HSL_MI35

PACKAGE SPECIFICATION HSL

1 SUMMARY

`HSL_MI35` computes **an incomplete factorization preconditioner** that may be used in the solution of **weighted sparse linear least squares problems.** Given an $m \times n$ ($m \geq n$) sparse matrix $A = \{a_{ij}\}$ and an $m \times m$ diagonal matrix of weights W, `HSL_MI35` computes an incomplete factorization of the matrix

$$C = A^T W^2 A.$$

The matrix C may be optionally reordered, scaled and, if necessary, shifted to avoid breakdown of the factorization. The computed lower triangular matrix L is such that LL^T is an incomplete factorization of

$$\overline{C} = SQ^T CQS + \alpha I,$$

where Q is a permutation matrix, S is a diagonal scaling matrix and α is a non-negative shift. The incomplete factorization may be used for preconditioning when solving the normal equations

$$A^T W^2 Ax = A^T W^2 b.$$

A separate entry performs the preconditioning operation

$$y = Pz$$

where $P = (\overline{L}\overline{L}^T)^{-1}$ is the incomplete factorization preconditioner and $\overline{L} = QS^{-1}L$.

The incomplete factorization is based on a matrix decomposition of the form

$$\overline{C} = (L+R)(L+R)^T - E, \tag{1.1}$$

where L is lower triangular with positive diagonal entries, R is a strictly lower triangular matrix with small entries that is used to stabilize the factorization process, and E has the form

$$E = RR^T + F + F^T, \tag{1.2}$$

where F is strictly lower triangle. E is not computed explicitly and all terms in F are ignored, while the matrix R is used in the computation of L but is then discarded. The user controls the dropping of small entries from L and R and the maximum number of entries within each column of L and R (and thus the amount of memory for L and the intermediate work and memory used in computing the incomplete factorization).

The incomplete factorization may be computed with or without forming C explicitly; the advantage of the latter option being that less memory is required.

ATTRIBUTES — Version: 2.1.1 (1 November 2023). **Interfaces:** Fortran, MATLAB. **Types:** Real (single, double). **Calls:** `KB07`, `MC61`, `HSL_MC64`, `HSL_MC68`, `MC77`, `_copy` and (optionally using METIS version 4.x) `METIS_NODEND`. **Language:** Fortran 2003 subset (F95+TR155581). **Date:** May 2015. **Origin:** J. A. Scott, STFC Rutherford Appleton Laboratory and M. Tůma, Institute of Computer Science, Academy of Sciences of the Czech Republic. **Remark:** The development of this package was partially supported by EPSRC grant EP/I013067/1 and by Grant Agency of the Czech Republic grant P201/13-06684S.

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

Access to the package requires a `USE` statement

Single precision version
```
      use hsl_MI35_single
```
Double precision version
```
      use hsl_MI35_double
```
If it is required to use more than one module at the same time, the derived types (see Section 2.2) must be renamed in one of the `USE` statements.

The following procedures are available to the user:

(a) `MI35_check_matrix` takes the matrix A and (optionally) the weights W and checks for out-of-range entries and duplicates. In addition, it removes null rows and columns, and rows and columns corresponding to zero weights. Use of this routine is optional.

(b) `MI35_factorize` computes an incomplete Cholesky factorization of the matrix C **without** explicitly forming the matrix C.

(c) `MI35_formC` forms the matrix C. Use of this routine is optional and is not needed if `MI35_factorize` is called.

(d) `MI35_factorizeC` takes the matrix C and computes an incomplete Cholesky factorization of it. As C is required to be held explicitly, this routine requires more memory than `MI35_factorize` but offers additional scaling options.

(e) `MI35_precondition` performs the preconditioning operation $y = Pz$, where P is the incomplete factorization preconditioner computed by `MI35_factorize` or `MI35_factorizeC`.

(f) `MI35_solve` solves the system $\overline{L}y = SQ^T z$ (or $\overline{L}^T S^{-1} Q^T y = z$), where $\overline{L}$ is the incomplete factor computed by `MI35_factorize` or `MI35_factorizeC`.

(g) `MI35_finalise` frees memory that has been allocated by `MI35_factorize` or by `MI35_formC` and `MI35_factorizeC`.

2.2 The derived data types

For each problem, the user must employ the derived types defined by the module to declare scalars of the types `MI35_keep`, `MI35_control` and `MI35_info`. The following pseudocode illustrates this.

```
      use hsl_MI35_double
      ...
      type (MI35_keep) :: keep
      type (MI35_control) :: control
      type (MI35_info) :: info

      ...
```

The components of `MI35_control` and `MI35_info` are explained in Sections 2.5 and 2.6. The components of `MI35_keep` are used to pass data between the subroutines of the package and must not be altered by the user.

