1 SUMMARY

HSL_MA97 solves one or more sets of $n \times n$ sparse **symmetric** equations $AX = B$ using a multifrontal method. The package covers the following cases:

1. A is **positive definite**. HSL_MA97 computes the **sparse Cholesky factorization**
$$A = PL(PL)^{\dagger}$$
where $L^{\dagger} = L^T$ (real or complex symmetric) or $L^{\dagger} = L^H$ (complex Hermitian), P is a permutation matrix and L is lower triangular.
2. A is **indefinite**. HSL_MA97 computes the sparse factorization
$$A = PLD(PL)^{\dagger}$$
where $L^{\dagger} = L^T$ (real or complex symmetric) or $L^{\dagger} = L^H$ (complex Hermitian), P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of size 1×1 and 2×2.

HSL_MA97 is designed to produce **bit-compatible solutions on any number of threads** (see Sections 2.3 and 2.4). That is to say, regardless of running in serial or parallel, it will always get the same answer (on the same machine with the same binary).

An option exists to scale the matrix. In this case, the factorization of the scaled matrix $\overline{A} = SAS$ is computed, where S is a diagonal scaling matrix.

For large problems where bit-compatible solutions are not required, HSL_MA86 (or HSL_MA87 for positive-definite systems) may provide significantly better parallel performance. For problems where the factors are too large to fit in memory, HSL_MA77 should be used (this allows the matrix data and computed factors to be held in files). HSL_MA77 may also be used for problems held in element form.

ATTRIBUTES — Version: 2.4.0 (2 September 2016) **Interfaces:** C, Fortran, MATLAB. **Types:** Real (single, double), Complex (single, double). **Uses:** MC30, HSL_MC34, HSL_MC64, HSL_MC68 (optionally using METIS), HSL_MC69, MC77, HSL_MC78, HSL_MC80, _axpy, _gemm, _gemv, _nrm2, _potrf, _swap, _syrk, _trmv, _trmm, _trsm, _trsv. **Original date:** November 2011. **Origin:** J.D. Hogg and J.A. Scott, Rutherford Appleton Laboratory. **Language:** Fortran 2003 subset (F95 + TR155581 + C interoperability). **Parallelism:** OpenMP 3.0. **Remark:** The development of HSL_MA97 was supported by the EPSRC grant 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 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_ma97s.h"
```
Double precision version
```
#include "hsl_ma97d.h"
```
Complex version
```
#include "hsl_ma97c.h"
```
Double complex version
```
#include "hsl_ma97z.h"
```

It is not possible to use more than one version at the same time.

The following procedures are available to the user:

- ma97_default_control sets default values for members of the ma97_control data type needed by other subroutines.
- ma97_analyse accepts the matrix data in compressed sparse column format and optionally checks it for duplicates and out-of-range entries. The user may supply an elimination order; otherwise one is generated. Using this elimination order, ma97_analyse analyses the sparsity pattern of the matrix and prepares the data structures for the factorization.
- ma97_analyse_coord is an alternative to ma97_analyse that may be used if the user has the matrix data held in coordinate format. Again, the user may supply an elimination order; otherwise one is generated. ma97_analyse_coord checks the matrix data for duplicates and out-of-range entries, stores it in compressed sparse column format and then proceeds in the same way as ma97_analyse.
- ma97_factor uses the data structures set up by ma97_analyse to compute a sparse factorization. More than one call to ma97_factor may follow a call to ma97_analyse (allowing more than one matrix with the same sparsity pattern but different numerical values to be factorized without multiple calls to ma97_analyse). An option exists to scale the matrix.
- ma97_factor_solve may be called in place of ma97_factor to factorize A and, at the same time, solve the system $AX = B$. Multiple calls to ma97_factor_solve may follow a call to ma97_analyse.
- ma97_solve uses the computed factors generated by ma97_factor or ma97_factor_solve to solve systems $AX = B$ for one or more right-hand sides B. Multiple calls to ma97_solve may follow a call to ma97_factor or ma97_factor_solve. An option is available to perform a partial solution.
- ma97_finalise should be called after all other calls are complete for a problem (including after an error return that does not allow the computation to continue). It frees memory allocated by the package.

In addition, the following routines may be called:

- ma97_free_akeep and ma97_free_fkeep may be called to free memory pointed to by akeep and fkeep respectively when a call to ma97_finalise is not appropriate (for example, if a further factorization is to be performed for a matrix with the same sparsity pattern).
- ma97_enquire_posdef may be called in the positive-definite case to obtain the pivots used.
- ma97_enquire_indef may be called in the indefinite case to obtain the pivot sequence used by the factorization and the entries of D^{-1}.

- ma97_alter may be called in the indefinite case to alter the entries of D^{-1}. Note that this means that $PLD(PL)^{\dagger}$ is no longer a factorization of A.
- ma97_solve_fredholm is an alternative solve routine that may be called in the indefinite case when the matrix A is found to be singular. It computes the same solution X as ma97_solve or the computed solution X_i corresponding to the $i-$th right-hand side B_i satisfies either $AX_i = B_i$ or $AX_i = 0$ and ${X_i}^{\dagger}B_i \neq 0$.
- ma97_lmultiply may be called to calculate a matrix-vector or matrix-matrix product with $S^{-1}PL$ or $(S^{-1}PL)^{\dagger}$.
- ma97_sparse_fwd_solve uses the computed factors generated by ma97_factor or ma97_factor_solve to solve the triangular system $PLX = SB$ for a single sparse right-hand side B. Multiple calls to ma97_sparse_fwd_solve may follow a call to ma97_factor or ma97_factor_solve.

2.3 OpenMP

OpenMP is used by HSL_MA97 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.

2.4 Achieving bit-compatibility

Care has been taken to allow bit-compatibility to be achieved using this solver. However, testing has revealed that this feature is dependant on the BLAS library used.

In tests it was found that bit-compatibility was impossible to achieve with the GotoBLAS. For the Intel MKL, bit-compatibility can be achieved by setting control.solve_blas3 to evaluating to true (using dgemv rather than dgemm during the backwards solve seems to trigger some form of bug). No problems were encountered using the ACML or ATLAS BLAS libraries.

2.5 The derived data types

For each problem, the user must employ the structures defined in the header file to declare scalars of the types ma97_control and ma97_info, and void * pointers for akeep, and fkeep. The following pseudo-code illustrates this.

