1 SUMMARY

To **solve a sparse unsymmetric system of linear equations.** Given a sparse matrix $\mathbf{A} = \{a_{ij}\}_{m\times n}$ and a vector $\mathbf{b}$, this subroutine solves the system $\mathbf{Ax} = \mathbf{b}$ or the system $\mathbf{A}^T\mathbf{x} = \mathbf{b}$. The matrix $\mathbf{A}$ can be rectangular. There is an option for iterative refinement and return of error estimates.

The package HSL_MA48 is an update to the package MA48, and offerse several additional features. For example, there is an option to analyse the matrix and generate the factors with a single call. The storage required for the factorization is chosen automatically and, if there is insufficient space for the factorization, more space is allocated and the factorization is continued. It also returns the number of entries in the factors and has facilities for computing the determinant when the matrix is square and for identifying the rows and columns that are treated specially when the matrix is singular or rectangular. In order to treat matrices with more entries than $2^{31}-1$ (around 2.1×10^9), long integers are used for components in some of the derived data types as indicated in the description of each data type.

ATTRIBUTES — Version: 3.3.0 (14 January 2014). **Interfaces:** C, Fortran, MATLAB. **Types:** Real (single, double). **Calls:** MC71, HSL_ZB01, HSL_ZD11, _AXPY, _DOT, _GEMM, _GEMV, _SWAP, _TRSM, _TRSV, _SCAL, I_AMAX. **Language:** Fortran 2003 subset (F95 + TR15581 + C interoperability). **Original date:** November 2001. **Origin:** Version 1 and 2: I.S. Duff and J.K. Reid (Rutherford Appleton Laboratory). Version 3: I.S. Duff (Rutherford Appleton Laboratory).

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 having rows and columns $0, 1, \ldots n-1$. 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, additional Fortran compiler libraries may need to be linked explicitly.

2.2 Calling sequences

Access to the package requires inclusion of the header file

Single precision version

```
#include "hsl_ma48s.h"
```

Double precision version

```
#include "hsl_ma48d.h"
```

If it is required to use more than one module at the same time, include both header files, but append _s (single) or _d (double) to all type and function names.

There are six principal subroutines for user calls:

ma48_default_control sets default values for members of the ma48_control data type needed by other routines. If non-default values are wanted for any of the control members, the corresponding members should be altered after the call to ma48_default_control.

ma48_initialize must be called to initialize the structure for the factors.

ma48_analyse accepts the pattern of **A** and chooses pivots for Gaussian elimination using a selection criterion to preserve sparsity. It will optionally find an ordering to block triangular form and exploit that structure. An option exists to restrict pivoting to the diagonal, which might reduce fill-in and operations if the matrix has a symmetric structure. It is possible to perform an analysis without generating the factors, in which case data on the costs of a subsequent factorization are returned to the user. The user can also input a desired pivotal sequence. In this case, the block triangular form will not be computed, the user's column ordering will be respected, but the row ordering might be changed for reasons of stability. It is also possible to request that a set of columns are pivoted on last in which case a subsequent factorization can avoid factorization operations on the earlier columns.

ma48_factorize factorizes a matrix **A** using the information from a previous call to ma48_analyse. The actual pivot sequence used may differ from that of ma48_analyse. An option exists for a fast factorization where the pivot sequence chosen is identical to the previous factorization and data structures from this earlier factorization are used.

ma48_solve uses the factors generated by ma48_factorize to solve a system of equations $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$.

ma48_finalize frees memory allocated by the package. It must be called when all the systems involving its matrix have been solved unless the structure is about to be used for the factors of another matrix.

There are also two auxiliary subroutines for user calls after a successful factorization:

ma48_determinant computes the determinant and is for use following a call of ma48_factorize.

ma48_special_rows_and_cols identifies the rows and columns that are treated specially when the matrix is singular or rectangular. It is for use following a call of ma48_factorize.

2.3 The derived data types

For each problem, the user must employ structures defined in the header file to declare structures for controlling the factorization and providing information, and a void * pointer to reference the factors. The following pseudocode illustrates this.

```
#include "hsl_ma48d.h"
...
void *factors;
struct ma48_control control;
struct ma48_ainfo info_analyse;
struct ma48_finfo info_factorize;
struct ma48_sinfo info_solve;
...
```

The members of ma48_control, ma48_ainfo, ma48_finfo, and ma48_sinfo are explained in Sections 2.3.2, 2.3.3, 2.3.4, and 2.3.5, respectively. The void * pointer is used to pass data between the subroutines of the package and must not be altered by the user.

