Science and Technology Facilities Council

# HSL_MA87

PACKAGE SPECIFICATION HSL

## 1 SUMMARY

HSL_MA87 uses a direct method to solve **large sparse positive-definite symmetric linear systems** of equations $AX = B$. This package uses **OpenMP** and is designed for **multicore architectures.** It computes the **sparse Cholesky factorization**

$$A = PL(PL)^{\dagger}$$

where $L^{\dagger} = L^T$ (real symmetric) or $L^{\dagger} = L^H$ (complex Hermitian), $P$ is a permutation matrix and $L$ is lower triangular. Options are provided for the complementary forward and backward substitutions.

The efficiency of HSL_MA87 is dependent on the user-supplied elimination order. The HSL package HSL_MC68 may be used to obtain a suitable ordering.

The lower triangular part of $A$ must be supplied in compressed sparse column format. The HSL package HSL_MC69 may be used to convert data held in other sparse matrix formats and also to check the user's matrix data for errors.

If $A$ is indefinite and pivoting for numerical stability is required, the package HSL_MA86 should be used.

**ATTRIBUTES — Version:** 2.6.6 (1 November 2023). **Interfaces:** Fortran, C, MATLAB. **Types:** Real (single, double), Complex (single, double). **Calls:** HSL_MC34 and HSL_MC78, BLAS subroutines _gemm, _gemv, _herk, _syrk, _trsm, _trsv, and the LAPACK subroutine _potrf. **Original date:** January 2009, Version 2.0.0 December 2010. **Origin:** J.D. Hogg, J.K. Reid and J.A. Scott, Rutherford Appleton Laboratory. **Language:** Fortran 95, plus allocatable components of derived types. **Parallelism:** Uses OpenMP and its runtime library. **Remark:** Development of HSL_MA87 was supported by EPSRC grants EP/F006535/1 and EP/E053351/1.

## 2 HOW TO USE THE PACKAGE

### 2.1 OpenMP

OpenMP is used by the HSL_MA87 package to provide parallelism for 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.

Although the code may be compiled and run in serial mode, we recommend it is run in parallel on a multicore machine (other HSL solvers, notably MA57 or HSL_MA77 may be more appropriate if a serial code is required).

### 2.2 Calling sequences

Access to the package requires a USE statement of the form

*Single precision version*

```
USE HSL_MA87_single
```

*Double precision version*

```
USE HSL_MA87_double
```

*Complex version*

```
USE HSL_MA87_complex
```

*Double complex version*

```
USE HSL_MA87_double_complex
```

If it is required to use more than one module at the same time, the derived types (Section 2.3) must be renamed in one of the use statements.

The following subroutines are available to the user:

(a) MA87_analyse analyses the sparsity pattern of the matrix and, using the user-supplied elimination order, prepares the data structures for the factorization.

(b) MA87_factor uses the data structures set up by MA87_analyse to compute a sparse factorization. More than one call to MA87_factor may follow a call to MA87_analyse.

(c) MA87_factor_solve may be called in place of MA87_factor to factorize $A$ and, at the same time, solve the system $AX = B$. Multiple calls to MA87_factor_solve may follow a call to MA87_analyse.

(d) MA87_solve uses the computed factors generated by MA87_factor or MA87_factor_solve to solve systems $AX = B$ for one or more right-hand sides $B$. Multiple calls to MA87_solve may follow a call to MA87_factor or MA87_factor_solve. An option is available to perform a partial solution.

(e) MA87_sparse_fwd_solve uses the computed factors generated by MA87_factor or MA87_factor_solve to solve the triangular system $LX = B$ for a single sparse right-hand side $B$. Multiple calls to MA87_sparse_fwd_solve may follow a call to MA87_factor or MA87_factor_solve.

(f) MA87_finalise should be called after all other calls are complete for a problem (including or after an error return that does not allow the computation to continue). It deallocates the components of the derived data types allocated by the package.

### 2.3 The derived data types

For each problem, the user must employ the derived types defined by the module to declare scalars of the types MA87_keep, MA87_info, and MA87_control. The following pseudocode illustrates this.

```
use HSL_MA87_double
...
type (MA87_control) :: control
type (MA87_info) :: info
type (MA87_keep) :: keep
```

The components of MA87_control and MA87_info are explained in Sections 2.4.9 and 2.4.10. The components of MA87_keep are private and are used for communication between subroutines and between threads.

