HSL_MA86

1 SUMMARY

HSL_MA86 uses a direct method to solve **large sparse symmetric indefinite linear systems** of equations $AX = B$. This package uses **OpenMP** and is designed for **multicore architectures.** It computes the sparse factorization

$$A = PLD(PL)^{\star}$$

where $L^{\star} = L^T$ (real symmetric or complex symmetric) or $L^{\star} = L^H$ (complex Hermitian, where L^H denotes the conjugate transpose of L), P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of size 1×1 and 2×2.

Options are provided for the complementary forward and backward substitutions.

The efficiency of HSL_MA86 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 known to be positive definite (so that pivoting for numerical stability is not required), we recommend HSL_MA87.

ATTRIBUTES — Version: 1.8.0 (27 January 2025). **Interfaces:** Fortran, C, MATLAB. **Types:** Real (single, double), Complex (single, double). **Calls:** HSL_MC34, MC64, MC77 and HSL_MC78, BLAS subroutines _copy, _axpy, _swap, _gemm, _gemv, _trsm, _trsv. **Original date:** January 2011. **Origin:** J.D. Hogg 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_MA86 was supported by EPSRC grant EP/E053351/1.

2 HOW TO USE THE PACKAGE

2.1 OpenMP

OpenMP is used by the HSL_MA86 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_MA86_single
```

Double precision version

```
USE HSL_MA86_double
```

Complex version

```
USE HSL_MA86_complex
```

Double complex version

```
USE HSL_MA86_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) `MA86_analyse` analyses the sparsity pattern of the matrix and, using the user-supplied elimination order, prepares the data structures for the factorization.

(b) `MA86_factor` uses the data structures set up by `MA86_analyse` to compute a sparse factorization. More than one call to `MA86_factor` may follow a call to `MA86_analyse`.

(c) `MA86_factor_solve` may be called in place of `MA86_factor` to factorize A and, at the same time, solve the system $AX = B$. Multiple calls to `MA86_factor_solve` may follow a call to `MA86_analyse`.

(d) `MA86_solve` uses the computed factors generated by `MA86_factor` or `MA86_factor_solve` to solve systems $AX = B$ for one or more right-hand sides B. Multiple calls to `MA86_solve` may follow a call to `MA86_factor` or `MA86_factor_solve`. An option is available to perform a partial solution.

(e) `MA86_finalise` should be called after all other calls are complete for a problem (including after an error return). 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 `MA86_keep`, `MA86_info`, and `MA86_control`. The following pseudocode illustrates this.

```
use HSL_MA86_double
...
type (MA86_control) :: control
type (MA86_info) :: info
type (MA86_keep) :: keep
```

The components of `MA86_control` and `MA86_info` are explained in Sections 2.4.8 and 2.4.9. The components of `MA86_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. There is no requirement that zero entries on the diagonal are explicitly included. **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_MA86` to fail in an unpredictable way. Before using `HSL_MA86`, 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 `MA86_factor` or `MA86_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 MA86_analyse(n,ptr,row,order,keep,control,info)
```

`n` is a scalar of type `INTEGER` that must hold the order of A.

`ptr` is an `INTENT(IN)` rank-one 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-one 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-one 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 `MA86_factor` (or `MA86_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 `MA86_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 `MA86_control` (see Section 2.4.8).

`info` is an `INTENT(OUT)` scalar of type `MA86_info`. Its components provide information about the execution of the subroutine, as explained in Section 2.4.9.

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 `MA86_analyse`:

The real case:

```
call MA86_factor(n,ptr,row,val,order,keep,control,info[,scale])
```

The complex case:

```
call MA86_factor(matrix_type,n,ptr,row,val,order,keep,control,info[,scale])
```

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:

The real case:

```
call MA86_factor_solve(n,ptr,row,val,order,keep,control,info,x1[,scale])
```

The complex case:

```
call MA86_factor_solve(matrix_type,n,ptr,row,val,order,keep,control,info,x1[,scale])
```

or, for more than one right-hand side,

The real case:

```
call MA86_factor_solve(n,ptr,row,val,order,keep,control,info,nrhs,lx,x[,scale])
```

The complex case:

```
call MA86_factor_solve(matrix_type,n,ptr,row,val,order,keep,control,info,nrhs,lx,x[,scale])
```

`matrix_type` is a scalar of type `INTEGER` that indicates the type of the matrix A in the complex case. It must be set to `-4` if A is Hermitian and to `-5` if A is complex symmetric. **Restriction:** `matrix_type = -4` or `-5`.

`n, ptr, row, order`: must be unchanged since the call to `MA86_analyse`.

`val` is an `INTENT(IN)` rank-one 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-one array of package type and of size at least `n`. On entry, `x1(i)` must hold the `i`th 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`≥`1`.

`lx` is an `INTENT(IN)` scalar of type `INTEGER` that holds the first extent of `x`. **Restriction:** `lx`≥`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 `i`th component of the `j`th right-hand side; on exit, it holds the corresponding solution.

`scale` is an optional `INTENT(INOUT)` rank-1 array of type `REAL` and extent `n`. If present and `control%scaling`=`0`, it specifies user-supplied scaling factors that are applied symmetricially to the matrix such that the matrix factored is SAS where $S = \text{diag}(\texttt{scale})$. If present and `control%scaling`≠`0`, the scaling calculated by `HSL_MA86` will be returned in `scale`.

2.4.6 To solve linear systems using the computed factors

After the call to `MA86_factor` (or `MA86_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 MA86_solve(x1,order,keep,control,info[,job])
```

or, for more than one right-hand side,

```
call MA86_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 $DX = B$

3 for solving $(PL)^\star X = B$

4 for solving $D(PL)^\star X = B$.

Restriction: job = 0,1,2,3,4.

Deprecated arguments:

scale is an optional INTENT(IN) rank-1 array of type REAL and extent n. If present it is ignored. This argument may be removed in future versions of HSL_MA86.

2.4.7 The finalisation subroutine

Once all other calls to HSL_MA86 routines are complete for a problem or after an error return, a call of the following form should be made to deallocate allocatable components of keep and info, and to destroy OpenMP locks.