2.3.1 Package types

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
```

2.3.2 Derived data type for control of the subroutines

The header file defines a structure, `struct ma48_control`, with the following components, which may be given default values through a call to `ma48_default_control`.

C only controls:

`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 `row`, `col`, `perm`, `endcol`, `rows`, and `cols` is assumed. Otherwise, if `f_arrays=0` (i.e. evaluates to false), then these arrays are copied a 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 lp` is used by the subroutines as the Fortran output unit for error messages. If it is negative, these messages will be suppressed. The default value is `6`.

`int wp` is used by the subroutines as the output unit for warning messages. If it is negative, these messages will be suppressed. The default value is `6`.

`int mp` is used by the subroutines as the output unit for diagnostic printing. If it is negative, these messages will be suppressed. The default value is `6`.

`int ldiag` is used by the subroutines to control diagnostic printing. If `ldiag` is less than `1`, no messages will be output. If the value is `1`, only error messages will be printed. If the value is `2`, then error and warning messages will be printed. If the value is `3`, scalar data and a few entries of array data on entry and exit from each subroutine will be printed. If the value is greater than `3`, all data will be printed on entry and exit. The default value is `2`.

Algorithmic controls:

`int btf` is used by `ma48_analyse` to define the minimum size of a block of the block triangular form other than the final block. If block triangularization is not wanted, `btf` should be set to a value greater than or equal to `n`. A non-positive value is regarded as the value `1`. For further discussion of this variable, see Section 2.6. The default value is `1`.

`pkgtype cgce` is used by `ma48_solve`. It is used to monitor the convergence of the iterative refinement. If successive corrections do not decrease by a factor of at least `cgce`, convergence is deemed to be too slow and `ma48_solve` terminates with `sing.flag` set to `-8`. The default value is `0.5`.

`int diagonal_pivoting` is used by `ma48_analyse` to limit pivoting to the diagonal. If `diagonal_pivoting` evaluates to true, it will do so. Otherwise, it will not. Its default value is `0` (false).

`pkgtype drop` is used by `ma48_analyse` and `ma48_factorize`. Any entry whose modulus is less than `drop` will be dropped from the factorization. The factorization will then require less storage but may be inaccurate. The default value is `0.0`.

int factor_blocking is used by ma48_factorize to determine the block size used for the Level 3 BLAS within the full factorization. If it is set to 1, Level 1 BLAS are used, if to 2, Level 2 BLAS are used. The default value is 32.

int fill_in is used by ma48_analyse to determine the initial storage allocation for the matrix factors. It will be set to fill_in times the value of argument ne. The default value is 3.

pkgtype multiplier is used by ma48_factorize when a real or integer array that holds data for the factors is too small. The array is reallocated with its size changed by the factor multiplier. The default value is 2.0. The value actually used in the code is the minimum of multiplier and 1.2.

int pivoting is used to control numerical pivoting by ma48_analyse. If pivoting has a positive value, each pivot search is limited to a maximum of pivoting columns. If pivoting is set to the value 0, a full Markowitz search technique is used to find the best pivot. This is usually only a little slower, but can occasionally be very slow. It may result in reduced fill-in. The default value is 3.

int solve_blas is used by ma48_solve to determine whether Level 2 BLAS are used (solve_blas $>$ 1) or not (solve_blas $\leq$ 1). The default value is 2.

int maxit is used by ma48_solve to limit the number of refinement iterations. If maxit is set to zero then ma48_solve will not perform any error analysis or iterative refinement. The default value is 10.

int struct_ is used by ma48_analyse. If struct_ evaluates to true, the subroutine will exit immediately structural singularity is detected. The default value is 0 (false).

pkgtype switch_ is used by ma48_analyse to control the switch from sparse to full matrix processing when factorizing the diagonal blocks. The switch is made when the ratio of the number of entries in the reduced matrix to the number that it would have as a full matrix is greater than switch. A value greater than 1.0 is treated as 1.0. The default value is 0.5.

int switch_mode is used by ma48_factorize. If it evaluates to true, a switch to slow mode is made when the fast mode is given an unsuitable pivot sequence. The default value is 0 (false).

