the factors of **K** between calls of `SSLS` procedures. This data should be preserved, untouched, from the initial call to `SSLS_initialize` to the final call to `SSLS_terminate`.

2.5 Argument lists and calling sequences

There are five procedures for user calls (see §2.7 for further features):

1. The subroutine `SSLS_initialize` is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine `SSLS_analyse` is called to form and analyse the structure of **K** prior to factorization.
3. The subroutine `SSLS_factorize` is called to factorize **K**. This may be called multiple times for matrices with identical structure but different numerical values.
4. The subroutine `SSLS_solve` is called to apply the factorization of **K**, that is to solve a linear system of the form (1.1).
5. The subroutine `SSLS_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `SSLS_analyse` at the end of the solution process.

2.5.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar `INTENT(INOUT)` argument of type `SSLS_data_type` (see §2.4.5). It is used to hold data about the problem being solved.

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

`inform` is a scalar `INTENT(OUT)` argument of type `SSLS_inform_type` (see Section 2.4.4). A successful call to `SSLS_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.6.

2.5.2 The subroutine for forming and analysing the matrix K

The matrix **K** is formed, and its structure analysed, prior to factorization as follows:

```
CALL SSLS_analyse( n, m, H, A, C, data, control, inform )
```

`n` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)` that specifies the number of rows of **H** (and columns of **A**).

`m` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)` that specifies the number of rows of **A** and **C**.

`H` is a scalar `INTENT(IN)` argument of type `SMT_type` whose components—strictly, the value of `H%val` is not needed at this stage—must be set to specify the data defining the matrix **H** (see §2.4.1).

`A` is a scalar `INTENT(IN)` argument of type `SMT_type` whose components—strictly, the value of `A%val` is not needed at this stage—must be set to specify the data defining the matrix **A** (see §2.4.1).

`C` is a scalar `INTENT(IN)` argument of type `SMT_type` whose components—strictly, the value of `C%val` is not needed at this stage—must be set to specify the data defining the matrix **C** (see §2.4.1).

`data` is a scalar `INTENT(INOUT)` argument of type `SSLS_data_type` (see §2.4.5). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `SSLS_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `SSLS_control_type` (see §2.4.2). Default values may be assigned by calling `SSLS_initialize` prior to the first call to `SSLS_solve`.

`inform` is a scalar `INTENT(OUT)` argument of type `SSLS_inform_type` (see §2.4.4). A successful call to `SSLS_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.6.

2.5.3 The subroutine for factorizing K

The matrix **K** is factorized as follows:

```
CALL SSLS_factorize( n, m, H, A, C, data, control, inform )
```

Components `n`, `m`, `data` and `control` are exactly as described for `SSLS_analyse` and must not have been altered **by the user** in the interim.

`H` is a scalar `INTENT(IN)` argument of type `SMT_type` whose component `H%val` must be set to specify the values defining the matrix **H** (see §2.4.1).

`A` is a scalar `INTENT(IN)` argument of type `SMT_type` whose component `A%val` must be set to specify the values defining the matrix **A** (see §2.4.1).

`C` is a scalar `INTENT(IN)` argument of type `SMT_type` whose component `C%val` must be set to specify the values defining the matrix **C** (see §2.4.1).

`inform` is a scalar `INTENT(OUT)` argument of type `SSLS_inform_type` (see §2.4.4). A successful call to `SSLS_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.6.

2.5.4 The subroutine for solving linear systems involving K

The factorization may be applied to solve a system of the form (1.1) as follows:

```
CALL SSLS_solve( n, m, SOL, data, control, inform )
```

Components `n`, `m`, `data` and `control` are exactly as described for `SSLS_factorize` and must not have been altered **by the user** in the interim.

`SOL` is a rank-one `INTENT(INOUT)` array of type default `REAL` and length at least `n+m`, that must be set on entry to hold the composite vector $(a^T \;\; b^T)^T$. In particular `SOL(`i`)`, $i = 1, \ldots$ `n` should be set to a_i, and `SOL(n+`j`)`, $j = 1, \ldots,$ `m` should be set to b_j. On successful exit, `SOL` will contain the solution $(x^T \;\; y^T)^T$ to (1.1), that is `SOL(`i`)`, $i = 1, \ldots,$ `n` will give x_i, and `SOL(n+`j`)`, $j = 1, \ldots,$ `m` will contain y_j.

`inform` is a scalar `INTENT(OUT)` argument of type `SSLS_inform_type` (see §2.4.4), that should be passed unaltered since the last call to `SSLS_factorize` or `SSLS_solve`. A successful call to `SSLS_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.6.

