# GALAHAD ULS

## 1 SUMMARY

This package **solves dense or sparse unsymmetric systems of linear equations** using variants of Gaussian elimination. Given a sparse symmetric matrix $\mathbf{A} = \{a_{ij}\}_{m \times n}$, and an $m$-vector $\mathbf{b}$ or a matrix $\mathbf{B} = \{b_{ij}\}_{m \times r}$, this subroutine solves the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ or the block system $\mathbf{A}\mathbf{X} = \mathbf{B}$. If $\mathbf{b}$ is an $n$-vector or $\mathbf{B} = \{b_{ij}\}_{n \times r}$, the subroutine may solve instead the system $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ or block system $\mathbf{A}^T\mathbf{X} = \mathbf{B}$. Both square ($m = n$) and rectangular ($m \neq n$) matrices are handled; one of an infinite class of solutions for consistent systems will be returned whenever $\mathbf{A}$ is not of full rank.

The method provides a common interface to a variety of well-known solvers from HSL. Currently supported solvers include `MA28/GLS` and `HSL_MA48`, Note that **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 that via this package.

**ATTRIBUTES — Versions:** `GALAHAD_ULS_single`, `GALAHAD_ULS_double`. **Calls:** `GALAHAD_SYMBOLS`, `GALAHAD_-SORT`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_STRING`, `GALAHAD_SMT`, `GALAHAD_GLS` and optionally `HSL_MA48`. **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

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

*Single precision version*

         `USE GALAHAD_ULS_single`

*Double precision version*

         `USE GALAHAD_ULS_double`

If it is required to use both modules at the same time, the derived types `SMT_type`, `ULS_control_type`, `ULS_data_type`, and `ULS_inform_type` (§2.6), and the subroutines `ULS_initialize`, `ULS_factorize`, `ULS_solve`, `ULS_terminate` (§2.7), and `ULS_enquire` (§2.9) must be renamed on one of the `USE` statements.

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

The subroutine `ULS_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 `ULS_initialize`.

`ULS_factorize` accepts the pattern of $\mathbf{A}$ and chooses pivots for Gaussian elimination using a selection criterion to preserve sparsity. The factors of $\mathbf{A}$ are generated using the calculated pivot order.

`ULS_solve` uses the factors generated by `ULS_factorize` to solve a system of equations with one ($\mathbf{A}\mathbf{x} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$) ) or several ($\mathbf{A}\mathbf{X} = \mathbf{B}$ or $\mathbf{A}^T\mathbf{X} = \mathbf{B}$) right-hand sides. Iterative refinement may be used to improve a given solution or set of solutions.

`ULS_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 ULS. Further details are provided in the references cited in §4.

| solver | out-of-core | parallelised |
|--------|-------------|--------------|
| GLS/MA28 | no | no |
| HSL_MA48 | no | no |

Table 2.1: External solver characteristics.

## 2.3 Matrix storage formats

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

### 2.3.1 Sparse co-ordinate storage format

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

### 2.3.2 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row $i$ appear directly before those in row $i+1$. For the $i$-th row of **A**, the $i$-th component of an integer array ptr holds the position of the first entry in this row, while ptr $(m+1)$ holds the total number of entries plus one. The column indices $j$ and values $a_{ij}$ of the entries in the $i$-th row are stored in components $l = \text{ptr}(i), \ldots, \text{ptr}(i+1)-1$ of the integer array col, and real array val, respectively.

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

### 2.3.3 Dense storage format

The matrix **A** is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. In particular, component $m*(i-1)+j$ of the storage array val will hold the value $a_{ij}$ for $1 \le i \le m$ and $1 \le j \le n$.

## 2.4 Integer kinds

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

## 2.5 OpenMP

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

The code may be compiled and run in serial mode.