2.3 METIS

The HSL_MI35 package optionally uses the METIS graph partitioning library available from the University of Minnesota website. If METIS is not available, the user must link with the supplied dummy subroutine METIS_NodeND. In this case, the METIS ordering option will not be available to the user and, if selected, MI35_factorize will return with an error.

Important: At present, HSL_MI35 only supports METIS version 4, not the latest version 5 releases.

2.4 Argument lists and calling sequences

2.4.1 Optional arguments

We use square brackets [ ] to indicate OPTIONAL arguments. In each call, optional arguments follow the argument info. Since we reserve the right to add additional optional arguments in future releases of the code, **we strongly recommend that all optional arguments be called by keyword, not by position**.

2.4.2 Integer and package types

INTEGER denotes default INTEGER and INTEGER(long) denotes INTEGER(kind=selected_int_kind(18)). We use the term **package type** to mean default real if the single precision version is being used and double precision real for the double precision version.

2.4.3 To check the matrix data

It is recommended that the user checks their data for errors, for explicit zeros and for null rows/columns; if this is not done, behaviour of the remaining routines is not guaranteed (the other routines make **no checks** of the user-supplied data). To check the matrix A for duplicate entries (they are summed) and out-of-range entries (they are removed), to check for explicit zeros (they are removed) and to check A for null rows/columns (they are removed) and optionally to check W for zero weights (the corresponding rows of WA are removed), a call of the following form should be made:

```
call MI35_check_matrix(m, n, ptr, row, val, control, info, weight, b)
```

m is an INTENT(INOUT) scalar of type INTEGER that must hold the number m of rows of A. On exit, m holds the row dimension after the removal of null rows and rows corresponding to zero weights.

n is an INTENT(INOUT) scalar of type INTEGER that must hold the number n of columns of A. On exit, n holds the column dimension after the removal of null columns. **Restriction:** After the removal of null rows and columns and rows corresponding to zero weights, m and n must satisfy m≥n≥1.

ptr is an INTENT(INOUT) rank-1 array of type INTEGER and size n+1. ptr(j) must be set by the user so that ptr(j) is the position in row of the first entry in column j and ptr(n+1) must be set to one more than the number of matrix entries being input by the user. ptr is only changed on exit if errors are found in A or W.

row is an INTENT(INOUT) rank-1 array of type INTEGER and size at least ptr(n+1)-1. It must hold the row indices of the entries of A with the row indices for the entries in column 1 preceding those for column 2, and so on. row is only changed on exit if errors are found in A or W.

val is an INTENT(INOUT) rank-1 array of package type and size at least ptr(n+1)-1. val(k) must hold the value of the entry in row(k). val is only changed on exit if errors are found in A or W.

control is an INTENT(IN) scalar of type MI35_control (see Section 2.5).

`info` is an `INTENT(OUT)` scalar of type `MI35_info`. Its components provide information about the execution of the subroutine, as explained in Section 2.6.

`weight` is an optional `INTENT(INOUT)` rank-one array of package type and size `m` that, if present, must be set by the user to hold the diagonal entries of the matrix W. On exit, zero weights are removed. A weight is considered to be a zero weight if its absolute value is less than or equal to `control%weight_tol`.

`b` is an optional `INTENT(INOUT)` rank-one array of package type and size `m` that, if present, must be set by the user to hold a vector b. On exit, entries corresponding to either null rows of A or zero weights are removed.

2.4.4 To compute an incomplete Cholesky factorization without forming *C*

To compute an incomplete Cholesky factorization **without** forming C, and a call of the following form should be made:

```
   call MI35_factorize(m, n, ptr, row, val, lsize, rsize, keep, control, info[, weight,
           scale, perm])
```

`m, n, ptr, row, val`: must be passed as output from the call to `MI35_check_matrix`.

`lsize` is an `INTENT(IN)` scalar of type `INTEGER` that determines the number of entries within the incomplete factor L. The maximum number of entries is `n+n`$\times$`lsize`, with a target of `1 + lsize` entries in each column. In general, increasing `lsize` improves the quality of the preconditioner but increases the time to compute and then apply the preconditioner (see Section 4). Values less than `0` are treated as `0`.

`rsize` is an `INTENT(IN)` scalar of type `INTEGER` that determines the maximum number of entries in the strictly lower triangular matrix R that is used in the computation of the preconditioner. A rank-1 array of type `INTEGER` and a rank-1 array of package type each of size `rsize`$\times$`n` are allocated internally to hold R. Thus the amount of memory used, as well as the amount of work involved in computing the preconditioner, depends on `rsize`. Setting `rsize` $>$ `0` generally leads to a higher quality preconditioner than using `rsize` $=$ `0` (and `rsize` $\geq$ `lsize` is generally recommended). Values less than `0` are treated as `0`.