pkgtype tolerance is used by ma48_analyse and ma48_factorize. If it is set to a positive value, any pivot whose modulus is less than tolerance will be treated as zero. The default value is 0.0.

pkgtype u is used by ma48_analyse and ma48_factorize. It holds the threshold parameter for the pivot control. The default value is 0.01. For problems requiring greater than average numerical care a higher value than the default would be advisable. Values greater than 1.0 are treated as 1.0 and less than 0.0 as 0.0.

2.3.3 Derived data type for information from ma48_analyse

The header file defines a structure, struct ma48_ainfo, with the following components

int flag is used to store the return code. The value zero indicates that the subroutine has performed successfully. For nonzero values, see Section 2.5.1.

int more provides further information in the case of an error, see Section 2.5.1.

int stat is set to the Fortran STAT value in the case of the failure of an allocate or deallocate statement.

long drop is set to the number of entries dropped from the data structure.

long dup is set to the number of duplicate entries.

int lblock holds the order of the largest non-triangular block on the diagonal of the block triangular form. If the matrix is rectangular, lblock will hold the number of rows.

long lena_analyse gives the number of floating point and integer words required for the analysis.

long lena_factorize gives the number of floating point words required for successful subsequent factorization assuming the same pivot sequence and set of dropped entries can be used.

long lenj_analyse gives the number of integer words required for an auxiliary array for the analysis.

long leni_factorize gives the number of integer words required for successful subsequent factorization assuming the same pivot sequence and set of dropped entries can be used. In the present version (3.0.0) of the code, leni_factorize is equal to lena_factorize.

int ncmpa holds the number of compresses of the internal data structure performed by ma48_analyse. If ncmpa is fairly large (say greater than 10), performance may be very poor.

long oor is set to the number of entries with one or both indices out of range.

pkgtype ops is set to the number of floating-point operations required by the factorization.

int rank gives an estimate of the rank of the matrix.

int sblock holds the sum of the orders of all the non-triangular blocks on the diagonal of the block triangular form. If the matrix is rectangular, sblock will hold the number of columns.

int struc_rank holds the structural rank of the matrix if btf is less than or equal to n. If btf is greater than n, struc_rank is set to min(m, n).

long tblock holds the total number of entries in all the non-triangular blocks on the diagonal of the block triangular form.

2.3.4 Derived data type for information from ma48_factorize and, optionally, from ma48_analyse

The header file defines a structure, struct ma48_finfo, with the following components

int flag is a return code. The value zero indicates that the subroutine has performed successfully. For nonzero values, see Section 2.5.2.

int more provides further information in the case of an error, see Section 2.5.2.

int stat is set to the Fortran STAT value in the case of the failure of an allocate or deallocate statement.

long drop is set to the number of entries dropped from the data structure.

pkgtype ops is set to the number of floating-point operations required by the factorization.

long lena_factorize gives the number of floating point words required for successful subsequent factorization assuming the same pivot sequence and set of dropped entries can be used.

long leni_factorize gives the number of integer words required for successful subsequent factorization assuming the same pivot sequence and set of dropped entries can be used. In the present version (3.0.0) of the code leni_factorize is equal to lena_factorize.

int rank gives an estimate of the rank of the matrix.

long size_factor gives the number of entries in the matrix factors.

2.3.5 Derived data type for information from `ma48_solve`

The header file defines a structure, `struct ma48_sinfo` with the following components

`int flag` is a return code. The value zero indicates that the subroutine has performed successfully. For nonzero values, see Section 2.5.3.

`int more` provides further information in the case of an error, see Section 2.5.3.

`int stat` is set to the Fortran `STAT` value in the case of the failure of an allocate or deallocate statement.

2.4 Argument lists

2.4.1 The default setting subroutine

Default values for members of the `ma48_control` structure may be set by a call to `ma48_default_control`.