### 2.4 Argument lists and calling sequences

#### 2.4.1 Optional arguments

We use square brackets [ ] to indicate OPTIONAL arguments, which are always at the end of the argument list. Since we reserve the right to modify the argument list and 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, real and package types

INTEGER denotes default integer and INTEGER(long) denotes INTEGER(kind=selected_int_kind(18)).
REAL denotes default real if the single precision version or the complex version is being used, and double precision real if the double precision or double precision complex version is being used.

We use the term **package type** to mean default real if the single precision version is being used, double precision real for the double precision version, default complex for the complex version and double precision complex for the double complex version.

### 2.4.3 Input of the matrix $A$

The user must supply the **lower** triangular part of the matrix $A$ in standard HSL format. This is a compressed sparse column format with the entries within each column ordered by increasing row index. **No checks** are made on the user's data. It is important to note that any out-of-range entries or duplicates may cause HSL_MA87 to fail in an unpredictable way. Before using HSL_MA87, the HSL package HSL_MC69 may be used to check for errors and to handle duplicates (HSL_MC69 sums them) and out-of-range entries (HSL_MC69 removes them).

If the user's data is held using another standard sparse matrix format (such as coordinate format or sparse compressed row format), we recommend using a conversion routine from HSL_MC69 to put the data into standard HSL format. The user may additionally call mc69_set_values before each call to MA87_factor or MA87_factor_solve to change the values of the entries of $A$ (without altering the sparsity pattern). The input of $A$ and of new values is illustrated in Section 5.

### 2.4.4 To analyse the sparsity pattern and prepare for the factorization

```
call MA87_analyse(n,ptr,row,order,keep,control,info)
```

`n` is an `INTENT(IN)` scalar of type `INTEGER` that must hold the order of $A$.

`ptr` is an `INTENT(IN)` rank-1 array of type `INTEGER` and size `n+1`. `ptr(j)` must be set 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 total number of entries in the `lower` triangular part of $A$.

`row` is an `INTENT(IN)` rank-1 array of type `INTEGER`. The first `ptr(n+1)-1` entries 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.

`order` is an `INTENT(INOUT)` rank-1 array of type `INTEGER` and size at least `n`. It must specify the elimination order, that is, `order(i)` must hold the position of variable $i$ in the pivot sequence. On exit, `order` contains the elimination order that `MA87_factor` (or `MA87_factor_solve`) will be given; this order may give slightly more fill-in than the user-supplied order.

`keep` is an `INTENT(INOUT)` scalar of type `MA87_keep`. It is used to hold data about the problem being solved and must be passed unchanged to the other subroutines.

`control` is an `INTENT(IN)` scalar of type `MA87_control` (see Section 2.4.9).

`info` is an `INTENT(OUT)` scalar of type `MA87_info`. Its components provide information about the execution of the subroutine, as explained in Section 2.4.10.

### 2.4.5 To factorize the matrix and optionally solve $AX = B$

To factorize the matrix, a call of the following form may be made after the call to `MA87_analyse`:

```
call MA87_factor(n,ptr,row,val,order,keep,control,info)
```

If the user wishes to solve $AX = B$ at the same time as factorizing the matrix, he or she should instead make a call of the following form for one right-hand side:

```
call MA87_factor_solve(n,ptr,row,val,order,keep,control,info,x1)
```

or, for more than one right-hand side,

```
call MA87_factor_solve(n,ptr,row,val,order,keep,control,info,nrhs,lx,x)
```

n, ptr, row, order: must be unchanged since the call to MA87_analyse.

val is an INTENT(IN) rank-1 array of package type. The first ptr(n+1)-1 entries must be set so that val(k) holds the value of the entry in position k of row(k).

keep, control, info: see Section 2.4.4.

x1 is an INTENT(INOUT) rank-1 array of package type and of size at least n. On entry, x1(i) must hold the ith component of the right-hand side; on exit, it holds the corresponding solution.

nrhs is an INTENT(IN) scalar of type INTEGER that holds the number of right-hand sides. **Restriction:** nrhs$\geq$1.

lx is an INTENT(IN) scalar of type INTEGER that holds the first extent of x. **Restriction:** lx$\geq$n.

x is an INTENT(INOUT) rank-2 array of package type with extents lx and nrhs. On entry, x(i,j) must hold the ith component of the jth right-hand side; on exit, it holds the corresponding solution.