`keep` is an `INTENT(OUT)` scalar of type `MI35_keep`. It is used to hold data about the problem being solved and must be passed unchanged to `MI35_precondition`. The following components may be of interest to the user:

- `fact_ptr` is an allocatable rank-1 array of type `INTEGER(long)`. On exit, it is allocated to have size `n+1`, `fact_ptr(j)` holds the position in `fact_row` of the first entry in column `j` of the computed factor L and `ptr(n+1)` is set to one more than the number of entries in L.
- `fact_row` is an allocatable rank-1 array of type `INTEGER`. On exit, the first `fact_ptr(n+1)-1` entries hold the row indices of the entries of the computed factor L, with the row indices for the entries in column 1 preceding those for column 2, and so on.
- `fact_val` is an allocatable rank-1 array of package type. On exit, `fact_val` holds the values of the entries in the computed factor L such that `fact_val(k)` is the value of the entry in `fact_row(k)`.
- `scale` is an allocatable rank-1 array of package type. On exit, if `control%iscale` $>$ `0`, it is allocated to have size `n` and holds the scaling factors for C.
- `invp` is an allocatable rank-1 array of type `INTEGER`. On exit, if `control%iorder` $>$ `0`, it is allocated to have size `n` and specifies the permutation such that the j-th column of the permuted matrix Q^TCQ is the `invp(j)`-th column of C (that is, `invp(j)` is the index of the `j`-th pivot).
- `perm` is an allocatable rank-1 array of type `INTEGER`. On exit, if `control%iorder` $>$ `0`, it is allocated to have size `n` and specifies the elimination ordering such that `perm(i)` holds the position of the `i`-th column of C in the elimination order.

`control` is an `INTENT(IN)` scalar of type `MI35_control` (see Section 2.5).

`info` is an `INTENT(INOUT)` scalar of type `MI35_info`. Its components provide information about the execution of the subroutine, as explained in Section 2.6.

`weight` is an optional `INTENT(IN)` rank-one array of package type and size `m` that, if present, must be set by the user to hold the diagonal entries of the matrix W. If `weight` is not present, $W = I$ is used.

`scale` is an optional `INTENT(IN)` rank-one array of package type and size `n` that must be present if `control%order=5`. In this case, `scale` must be set by the user to hold scaling factors for C.

`perm` is an optional `INTENT(IN)` rank-one array of type `INTEGER` and size `n` that must be present if `control%iorder=3`. In this case, the user must supply an elimination ordering such that `perm(i)` holds the position of the `i`-th column of C in the elimination order.

2.4.5 To compute the matrix C

The user may optionally choose to compute and store the lower triangular part of the matrix C and then use `MI35_factorizeC` in place of `MI35_factorize`. To compute the lower triangular part of C, a call of the following form may be made:

```
call MI35_formC(m,n,ptr,row,val,ptrC,rowC,valC,control,info,weight)
```

`m, n, ptr, row, val, control, info, weight`: see Section 2.4.4.

`ptrC` is an `INTENT(OUT)` rank-1 array of type `INTEGER` and size `n+1`. On exit, `ptrC(j)` is the position in `rowC` of the first entry in column `j` and `ptrC(n+1)` is one more than the number of entries in the lower triangular part of C.

`rowC` is an `INTENT(OUT)` rank-1 allocatable array of type `INTEGER`. On exit, it will be allocated to have size at least `ptrC(n+1)-1` and will hold the row indices of the entries of the lower triangular part of C with the row indices for the entries in column 1 preceding those for column 2, and so on. The diagonal entry will be the first entry in each column and the column indices will be in increasing order.

`valC` is an `INTENT(INOUT)` rank-1 allocatable array of package type. On exit, it will be allocated to have size at least `ptrC(n+1)-1` and `valC(k)` will hold the value of the entry in `rowC(k)`.

2.4.6 To compute an incomplete Cholesky factorization of C

To compute an incomplete Cholesky factorization when the lower triangular part of C is available explicitly, a call of the following form should be made:

```
call MI35_factorizeC(n, ptrC, rowC, valC, lsize, rsize, keep, control, info[, &
        scale, perm])
```

`n` is an `INTENT(IN)` scalar of type `INTEGER` that must hold the dimension of C.

`ptrC` is an `INTENT(IN)` rank-1 array of type `INTEGER` and size `n+1`. `ptrC(j)` must be set by the user so that `ptrC(j)` is the position in `rowC` of the first entry in column `j` and `ptrC(n+1)` must be set to one more than the number of entries in the lower triangular part of C.

`rowC` is an `INTENT(IN)` rank-1 array of type `INTEGER` and size at least `ptrC(n+1)-1`. It must hold the row indices of the entries of the lower triangular part of C with the row indices for the entries in column 1 preceding those for column 2, and so on. The diagonal must be present and must be the first entry in each column.

`valC` is an `INTENT(INOUT)` rank-1 array of package type and size at least `ptrC(n+1)-1`. `valC(k)` must hold the value of the entry in `rowC(k)`.

