



1 SUMMARY

Given an $n \times n$ sparse symmetric matrix $A = \{a_{ij}\}$, HSL_MI28 **computes an incomplete Cholesky factorization** that may be used as a preconditioner. The matrix A is optionally reordered, scaled and, if necessary, shifted to avoid breakdown of the factorization so that the LL^T incomplete factorization of the matrix

$$\bar{A} = SQ^T AQS + \alpha I$$

is computed, 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 sparse symmetric linear system $Ax = b$. A separate entry performs the preconditioning operation

$$y = Pz$$

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

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

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

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).

ATTRIBUTES — **Version:** 2.4.2 (1 November 2023). **Interfaces:** C, Fortran, MATLAB. **Types:** Real (single, double). **Calls:** KB07, MC61, HSL_MC64, HSL_MC68, HSL_MC69, MC77, `_copy` and (optionally using METIS version 4.x) `METIS_NODEND`. **Original date:** March 2013. **Origin:** J. A. Scott, STFC Rutherford Appleton Laboratory and M. Tůma, Institute of Computer Science, Academy of Sciences of the Czech Republic. **Language:** Fortran 2003 subset (F95 + TR155581 + C interoperability). **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_mi28_single
```

Double precision version

```
use hsl_mi28_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:

All use is subject to licence.

- (a) `mi28_factorize` computes an incomplete Cholesky factorization.
- (b) `mi28_precondition` performs the preconditioning operation $y = Pz$, where P is the incomplete factorization preconditioner computed by `mi28_factorize`.
- (c) `mi28_solve` solves the system $\bar{L}y = SQ^T z$ (or $\bar{L}^T S^{-1} Q^T y = z$), where \bar{L} is the incomplete factor computed by `mi28_factorize`.
- (d) `mi28_finalise` frees memory that has been allocated by `mi28_factorize`.

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 `mi28_keep`, `mi28_control` and `mi28_info`. The following pseudocode illustrates this.

```

use hsl_mi28_double
...
type (mi28_keep) :: keep
type (mi28_control) :: control
type (mi28_info) :: info

...

```

The components of `mi28_control` and `mi28_info` are explained in Sections 2.5 and 2.6. The components of `mi28_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_MI28 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, `mi28_factorize` will return with an error.

Important: At present, HSL_MI28 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 compute an incomplete Cholesky factorization

To compute an incomplete Cholesky factorization, the lower triangular part of the matrix A must be held in compressed column storage and a call of the following form must be made:

```
call mi28_factorize(n, ptr, row, val, lsize, rsize, keep, control, info[, scale, perm])
```

n is an INTENT(IN) scalar of type INTEGER that must hold the order of A . **Restriction:** $n \geq 1$.

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 $control\%check$ is set to `.true.` (the default) and duplicates and/or out-of-range indices are found.

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 the lower triangular part of A with the row indices for the entries in column 1 preceding those for column 2, and so on. Within each column, the row indices must be in increasing order and the diagonal must be present. If $control\%check$ is set to `.true.` (the default), row is checked for errors and duplicates and out-of-range indices are removed; otherwise, row is unchanged.

val is an INTENT(INOUT) rank-1 array of package type and size at least $ptr(n+1)-1$. The data in $val(k)$ must hold the value of the entry in $row(k)$. If $control\%check$ is set to `.true.` (the default), on exit duplicates are summed and out-of-range indices removed; otherwise, val is unchanged.

$lsize$ is an INTENT(IN) scalar of type INTEGER that determines the maximum number of fill entries within each column of the incomplete factor L . 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 within each column of 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 `mi28_keep`. It is used to hold data about the problem being solved and must be passed unchanged to `mi28_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 $fact_row(k)$. The diagonal entry will always be the first entry in each column, and $1/l_{ii}$ is stored rather than l_{ii} .

$scale$ 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 A .

`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 $Q^T A Q$ is the `invp(j)`-th column of A (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 A in the elimination order.

`control` is an `INTENT(IN)` scalar of type `mi28_control` (see Section 2.5).

`info` is an `INTENT(OUT)` scalar of type `mi28_info`. Its components provide information about the execution of the subroutine, as explained in Section 2.6.

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

`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 A in the elimination order.

2.4.4 To perform preconditioning operations

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

```
call mi28_precondition(n, keep, z, y, info)
```

`n`, `keep`: must be unchanged since the call to `mi28_factorize`.

`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 `mi28_info`. Only the component `info%flag` is accessed (see Section 2.6).

2.4.5 To perform solve operations

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

```
call mi28_solve(trans, n, keep, z, y, info)
```

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

`n`, `keep`: must be unchanged since the call to `mi28_factorize`.

`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 `mi28_info`. Only the component `info%flag` is accessed (see Section 2.6).

2.4.6 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_mi28_factorize` using a call to `mi28_finalise`.

```
call mi28_finalise(keep, info)
```

`keep` is an `INTENT(INOUT)` scalar of type `mi28_keep` that must be passed unchanged. On exit, allocatable components will have been deallocated.