### 2.4.6 To solve linear systems using the computed factors

After the call to MA87_factor (or MA87_factor_solve), one or more calls of the following form may be made to solve $AX = B$. Partial solutions may be performed by appropriately setting the optional parameter job. For a single right-hand side,

```
call MA87_solve(x1,order,keep,control,info[,job])
```

or, for more than one right-hand side,

```
call MA87_solve(nrhs,lx,x,order,keep,control,info[,job])
```

x1, nrhs, lx, x, order: see Section 2.4.5.

keep, control, info: see Section 2.4.4.

job is an optional INTENT(IN) scalar of type INTEGER. If job = 0 or job is absent, $AX = B$ is solved. A partial solution may be computed by setting job to have one of the following values:

1 for solving $PLX = B$

2 for solving $(PL)^{\dagger}X = B$

**Restriction:** job = 0,1,2.

### 2.4.7 To solve $LX = B$ for sparse $B$

After the call to MA87_factor (or MA87_factor_solve), one or more calls of the following form may be made to solve $LX = B$ for a single sparse right-hand side

```
call MA87_sparse_fwd_solve(nbi,bindex,b,order,invp,nxi,index,x,w,keep,control,info)
```

nbi is an INTENT(IN) scalar of type INTEGER that must hold the number of nonzero entries in the right-hand side. **Restriction:** 1≤nbi≤n.

bindex is an INTENT(INOUT) rank-1 array of type INTEGER. The first nbi entries must hold the indices of the nonzero entries in the right-hand side. These entries are altered during the solve but are reset to the user-supplied indices on successful exit.

b is an INTENT(IN) rank-1 array of package type and of size at least n. If bindex(i)=k, b(k) must hold the k-th nonzero component of the right-hand side; other entries of b are not accessed.

order is an INTENT(IN) rank-1 array of type INTEGER and size at least n. It must be unchanged since the call to MA87_analyse, that is, order(i) must hold the position of variable $i$ in the elimination order that was passed to MA87_factor or MA87_factor_solve.

invp is an INTENT(IN) rank-1 array of type INTEGER and size at least n. invp(j) must hold the variable that is in the $j-$th position in the elimination order (thus if j=order(i), the user must set invp(j)=i).

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

index is an INTENT(OUT) rank-1 array of package type. On exit, the first nxi entries hold the indices of the nonzero entries in the solution.

x is an INTENT(INOUT) rank-1 array of package type and size at least n. On entry, it must be set by the user to zero. On exit, if index(i)=k, x(k) holds the k-th nonzero component of the solution; all other entries of x are zero.

w is an INTENT(OUT) rank-1 array of package type and size at least n. It is used as workspace.

keep, control, info: see Section 2.4.4.

### 2.4.8 The finalisation subroutine

Once all other calls to HSL_MA87 routines are complete for a problem or after an error return that does not allow the computation to continue, a call of the following form should be made to deallocate allocatable components of keep and info, and to destroy OpenMP locks.

```
call MA87_finalise(keep,control)
```

keep is an INTENT(INOUT) scalar of type MA87_keep that must be passed unchanged. On exit, allocatable components will be deallocated.

control is an INTENT(IN) scalar of type MA87_control. Only the components that control printing are accessed (see Section 2.4.9).

### 2.4.9 The derived data type for holding control parameters

The derived data type MA87_control is used to hold controlling data. The components, which are automatically given default values in the definition of the type, are:

**Printing controls**

diagnostics_level is a scalar of type INTEGER that is used to control the level of diagnostic printing. The different levels are:

- < 0 No printing.
- = 0 Error and warning messages only.
- = 1 As 0, plus basic diagnostic printing.
- = 2 As 1, plus some additional diagnostic printing.
- = 3 As 2, plus all entries of user-supplied arrays.