`lsize, rsize, keep, control, info, scale, perm`: see Section 2.4.4.

2.4.7 To perform preconditioning operations

The incomplete Cholesky factorization preconditioner may be applied to compute $y = Pz$ by making a call as follows.

```
call MI35_precondition(n, keep, z, y, info)
```

`n, keep`: must be unchanged since the call to `MI35_factorize` or `MI35_factorizeC`.

`z` is an `INTENT(IN)` rank-1 array of package type and size `n`. It must be set by the user to hold the vector z to which the incomplete factorization preconditioner P is to be applied.

`y` is an `INTENT(OUT)` rank-1 array of package type and size `n`. On exit, `y` contains Pz.

`info` is an `INTENT(INOUT)` scalar of type `MI35_info`. Only the component `info%flag` is accessed (see Section 2.6).

2.4.8 To perform solve operations

The system $\overline{L}y = SQ^Tz$ or $\overline{L}^T S^{-1}Q^Ty = z$ may be solved by making a call as follows.

```
call MI35_solve(trans, n, keep, z, y, info)
```

`trans` is an `INTENT(IN)` scalar of type `LOGICAL` that must to `.false` if the solution of $\overline{L}y = SQ^Tz$ is required and to `.true.` if the solution of $\overline{L}^T S^{-1}Q^Ty = z$ is required.

`n, keep`: must be unchanged since the call to `MI35_factorize` or `MI35_factorizeC`.

`z` is an `INTENT(IN)` rank-1 array of package type and size `n`. It must be set by the user to the right-hand side vector z.

`y` is an `INTENT(OUT)` rank-1 array of package type and size `n`. On exit, `y` contains the solution vector y.

`info` is an `INTENT(INOUT)` scalar of type `MI35_info`. Only the component `info%flag` is accessed (see Section 2.6).

2.4.9 The finalisation subroutine

Once all other calls are complete for a problem or after an error return, a call should be made to free memory allocated by `hsl_MI35_factorize` or `hsl_MI35_factorizeC` using a call to `MI35_finalise`.

```
call MI35_finalise(keep, info)
```

`keep` is an `INTENT(INOUT)` scalar of type `MI35_keep` that must be passed unchanged. On exit, allocatable components will have been deallocated.

`info` is an `INTENT(INOUT)` scalar of type `MI35_info`. Only the components `info%flag` and `info%stat` are accessed (see Section 2.6).

2.5 The control derived data type

The derived data type `MI35_control` is used to hold controlling data. The components are automatically given default values in the definition of the type.

Components that control printing

`unit_error` is a scalar of type `INTEGER` with default value `6` that is used as the output stream for error messages. If it is negative, these messages will be suppressed.

`unit_warning` is a scalar of type `INTEGER` with default value `6` that is used as the output stream for warning messages. If it is negative, these messages will be suppressed.

Components that control the initial and subsequent choice of the shift α.

Note that α needs to be chosen to avoid breakdown of the Cholesky factorization process; it should be small but not so small as to cause instability (see Section 4).

`alpha` is a scalar of package type with default value 0.0 that holds the initial shift α. Values less than zero are treated as zero.

`lowalpha` is a scalar of package type with default value 0.001 that controls the choice of the first non-zero shift in the event of a breakdown. Values less than or equal to zero are treated as the default.

`maxshift` is a scalar of type `INTEGER` with default value `3` that controls the maximum number of times the shift can be decreased after a successful factorization with a positive shift. See Section 4 for details. Limiting `maxshift` may reduce the factorization time but may result in a poorer quality preconditioner.

`shift_factor` is a scalar of package type with default value 2.0 that controls how rapidly the shift is increased after a breakdown. See Section 4 for details. Increasing `shift_factor` may reduce the factorization time but may result in a poorer quality preconditioner. Values less than one are treated as the default.

`shift_factor2` is a scalar of package type with default value 4.0 that controls how rapidly the shift is decreased after a successful factorization with a positive shift. See Section 4 for details. Values less than one are treated as the default.

`small` is a scalar of type `REAL`. Any pivot whose modulus is less than `small` is treated as zero and, if such a pivot is encountered, the factorization breaks down, the shift is increased and the factorization restarted. The default in the double version is 10^{-20} and in the single version is 10^{-12}.

Components that control the dropping of small entries

`tau1` and `tau2` are scalars of package type with default values 0.001 and 0.0001. They control the dropping of entries from L and R. In the computation of the incomplete factorization, entries of magnitude less than `|tau1|` are dropped from L; those that are at least `|tau2|` but less than `|tau1|` may be included in R while those less than `|tau2|` are dropped from R.

`weight_tol` is a scalar of package type with default value `0.0`. Weights (entries of W) that have absolute value less than or equal to `weight_tol` are treated as zero (and are referred to as zero weights).