## 2.6 The derived data types

Four 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:

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

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

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

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

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

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

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

val    is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_ULS_double`), that holds the values of the entries of the matrix **A** for each of the storage schemes discussed in §2.3. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row    is a rank-one allocatable array of type default `INTEGER`, that holds the row indices of **A** in the sparse co-ordinate storage scheme (see §2.3.1). It need not be allocated for any of the other schemes. Any entry whose row index lies out of the range [1,n] will be ignored.

col    is a rank-one allocatable array variable of type default `INTEGER`, that holds the column indices of **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 row index lies out of the range [1,m] or column index lies out of the range [1,n] will be ignored.

ptr    is a rank-one allocatable array of size m+1 and type default `INTEGER`, that holds the starting position 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, `MA28/GLS` and `HSL_MA48` 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 `ULS_control_type` is used to hold controlling data. Default values specifically for the desired solver may be obtained by calling `ULS_initialize` (see §2.7.1), while components may be changed at run time by calling `ULS_read_specfile` (see §2.10.1). The components of `ULS_control_type` are:

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

---

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

print_level  is a scalar variable of type default `INTEGER`, that is used to control the amount of informational output that is required. No informational output will occur if `print_level` $\le 0$. If `print_level` $\ge 1$ a single line of output will be produced for each step of iterative refinement performed. If `print_level` $\ge 2$ this output may be increased to provide significant detail of the factorization process. The default is `print_level = 0`.

print_level_solver  is a scalar variable of type default `INTEGER`, that is used to control the amount of informational output that is required by the exteral solver. No informational output will occur if `print_level` $\le 0$. If `print_level` $\ge 1$ the amount of output produced is solver dependent. The default is `print_level_solver = 0`.

initial_fill_in_factor  is a scalar variable of type default `INTEGER`, that gives a prediction of the factor by which the fill-in will exceed the initial number of entries in **a**. The default is `initial_fill_in_factor = 3`.

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

blas_block_size_factorize  is a scalar variable of type default `INTEGER`, 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 default `INTEGER`, 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`.

pivot_control  is a scalar variable of type default `INTEGER`, that is used to control numerical pivoting by `ULS_factorize`. Possible values are:

     1 Threshold partial pivoting will be performed, with relative pivot tolerance given by the component `relative_pivot_tolerance`.

     2 Threshold rook pivoting is desired, with relative pivot tolerance given by the component `relative_pivot_tolerance`.

     3 Threshold complete pivoting is desired, with relative pivot tolerance given by the component `relative_pivot_tolerance`.

     4 Threshold symmetric pivoting is desired, with relative pivot tolerance given by the component `relative_pivot_tolerance`.

     5 Threshold diagonal pivoting is desired, with relative pivot tolerance given by the component `relative_pivot_tolerance`.

The default is `pivot_control = 1`, and any value outside of $[1, 5]$ will be reset to the default. If a desired value is not available, the default will be substituted.

---

**All use is subject to licence. See** `http://galahad.rl.ac.uk/galahad-www/cou.html`**.**
**For any commercial application, a separate license must be signed.**

pivot_search_limit is a scalar variable of type default INTEGER, that controls the maximum number of row and columns searched for a pivot. If pivot_search_limit $\leq$ 0, the search is unlimited. The default is pivot_search_-limit = 0.

minimum_size_for_btf is a scalar variable of type default INTEGER, that specifies the minimum size of a block within any block-triangular form found during the factorization. The default is minimum_size_for_btf = 1.

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

stop_if_singular is a scalar variable of type default LOGICAL, that should be set .TRUE. if the factorization is to be terminated if **A** is found to be singular, and .FALSE. if the factorization should continue. The default is stop_if_singular = .FALSE..

array_increase_factor is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), 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.

switch_to_full_code_density is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), that specifies the density at which a switch to full/dense code to perform the remaining factorization occurs. The default is switch_to_full_code_density = 0.5.

array_decrease_factor is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), that holds a factor which 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.

relative_pivot_tolerance is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), 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 and less than 0.0 as 0.0.

absolute_pivot_tolerance is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), that holds the absolute pivot tolerance which is used to control the stability of the factorization. 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_ULS_double).

zero_tolerance is a scalar variable of type REAL (double precision in GALAHAD_ULS_double), 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.

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

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.

---

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

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

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

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

`bad_alloc` is a scalar variable of type default `CHARACTER` and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if there have been no allocation or deallocation errors.

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

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

`entries_dropped` is a scalar variable of type long `INTEGER`, that is set to the number of small entries dropped from the factorization.

`workspace_factors` is a scalar variable of type default `INTEGER`, that gives the total number of reals and integers needed to hold the factors.

`compresses` is a scalar variable of type default `INTEGER`, that gives the number of compresses of the workspace data required.

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

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

`structural_rank` is a scalar variable of type default `INTEGER`, that gives the structural rank of **A**.

