Science and Technology Facilities Council

HSL_MA87

C INTERFACE 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 2003 subset (F95 + TR15581 + C interoperability). **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 C interface to Fortran code

This package is written in Fortran and a wrapper is provided for C programmers. This wrapper may only implement a subset of the full functionality described in the Fortran user documentation.

The wrapper will automatically convert between 0-based (C) and 1-based (Fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing. The conversion may be disabled by setting the control parameter `control.f_arrays=1` and supplying all data using 1-based indexing. With 0-based indexing, the matrix is treated as have rows and column $0, 1, \ldots n-1$. In this document, we assume 0-based indexing.

The wrapper uses the Fortran 2003 interoperability features. **Matching C and Fortran compilers must be used**, for example gcc and gfortran, or icc and ifort. If the Fortran compiler is not used to link the user's program, then additional Fortran compiler libraries may need to be linked explicitly.

2.2 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).

2.3 Calling sequences

Access to the package requires inclusion of the header file

Single precision version

```
#include "hsl_ma87s.h"
```

Double precision version

```
#include "hsl_ma87d.h"
```

Complex version

```
#include "hsl_ma87c.h"
```

Double complex version

```
#include "hsl_ma87z.h"
```

It is not possible to use more than one version at the same time.
The following subroutines are available to the user:

(a) `ma87_default_control` sets default values for members of the `ma87_control` data type needed by other subroutines.

(b) `ma87_analyse` analyses the sparsity pattern of the matrix and, using the user-supplied elimination order, prepares the data structures for the factorization.

(c) `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`.

(d) `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`.

(e) `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.

(f) `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`.

(g) `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 frees memory allocated by the package.

2.4 The derived data types

For each problem, the user must employ the structures defined in the header file to declare scalars of the types `struct ma87_info`, `struct ma87_control`, and a `void *` pointer `keep`. The following pseudocode illustrates this.

```
#include "hsl_ma87d.h"
...
struct ma87_control control;
struct ma87_info info;
void * keep;
...
```

The members of `ma87_control` and `ma87_info` are explained in Sections 2.5.9 and 2.5.10. The `void *` pointer is used to pass data between subroutines of the package.

2.5 Argument lists and calling sequences

2.5.1 Package types

The complex versions require C99 support for the `double complex` and `float complex` types. The real versions do not require C99 support.

We use the following type definitions in the different versions of the package:

Single precision version

```
typedef float pkgtype
```

Double precision version

```
typedef double pkgtype
```

Complex version

```
typedef float complex pkgtype
```

Double complex version

```
typedef double complex pkgtype
```

Elsewhere, for *single* and *single complex* versions replace `double` with `float`.

2.5.2 Input of the matrix A

The user must input the **lower** triangular part of the matrix A using the arguments `ptr` and `row` as described in Section 2.5.4.

We recommend that standard HSL format is used to ensure compatibility with other HSL routines. 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.

2.5.3 The default setting subroutine

Default values for members of the `ma87_control` structure may be set by a call to `ma87_default_control`.

```
void ma87_default_control(struct ma87_control *control)
```

`control` has its members set to their default values, as described in Section 2.5.9.

2.5.4 To analyse the sparsity pattern and prepare for the factorization

```
void ma87_analyse(int n, const int ptr[], const int row[], int order[],
      void **keep, const struct ma87_control *control, struct ma87_info *info)
```

`n` must hold the order of A.

`ptr` is a rank-1 array of 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]` must be set to the total number of entries in the `lower` triangular part of A.

row is a rank-1 array of size ptr[n]. 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 0 preceding those for column 1, and so on.

order is a rank-1 array of size 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 will be set to point at an area of memory allocated using a Fortran allocate statement that will be used to hold data about the problem being solved. It must be passed unchanged to the other subroutines. To avoid a memory leak the subroutine ma87_finalise must be used to clean up and deallocate this memory after the factorization is no longer required.

control is used to control the actions of the package, see Section 2.5.9.

info is used to return information about the execution of the package, as explained in Section 2.5.10.

2.5.5 To factorize the matrix and optionally solve $AX = B$

To factorize the matrix, a call to ma87_factor may be made after the call to ma87_analyse:

```
void ma87_factor(int n, const int ptr[], const int row[], const pkgtype val[],
      const int order[], void **keep, const struct ma87_control *control,
      struct ma87_info *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 to ma87_factor_solve.