The default is diagnostics_level=0.

unit_diagnostics is a scalar of type INTEGER that holds the unit number for diagnostic printing. Printing is suppressed if unit_diagnostics<0. The default is unit_diagnostics=6.

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

unit_warning is a scalar of type INTEGER that holds the unit number for warning messages. Printing of warning messages is suppressed if unit_warning<0. The default is unit_warning=6.

**Controls used by** MA87_analyse

nemin is a scalar of type INTEGER that controls node amalgamation. A child node is merged with its parent node in the assembly tree if they both involve fewer than nemin eliminations. The default is nemin=32. The default is used if nemin<1.

nb is a scalar of type INTEGER. The factor $L$ is held using a block structure (see Section 4.1) and nb controls the size of the blocks. The target number of rows in each block is nb. The default is nb=256. The default is used if nb<1.

**Controls used by** MA87_factor **and** MA87_factor_solve

pool_size is a scalar of type INTEGER that holds the initial size of the arrays that store the task pool (see Section 4). Whenever the size of these arrays is found to be too small, their size is doubled. The default is pool_size=25000. The default is used if pool_size<1.

diag_zero_minus and diag_zero_plus are scalars of package type that specify tolerances for the detection of matrix inertia. They may be set to non-default values to allow the factorization of semi-definite matrices such as those arising, for example, from the solution of least squares problems by the method of normal equations. After $i-1$ eliminations, the diagonal value $a_{ii}$ is classified as either:

| | | |
|---|---|---|
| positive | if diag_zero_plus $< a_{ii}$ | (proceed with standard factorization); |
| zero | if diag_zero_minus $< a_{ii} \leq$ diag_zero_plus | (replace column $L_i$ with $i$-th column of the identity matrix and set warning +2 (semi-definite)); or |
| negative | if $a_{ii} <$ min(diag_zero_minus, diag_zero_plus) | (return immediately with error -3 (indefinite)). |

The default values of diag_zero_minus=0.0 and diag_zero_plus=0.0 correspond to the same behaviour as the LAPACK function _potrf. Using non-default values may reduce performance. **The semi-definite factorization algorithm resulting from the use of non-default values is unstable. This option is offered for expert users only, and any results must be treated with caution.**

#### 2.4.10 The derived data type for holding information

The derived data type MA87_info is used to hold parameters that give information about the progress and needs of the algorithm. The components of MA87_info (in alphabetical order) are:

detlog is a scalar of type REAL. On exit from MA87_factor or MA87_factor_solve, it holds the logarithm of the absolute value of the determinant of $A$.

flag is a scalar of type INTEGER that gives the exit status of the algorithm (details in Section 2.5).

maxdepth is a scalar of type INTEGER. On exit from MA87_analyse, it holds the maximum depth of the assembly tree.

num_factor is a scalar of type INTEGER(long). On exit from MA87_analyse, it holds the number of entries that will be in the $L$ factor.

num_flops is a scalar of type INTEGER(long). On exit from MA87_analyse, it holds the number of floating-point operations that will be needed to perform the factorization, assuming the pivot sequence can be used without modification. On exit from MA87_factor and MA87_factor_solve, it holds the number of floating-point operations performed.

num_nodes is a scalar of type INTEGER. On exit from MA87_analyse, it holds the number of nodes in the assembly tree.

num_zero is a scalar of type INTEGER. On exit from MA87_factor and MA87_factor_solve, it holds the number of zero diagonal pivots encountered (those in the range (control%diag_zero_minus, control%diag_zero_plus]). These pivots were replaced with columns of the identity matrix, and n − info%num_zero is an estimate of the rank of $A$ if $A$ is positive semi-definite.

pool_size is a scalar of type INTEGER. On exit from MA87_factor and MA87_factor_solve, it holds the maximum number of tasks that are in the task pool during the factorization. Note that on repeated runs using the same matrix data, this may vary.

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

### 2.5 Warning and error messages

A successful return from a subroutine in the package is indicated by 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 Allocation error. The stat parameter is returned in info%stat.

−2 Returned by MA87_analyse if an error is found in the user-supplied elimination order (held in order).

−3 Returned by MA87_analyse if some variables are unused. Also returned by MA87_factor and MA87_factor_solve if the matrix is found not to be positive definite (if control%diag_zero_minus≥control%diag_zero_plus) or indefinite (otherwise). The user may reset the matrix values in val and recall MA87_factor or MA87_factor_solve.