`pivot_control` is a scalar variable of type default `INTEGER`, that specifies what form of numerical pivoting has been used. Possible values are:

  1 Threshold partial pivoting has been performed.
  2 Threshold rook pivoting has been performed.
  3 Threshold complete pivoting has been performed.
  4 Threshold symmetric pivoting has been performed.
  5 Threshold diagonal pivoting has been performed.

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

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

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

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

---

`ma48_ainfo` is a scalar variable of type `ma48_ainfo`, that corresponds to the output value `ma48_ainfo` from `HSL_MA48`. See the documentation for `HSL_MA48` for further details.

`ma48_finfo` is a scalar variable of type `ma48_finfo`, that corresponds to the output value `ma48_finfo` from `HSL_MA48`. See the documentation for `HSL_MA48` for further details.

`ma48_sinfo` is a scalar variable of type `ma48_sinfo`, that corresponds to the output value `ma48_sinfo` from `HSL_MA48`. See the documentation for `HSL_MA48` for further details.

`PARDISO_error` is a scalar variable of type `INTEGER`, 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`, 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` (double precision in `GALAHAD_ULS_double`), whose components correspond to those in the output array `DPARM` from `PARDISO`. See the documentation for `PARDISO` for further details.

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

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

## 2.7 Argument lists and calling sequences

### 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 ULS_initialize( solver, data, control, inform )
```

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

  `gls` if the GALAHAD solver `GLS` is desired.

  `ma28` is an alias for `gls` that reflects the fact that the GALAHAD solver `GLS` is a Fortran-90 encapsulation of the Fortran-77 package `MA28` from HSL.

  `ma48` if the HSL solver `HSL_MA48` is desired. This is a more advanced version of `GLS/MA28`.

  Other solvers may be added in the future.

`data` is a scalar `INTENT(OUT)` argument of type `ULS_data_type` (see §2.6.4). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `ULS_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 `ULS_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `ULS_inform_type` (see §2.6.3). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

---

### 2.7.2 The factorization subroutine

The matrix **A** may be factorized as follows:

        CALL ULS_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). Incorrectly-set components will result in errors flagged in `inform%status`, see §2.8.

`data` is a scalar `INTENT(INOUT)` argument of type `ULS_data_type` (see §2.6.4). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to `ULS_initialize` and not altered by the user in the interim.

`control` is scalar `INTENT(IN)` argument of type `ULS_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 `ULS_inform_type` (see §2.6.3). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

### 2.7.3 The solution subroutine

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

        CALL ULS_solve( matrix, RHS, X, data, control, inform, trans )

`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 `ULS_factorize` must not have been altered in the interim.

`RHS` is an `INTENT(IN)` assumed-shape array argument of rank 1 or 2 and of type `REAL` (double precision in `GALAH-AD_ULS_double`). On entry, `RHS` 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** occupies the `i`-th component of `RHS`. When there are multiple right-hand sides, the `i`-th component of the $j$-th right-hand side $B_{i,j}$ occupies the `i,j`-th component of `RHS`.

`X` is an `INTENT(INOUT)` assumed-shape array argument of rank 1 or 2 and of type `REAL` (double precision in `GALAH-AD_ULS_double`). On successful return it holds the solution **x** or **X**. For the single right-hand side case, the `i`-th component of the solution **x** occupies the `i`-th component of `X`. When there are multiple right-hand sides, the resulting solution $X_{i,j}$ occupies the `i,j`-th component of `X`.

`data` is a scalar `INTENT(INOUT)` argument of type `ULS_data_type` (see §2.6.4). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to `ULS_initialize` and not altered by the user in the interim.

`control` is scalar `INTENT(IN)` argument of type `ULS_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 `ULS_inform_type` (see §2.6.3). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.8.

`trans` is a scalar `INTENT(IN)` argument of type `LOGICAL` that should be set `.TRUE.` if the solution to $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{X} = \mathbf{B}$ is sought, and `.FALSE.` if the solution to $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{AX} = \mathbf{B}$ is required.

---

### 2.7.4 The termination subroutine

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

        CALL ULS_terminate( data, control, inform )

data is a scalar `INTENT(INOUT)` argument of type `ULS_data_type` (see §2.6.4). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to `ULS_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 `ULS_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 `ULS_inform_type` (see §2.6.3). 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%m > 0` or `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.

-26 The requested solver is not available.

-29 This option is not available with this solver.

-32 The integer workspace required is larger than `max_factor_size`.

-33 The real workspace required is larger than `max_factor_size`.

-39 The input permutation/pivot order is not a permutation or is faulty in some other way.

-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 a feature for enquiring about the factorization constructed. This feature will not be needed by a user who wants simply to solve systems of equations with matrix $\mathbf{A}$ or $\mathbf{A}^T$.

The solvers used each produce an $\mathbf{P}_R\mathbf{L}\mathbf{U}\mathbf{P}_C$ factorization of $\mathbf{A}$, where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrices, and $\mathbf{P}_R$ and $\mathbf{P}_C$ are row and column permutation matrices respectively. The following subroutine is provided:

ULS_enquire returns the row and column permutations that define $\mathbf{P}_R$ and $\mathbf{P}_C$.

Support for this feature from the solvers available with ULS is summarised in Table 2.2.

| solver | ULS_enquire |
|---|---|
| GLS/MA28 | √ |
| HSL_MA48 | √ |

Table 2.2: Options supported.

### 2.9.1 To return $\mathbf{P}_R$ and $\mathbf{P}_C$