2.5.5 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

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

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

2.6 Warning and error messages

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

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

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

-3. One of the restrictions prob%n > 0 or prob%m ≥ 0 or requirements that prob%A_type, prob%H_type and prob%C_type contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL', 'SCALED_IDENTITY', 'IDENTITY', 'ZERO' or 'NONE' has been violated.

-9. An error was reported by SLS_analyse. The return status from SLS_analyse is given in inform%SLS_inform%status. See the documentation for the GALAHAD package SLS for further details.

-10. An error was reported by SLS_factorize. The return status from SLS_factorize is given in inform%SLS_inform%status. See the documentation for the GALAHAD package SLS for further details.

-11. An error was reported by SLS_solve. The return status from SLS_solve is given in inform%SLS_inform%status. See the documentation for the GALAHAD package SLS for further details.

-26. The requested linear equation solver is not available.

A positive value of inform%status on exit from SSLS_factorize warns of unexpected behaviour. A possible values is:

1. The matrx **A** is rank defficient.

2.7 Further features

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

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

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

```
( .. lines ignored by SSLS_read_specfile .. )
  BEGIN SSLS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by SSLS_read_specfile .. )
```

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

and

```
END SSLS SPECIFICATION
```

are acceptable. Furthermore, between the “BEGIN SSLS” and “END” delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to ”comment out” some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are ”ON”, ”TRUE”, ”.TRUE.”, ”T”, ”YES”, ”Y”, or ”OFF”, ”NO”, ”N”, ”FALSE”, ”.FALSE.” and ”F”. Empty values are also allowed for logical control parameters, and are interpreted as ”TRUE”.

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

Control parameters corresponding to the components SLS_control may be changed by including additional sections enclosed by “BEGIN SLS” and “END SLS”. See the specification sheets for the package GALAHAD_SLS for further details.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL SSLS_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type SSLS_control_type (see §2.4.2). Default values should have already been set, perhaps by calling SSLS_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 §2.4.2) of control that each affects are given in Table 2.1.

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

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
symmetric-linear-equation-solver	%symmetric_linear_solver	character
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

2.8 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control-%out. If control%print_level $= 1$, statistics concerning the factorization, as well as warning and error messages will be reported. If control%print_level $= 2$, additional information about the progress of the factorization and the solution phases will be given. If control%print_level > 2, debug information, of little interest to the general user, may be returned.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: SSLS_solve calls the GALAHAD packages GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SLS, GALAHAD_ULS and GALAHAD_SPECFILE,

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

Restrictions: prob%n > 0, prob%m ≥ 0, prob%H_type, prob%A_type and prob%C_type $\in$ {'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'DIAGONAL', 'SCALED_IDENTITY', 'IDENTITY' }, 'ZERO', 'NONE'.

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

4 METHOD

The method simply assembles **K** from its components, and then relies of SLS for analysis, factorization and solves.

5 EXAMPLE OF USE

Suppose we wish to solve the linear system (1.1) with matrix data

$$\mathbf{H} = \begin{pmatrix} 1 & & 4 \\ & 2 & \\ 4 & & 3 \end{pmatrix}, \ \mathbf{A} = \begin{pmatrix} 2 & 1 & \\ & 1 & 1 \end{pmatrix} \text{ and } \mathbf{C} = \begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$$