`info` is an `INTENT(INOUT)` scalar of type `mi28_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 `mi28_control` is used to hold controlling data; it is used by `mi28_factorize` only. 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 the aim is to choose α to be as small as possible to avoid breakdown of the Cholesky factorization process (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} while in the single version it 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 $|\text{tau1}|$ are dropped from L ; those that are at least $|\text{tau2}|$ but less than $|\text{tau1}|$ may be included in R while those less than $|\text{tau2}|$ are dropped from R .

Other components

`check` is a scalar of type LOGICAL with default value `.true.`. If `.true.`, the matrix data is checked for errors and the cleaned matrix (duplicates are summed, out-of-range entries discarded and, within each column, the entries are ordered by increasing row index) overwrites the user-supplied data in `ptr`, `row` and `val`. Otherwise, no checking of the matrix data is carried out (it is important to note that any out-of-order entries or out-of-range entries or duplicates may cause HSL_MI28 to fail in an unpredictable way) and so it is recommended that the matrix data is checked.

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

≤ 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 1 that indicates the scaling that is required. Options available are:

≤ 0 No scaling.

1 Scaling generated using the l_2 -norm of the columns of A . This is the default.

2 Scaling 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).

3 Scaling generated from a weighted bipartite matching using the package HSL_MC64.

4 Diagonal scaling is used.

5 User-supplied scaling is used. The user must supply scaling factors for A .

If `iscale` > 5, the default is used.

`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 `mi28_info` is used to hold information from the execution of `mi28_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).

`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).

`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.

`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 n is out of range.
- 5 Error in the array `ptr`.
- 6 One or more diagonal entries in A is missing.
- 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`.
- 9 A is found to be singular.
- 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 Unexpected error returned by `HSL_MC68`. The `HSL_MC68` exit status is returned in `info%flag68`. Note that this error is returned if `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`.

Positive values for `info%flag` are associated with a warning and can only be returned by `mi28_factorize`. 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 A warning has been issued by `HSL_MC64` that the computed scaling factors are large and may cause overflow when used to scale the matrix. No scaling is used.
- +4 A warning has been issued by `MC61`. The `MC61` exit status is returned in `info%flag61`.
- +5 One or more of the diagonal entries of the matrix is non-positive. A non-zero diagonal shift is used.

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: $n \geq 1$.

4 METHOD

`mi28_factorize` starts by optionally checking the matrix data for errors; this is done using `HSL_MC69`. Checking removes out-of-range entries, sums duplicates, and reorders the entries within each column by increasing row index. A scaling and/or ordering is then optionally computed; HSL packages are used to do this. Unless a problem is known to be well scaled, **scaling is highly recommended**.

A left-looking sparse Cholesky algorithm is used to compute the incomplete factorization, one column at a time. 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 the computed incomplete factor L (at most `lsize` fill entries are permitted in each column of 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]. 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 `mi28_precondition` and `mi28_solve`. In experiments, reported in [2] it has been found that using values of `lsize` and `rsize` equal to 10 gives good results but, if the preconditioner is to be used for many problems, it may be worthwhile to experiment with a range of values to try and get the best overall performance.

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 $\beta = \min(\bar{a}_{ii}) > 0$, then $\alpha_0 = 0.0$; otherwise, $\alpha_0 = -\beta + \text{control\%lowalpha}$. The incomplete factorization algorithm is applied to $\bar{A}_0 = SQ^T AQS + \alpha_0 I$. If breakdown occurs, a larger shift

$$\alpha_1 = \max(\text{control\%lowalpha}, \alpha_0 \times \text{control\%shift_factor}),$$

is tried. The process continues until an incomplete factorization of $\bar{A}_k = SQ^T AQS + \alpha_k I$ 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 $2 \times \text{control\%shift_factor}$. Conversely, if $\alpha_k = \text{control\%lowalpha}$, to prevent an unnecessarily large shift from being used, the shift is decreased by setting

$$\alpha_{k+1} = \alpha_k / \text{control\%shift_factor2},$$

and applying the incomplete factorization algorithm to $\bar{A}_{k+1} = SQ^T AQS + \alpha_{k+1} I$. 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`).

An outline of the algorithm is given below; further details of the factorization algorithm may be found in [2]. In Figure 1, $A = \{a_{ij}\}$ and a_j, l_j and r_j denote the j -th columns of the lower triangular parts of A, L and R , respectively, n_j is the number of entries in the j -th column of the strictly lower triangular part of A ; w is a workarray of length n .

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. (2013). `HSL_MI28`: an efficient and robust limited-memory incomplete Cholesky factorization code. RAL Technical Report. RAL-P-2013-004. See also *ACM Trans. Math. Software* **40** (2014), 24:1–24:19.