```
call MA86_finalise(keep,control)
```

keep is an INTENT(INOUT) scalar of type MA86_keep that must be passed unchanged. On exit, allocatable components will be deallocated.

control is an INTENT(IN) scalar of type MA86_control. Only the components that control printing are accessed (see Section 2.4.8).

2.4.8 The derived data type for holding control parameters

The derived data type MA86_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 MA86_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 MA86_factor **and** MA86_factor_solve

action is a scalar of type default LOGICAL. If A is found to be singular (that is, to have rank less than n), the computation continues after issuing a warning if action has the value .true. or terminates (see error -11) if it has the value .false. The default is action=.true.

nbi is a scalar of type INTEGER that holds the inner block size used in the factorize_column tasks (see Section 4). Using a value of nbi that is smaller than nb increases the amount of computation performed using Level 3 BLAS. The default is nbi=16. The default is used if nbi<1.

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.

small is a scalar of type REAL. Any pivot whose modulus is less than small is treated as zero. The default is small= 10^{-20}.

scaling is a scalar of type INTEGER that is used to control scaling. The available options are:

- $\leq$ 0 No scaling (scale optional argument not present), or user-supplied scaling (scale optional argument present).
- $=$ 1 Generate a scaling using a weighted bipartite matching using the package MC64.
- $\geq$ 2 Generate a scaling by applying the iterative method of the package MC77 for one iteration in the infinity norm and three iterations in the one norm.

The default is scaling=0.

static is a scalar of type REAL that is used to control static pivoting. If static>0.0 and if, at any stage of the computation, fewer than the expected number of pivots can be found with relative pivot tolerance greater than umin, diagonal entries are accepted as pivots. If a candidate diagonal entry has absolute value at least static, it is selected as a pivot; otherwise, the pivot is given the value that has the same sign but absolute value static. Further details are given in Section 4.3. The default value is 0.0. **Restriction:** Either static=0.0 or static≥small.

u is a scalar of type REAL that holds the initial value of the relative pivot tolerance u used. The default is u=0.01 in the double precision version and u=0.1 in the single precision. Values outside the range $[0, 1.0]$ are treated as the default.

umin is a scalar of type REAL that holds the minimum value of the relative pivot tolerance. If, at any stage of the computation, fewer than the expected number of stable pivots have been found using the current tolerance u and the candidate pivot with greatest relative pivot tolerance has tolerance $v \geq$umin, this is accepted as a pivot and the tolerance u is set to v. The default is umin=0.01. Values of umin greater than u are treated as u and values less than 0 are treated as 0.

2.4.9 The derived data type for holding information

The derived data type MA86_info is used to hold parameters that give information about the progress and needs of the algorithm. The components of MA86_info (in alphabetical order) are:

detlog is a scalar of type REAL. On exit from MA86_factor or MA86_factor_solve, it holds the logarithm of the absolute value of the determinant of A or zero if the determinant is zero.

detarg is a scalar of package type that is absent in the real case. On exit from MA86_factor or MA86_factor_solve in the complex symmetric case, it holds the determinant of A divided by its absolute value or one if the determinant is zero.

detsign is a scalar of type default INTEGER. On exit from MA86_factor or MA86_factor_solve in the real or complex Hermitian case, it holds the sign of the determinant of A or zero if the determinant of A is zero.

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

matrix_rank is a scalar of type INTEGER. On exit from MA86_factor and MA86_factor_solve, it holds the rank of the factorized matrix.

maxdepth is a scalar of type INTEGER. On exit from MA86_analyse, it holds the maximum depth of the assembly tree.

num_delay is a scalar of type INTEGER. On exit from MA86_factor and MA86_factor_solve, it holds the number of eliminations that were delayed, that is, the total number of pivot candidiates that were passed to the parent node in the assembly because of stability considerations. If a variable is passed further up the assembly tree, it will be counted again. A large value of num_delay indicates that substantial modifications were made to the pivot sequence to ensure stability (and the number of entries num_factor in L and number of flops num_flops used to compute L will be significantly more than predicted by MA86_analyse).

num_factor is a scalar of type INTEGER(long). On exit from MA86_analyse, it holds the number of entries that will be in the L factor, assuming the pivot sequence can be used without modification. On exit from MA86_factor and MA86_factor_solve, it holds the actual number of entries in the L factor. Note that, $2n$ entries of D^{-1} are also held.

num_flops is a scalar of type INTEGER(long). On exit from MA86_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 MA86_factor and MA86_factor_solve, it holds the number of floating-point operations performed.

num_neg is a scalar of type INTEGER. On exit from MA86_factor or MA86_factor_solve in the real or complex Hermitian case, it holds the number of negative eigenvalues of the matrix D and is set to zero otherwise.

num_nodes is a scalar of type INTEGER. On exit from MA86_analyse, it holds the number of nodes in the assembly tree.

num_nothresh is a scalar of type INTEGER. On successful exit from MA86_factor and MA86_factor_solve, it holds the number of pivots which did not satisfy the threshold criteria based on the value of control%u.

num_perturbed is a scalar of type INTEGER. On successful exit from MA86_factor and MA86_factor_solve, it holds number of pivots that were replaced by control%static.

num_sup is a scalar of type INTEGER. On exit from the final call to MA86_analyse, it holds the number of supervariables in the problem.

num_two is a scalar of type INTEGER. On exit from MA86_factor and MA86_factor_solve, it holds the number of 2×2 blocks in D.

pool_size is a scalar of type INTEGER. On exit from MA86_factor and MA86_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.

usmall is a scalar of type REAL. On successful exit from MA86_factor and MA86_factor_solve, if num_perturbed=0, usmall holds the threshold parameter that was used; otherwise usmall is set to zero.

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 MA86_analyse if an error is found in the user-supplied elimination order (held in order).

-3 Returned by MA86_factor and MA86_factor_solve if control%action = .false. and the matrix is found to be singular.

-4 Returned by MA86_factor_solve and MA86_solve if there is an error in the size of array x (that is, lx<n or nrhs<1).

-5 Returned by MA86_factor and MA86_factor_solve if IEEE infinities found in the factorization, probably caused by control%small or control%u having too small a value.

-6 Returned by MA86_solve if job is out of range.

-7 Immediate return from MA86_factor and MA86_factor_solve if control%static < abs(control%small) and control%static ≠ 0.0.

-8 Returned by MA86_factor and MA86_factor_solve in the complex case if matrix_type is invalid.

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

$+1$ Returned by MA86_factor and MA86_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 MA86_factor and MA86_factor_solve if control%action=.true. and the matrix is found to be singular.

$+3$ Returned by MA86_factor and MA86_factor_solve if both of the above two warnings are issued (that is, the task pool is found to be too small and the matrix is found to be singular).

3 GENERAL INFORMATION

Workspace: `HSL_MA86` handles its own memory allocations.

Other routines called directly: HSL packages `HSL_MC34`, `MC64`, `MC77` and `HSL_MC78`, BLAS routines `_copy`, `_axpy`, `_swap`, `_gemm`, `_gemv`, `_trsm`, `_trsv`.

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`$\geq$`1`, `lx`$\geq$`n`, `job = 0,1,2,3,4`. Complex case `matrix_type = -4` or `-5`.

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

4 METHOD

`HSL_MA86` divides the sparse factorization into tasks, each of which alters a single block or a block column. The three different types of tasks are referred to as the factorize_column task, the update_internal task, and the update_between task; they are discussed in detail in [1]. The 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

$$\begin{array}{ccc|cc}
1 & & & & \\
4 & 5 & & & \\
7 & 8 & 9 & & \\
\cline{1-3}
10 & 11 & 12 & 25 & \\
13 & 14 & 15 & 27 & 28 \\
16 & 17 & 18 & 29 & 30 \\
\hline
19 & 20 & 21 & 31 & 32 \\
22 & 23 & 24 & 33 & 34
\end{array}$$

4.2 The analyse phase

The analyse phase uses only the sparsity pattern of A. It requires the user to input the lower and upper triangular parts 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.

4.3 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. We also hold a dependency count for each block column. This equal to the sum of the dependency counts of the blocks within the block column. Each block of L is set to zero the first time that it is accessed and the entries of A are added into the blocks within a block column at the start of a factorize_column task.

During factorize, the block count (and corresponding block column count) is decremented by one after the completion of each update for it. When the block column count reaches zero, a factorize_column task is added to the task pool.

When a factorize_column 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.

The factorize_column task incorporates threshold partial pivoting. The relative pivot tolerance u is initially set to `control%u`. If a pivot candidate does not satisfy the threshold pivot criteria, the action taken depends on the control parameters `control%umin` and `control%static`. If static pivoting is not requested (`control%static=0.0`) and `control%umin=control%u`, the pivot is delayed (this is the default case). In our experience, if a large number of pivots are delayed (see `info%num_delay`), the performance of `HSL_MA86` can be badly effected and so we have included options that can help limit the number of delayed pivots (possibly at the cost of a less stable factorization). If `control%umin<control%u` and the relative pivot tolerance for the pivot candidate is $v \geq$`control%umin`, the candidate is accepted and u is set to v. The factorization continues using this new value. If $v <$`control%umin`, the pivot is delayed unless static pivoting is being used. In this case, if the candidate has absolute value at least `control%static`, it is selected and `info%num_nothresh` is incremented by one; otherwise, the pivot is given the value that has the same sign but absolute value `control%static` and `info%num_perturbed` is incremented by one. Note that if a small relative pivot tolerance is used and/or static pivoting is used, the factorization is likely to be inaccurate and an iterative procedure (such as iterative refinement) may be needed once the factorization is complete to try and restore accuracy. Our experience is that the accuracy can be very sensitive to the choice of `control%static`; in our tests in double precision, a value of $\|\mathbf{A}\| * 10^{-6}$ was an appropriate choice.

If the user wishes to solve $AX = B$ at the same time as factorizing the matrix, the call to `MA86_factor` should be replaced by a call to `MA86_factor_solve`. The user must pass right-hand vectors to `MA86_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, `MA86_factor_solve` performs the back substitutions by calling `MA86_solve` with `job = 4`. Using `MA86_factor_solve` is more efficient than calling `MA86_factor` followed by `MA86_solve`.

4.4 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 and J.A. Scott. (2010). An indefinite sparse direct solver for multicore machines Technical Report TR-RAL-2010-011.
Available from `http://www.numerical.rl.ac.uk/reports/reports.shtml`

5 EXAMPLE OF USE

5.1 Example 1

We wish to solve the indefinite system

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

The following code may be used.