and right-hand sides

$$\mathbf{a} = \begin{pmatrix} 7 \\ 4 \\ 8 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Then storing the matrices in sparse co-ordinate format, we may use the following code:

```
! THIS VERSION: GALAHAD 5.3 - 2025-07-24 AT 10:30 GMT.
   PROGRAM GALAHAD_SSLS_EXAMPLE
   USE GALAHAD_SSLS_double          ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   TYPE ( SMT_type ) :: H, A, C
   REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : ) :: SOL
   TYPE ( SSLS_data_type ) :: data
   TYPE ( SSLS_control_type ) :: control
   TYPE ( SSLS_inform_type ) :: inform
   INTEGER :: s
   INTEGER :: n = 3, m = 2, h_ne = 4, a_ne = 4, c_ne = 1
! start problem data
   ALLOCATE( SOL( n + m ) )
   SOL( 1 : n ) = (/ 7.0_wp, 4.0_wp, 8.0_wp /)  ! RHS a
   SOL( n + 1 : n + m ) = (/ 2.0_wp, 1.0_wp /)  ! RHS b
! sparse co-ordinate storage format
   CALL SMT_put( H%type, 'COORDINATE', s )  ! Specify co-ordinate
   CALL SMT_put( A%type, 'COORDINATE', s )  ! storage for H, A and C
   CALL SMT_put( C%type, 'COORDINATE', s )
   ALLOCATE( H%val( h_ne ), H%row( h_ne ), H%col( h_ne ) )
   ALLOCATE( A%val( a_ne ), A%row( a_ne ), A%col( a_ne ) )
   ALLOCATE( C%val( c_ne ), C%row( c_ne ), C%col( c_ne ) )
   H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /) ! matrix H
   H%row = (/ 1, 2, 3, 3 /)                     ! NB lower triangle
   H%col = (/ 1, 2, 3, 1 /) ; H%ne = h_ne
   A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! matrix A
   A%row = (/ 1, 1, 2, 2 /)
   A%col = (/ 1, 2, 2, 3 /) ; A%ne = a_ne
   C%val = (/ 1.0_wp /) ! matrix C
   C%row = (/ 2 /) ! NB lower triangle
   C%col = (/ 1 /) ; C%ne = c_ne
! problem data complete
   CALL SSLS_initialize( data, control, inform ) ! Initialize control parameters
   control%symmetric_linear_solver = "sytr "
! factorize matrix
   CALL SSLS_analyse( n, m, H, A, C, data, control, inform )
   IF ( inform%status < 0 ) THEN                ! Unsuccessful call
     WRITE( 6, "( ' SSLS_analyse exit status = ', I0 )" ) inform%status
     STOP
   END IF
   CALL SSLS_factorize( n, m, H, A, C, data, control, inform )
   IF ( inform%status < 0 ) THEN                ! Unsuccessful call
     WRITE( 6, "( ' SSLS_factorize exit status = ', I0 )" ) inform%status
     STOP
   END IF
! solve system
   CALL SSLS_solve( n, m, SOL, data, control, inform )
   IF ( inform%status == 0 ) THEN               ! Successful return
```

```
    WRITE( 6, "( ' SSLS: Solution = ', /, ( 5ES12.4 ) )" ) SOL
  ELSE                                          !  Error returns
    WRITE( 6, "( ' SSLS_solve exit status = ', I6 )" ) inform%status
  END IF
  CALL SSLS_terminate( data, control, inform )  ! delete internal workspace
  END PROGRAM GALAHAD_SSLS_EXAMPLE
```

This produces the following output:

```
 SSLS: Solution =
  1.0000E+00  1.0000E+00  1.0000E+00  1.0000E+00  1.0000E+00
```

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

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

by

```
! sparse row-wise storage format
   CALL SMT_put( H%type, 'SPARSE_BY_ROWS', s )  ! Specify sparse-by-rows
   CALL SMT_put( A%type, 'SPARSE_BY_ROWS', s )  ! storage for H, A and C
   CALL SMT_put( C%type, 'SPARSE_BY_ROWS', s )
   ALLOCATE( H%val( h_ne ), H%col( h_ne ), H%ptr( n + 1 ) )
   ALLOCATE( A%val( a_ne ), A%col( a_ne ), A%ptr( m + 1 ) )
   ALLOCATE( C%val( c_ne ), C%col( c_ne ), C%ptr( m + 1 ) )
   H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /) ! matrix H
   H%col = (/ 1, 2, 3, 1 /)                     ! NB lower triangular
   H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row pointers
   A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! matrix A
   A%col = (/ 1, 2, 2, 3 /)
   A%ptr = (/ 1, 3, 5 /)                        ! Set row pointers
   C%val = (/ 1.0_wp /)                         ! matrix C
   C%col = (/ 1 /)                              ! NB lower triangular
   C%ptr = (/ 1, 1, 2 /)                        ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( H%type, 'DENSE', s )  ! Specify dense
   CALL SMT_put( A%type, 'DENSE', s )  ! storage for H, A and C
   CALL SMT_put( C%type, 'DENSE', s )
   ALLOCATE( H%val( n * ( n + 1 ) / 2 ) )
   ALLOCATE( A%val( n * m ) )
   ALLOCATE( C%val( m * ( m + 1 ) / 2 ) )
   H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 4.0_wp, 0.0_wp, 3.0_wp /) ! H
   A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! A
   C%val = (/ 0.0_wp, 1.0_wp, 0.0_wp /)                         ! C
! problem data complete
```

respectively.

If instead **H** had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for **H**, and in this case we would instead

```
CALL SMT_put( prob%H%type, 'DIAGONAL', s )  ! Specify dense storage for H
ALLOCATE( p%H%val( n ) )
p%H%val = (/ 1.0_wp, 0.0_wp, 3.0_wp /) ! Hessian values
```

Notice here that zero diagonal entries are stored.