Figure 1: Outline of the HSL_MI28 algorithm

```

! Reorder
Compute an ordering  $Q$  for  $A$ 
Permute the matrix:  $A \leftarrow Q^T A Q$ 

! Scale
Compute a diagonal scaling  $S$ 
Scale the matrix:  $A \leftarrow SAS$ 

! Diagonal shift
If control%alpha not input, choose  $\alpha$  such that the  $\min(a_{ii}) + \alpha > \text{control\%small}$ 
Initialise breakdown = false and  $\alpha_0 = 0$ 

! Loop over shifts
Set  $w = 0$ 
do
  Set  $A \leftarrow A + (\alpha - \alpha_0)I$  and  $d(1:n) = (a_{11}, a_{22}, \dots, a_{nn})$ 

  ! Factorization : loop over columns
  for  $j = 1 : n$  do
    Copy  $a_j$  into  $w$ 
    Apply  $LL^T + RL^T + LR^T$  updates from columns  $1 : j - 1$  to  $w$ 
    Apply  $LL^T + RL^T + LR^T$  updates from columns  $1 : j - 1$  to  $d(j+1:n)$ 
    if (control%rrt) apply  $RR^T$  updates from columns  $1 : j - 1$  that cause
      no additional fill-in in  $w$ 
    if ( $\min(d(j+1:n)) < \text{control\%small}$ ) then
      Set breakdown = true,  $\alpha_0 = \alpha$  and increase  $\alpha$ 
    exit
  end if
  Sort entries in  $w$  by magnitude
  Keep at most  $n_j + \text{lsize}$  entries of largest magnitude in  $l_j$  such that
    they are all at least control%tau1
  Keep at most rsize additional entries that are next largest in magnitude
    in  $r_j$  such that they are all at least control%tau2
  Reset entries of  $w$  to zero
  end do
  if breakdown = false exit
end do

```

[3] J. A. Scott and M. Tůma. (2013). On positive semidefinite modification schemes for incomplete Cholesky factorization. RAL Technical Report. RAL-P-2013-005. See also SIAM J. Sci. Computing 36 (2014), A609–A633.

5 EXAMPLE OF USE

Suppose we wish to use preconditioned conjugate gradients to solve the linear system $Ax = b$ with

$$A = \begin{pmatrix} 6 & 1 & 0 & 1 & -2 \\ 1 & 7 & 0 & 0 & 3 \\ 0 & 0 & 4 & -1 & 0 \\ 1 & 0 & -1 & 4 & 1 \\ -2 & 3 & 0 & 1 & 3 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 6 \\ 11 \\ 3 \\ 5 \\ 5 \end{pmatrix}.$$

We may use the following code:

```
program mi28_spec_double

  use hsl_mi28_double
  implicit none

  integer, parameter :: wp = kind(1.0d0)

  type(mi28_control) :: control
  type(mi28_info) :: info
  type(mi28_keep) :: keep

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

  ! Arrays and scalars required by the CG code mi21
  real(wp) :: resid
  real(wp) :: cnt121(5), rsave21(6)
  integer :: icnt121(8), isave21(10), info21(4)

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

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

  allocate (ptr(n+1), row(nz), val(nz), w(n, 4))
  read (5, *) ptr(1:n+1)
  read (5, *) row(1:nz)
  read (5, *) val(1:nz)
  read (5, *) w(1:n, 1) ! Right-hand side array

  ! Choose lsize and rsize
  lsize = 1
  rsize = 1
```

```

! Compute the preconditioner
call mi28_factorize(n, ptr, row, val, lsize, rsize, keep, control, info)
if (info%flag.lt.0) then
  write (*,'(a,i4)') &
  ' Unexpected error from mi28_factorize. flag = ',info%flag
  go to 99
end if

! Prepare to use the CG code mi21 with preconditioning
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,i3,a)') &
      ' CG Convergence in ',info21(2),' iterations'
    write (*,'(a)') ' Solution = '
    write (*,'(5es12.4)') w(1:n,2)
    !write (*,'(a,es12.4)') " 2-norm of residual =", resid
    exit

  else if (iact == 2) then
    call mxmult(n,ptr,row,val,w(1:n,locz),w(1:n,locy))

  else if (iact == 3) then
    call mi28_precondition(n, keep, w(1:n,locz), w(1:n,locy), info)
  end if
end do
99 call mi28_finalise(keep,info)

contains
!*****
! sparse matrix-vector multiplication y=A*x.
! Lower triangle of sparse matrix held, with diagonal
! entry the FIRST entry in each column.

  subroutine mxmult(n,ptr,row,val,x,y)

  real(wp), parameter :: zero = 0.0_wp

  integer, intent(in) :: n

```

```
integer, intent(in) :: ptr(n+1),row(:)
real(wp), intent(in) :: val(:),x(n)
real(wp), intent(out) :: y(n)

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

y = zero
do i = 1,n
  sum = zero
  y(i) = y(i) + val(ptr(i))*x(i)
  do j = ptr(i)+1,ptr(i+1)-1
    k = row(j)
    y(k) = y(k) + val(j)*x(i)
    sum = sum + val(j)*x(k)
  end do
  y(i) = y(i) + sum
end do

end subroutine mxmult

end program mi28_spec_double
```

With the input data:

```
5 11
1 5 7 9 11 12
1 2 4 5 2 5 3 4 4 5 5
6. 1. 1. -2. 7. 3. 4. -1. 4. 1. 3.
6. 11. 3. 5. 5.
```

we obtain the following output:

```
CG Convergence in 1 iterations
Solution =
1.0000E+00 1.0000E+00 1.0000E+00 1.0000E+00 1.0000E+00
```