```
void ma48_default_control(struct ma48_control *control)
```

`control` has its members set to their default values, as described in Section 2.3.2.

2.4.2 The initialization subroutine

The initialization subroutine must be called to allocate space to store the factors. Space allocated using this routine can only be freed through a call to `ma48_finalize`.

```
void ma48_initialize(void **factors)
```

`factors` will be set to point at an area of memory allocated using a Fortran `allocate` statement that will be used to hold the factorization generated in later calls. To avoid a memory leak, the subroutine `ma48_finalize` must be used to clean up and deallocate this memory once the factorization is no longer required.

2.4.3 To analyse the sparsity pattern

```
void ma48_analyse(int m, int n, long ne, const int row[], const int col[],
   const pkgtype val[], void *factors, const struct ma48_control *control,
   struct ma48_ainfo *ainfo, struct ma48_finfo *finfo, const int perm[],
   const int endcol[])
```

`m` must hold the number of rows in the matrix. **Restriction:** `m` $\geq$ `1`.

`n` must hold the number of columns in the matrix. **Restriction:** `n` $\geq$ `1`.

`ne` must hold the number of entries in the matrix. **Restriction:** `ne` $\geq$ `0`.

`row` is a rank-1 array of size `ne`. It must be set such that `row[i]` holds the row of the `i`-th entry of the matrix.

`col` is a rank-1 array of size `ne`. It must be set such that `col[i]` holds the column of the `i`-th entry of the matrix.

`val` is a rank-1 array of size `ne`. It must be set such that `val[i]` holds the value of the `i`-th entry of the matrix.

`factors` must have been initialized by a call to `ma48_initialize` or have been used for a previous calculation. In the latter case, the previous data will be lost but fewer memory allocations may be required.

control is used to control the actions of the package, see Section 2.3.2 for details.

ainfo is used to return information about the execution, as explained in Section 2.3.3.

finfo may be NULL. If it is not NULL, the call to ma48_analyse will additionally compute and store the factorization of the matrix. Its members provide information about the execution of the factorization, as explained in Section 2.3.4.

perm must be NULL or a rank-1 array of size $(m+n)$. If it is not NULL, perm[i], i = 0, 1, ..., m-1, should be set to the position of row i in the permuted matrix, and perm[m+j], j = 0, 1, ..., n-1, should be set to the index of the column that is in position j in the permuted matrix. In this case, the block triangular form will not be computed. The routine will try to use this input pivotal sequence but may change the row ordering if necessary for stability reasons.

endcol must be NULL or a rank-1 array of size n. If not NULL, ma48_analyse will place each column j for which endcol[j]=-1 (0 if control.f_arrays evaluates to true) at the end of the pivot sequence within its block. A subsequent call to ma48_factorize can save work by assuming that only these columns are changed since the previous call.

2.4.4 To perform a factorization

```
void ma48_factorize(int m, int n, long ne, const int row[], const int col[],
   const pkgtype val[], void *factors, const struct ma48_control *control,
   struct ma48_finfo *finfo, int fast, int partial)
```

m, n, ne, row, col: see Section 2.4.3; must be unaltered since call to ma48_analyse.

val is a rank-1 array of size ne. It must be set such that val[i] holds the value of the i-th entry of the matrix.

factors must be unaltered since the call to ma48_analyse or a subsequent call to ma48_factorize.

control see Section 2.4.3.

finfo provides information about the execution, as explained in Section 2.3.4.

fast controls use of a fast factorization, which may only be used if there has been at least one successful factorization on a previous matrix. If fast evaluates to true, the factorization will use the same pivot sequence as the previous factorization. It will also utilize data structures from the earlier factorization to effect a more rapid factorization. If, however, entries were dropped from the previous analysis or factorization, this option will be inoperative and the matrix will be factorized using the same pivot sequence but will regenerate the data structures during the factorization.

partial controls use of a partial factorization. If partial evaluates to true, the factorization will be performed on only the last columns of the matrix that were flagged by the argument endcol during the call to ma48_analyse. If, however, entries were dropped from the previous analysis or factorization, this option will be inoperative and the matrix will be factorized using the same pivot sequence but will regenerate the data structures during the factorization.

2.4.5 To solve a set of equations