```
CALL ULS_enquire( data, inform, ROWS, COLS )
```

data is a scalar INTENT(INOUT) argument of type ULS_data_type (see §2.6.4). It is used to hold the factors and other data about the problem being solved. It must have been initialized by a call to ULS_initialize and not altered by the user in the interim.

inform is a scalar INTENT(INOUT) argument of type ULS_inform_type (see §2.6.3). A successful call is indicated when the component status has the value 0. For other return values of status, see §2.8.

ROWS is a rank-one default INTEGER array argument of INTENT(OUT) and length $m$. The ROW($i$)th row of **A** is the $i$th row in the factors, $1 \le i \le m$.

COLS is a rank-one default INTEGER array argument of INTENT(OUT) and length $n$. The COL($j$)th column of **A** is the $j$th column in the factors, $1 \le j \le n$.

### 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 ULS_control_type (see §2.6.2), by reading an appropriate data specification file using the subroutine ULS_read_specfile. This facility is useful as it allows a user to change ULS control parameters without editing and recompiling programs that call ULS.

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

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

```
( .. lines ignored by ULS_read_specfile .. )
  BEGIN ULS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by ULS_read_specfile .. )
```

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

and

```
END ULS SPECIFICATION
```

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

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

The specification file must be open for input when ULS_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 ULS_read_specfile.

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

Control parameters may be read from a file as follows:

```
CALL ULS_read_specfile( control, device )
```

control  is a scalar INTENT(INOUT) argument of type ULS_control_type (see §2.6.2). Default values should have already been set, perhaps by calling ULS_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 default INTEGER, that must be set to the unit number on which the specification file has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

## 3   GENERAL INFORMATION

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

**Other modules used directly:** GALAHAD_SYMBOLS, GALAHAD_SORT_single/double, GALAHAD_SPACE_single/double, GALAHAD_SPECFILE_single/double, GALAHAD_STRING_single/double, GALAHAD_SMT_single/double, GALAHAD_GLS_single/double and optionally HSL_MA48_single/double.

**Other routines called directly:** None.

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

**Restrictions:** matrix%n ≥ 1, matrix%ne ≥ 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 OpenNP. The package is thread-safe.

---

| command | component of `control` | value type |
|---|---|---|
| `error-printout-device` | `%error` | `integer` |
| `warning-printout-device` | `%warning` | `integer` |
| `printout-device` | `%out` | `integer` |
| `print-level` | `%print_level` | `integer` |
| `print-level-solver` | `%print_level_solver` | `integer` |
| `maximum-block-size-for-btf` | `%maximum_block_size_for_btf` | `integer` |
| `blas-block-for-size-factorize` | `%blas_block_size_factorize` | `integer` |
| `blas-block-size-for-solve` | `%blas_block_size_solve` | `integer` |
| `initial-fill-in-factor` | `%initial_fill_in_factor` | `integer` |
| `minimum-real-factor-size` | `%min_real_factor_size` | `integer` |
| `minimum-integer-factor-size` | `%min_integer_factor_size` | `integer` |
| `maximum-factor-size` | `%max_factor_size` | `integer(long)` |
| `pivot-control` | `%pivot_control` | `integer` |
| `pivot-search-limit` | `%pivot_search_limit` | `integer` |
| `max-iterative-refinements` | `%max_iterative_refinements` | `integer` |
| `stop-if-singular` | `%stop_if_singular` | `logical` |
| `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` |
| `switch-to-full-code-density` | `%switch_to_full_code_density` | `real` |
| `acceptable-residual-relative` | `%acceptable_residual_relative` | `real` |
| `acceptable-residual-absolute` | `%acceptable_residual_absolute` | `real` |
| `output-line-prefix` | `%prefix` | `character` |

Table 2.3: Specfile commands and associated components of `control`.

## 4 METHOD

Variants of sparse Gaussian elimination are used.

The solver `GLS` is available as part of GALAHAD and relies on the HSL Archive packages `MA33`. To obtain HSL Archive packages, see

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

The solver `HSL_MA48` is part of HSL 2007. To obtain HSL 2007 packages, see

    http://hsl.rl.ac.uk/hsl2007/ .

### References:

The methods used are described in the user-documentation for

HSL 2007, A collection of Fortran codes for large-scale scientific computation (2007).
`http://www.cse.clrc.ac.uk/nag/hsl`

and papers

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

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

## 5 EXAMPLE OF USE

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

$$\begin{pmatrix} 11 & 12 & \\ 21 & 22 & 23 \\ & 32 & 33 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 23 \\ 66 \\ 65 \end{pmatrix}$$

(Note that this example does not illustrate all the facilities). Then, choosing the solver `HSL_MA48`, we may use the following code:

```
! THIS VERSION: GALAHAD 2.4 - 30/09/2009 AT 16:00 GMT.
   PROGRAM GALAHAD_ULS_example
   USE GALAHAD_ULS_DOUBLE
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )
   INTEGER :: i, m, n, ne, s
   TYPE ( SMT_type ) :: matrix
   TYPE ( ULS_data_type ) :: data
   TYPE ( ULS_control_type ) control
   TYPE ( ULS_inform_type ) :: inform
   INTEGER, ALLOCATABLE, DIMENSION( : ) :: ROWS, COLS
   REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : ) :: B, X
! Read matrix order and number of entries
   READ( 5, * ) m, n, ne
   matrix%m = m ; matrix%n = n ; matrix%ne = ne
! Allocate arrays of appropriate sizes
   ALLOCATE( matrix%val( ne ),  matrix%row( ne ),  matrix%col( ne ) )
   ALLOCATE( B( m ), X( n ), ROWS( m ), COLS( n ) )
! Read matrix
   READ( 5, * ) ( matrix%row( i ), matrix%col( i ), matrix%val( i ), i = 1, ne )
```

```
   CALL SMT_put( matrix%type, 'COORDINATE', s )     ! Specify co-ordinate
! Specify the solver (in this case HSL_MA48)
   CALL ULS_initialize( 'ma48', data, control, inform )
! Factorize the matrix
   CALL ULS_factorize( matrix, data, control, inform )
   IF ( inform%status < 0 ) THEN
     WRITE( 6, '( A, I0 )' )                                              &
       ' Failure of ULS_factorize with inform%status = ',  inform%status
     STOP
   END IF
! Write row and column reorderings
   CALL ULS_enquire( data, inform, ROWS, COLS )
   WRITE( 6, "( A, /, ( 10I5 ) )" ) ' row orderings:', ROWS( : inform%rank )
   WRITE( 6, "( A, /, ( 10I5 ) )" ) ' column orderings:', COLS( : inform%rank )
! Read right-hand side and solve system
   READ( 5, * ) B
   CALL ULS_solve( matrix, B, X, data, control, inform, .FALSE. )
   IF ( inform%status == 0 ) WRITE( 6, '( A, /,( 6ES11.3 ) )' )          &
     ' Solution of set of equations without refinement is', X
! Clean up
   CALL ULS_terminate( data, control, inform )
   DEALLOCATE( matrix%val, matrix%row, matrix%col, X, B, ROWS, COLS )
   STOP
   END PROGRAM GALAHAD_ULS_example
```

with the following data

```
   3     3     7
   1     1     11.0
   2     3     23.0
   3     3     33.0
   2     1     21.0
   1     2     12.0
   3     2     32.0
   2     2     22.0
  23.0  66.0  65.0
```

This produces the following output:

```
 row orderings:
    1    2    3
 column orderings:
    1    3    2
 Solution of set of equations without refinement is
  1.000E+00  1.000E+00  1.000E+00
```