−4 Returned by MA87_factor_solve and MA87_solve if there is an error in the size of array x (that is, lx<n or nrhs<1). The user may reset lx and/or nrhs and recall MA87_factor_solve or MA87_solve.

−5 Returned by MA87_factor and MA87_factor_solve if IEEE infinities found in the factorization. The user may reset the matrix values in val and recall MA87_factor or MA87_factor_solve.

−6 Returned by MA87_solve if job is out of range. The user may reset job and recall MA87_solve.

−7 Returned by MA87_sparse_fwd_solve if nbi is out of range. The user may reset nbi and recall MA87_sparse_fwd_solve.

A positive value for info%flag is used for warnings. Possible values are:

+1 Returned by MA87_factor and MA87_factor_solve if control%pool_size found to be too small. The size of the task pool used is returned in info%pool_size.

+2 Returned by MA87_factor and MA87_factor_solve if the matrix is semi-positive definite (that is pivots in the range (control%diag_zero_minus, control%diag_zero_plus] were encountered). The number of such pivots is returned in info%num_zero.

## 3 GENERAL INFORMATION

**Other routines called directly:** HSL packages HSL_MC34 and HSL_MC78, BLAS routines _gemm, _gemv, _herk, _syrk, _trsm, _trsv, and LAPACK routine _potrf.

**Input/output:** Output is provided under the control of control%diagnostics_level, which allows error, warning and diagnostics messages to be printed on units control%unit_error, control%unit_warning and control%unit_diagnostics, respectively.

**Restrictions:** nrhs≥1, lx≥n, job = 0,1,2, 1≤nbi≤n.

**Portability:** Fortran 95, plus allocatable components of derived types.

**Changes from Version** 1
The interface has changed. In particular, the entry MA87_input is no longer available, the derived type HSL_ZD11 is not used, the entry MA87_factor_solve has been added and separate entries are available for solving for a single right-hand side (to avoid requiring the solution x having to be of rank 2). The analyse phase now calls HSL_MC78. A complex Hermitian version has been added.

**Changes from Version** 2.3
Added support for factorization of semi-definite matrices. Additional components control%diag_zero_minus, control%diag_zero_plus and info%num_zero added for this purpose.

## 4 METHOD

HSL_MA87 divides the sparse factorization into tasks, each of which alters a single block (details in Section 4.2). These tasks are partially ordered; for example, the updating of a block from a block column of $L$ has to wait until all the rows that it needs from the block column have been calculated. As soon as all the data that a task needs are available, the task is placed in a pool of tasks for execution by any processor. The whole factorization is thus represented by a directed acyclic graph (DAG) with vertices representing tasks and edges representing dependencies.

### 4.1 Data structures

A node of the assembly tree represents a set of contiguous columns of $L$ with the same (or nearly the same) sparsity structure below a dense (or nearly dense) triangular submatrix. Each nodal matrix is held as a dense trapezoidal matrix. We store this matrix using a row hybrid blocked structure and use "full" storage for the blocks on the diagonal (which allows us to exploit efficient BLAS and LAPACK routines). If the number of columns in the nodal matrix is large, we use the block size *nb* specified through the control parameter `control%nb` and the blocks will be of size near $nb \times nb$. For example, if the block size was 3, a node with 5 columns and 8 rows would be stored as

| | | | | |
|---|---|---|---|---|
| 1 | | | | |
| 4 | 5 | | | |
| 7 | 8 | 9 | | |
| 10 | 11 | 12 | 25 | |
| 13 | 14 | 15 | 27 | 28 |
| 16 | 17 | 18 | 29 | 30 |
| 19 | 20 | 21 | 31 | 32 |
| 22 | 23 | 24 | 33 | 34 |

### 4.2 Tasks

We divide the sparse Cholesky factorization into the following tasks:

**factorize**(`diag`) Computes the traditional dense Cholesky factor $L_{triang}$ of the triangular part of a block `diag` that is on the diagonal using the LAPACK subroutine `_potrf`. If the block is trapezoidal, this is followed by a triangular solve of its rectangular part

$$L_{rect} \Leftarrow L_{rect} L_{triang}^{-T}$$

using the BLAS subroutine `_trsm`.