```
void ma48_solve(int m, int n, long ne, const int row[], const int col[],
   const pkgtype val[], const void *factors, const pkgtype rhs[], pkgtype x[],
   const struct ma48_control *control, struct ma48_sinfo *sinfo, int trans,
   pkgtype resid[], pkgtype *error)
```

m, n, ne, row, col, val, factors: see Section 2.4.3; must be unaltered since call to ma48_analyse or ma48_factor.

rhs is a rank-1 array of size n. It must be set by the user to the vector **b**.

x is a rank-1 array of size n. On return it holds the solution **x**.

control see Section 2.4.3.

sinfo provides information about the execution, as explained in Section 2.3.5.

trans controls transposed solution. If trans evaluates to true, $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ is solved, otherwise the solution is obtained for $\mathbf{A}\mathbf{x} = \mathbf{b}$.

resid must be NULL or a rank-1 array of size 2. If not NULL and control.maxit $\geq$ 1, the scaled residual for the two categories of equations (see Section 2.9) will be held in resid[0] and resid[1], respectively.

error may be NULL. If not NULL and control.maxit $\geq$ 1, an estimate of the error in solving the equations (see Section 2.9) will be held in *error.

2.4.6 The finalization subroutine

```
int ma48_finalize(void **factors, const struct ma48_control *control)
```

factors will have all associated memory deallocated and *factors will be NULL on exit.

control see Section 2.4.3.

Return code:
On return, the value 0 indicates success. Any other value is the Fortran STAT value of a DEALLOCATE statement that has failed.

2.4.7 To compute the determinant following a successful factorization

```
int ma48_determinant(const void *factors, int *sgndet, pkgtype *logdet,
   const struct ma48_control *control)
```

factors see Section 2.4.4; must be unaltered since last call to ma48_analyse or ma48_factorize.

*sgndet holds, on return, the value 1 if the determinant is positive, -1 if the determinant is negative, or 0 if the determinant is zero or the matrix is not square.

*logdet holds, on return, the logarithm of the absolute value of the determinant, or zero if the determinant is zero or the matrix is not square.

Return code:
On return, its value is 0 if the call was successful and is -1 if the allocation of a temporary array failed.

2.4.8 To identify the rows and columns that are treated specially following a successful factorization