Other components

`iorder` is a scalar of type `INTEGER` with default value `6` that indicates the ordering of C that is required. Options available are:

$\leq$ `0` no ordering.

`1` A reverse Cuthill-McKee (RCM) ordering (computed using `MC61`) is used.

`2` An approximate minimum degree (AMD) ordering (computed using `HSL_MC68`) is used.

`3` User-supplied ordering is used.

`4` The rows are ordered by ascending degree.

`5` METIS (nested dissection) ordering with default settings is used. If METIS is not supplied and this option is requested, the routine will return immediately with an error.

`6` A Sloan profile reduction ordering (computed using `MC61`) is used. This is the default.

If `iorder`>`6`, the default is used.

`iscale` is a scalar of type `INTEGER` with default value `4` that indicates the scaling that is required. Options available are:

$\leq$ `0` No scaling.

`1` Scaling generated using the l_2-norm of the columns of C. This is the default. Note that in `MI35_factorize`, this option will incur an overhead as the columns of C must be computed one at a time and then discarded after use.

`2` Scaling of C generated by applying the iterative method of the package `MC77` for one iteration in the infinity norm and three iterations in the one norm (equilibration ordering). Only available in `MI35_factorizeC` (in `MI35_factorize`, the default ordering is used).

`3` Scaling of C generated from a weighted bipartite matching using the package `HSL_MC64`. Only available in `MI35_factorizeC` (in `MI35_factorize`, the default ordering is used).

`4` Diagonal scaling is used so that the diagonal entries of SCS are one.

`5` User-supplied scaling of C is used. The user must supply scaling factors for C.

For all other values of `iscale`, the default is used.

`limit_rowA` is a scalar of type `INTEGER` with default value `-1`. If a row of A has more than `limit_rowA entries`, `MI35_checkA` terminates. If `limit_rowA` is negative, the limit is `n`.

`limit_colC` is a scalar of type `INTEGER` with default value `-1`. If a column of C has more than `limit_colC` entries, the computation terminates If `limit_colC` is negative, the limit is `n`.

`limit_C` is a scalar of type `INTEGER` with default value `-1`. If C (or the pattern of C) is formed explicitly and it is found to have more than `limit_C` entries (upper and lower triangular parts), the computation terminates. If `limit_C` is negative, the limit is `huge(1)`.

`rrt` is a scalar of type `LOGICAL` with default value `.false.` that is used to control whether the entries of RR^T (see (1.2)) that cause no additional fill-in in (1.1) are allowed (`rsize` $>$ `0` only). Allowing such entries can improve the quality of the preconditioner (although this is not guaranteed) but at some additional computational cost in the factorization process. If `rrt`=`.true.` such entries are allowed; otherwise, they are not allowed.

2.6 The derived data type for holding information

The derived data type `MI35_info` is used to hold information from the execution of `MI35_factorize`. The components are:

`alpha` is a scalar of package type that holds the final shift (it is set to zero if no shift is used).

`avlenC` is a scalar of type `INTEGER` that holds the average number of entries in a row/column of C.

`band_after` is a scalar of type `INTEGER`. If `control%iorder=1` or `6`, `band_after` holds the semibandwidth of A after reordering; otherwise, it is set to 0.

`band_before` is a scalar of type `INTEGER`. If `control%iorder=1` or `6`, `band_before` holds the semibandwidth of A before reordering; otherwise, it is set to 0.

`dup` is a scalar of type `INTEGER` that holds the number of duplicated indices removed from `row`.

`flag` is a scalar of type `INTEGER` that gives the exit status of the algorithm (details in Section 2.7).

`flag61` is a scalar of type `INTEGER` that holds the exit status on return from `MC61` (and is set to 0 if `MC61` is not used).

`flag64` is a scalar of type `INTEGER` that holds the exit status on return from `HSL_MC64` (and is set to 0 if `HSL_MC64` is not used).

`flag68` is a scalar of type `INTEGER` that holds the exit status on return from `HSL_MC68` (and is set to 0 if `HSL_MC68` is not used).

`flag77` is a scalar of type `INTEGER` that holds the exit status on return from `MC77` (and is set to 0 if `MC77` is not used).

`maxlen` is a scalar of type `INTEGER` that holds the largest number of entries in a row of A (after removal of duplicates and out-of-range entries).

`maxlenC` is a scalar of type `INTEGER` that holds the largest number of entries in a row/column of C.

`nrestart` is a scalar of type `INTEGER` that holds the number of restarts (after reducing the shift).

`nshift` is a scalar of type `INTEGER` that holds the number of non-zero shifts used.

`nnz_C` is a scalar of type `INTEGER` that holds the number of entries in the lower triangular part of C.

`nzero` is a scalar of type `INTEGER` that holds the number of explicit zero entries found and removed from A.

`nzero_weight` is a scalar of type `INTEGER` that holds the number of zero weights (that is, the number of weights that have absolute value less than or equal to `control%weight_tol`) that have been removed from W.