**solve**(`dest`, `diag`) Performs a triangular solve of the off-diagonal block `dest` by the Cholesky factor $L_{triang}$ of the block `diag` on its diagonal. i.e.

$$L_{dest} \Leftarrow L_{dest} L_{triang}^{-T}$$

using the BLAS subroutine `_trsm`.

**update_internal**(`dest`,`rsrc`,`csrc`) Forms the outer product of blocks `rsrc` and `csrc` and applies it to the block `dest` within the same node. This task is accomplished using the BLAS routines `_syrk` and `_gemm`.

**update_between**(`dest`,`snode`,`scol`) Perform the outer product update of block `dest` using the relevant portion of block column `scol` in node `snode`. This task exploits the fact that row boundaries in the nodal data structure are artificial and can be ignored to perform a direct outer product into a buffer. This buffer is then expanded out into the destination node using a sparse mapping which is calculated on the fly.

The dependency graph can be computed through examination of the assembly tree observing that we cannot perform the factorization of a diagonal block until all required updates to it have been performed, and we cannot perform a solve until all required updates are complete and the relevant diagonal block has been factorised. Updates can be performed as soon as the source data has had the relevant factor or solve operation performed upon it.

Further details are given in [1].

### 4.3 The analyse phase

The analyse phase uses only the sparsity pattern of $A$. It requires the user to input the lower triangular part of the matrix in compressed sparse column format; no checks are made on the matrix data. The user must also input an elimination order (which may be computed using, for example, HSL_MC68). For the given elimination order, the analyse phase computes the assembly tree using HSL_MC78. A child node is merged with its parent node if they both involve fewer than control%nemin eliminations. The block structure of $L$ is computed and the task DAG is established and to track which tasks are ready, a dependency count for each block is computed. If the block is on the diagonal, the count is the number of updates that will be applied to it. If the block is not on the diagonal, the count is one more than the number of updates that will be applied to it.

### 4.4 The factorize phase

The factorize phase uses the data structures prepared in the analyse phase and takes a copy of the dependency counts computed by the analyse phase. It starts by initialising the numeric representation of the factors to zero and then copying the entries of $A$ into the correct locations in $L$. The leaf tasks (one for each leaf node in the assembly tree) are then added into the task pool. Processors take tasks from the pool one at a time and execute them, placing any new tasks as they become ready into the pool.

During factorize, the block count (and corresponding block column count) is decremented by one after the completion of each update for it. When the count for a block on the diagonal reaches zero, a factorize_block task for it is added to the task pool. Once this task completes, the count of all the blocks in its block column is decremented. When the count for an off-diagonal block reaches zero, its solve_block task is added to the task pool.

When a factorize_block or solve_block task completes, its count is decremented to flag this event with a negative value. A column lock is set and each update task that depends on the completion of this task and does not depend on a task that has not yet completed is added to the task pool. Once this has been done, the lock is released.

An optimisation of this approach used for machines with separate caches is to give each cache its own task stack, which overflows into the task pool. If the local stack and task pool are empty, workstealing is used to obtain tasks — if another cache has spare tasks in its stack, half of these are moved to the task pool.

If the user wishes to solve $AX = B$ at the same time as factorizing the matrix, the call to MA87_factor should be replaced by a call to MA87_factor_solve. The user must pass right-hand vectors to MA87_factor_solve using the argument x1 (single right-hand side) or x (multiple right-hand sides). The forward substitutions are performed as the factor entries are generated. Once the factorization is complete, MA87_factor_solve performs the back substitutions by calling MA87_solve with job = 2. Using MA87_factor_solve is more efficient than calling MA87_factor followed by MA87_solve.

#### 4.4.1 Semi-definite factorization

If control%diag_zero_plus $= 0.0$ and control%diag_zero_minus $\geq$ control%diag_zero_plus, the LAPACK routine _potrf is used for factorization of the diagonal block. Otherwise the following pivoting logic is used.