```
program hsl_ma86ds
   use hsl_ma86_double
   use hsl_mc69_double
   implicit none

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

   type (ma86_keep)    :: keep
   type (ma86_control) :: control
   type (ma86_info)    :: info

   integer :: i, n
   ! integer :: flag, more ! uncomment for error 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_INDEF, 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 ma86_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 ma86_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 ma86_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 ma86_finalise(keep, control)

end program hsl_ma86ds
```

The input file is:

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

$$\begin{pmatrix} -3. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 4. & \\ & 1. & & & 2. \end{pmatrix} X = \begin{pmatrix} -1. \\ 12. \\ 10. \\ 8. \\ 4. \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} -5. & 2. & & & \\ 2. & 9. & 3. & & -2. \\ & 3. & 6. & 1. & \\ & & 1. & -5. & \\ & -2. & & & 6. \end{pmatrix} X = \begin{pmatrix} -1. & -11. \\ 19. & 21. \\ 28. & 14. \\ -17. & -9. \\ 26. & 14. \end{pmatrix},$$

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

```
program hsl_ma86ds1
   use hsl_ma86_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(ma86_keep)    :: keep
   type(ma86_control) :: control
   type(ma86_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_INDEF, 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 ma86_analyse(n, ptr, row, order, keep, control, info)
   call stop_on_bad_flag("analyse", info%flag)

   ! Factor
   call ma86_factor(n, ptr, row, val, order, keep, control, info)
   call stop_on_bad_flag("factor", info%flag)

   ! Solve
   call ma86_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_INDEF, lmap, map, cval, &
      ptr(n+1)-1, val)

   ! Perform second factorization and solve
   call ma86_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 ma86_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_ma86ds1
```

The input file is:

```
  5  9
  1  1  2  2  2  3  3  4  5
  1  2  2  3  5  3  4  4  5
 -3. 1. 4. 1. 1. 3. 2. 4. 2.
 -1. 12. 10. 8. 4.
 -5. 2. 9. 3. -2. 6. 1. -5. 6.
  2
 -1. 19. 28. -17. 26.
 -11. 21. 14. -9. 14.
```

This produces the following output:

```
 Computed solution:
     1.000     2.000     2.000     1.000     1.000
 Computed solutions:
     1.000     2.000     3.000     4.000     5.000
     3.000     2.000     1.000     2.000     3.000
```