```
void ma87_factor_solve(int n, const int ptr[], const int row[],
      const pkgtype val[], const int order[], void **keep,
      const struct ma87_control *control, struct ma87_info *info,
      const int nrhs, const int ldx, pkgtype x[])
```

n, ptr, row, order: must be unchanged since the call to ma87_analyse.

val is a rank-1 array of size ptr[n]. The 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 2.5.4.

nrhs holds the number of right-hand sides. **Restriction:** nrhs$\geq$1.

ldx holds the length of the leading dimension of x (only the first n locations are accessed). Note that this is the leading dimension in memory, not in C notation. **Restriction:** ldx$\geq$n.

x is a rank-2 array with size x[nrhs][ldx]. On entry, x[j][i] must hold the ith component of the jth right-hand side; on exit, it holds the corresponding solution.

2.5.6 To solve linear systems using the computed factors

After the call to ma87_factor (or ma87_factor_solve), one or more calls to ma87_solve may be made to solve $AX = B$. Partial solutions may be performed by appropriately setting the optional parameter job.

```
void ma87_solve(int job, int nrhs, int ldx, pkgtype x[], const int order[],
      void **keep, const struct ma87_control *control,
      struct ma87_info *info)
```

job If job = 0, $AX = B$ 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.

nrhs, ldx, x: see Section 2.5.5.

order, keep, control, info: see Section 2.5.4.

2.5.7 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

```
void ma87_sparse_fwd_solve(int nbi, int bindex[], const pkgtype b[],
   const int order[], const int invp[], int *nxi, int index[], pkgtype x[],
   pkgtype w[], void **keep, const struct ma87_control *control,
   struct ma87_info *info);
```

nbi must hold the number of nonzero entries in the right-hand side. **Restriction:** 1≤nbi≤n.

bindex is a rank-1 array of size nbi. It 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 a rank-1 array 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 a rank-1 array of 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 a rank-1 array of 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 holds, on exit, the number of nonzero entries in the solution.

index is a rank-1 array. On exit, the first nxi entries hold the indices of the nonzero entries in the solution.

x is a rank-1 array of 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 a rank-1 array of package type and size at least n. It is used as workspace.

keep, control, info: see Section 2.5.4.

2.5.8 The finalisation subroutine

Once all other calls to HSL_MA87 routines are complete for a problem or after an error return, a call to ma87_finalise should be made to free memory and resources associated with keep.

```
void ma87_finalise(void **keep, const struct ma87_control *control)
```

keep must be passed unchanged. On exit, all memory associated with keep will have been freed and keep will be set to null.

control will be used to control printing. Only the members that control printing are accessed (see Section 2.5.9).

2.5.9 The data type for holding control parameters

The data type struct ma87_control is used to hold controlling data. The members, and default values that are set by a call to ma87_default_control, are:

C only controls

int f_arrays indicates whether to use C or Fortran array indexing. If f_arrays!=0 (i.e. evaluates to true) then 1-based indexing of the arrays ptr, row and order is assumed. Otherwise, if f_arrays=0 (i.e. evaluates to false), then these arrays are copied and converted to 1-based indexing in the wrapper function. All descriptions in this documentation assume f_arrays=0. The default is f_arrays=0 (false).

Printing controls

int diagnostics_level 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.

int unit_diagnostics holds the Fortran unit number for diagnostic printing. Printing of diagnostics is suppressed if unit_diagnostics<0. The default is unit_diagnostics=6 (stdout).

int unit_error holds the Fortran unit number for error messages. Printing of error messages is suppressed if unit_error<0. The default is unit_error=6 (stdout).

int unit_warning holds the Fortran unit number for warning messages. Printing of warning messages is suppressed if unit_warning<0. The default is unit_warning=6 (stdout).

Controls used by ma87_analyse

int nemin 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.

int nb controls the target number of rows used in the block data structure used to hold the factor L (see Section 4.1). 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

int pool_size 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.

double diag_zero_minus and double diag_zero_plus 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.**

2.5.10 The data type for holding information

The data type struct ma87_info is used to hold parameters that give information about the progress and needs of the algorithm. The members of ma87_info (in alphabetical order) are:

double detlog holds, on exit from ma87_factor or ma87_factor_solve, the logarithm of the absolute value of the determinant of A.

int flag gives the exit status of the algorithm (details in Section 2.6).

int maxdepth holds, on exit from ma87_analyse, the maximum depth of the assembly tree.

long num_factor holds, on exit from ma87_analyse, the number of entries that will be in the L factor.

long num_flops holds, on exit from ma87_analyse, 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.

int num_nodes holds, on exit from ma87_analyse, the number of nodes in the assembly tree.

int num_zero holds, on exit from ma87_factor and ma87_factor_solve, 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.

int pool_size holds, on exit from ma87_factor and ma87_factor_solve, 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.

int stat holds the Fortran stat parameter.

2.6 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 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

Workspace: HSL_MA87 handles its own memory allocations.

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.

Portability: Fortran 2003 subset (F95 + TR15581 + C interoperability).

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. Size of control and info structures is changed, and additional space reserved for any future additions.

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.