Let $d_{jj} = \text{real}(a_{jj}^{(j)})$.
**if** $d_{jj} \leq$ control%diag_zero_plus
  **if** $d_{jj} \leq$ control%diag_zero_minus
    Immediate return error -3 (indefinite).
  **else**
    Set $l_{jj} = 1.0$ and $l_{ij} = 0.0, i = j+1, \ldots n$.
    Set warning -2 (semi-definite).

Observe update for this column is now a zero matrix.

**else**

*! Standard Cholesky step*

Set $l_{jj} = \sqrt{d_{jj}}$ and $l_{ij} = a_{ij}^{(j)}/l_{jj}, i = j+1,\ldots n$.

Update trailing matrix.

The above pivoting algorithm is unstable, and any results should be treated with caution.

### 4.5 The solve phase

The solve phase uses the data structures prepared by the factorize phase to perform a full or partial solution of the equation

$$AX = B$$

The matrix factor $L$ must be accessed once for the forward substitution and once for the back substitution. This is independent of the number of right-hand sides so that solving for several right-hand sides at once is significantly faster than repeatedly solving for a single right-hand side.

## References:

[1] J.D. Hogg, J.K Reid and J.A. Scott. (2009). Design of a multicore sparse Cholesky factorization using DAGs. Technical Report TR-RAL-2009-027.

Available from `http://www.numerical.rl.ac.uk/reports/reports.shtml`

## 5 EXAMPLE OF USE

### 5.1 Example 1

We wish to solve the system

$$\begin{pmatrix} 2. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 4. & \\ & 1. & & & 2. \end{pmatrix} X = \begin{pmatrix} 4. \\ 12. \\ 10. \\ 8. \\ 4. \end{pmatrix}.$$

The following code may be used.

```
program hsl_ma87ds
   use hsl_ma87_double
   use hsl_mc69_double
   implicit none

   integer, parameter :: wp = kind(0d0)

   type (ma87_keep)    :: keep
   type (ma87_control) :: control
   type (ma87_info)    :: info

   integer :: i, n
```

```
! integer :: flag, more ! uncomment for checking
integer, dimension(:), allocatable  :: ptr, row, order
real(wp), dimension(:), allocatable :: val, x

! Read the lower triangle of the matrix
read(*,*) n
allocate(ptr(n+1));        read(*,*) ptr(:)
allocate(row(ptr(n+1)-1)); read(*,*) row(:)
allocate(val(ptr(n+1)-1)); read(*,*) val(:)
! Read the right hand side
allocate(x(n)); read(*,*) x(:)

! Use the input order
allocate(order(n))
do i = 1,n
   order(i) = i
end do

! Uncomment the following lines to enable checking (performance overhead)
!call mc69_verify(6, HSL_MATRIX_REAL_SYM_PSDEF, n, n, ptr, row, flag, more)
!if(flag.ne.0) then
!   write(*,*) "Matrix not in HSL standard format. flag, more = ", flag, more
!   stop
!endif

! Analyse
call ma87_analyse(n, ptr, row, order, keep, control, info)
if(info%flag.lt.0) then
   write(*,*) "Failure during analyse with info%flag = ", info%flag
   stop
endif

! Factor
call ma87_factor(n, ptr, row, val, order, keep, control, info)
if(info%flag.lt.0) then
   write(*,*) "Failure during factor with info%flag = ", info%flag
   stop
endif

! Solve
call ma87_solve(x, order, keep, control, info)
if(info%flag.lt.0) then
   write(*,*) "Failure during solve with info%flag = ", info%flag
   stop
endif

write(*,'(a)') ' Computed solution:'
write(*,'(8f10.3)') x(1:n)
```

```
   ! Finalize
   call ma87_finalise(keep, control)

end program hsl_ma87ds
```

The input file is:

```
 5
 1  3  6  8  9 10
 1  2  2  3  5  3  4  4  5
 2. 1. 4. 1. 1. 3. 2. 4. 2.
 4. 12. 10. 8. 4.
```

This produces the following output:

```
 Computed solution:
    1.000     2.000     2.000     1.000     1.000
```