```
int ma48_special_rows_and_cols(const void *factors, int *rank,
   int rows[], int cols[], const struct ma48_control *control)
```

`factors` see Section 2.4.4; must be unaltered since last call to `ma48_analyse` or `ma48_factorize`.

`*rank` holds, on return, the calculated rank of the matrix (it is the rank of the matrix actually factorized).

`rows` is a rank-1 array of size `m`. On return, it holds a permutation. The indices of the rows that are taken into account when solving $\mathbf{Ax} = \mathbf{b}$ are `rows[i]`, `i` $<$ `rank`.

`cols` is a rank-1 array of size `n`. On return, it holds a permutation. The indices of the columns that are taken into account when solving $\mathbf{Ax} = \mathbf{b}$ are `cols[i]`, `i` $<$ `rank`.

Return code:
On return, its value is `0` if the call was successful and is `-1` if the allocation of a temporary array failed.

2.5 Error diagnostics

2.5.1 When performing the analysis.

A successful return from `ma48_analyse` is indicated by `ainfo.flag` having the value zero. A negative value is associated with an error message that will be output on Fortran unit `control.lp`. Possible negative values are:

`-1` Value of `m` out of range. `m` $<$ `1`. `ainfo.more` is set to the value of `m`.

`-2` Value of `n` out of range. `n` $<$ `1`. `ainfo.more` is set to the value of `n`.

`-3` Value of `ne` out of range. `ne` $<$ `0`. `ainfo.more` is set to the value of `ne`.

`-4` Failure of an allocate or deallocate statement. `ainfo.stat` is set to the Fortran `STAT` value.

`-5` On a call where `control.struct_` evaluates to true, the matrix is structurally rank deficient. The structural rank is given by `struc_rank`.

`-6` The array `perm` does not hold valid permutations. `ainfo.more` holds the first component at which an error was detected.

`-7` An error occurred in a call to `ma48_factorize` (when `finfo` was not `NULL` in the call). The only reason why this can happen is because of an allocation error in `ma48_factorize`.

A positive flag value is associated with a warning message that will be output on Fortran unit `ainfo.wp`. Possible positive values are:

`+1` Index (in `row` or `col`) out of range. Action taken by subroutine is to ignore any such entries and continue. `ainfo.oor` is set to the number of such entries. Details of the first ten are optionally printed on unit `control.mp`.

`+2` Duplicate indices. Action taken by subroutine is to sum corresponding reals. `ainfo.dup` is set to the number of duplicate entries. Details of the first ten are optionally printed on unit `control.mp`.

`+3` Combination of a `+1` and a `+2` warning.

`+4` The matrix is rank deficient with estimated rank `ainfo.rank`.

+5 Combination of a +1 and a +4 warning.

+6 Combination of a +2 and a +4 warning.

+7 Combination of a +1, a +2, and a +4 warning.

+8 Not possible to choose all pivots from diagonal (call with control.diagonal_pivoting evaluating to true).

+9 to +15 Combination of warnings that sum to this total.

2.5.2 When factorizing the matrix

A successful return from ma48_factorize is indicated by finfo.flag having the value zero. A negative value is associated with an error message that will be output on Fortran unit control.lp. In this case, no solution will have been calculated. Possible negative values are:

-1 Value of m differs from the ma48_analyse value. finfo.more holds the value of m.

-2 Value of n differs from the ma48_analyse value. finfo.more holds the value of n.

-3 Value of ne out of range. ne $<$ 0. finfo.more holds the value of ne.

-4 Failure of an allocate statement. finfo.stat is set to the Fortran STAT value.

-10 ma48_factorize has been called without a prior call to ma48_analyse.

-11 ma48_factorize has been called with fast evalutating to true, but the matrix entries are unsuitable for this.

A positive flag value is associated with a warning message that will be output on Fortran unit control.mp. In this case, a factorization will have been calculated.

+4 Matrix is rank deficient. In this case, finfo%rank will be set to the rank of the factorization. In the subsequent solution, all columns in the singular block will have the corresponding component in the solution vector set to zero.

2.5.3 When using factors to solve equations

A successful return from ma48_solve is indicated by sinfo.flag having the value zero. A negative value is associated with an error message that will be output on Fortran unit control.lp. In this case, the solution will not have been completed. Possible negative values are:

-1 Value of m differs from the ma48_analyse value. sinfo.more holds the value of m.

-2 Value of n differs from the ma48_analyse value. sinfo.more holds the value of n.

-3 Value of ne out of range. ne $<$ 0. sinfo.more holds the value of ne.

-8 Iterative refinement has not converged. This is an indication that the system is very ill-conditioned. The solution may not be accurate although estimates of the error can still be obtained by ma48_solve.

-9 A problem has occurred in the calculation of matrix norms using MC71A/AD. See the documentation for this routine.

-10 ma48_solve has been called without a prior call to ma48_factorize.

2.6 Rectangular and rank deficient matrices

Rectangular matrices are handled by the code although no attempt is made at prior block triangularization. Rank deficient matrices are also factorized and a warning flag is set (`ainfo.flag` or `finfo.flag` set to `+4`). If `control.struct_` evaluates to true, then an error return occurs (`ainfo.flag = -5`) if block triangularization is attempted and the matrix is structurally singular.

The package identifies a square submatrix of $\mathbf{A}$ that it considers to be nonsingular. When solving $\mathbf{Ax} = \mathbf{b}$, equations outside this submatrix are ignored and solution components that correspond to columns outside the submatrix are set to zero. `ma48_special_rows_and_cols` identifies the rows and columns of this submatrix from stored integer data.