`oor` is a scalar of type `INTEGER` that holds the number of out-of-range indices removed from `row`.

`profile_after` is a scalar of package type. If `control%iorder=1` or `6`, `profile_after` holds the profile of A after reordering; otherwise, it is set to 0.0.

`profile_before` is a scalar of package type. If `control%iorder=1` or `6`, `profile_before` holds the profile of A before reordering; otherwise, it is set to 0.0.

`size_r` is a scalar of type `INTEGER(long)` that holds the size of the integer and real arrays that are used during the factorization to hold R.

`stat` is a scalar of type `INTEGER` that holds the Fortran stat parameter.

2.7 Warning and error messages

A successful return from a subroutine in the package is indicated by `info%flag` having the value zero. A negative value is associated with an error message that by default will be output on unit `control%unit_error`.

Possible negative values are:

`-1` memory allocation failed. The `stat` parameter is returned in `info%stat`.

`-2` The array `row` is too small.

`-3` The array `val` is too small.

`-4` `m`$\geq$`n`$\geq$`1` is not satisfied. Note that this is checked after the removal of null rows and columns and rows corresponding to zero weights,

`-5` Error in the array `ptr`.

`-6` All columns of *A* are null columns (this could happen if all the row indices are out-of-range or if all entries of *A* are zero).

`-7` Unexpected error returned by `MC77`. The `MC77` exit status is returned in `info%flag77`.

`-8` Unexpected error returned by `HSL_MC64`. The `HSL_MC64` exit status is returned in `info%flag64`.

`-10` The optional argument `scale` is not present when it should be.

`-11` The optional argument `invp` is either not present when it should be or it does not hold a permutation.

`-12` Unexpected error returned by `MC61`. The `MC61` exit status is returned in `info%flag61`. Note that, if the matrix has not been checked for errors and there are duplicated or out-of-range entries in `row`, `mc61` will return an error flag of `-4` and the computation will terminate.

`-13` METIS ordering has been requested (`control%iorder` = `5`) but METIS is not linked.

`-14` Memory deallocation failed. The `stat` parameter is returned in `info%stat`.

`-15` One or more columns of *C* has more than `control%limit_colC` entries.

`-16` *C* has more than `control%limit_C` entries.

`-17` One or more rows of *A* has more than `control%limit_rowA` entries.

Positive values for `info%flag` are associated with a warning. Possible positive values are:

`+1` Out-of-range indices have been removed from `row`. The number of such entries is given in `info%oor`.

`+2` Duplicated entries were found in `row`; these have been removed and the corresponding entries in `val` have been summed. The number of such entries is given in `info%dup`.

`+3` Explicit zeros have been removed from `val`.

`+4` Null rows have been found and removed.

`+5` Null columns have been found and removed.

`+6` Zero weight(s) (that is, weights that have absolute value less than or equal to `control%weight_tol`) found and removed. The corresponding rows have been removed.

+7 A problem was encountered when computing the requested ordering. The original ordering is retained.

+8 Either HSL_MC64 found C to be structurally singular. or the computed scaling factors are large and may cause overflow when used to scale the matrix. No scaling is used.

+9 Warning returned by MC61. Profile not reduced so original order retained.

3 GENERAL INFORMATION

Input/output: Error messages on unit control%lp and warning and diagnostic messages on units control%wp and control%mp, respectively. These have default value 6; printing of these messages is suppressed if the relevant unit number is negative or if print_level is negative.

Restrictions: m≥n≥1.

Changes from Version 1
Version 2 includes a new control parameter control%weight_tol that is used to define whether a weight is to treated as zero. Routine MI35_formC has been revised so that, within each column, the entries of the array rowC are ordered by increasing row index.

4 METHOD

MI35_checkA checks the matrix data for errors. Checking removes out-of-range entries, sums duplicates, removes any entries with value zero, and removes any null rows or columns. In addition, if weights are supplied, any zero weights are removed and the corresponding rows of A are removed. A weight is considered to be zero if its absolute value is less than or equal to control%weight_tol (with default value 0.0). If a vector b is input, entries of b corresponding to removed rows of A are also removed. Checking of the data is recommended as no further checking of the user-supplied data is carried out and errors (including explicit zeros and/or null rows/columns) may lead to unpredictable behaviour later in the computation.

The user has a choice of computing the matrix C explicitly and then calling MI35_factorizeC to form the incomplete factorization or avoiding forming C by calling MI35_factorize. The latter requires less memory (note that C will typically have many more entries than A) but as some scaling routines (MC64 and MC77) require C to be held explicitly, limited scaling options are available for use with MI35_factorize. Unless a problem is known to be well scaled, **scaling is highly recommended**; we recommend using the default of diagonal scaling. This normalizes the columns of WA to have length 1 and the diagonals of SQ^TCQS are then 1. If MI35_factorize is called, columns of C are computed as they are required (and the pattern of C is computed and stored if control%iorder $= 1,2,5$ or 6).