### 5.2 Example 2

We wish to solve the two pattern-identical systems

$$\begin{pmatrix} 2. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 4. & \\ & 1. & & & 2. \end{pmatrix} X = \begin{pmatrix} 4. \\ 12. \\ 10. \\ 8. \\ 4. \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} 5. & 2. & & & \\ 2. & 9. & 3. & & -2. \\ & 3. & 6. & 1. & \\ & & 1. & 5. & \\ & -2. & & & 6. \end{pmatrix} X = \begin{pmatrix} 9. & 19. \\ 24. & 21. \\ 19. & 14. \\ 7. & 11. \\ 2. & 14. \end{pmatrix},$$

where input is in coordinate form. The following code may be used.

```
program hsl_ma87ds1
   use hsl_ma87_double
   use hsl_mc68_double
   use hsl_mc69_double
   implicit none

   integer, parameter :: wp = kind(0d0)

   type(mc68_control) :: control68
   type(mc68_info) :: info68

   type(ma87_keep)    :: keep
   type(ma87_control) :: control
   type(ma87_info)    :: info

   integer :: n, ne, nrhs, lmap, flag
   integer, dimension(:), allocatable  :: crow, ccol, ptr, row, order, map
   real(wp), dimension(:), allocatable :: cval, val, x
   real(wp), dimension(:,:), allocatable :: x2

   ! Read the first matrix in coordinate format
```

```
read(*,*) n, ne
allocate(ccol(ne)); read(*,*) ccol(:)
allocate(crow(ne)); read(*,*) crow(:)
allocate(cval(ne)); read(*,*) cval(:)
! Read the first right hand side
allocate(x(n)); read(*,*) x(:)

! Convert to HSL standard format
allocate(ptr(n+1))
call mc69_coord_convert(HSL_MATRIX_REAL_SYM_PSDEF, n, n, ne, crow, ccol, &
   ptr, row, flag, val_in=cval, val_out=val, lmap=lmap, map=map)
call stop_on_bad_flag("mc69_coord_convert", flag)

! Call mc68 to find a fill reducing ordering (1=AMD)
allocate(order(n))
call mc68_order(1, n, ptr, row, order, control68, info68)
call stop_on_bad_flag("mc68_order", info68%flag)

! Analyse
call ma87_analyse(n, ptr, row, order, keep, control, info)
call stop_on_bad_flag("analyse", info%flag)

! Factor
call ma87_factor(n, ptr, row, val, order, keep, control, info)
call stop_on_bad_flag("factor", info%flag)

! Solve
call ma87_solve(x, order, keep, control, info)
call stop_on_bad_flag("solve", info%flag)

write(*,'(a)') ' Computed solution:'
write(*,'(8f10.3)') x(1:n)

! Read the values of the second matrix and the new right hand sides
read(*,*) cval(:)
read(*,*) nrhs
allocate(x2(n,nrhs)); read(*,*) x2(:,:)

! Convert the values to HSL standard form
call mc69_set_values(HSL_MATRIX_REAL_SYM_PSDEF, lmap, map, cval, &
   ptr(n+1)-1, val)

! Perform second factorization and solve
call ma87_factor_solve(n, ptr, row, val, order, keep, control, info, &
   nrhs, n, x2)
call stop_on_bad_flag("factor_solve", info%flag)

write(*,'(a)') ' Computed solutions:'
write(*,'(8f10.3)') x2(1:n,1)
```

```
   write(*,'(8f10.3)') x2(1:n,2)

   ! Finalize
   call ma87_finalise(keep, control)

contains
   subroutine stop_on_bad_flag(context, flag)
      character(len=*), intent(in) :: context
      integer, intent(in) :: flag

      if(flag.eq.0) return
      write(*,*) "Failure during ", context, " with flag = ", flag
      stop
   end subroutine stop_on_bad_flag
end program hsl_ma87ds1
```

The input file is:

```
  5  9
  1  1  2  2  2  3  3  4  5
  1  2  2  3  5  3  4  4  5
  2. 1. 4. 1. 1. 3. 2. 4. 2.
  4. 12. 10. 8. 4.
  5. 2. 9. 3. -2. 6. 1. 5. 6.
  2
  9. 24. 19. 7. 2.
  19. 21. 14. 11. 14.
```

This produces the following output:

```
 Computed solution:
     1.000     2.000     2.000     1.000     1.000
 Computed solutions:
     1.000     2.000     2.000     1.000     1.000
     3.000     2.000     1.000     2.000     3.000
```