It should be emphasized that the primary purpose of the package is to solve square nonsingular sets of equations. The rank is determined from the number of pivots that are not small or zero. There are more reliable (but much more expensive) ways of determining numerical rank.

2.7 Block upper triangular form

Many large unsymmetric square matrices can be permuted to the form

$$\mathbf{PAQ} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdot & \cdot & \cdot & \cdot \\ & \mathbf{A}_{22} & \cdot & \cdot & \cdot & \cdot \\ & & \mathbf{A}_{33} & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot \\ & & & & & \mathbf{A}_{ll} \end{pmatrix}$$

whereupon the system

$\mathbf{Ax} = \mathbf{b}$ (or $\mathbf{A}^T\mathbf{x} = \mathbf{b}$)

can be solved by block back-substitution giving a saving in storage and execution time if the matrices $\mathbf{A}_{ii}$ are much smaller than $\mathbf{A}$.

Since it is not very efficient to process a small block (for example a 1×1 block), any block of size less than `control.btf` other than the final block is merged with its successor.

2.8 Badly-scaled systems

If the user's input matrix has entries differing widely in magnitude, then an inaccurate solution may be obtained. In such cases, the user is advised to first use a scaling routine (for example, `MC29A/AD`, `MC64A/AD`, or `MC77A/AD`) to obtain scaling factors for the matrix and then explicitly scale it prior to calling this package.

2.9 Error estimates

We calculate an estimate of the sparse backward error using the theory and measure developed by Arioli, Demmel, and Duff (1989). We use the notation $\bar{\mathbf{x}}$ for the computed solution and a modulus sign on a vector or matrix to indicate the vector or matrix obtained by replacing all entries by their moduli. The scaled residual

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}||\bar{\mathbf{x}}|)_i} \tag{2.1}$$

is calculated for all equations except those for which the numerator is nonzero and the denominator is small. For the exceptional equations,

$$\frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{A}||\bar{\mathbf{x}}|)_i + ||\mathbf{A}_i||_\infty ||\bar{\mathbf{x}}||_\infty} \tag{2.2}$$

is used instead, where $\mathbf{A}_i$ is row i of $\mathbf{A}$. The largest scaled residual (2.1) is returned in `resid[0]` and the largest scaled residual (2.2) is returned in `resid[1]`. If all equations are in category (2.1), zero is returned in `resid[1]`. The computed solution $\bar{\mathbf{x}}$ is the exact solution of the equation

$$(\mathbf{A}+\delta\mathbf{A})\mathbf{x} \;=\; (\mathbf{b}+\delta\mathbf{b})$$

where $\delta\mathbf{A}_{ij} \leq \max(\texttt{resid[0]},\texttt{resid[1]})|\mathbf{A}|_{ij}$ and $\delta\mathbf{b}_i \leq \max(\texttt{resid[0]}|\mathbf{b}|_i,\texttt{resid[1]}||\mathbf{A}_i||_\infty||\bar{\mathbf{x}}||_\infty)$. Note that $\delta\mathbf{A}$ respects the sparsity in $\mathbf{A}$. For the square case, `resid[0]` and `resid[1]` can also be used with appropriate condition numbers to obtain an estimate of the relative error in the solution,

$$\frac{||\mathbf{x}-\bar{\mathbf{x}}||_\infty}{||\bar{\mathbf{x}}||_\infty},$$

which is returned in `*error`.

Reference [1] Arioli, M. Demmel, J. W., and Duff, I. S. (1989). Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.* **10**, 165-190.

2.10 Determinant

For the determinant, the package computes the parity of the permutations that have been applied and the product of the diagonal entries of the triangular factors.

3 GENERAL INFORMATION

Other routines called directly: The auxiliary routines that are revised Fortran 95 versions of routines from the packages `MA48`, `MA50`, `MA51`, `MC21`, and `MC13` are internal to the package. Packages `MC71`, `HSL_ZD11` and `HSL_ZB01` are also used.

Input/output: Error, warning and diagnostic messages only. Error messages on unit `control.lp` and warning and diagnostic messages on unit `control.WP` and `control.mp`, respectively. These have default value `6`, and printing of these messages is suppressed if the relevant unit number is set negative. These messages are also suppressed if `control.ldiag` is less than `1`.

Restrictions: `m` $\geq$ `1`, `n` $\geq$ `1`, `ne` $\geq$ `0`.

4 METHOD

A version of sparse Gaussian elimination is used.

The `ma48_analyse` entry uses a sparse variant of Gaussian elimination to compute a pivot ordering for the decomposition of $\mathbf{A}$ into its $\mathbf{LU}$ factors. It uses pivoting to preserve sparsity in the factors and requires each pivot a_{pj} to satisfy the stability test

$$|a_{pj}| \geq u max_i |a_{ij}|$$

within the reduced matrix, where u is the threshold held in `control.u`, with default value `0.01`. It then optionally computes the numerical factors.

The `ma48_factorize` entry factorizes the matrix by using information generated by `ma48_analyse`. The initial call uses the same stability test as above and generates data structures to allow for the possibility of the faster factorization of subsequent matrices, using the parameter `fast`.

The `ma48_solve` entry uses the factors from `ma48_factorize` to solve systems of equations. Iterative refinement can be performed to improve the accuracy of the solution and to obtain error estimates as discussed in Section 2.9.

A discussion of the design of the MA48 routines called by this package is given by Duff and Reid, *MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations*, Report RAL-93-072, Rutherford Appleton Laboratory (obtainable from the Web page http://www.stfc.ac.uk/CSE/36276.aspx) and in ACM Trans Math Softw **22**, 1996, 187-226.

5 EXAMPLE OF USE

In the example code shown below, we first decompose a matrix and use information from this decomposition to solve a square set of linear equations. Then we factorize a matrix of a similar sparsity pattern and solve another set of equations with iterative refinement and error estimation.

Program