MI35_factorize (and MI35_factorizeC) employ a left-looking sparse Cholesky algorithm to compute the incomplete factorization, one column at a time. The algorithm used is the same as that employed by HSL_MI28 but note that, since C may have one or more dense (or close to dense) columns, in MI35_factorize (and MI35_factorizeC) lsize is the desired number of entries in each column j of L (whereas in HSL_MI28 the number of entries in column j is lsize$+nz_j$, where nz_j is the number of entries in column j of the matrix that is being factorized).

The parameters lsize and rsize control the amount of memory used as well as the amount of work involved in computing the factorization. lsize controls the number of entries in each column of the computed incomplete factor L and rsize limits the number of entries in each column of the matrix R. If rsize $= 0$ and control%tau1 $= 0.0$, the incomplete factorization is essentially that of [1] applied to C. However, it is generally advantageous (in terms of the quality of the preconditioner) to use rsize > 0. Increasing lsize and/or rsize increases the cost of the factorization (in terms of time and memory). Furthermore, increasing lsize leads to a denser incomplete factorization (but one that is, in general, a better preconditioner), increasing the cost of each call to MI35_precondition and MI35_solve.

Dropping parameters control%tau1 and control%tau2 are used to further sparsify L and R, respectively. As each column of L is computed, entries of absolute value less than control%tau1 are dropped. These may be included in R but entries less than control%tau2 are dropped from R.

In the event of breakdown within the factorization (that is, a pivot is encountered that is smaller in absolute value than control%small), a global diagonal shift α is used. It is important to try and use as small a shift as possible but also to limit the number of breakdowns. The user can supply an initial shift α_0. Otherwise, if $\beta = \min(\overline{a}_{ii}) > 0$, then $\alpha_0 = 0.0$; otherwise, $\alpha_0 = -\beta + \texttt{control\%lowalpha}$. The incomplete factorization algorithm is applied to $\overline{A}_0 = SQ^T AQS + \alpha_0 I$. If breakdown occurs, a larger shift

$$\alpha_1 = \max(\texttt{control\%lowalpha},\ \alpha_0 \times \texttt{control\%shift\_factor}),$$

is tried. The process continues until an incomplete factorization of $\overline{A}_k = SQ^T AQS + \alpha_k I$ is successful. If breakdown occurs at the same (or nearly the same) stage of the factorization for two successive shifts, to try and limit the number of restarts, α is increased by a factor of $2 \times \texttt{control\%shift\_factor}$. Conversely, if $\alpha_k = \texttt{control\%lowalpha}$, to prevent an unnecessarily large shift from being used, the shift is decreased by setting

$$\alpha_{k+1} = \alpha_k / \texttt{control\%shift\_factor2},$$

and applying the incomplete factorization algorithm to $\overline{A}_{k+1} = SQ^T AQS + \alpha_{k+1} I$. If this factorization is also breakdown free, the process is repeated (up to control%maxshift times). In all cases, the value of the final shift is returned to the user in info%alpha, along with the number of shifts tried and the number of restarts (info%nrestart).

Further details of the factorization algorithm may be found in [2]; an outline of the algorithm is also given in the documentation for HSL_MI28.

References:

[1] C.-J. Lin and J. J. Moré. (1999). Incomplete Cholesky factorizations with limited memory. SIAM Journal on Scientific Computing, **21**, 24-45.

[2] J. A. Scott and M. Tůma. (2014). HSL_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code. ACM Transactions on Mathematical Software, 40, 24:1–24:19.

[3] J. A. Scott and M. Tůma. (2014). On positive semidefinite modification schemes for incomplete Cholesky factorization. SIAM J. Sci. Computing, 36, pp A609-A633.

5 EXAMPLE OF USE

Suppose we wish to compute the vector x that minimizes $||W(Ax-b)||_2$, where

$$A = \begin{pmatrix} 2. & 3. & 1. \\ 0. & 0. & 4. \\ 0. & 1. & 0. \\ 0. & 0. & 5. \end{pmatrix} \quad W = \begin{pmatrix} 2. & & & \\ & 1. & & \\ & & 2. & \\ & & & 1. \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 8. \\ 12. \\ 2. \\ 15. \end{pmatrix}.$$