```
/* hsl_ma87ds.c */
#include <stdio.h>
#include <stdlib.h>
#include "hsl_mc69d.h"
#include "hsl_ma87d.h"
```

```
/* Simple code to illustrate use of hsl_ma87 */
int main(void) {
   void *keep;
   struct ma87_control control;
   struct ma87_info info;

   int i, n, flag, more;
   int *ptr, *row, *order;
   double *val, *x;

   /* Read the lower triangle of the matrix */
   scanf("%d\n", &n);
   ptr = (int *) malloc(sizeof(int)*(n+1));
   for(i=0; i<n+1; i++) scanf("%d", &ptr[i]);
   row = (int *) malloc(sizeof(int)*(ptr[n]));
   for(i=0; i<ptr[n]; i++) scanf("%d", &row[i]);
   val = (double *) malloc(sizeof(double)*(ptr[n]));
   for(i=0; i<ptr[n]; i++) scanf("%lf", &val[i]);
   /* Read the right hand side */
   x = (double *) malloc(sizeof(double)*n);
   for(i=0; i<n; i++) scanf("%lf", &x[i]);

   /* Use the input order */
   order = (int *) malloc(sizeof(int)*n);
   for(i=0; i<n; i++) order[i] = i;

   /* Uncomment the following lines to enable checking (performance overhead) */
   /*
   flag = mc69_verify(6, HSL_MATRIX_REAL_SYM_PSDEF, 0, n, n, ptr, row, NULL,
      &more);
   if(flag != 0) {
      write(*,*) "Matrix not in HSL standard format. flag, more = ", flag, more
      printf("Matrix not in HSL standard format. flag, more = %i, %i\n",
         flag, more);
      return 1;
   }
   */

   /* Set up control */
   ma87_default_control(&control);

   /* Analyse */
   ma87_analyse(n, ptr, row, order, &keep, &control, &info);
   if(info.flag < 0) {
      printf("Failure during analyse with info.flag = %i\n", info.flag);
      return 1;
   }

   /* Factor */
```

```
   ma87_factor(n, ptr, row, val, order, &keep, &control, &info);
   if(info.flag < 0) {
      printf("Failure during factor with info.flag = %i\n", info.flag);
      return 1;
   }

   /* Solve */
   ma87_solve(0, 1, n, x, order, &keep, &control, &info);
   if(info.flag < 0) {
      printf("Failure during solve with info.flag = %i\n", info.flag);
      return 1;
   }

   printf(" Computed solution:\n");
   for(i=0; i<n; i++) printf("%10.3lf ", x[i]);
   printf("\n");

   /* Clean up */
   ma87_finalise(&keep, &control);
   free(ptr); free(row); free(val);
   free(order);
   free(x);

   return 0;
}
```

The input file is:

```
5
0  2  5  7  8  9
0  1  1  2  4  2  3  3  4
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 similar indefinite 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.

```
/* hsl_ma87ds1.c */
#include <stdio.h>
#include <stdlib.h>
#include "hsl_mc68i.h"
#include "hsl_mc69d.h"
#include "hsl_ma87d.h"

void stop_on_bad_flag(const char *context, const int flag);

/* Code to illustrate use of more complex hsl_ma87 features */
int main(void) {
   typedef double pkgtype;

   struct mc68_control control68;
   struct mc68_info info68;

   void *keep;
   struct ma87_control control;
   struct ma87_info info;

   int i, j, n, ne, nrhs, lmap, lrow, noor, ndup, flag;
   int *crow, *ccol, *ptr, *row, *order, *map;
   pkgtype *cval, *val, *x, *x2;

   /* Read the first matrix in coordinate format */
   scanf("%d %d\n", &n, &ne);
   ccol = (int *) malloc(sizeof(int)*ne);
   for(i=0; i<ne; i++) scanf("%d", &ccol[i]);
   crow = (int *) malloc(sizeof(int)*ne);
   for(i=0; i<ne; i++) scanf("%d", &crow[i]);
   cval = (pkgtype *) malloc(sizeof(pkgtype)*ne);
   for(i=0; i<ne; i++) scanf("%lf", &cval[i]);
   /* Read the first right hand side */
   x = (pkgtype *) malloc(sizeof(pkgtype)*n);
   for(i=0; i<n; i++) scanf("%lf", &x[i]);

   /* Convert to HSL standard format */
   ptr = (int *) malloc(sizeof(int)*(n+1));
   lrow = ne; /* maximum size of output matrix cannot exceed size of input */
   lmap = 2*ne; /* large enough for worst case */
   row = (int *) malloc(sizeof(int)*lrow);
   val = (pkgtype *) malloc(sizeof(pkgtype)*lrow);
   map = (int *) malloc(sizeof(int)*lmap);
   flag = mc69_coord_convert(6, HSL_MATRIX_REAL_SYM_PSDEF, 0, n, n, ne,
      crow, ccol, cval, ptr, lrow, row, val, &noor, &ndup, &lmap, map);
   stop_on_bad_flag("mc69_coord_convert", flag);

   /* Call mc68 to find a fill reducing ordering (1=AMD) */
   order = (int *) malloc(sizeof(int)*n);
```

```
mc68_default_control(&control68);
mc68_order(1, n, ptr, row, order, &control68, &info68);
stop_on_bad_flag("mc68_order", info68.flag);

/* Initialize control parameters */
ma87_default_control(&control);

/* Analyse */
ma87_analyse(n, ptr, row, order, &keep, &control, &info);
stop_on_bad_flag("analyse", info.flag);

/* Factor */
ma87_factor(n, ptr, row, val, order, &keep, &control, &info);
stop_on_bad_flag("factor", info.flag);

/* Solve */
ma87_solve(0, 1, n, x, order, &keep, &control, &info);
stop_on_bad_flag("solve", info.flag);

printf(" Computed solution:\n");
for(i=0; i<n; i++) printf(" %lf", x[i]);

/* Read the values of the second matrix and the new right hand sides */
for(i=0; i<ne; i++) scanf("%lf", &cval[i]);
scanf("%d\n", &nrhs);
x2 = (pkgtype *) malloc(sizeof(pkgtype)*n*nrhs);
for(i=0; i<nrhs; i++) {
   for(j=0; j<n; j++) {
      scanf("%lf", &x2[i*n+j]);
   }
}
printf("\n");

/* Convert the values to HSL standard form */
mc69_set_values(HSL_MATRIX_REAL_SYM_PSDEF, lmap, map, cval,
   ptr[n], val);

/* Perform second factorization and solve */
ma87_factor_solve(n, ptr, row, val, order, &keep, &control, &info,
   nrhs, n, x2);
stop_on_bad_flag("factor_solve", info.flag);

printf(" Computed solutions:\n");
for(i=0; i<nrhs; i++) {
   for(j=0; j<n; j++) printf(" %lf", x2[i*n+j]);
   printf("\n");
}

/* Clean up */
```

```
   ma87_finalise(&keep, &control);
   free(crow); free(ccol); free(cval);
   free(ptr); free(row); free(val);
   free(map); free(order);
   free(x); free(x2);

   return 0;
}

void stop_on_bad_flag(const char *context, const int flag) {
   if(0==flag) return;
   printf("Failure during %s  with flag = %d\n", context, flag);
   exit(1);
}
```

The following input, supplied on stdin,

```
  5  9
  0  0  1  1  1  2  2  3  4
  0  1  1  2  4  2  3  3  4
  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.
```

produces the following output.

```
 Computed solution:
 1.000000 2.000000 2.000000 1.000000 1.000000
 Computed solutions:
 1.000000 2.000000 2.000000 1.000000 1.000000
 3.000000 2.000000 1.000000 2.000000 3.000000
```