```
/* Simple example of use of HSL_MA48 */

#include <stdio.h>
#include <stdlib.h>
#include "hsl_ma48d.h"

int main(int argc, char **argv) {
   struct ma48_control control;
   struct ma48_ainfo ainfo;
   struct ma48_finfo finfo;
   struct ma48_sinfo sinfo;
   void *factors;

   double *val, *b, *x, res[2], err;
   int i, m, n, *row, *col;
   long ne;

   /* Read matrix order and number of entries */
   scanf("%d %d %ld\n", &m, &n, &ne);

   /* Allocate arrays of appropriate sizes */
   row = (int *) malloc(ne*sizeof(int));
   col = (int *) malloc(ne*sizeof(int));
   val = (double *) malloc(ne*sizeof(double));
   b = (double *) malloc(n*sizeof(double));
   x = (double *) malloc(n*sizeof(double));

   /* Read matrix and right-hand side */
   for(i=0; i<ne; i++) scanf("%d %d %lf\n", &row[i], &col[i], &val[i]);
   for(i=0; i<n; i++) scanf("%lf\n", &b[i]);

   /* Initialize the factors and control*/
   ma48_initialize(&factors);
   ma48_default_control(&control);

   /* Analyse and factorize */
   ma48_analyse(m,n,ne,row,col,val,factors,&control,&ainfo,&finfo,NULL,NULL);
```

```
   if(ainfo.flag != 0) {
      printf("Failure of ma48_analyse with ainfo.flag = %d\n", ainfo.flag);
      return 1;
   }

   /* Solve without iterative refinement */
   ma48_solve(m,n,ne,row,col,val,factors,b,x,&control,&sinfo,0,NULL,NULL);
   if(sinfo.flag == 0) {
      printf("Solution of first set of equations without refinement is:\n");
      for(i=0; i<n; i++) printf("%10.3lf ", x[i]);
      printf("\n\n");
   }

   /* read new matrix and right-hand side */
   for(i=0; i<ne; i++) scanf("%lf\n", &val[i]);
   for(i=0; i<n; i++) scanf("%lf\n", &b[i]);

   /* fast factorize */
   ma48_factorize(m,n,ne,row,col,val,factors,&control,&finfo,1,0);
   if(finfo.flag != 0) {
      printf("Failure of ma48_factorize with finfo.flag=%d\n", finfo.flag);
      return 1;
   }

   /* solve with iterative refinement */
   ma48_solve(m,n,ne,row,col,val,factors,b,x,&control,&sinfo,0,res,&err);
   if(sinfo.flag == 0) {
      printf("Solution of second system with refinement is:\n");
      for(i=0; i<n; i++) printf("%10.3lf ", x[i]);
      printf("\nScaled residual is %10.3le %10.3le\n", res[0], res[1]);
      printf("Estimated error is %10.3le\n", err);
   }

   /* clean up */
   ma48_finalize(&factors, &control);
   free(row); free(col); free(val);
   free(b); free(x);

   return 0;
}
```

Thus if, in this example, we wish to solve:

$$\begin{pmatrix} 3.14 & 7.5 & \\ 4.1 & 3.2 & 0.3 \\ & 1.0 & 4.1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}$$

followed by the system:

$$\begin{pmatrix} 4.7 & 6.2 & \\ 3.2 & 0.0 & 0.31 \\ & 3.1 & 0.0 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1.1 \\ 2.1 \\ 3.1 \end{pmatrix}$$

we have as input

```
3     3     7
0     0     3.14
1     2     0.30
2     2     4.1
1     0     4.1
0     1     7.5
2     1     1.0
1     1     3.2
1.0       2.0          3.0
4.7       0.31         0.0          3.2          6.2
3.1       0.0
1.1       2.1          3.1
```

and the output would be

```
Solution of first set of equations without refinement is:
     0.489     -0.071      0.749

Solution of second system with refinement is:
    -1.085      1.000     17.975
Scaled residual is  7.163e-17  0.000e+00
Estimated error is  3.603e-16
```