We may use the following code:

```
program mi35_spec_double

  ! example to illustrate use of HSL_MI35 without forming C = A^T * W^2 *W
  ! explicitly
  use hsl_mi35_double
```

```
implicit none

integer, parameter :: wp = kind(1.0d0)
real(wp), parameter :: zero = 0.0_wp

type(mi35_control) :: control
type(mi35_info) :: info
type(mi35_keep) :: keep

integer,  allocatable :: ptr(:),row(:)
real(wp), allocatable :: b(:),val(:),weight(:),w(:,:),y(:)

! Arrays and scalars required by the CG code mi21
real(wp) :: resid
real(wp) :: cntl21(5),rsave21(6)
integer  :: icntl21(8),isave21(10),info21(4)

integer :: iact,locy,locz,lsize,m,n,nz,rsize

! Read in the matrix data
read (5,*) m,n,nz

allocate (ptr(n+1),row(nz),val(nz),w(n,4),weight(m),b(m),y(m))
read (5,*) ptr(1:n+1)
read (5,*) row(1:nz)
read (5,*) val(1:nz)
read (5,*) weight(1:m)
read (5,*) b(1:m)


call mi35_check_matrix(m,n,ptr,row,val,control,info,weight=weight)
if (info%flag.lt.0) then
   print *, "mi35_check_matrix failed with error ", info%flag
   go to 99
end if

! switch off ordering and scaling
control%iorder = 0
control%iscale = 0

! Set space parameters
lsize = 1
rsize = 1
call mi35_factorize(m,n,ptr,row,val,lsize,rsize,keep,control,info, &
      weight=weight)
if(info%flag.lt.0) then
   print *, "mi35_factorize failed with error ", info%flag
   go to 99
end if
```

```
   ! Prepare to run CG on the normal equations using mc21
   ! Compute A^T * W^2 * b (right hand side)
   call matrix_mult_trans(m,n,ptr,row,val,b,w(1:n,1),weight=weight)

   call mi21id(icntl21, cntl21, isave21, rsave21)
   icntl21(3) = 1 ! we will use preconditioning

   iact = 0
   do
     call mi21ad(iact, n, w, n, locy, locz, resid, icntl21,  &
                 cntl21, info21, isave21, rsave21)

     if (iact == -1) then
        write (*,'(a,i4)') ' Unexpected error from mi21. flag = ',info21(1)
        exit

     else if (iact == 1) then
        write (*,'(a)') ' CG Converged'
        write (*,'(a)') ' Solution = '
        write (*,'(5es12.4)') w(1:n,2)
        exit

     else if (iact == 2) then
         ! Perform product with C, without forming C explicitly.
         ! First, compute y = W^2 * A * w(:,locz)
         call matrix_mult(m,n,ptr,row,val,w(1:n,locz),y,weight=weight)

         ! Now compute w(:,locy) = A^T * y
         call matrix_mult_trans(m,n,ptr,row,val,y,w(1:n,locy))

     else if (iact == 3) then
         call mi35_precondition(n,keep,w(1:n,locz),w(1:n,locy),info)
     end if
   end do

99 call mi35_finalise(keep,info)

contains
!************************************************************

      ! Takes b and computes z = A^T * W^2 * b (or A^T * b)

      subroutine matrix_mult_trans(m,n,ptr,row,val,b,z,weight)

      integer, intent(in) :: m,n
      integer, intent(in) :: ptr(n+1),row(:)
      real(wp), intent(in) :: val(:),b(m)
      real(wp), intent(out) :: z(n)
      real(wp), optional, intent(in) :: weight(m)

      integer:: i,j,k
      real(wp) :: sum

      if (present(weight)) then
         do j = 1,n
            sum = zero
            do k = ptr(j),ptr(j+1)-1
               i = row(k)
               sum = sum + val(k)*weight(i)*b(i)
            end do
            z(j) = sum
         end do
      else
         do j = 1,n
            sum = zero
            do k = ptr(j),ptr(j+1)-1
               i = row(k)
               sum = sum + val(k)*b(i)
            end do
            z(j) = sum
         end do
      end if

      end subroutine matrix_mult_trans
!**************************************************************
      ! Takes b and computes z = W^2 * A * b (or A * b)

      subroutine matrix_mult(m,n,ptr,row,val,b,z,weight)

      integer, intent(in) :: m,n
      integer, intent(in) :: ptr(n+1),row(:)
      real(wp), intent(in) :: val(:),b(n)
      real(wp), intent(out) :: z(m)
      real(wp), optional, intent(in) :: weight(m)

      integer:: i,j,k
      real(wp) :: sum

      z = zero

      do j = 1,n
         do k = ptr(j),ptr(j+1)-1
            i = row(k)
            z(i) = z(i) + val(k)*b(j)
         end do
      end do
      if (present(weight)) then
```

```
         do j = 1,m
            z(j) = z(j) * weight(j)
         end do
      end if

      end subroutine matrix_mult
 end program mi35_spec_double
```

With the input data:

```
  4  3  5
  1  2  4  6
  1  1  3  2  4
  2.0  3.0  1.0  4.0  5.0
  2.0  1.0  2.0  1.0
  8.0 12.0  2.0 15.0
```

we obtain the following output:

```
 CG Converged
 Solution =
  1.0000E+00  2.0000E+00  3.0000E+00
```