```
#include "hsl_ma97d.h"
...
struct ma97_control control;
struct ma97_info info;
void *akeep, *fkeep;
...
```

The members of ma97_control and ma97_info are explained in Sections 2.8.1 and 2.8.2. The void * pointers are used to pass data between the subroutines of the package and must not be altered by the user.

2.6 METIS

The HSL_MA97 package optionally uses the METIS graph partitioning library available from the University of Minnesota website. If METIS is not available, the user must link with the supplied dummy subroutine METIS_NodeND. In this case, the METIS ordering option will not be available to the user and, if selected, ma97_analyse and ma97_analyse_coord will return with an error.

Note that if HSL_MA97 is to be run in parallel, it is recommended that either MeTiS is used or the user supplies an elimination computed using a nested dissection-based algorithm.

Important: At present, HSL_MA97 only supports MeTiS version 4, not the latest version 5 releases.

2.7 Argument lists and calling sequences

2.7.1 Optional arguments

We use square brackets [ ] to indicate OPTIONAL arguments. In each call, optional arguments follow the argument info. Since we reserve the right to add additional optional arguments in future releases of the code, **we strongly recommend that all optional arguments be called by keyword, not by position**.

2.7.2 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.

2.7.3 The default setting subroutine

Default values for members of the ma97_control structure may be set by a call to ma97_default_control.

```
void ma97_default_control(struct ma97_control *control)
```

control has its members set to their default values, as described in Section 2.8.1.

2.7.4 To analyse the sparsity pattern and prepare for the factorization: CSC format

If the matrix data is held in compressed sparse column (CSC) format, the analyse phase optionally checks the user's data for out-of-range and duplicate entries. Only the lower triangular part of the matrix A is required; any entries in the upper triangular part are regarded as out of range. Entries on the diagonal that are zero do not need to be entered explicitly. If checking is carried out, the cleaned matrix data (duplicates are summed during the factorization and out-of-range entries discarded) is held within memory pointed to by akeep and the user data ptr and row is not required by any of the remaining subroutines in the package. If the data is not checked, ptr and row must be passed unchanged to the factorization routines. Note that in this case, the presence of out-of-range or duplicates may cause this routine or any of the other routines in the package to fail in an unpredictable way.

A call of the following form should be made:

```
void ma97_analyse(int check, int n, const int ptr[], const int row[], pkgtype *val,
      void **akeep, const struct ma97_control *control, struct ma97_info *info,
      int order[])
```

check If check!=0 (i.e. evaluates to true) the matrix data is checked for errors and the cleaned matrix (duplicates are summed and out-of-range entries discarded) is stored in akeep. Otherwise, if check==0 (i.e. evaluates to false), no checking of the matrix data is carried out and ptr and row must be passed unchanged to the factorization routines.

n must hold the order of A. **Restriction:** n≥0.

ptr is a INTENT(IN) rank-1 array of size n+1. ptr[j] must be set by the user so that ptr[j] is the position in row of the first entry in column j and ptr[n] must be set to the number of matrix entries being input by the user.

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 (within each column, the row indices may be in arbitrary order). If check==0 (false), row must contain no duplicates or out-of-range entries (including no entries in the upper triangular part).

val may be NULL. If it is not NULL, it must be a rank-1 array. and val(k) must hold the value of the entry in row(k). val must not be NULL if a matching-based elimination ordering is required (control%ordering=7 or 8).

akeep 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, one of the subroutines ma97_free_akeep or ma97_finalise must be used to clean up and deallocate this memory after the symbolic factorization is no longer required.

control is used to control the actions of the package, see Section 2.8.1.

info is used to return information about the execution of the package, as explained in Section 2.8.2.

order may be NULL, otherwise it is a rank-1 array of size n. If control.ordering is set to 0, order must not be NULL and order[i] must hold the position of variable i in the elimination order. If control.ordering is 1, 2, 3, 4 or 5 then an ordering is computed by the analyse phase. If order is not NULL, on exit it contains the elimination order that ma97_factor or ma97_factor_solve will be given (it is passed to these routines as part of akeep); this order may give slightly more fill-in than the user-supplied order and, in the indefinite case, may be modified by ma97_factor or ma97_factor_solve to maintain numerical stability.

2.7.5 To analyse the sparsity pattern and prepare for the factorization: coordinate format

If the matrix data is held in coordinate format, entries in the upper and/or lower triangular part of A may be input using a call of the following form:

```
void ma97_analyse_coord(int n, int ne, const int row[], const int col[],
      const pkgtype *val, void **akeep, const struct ma97_control *control,
      struct ma97_info *info, int order[])
```

n must hold the order of A. **Restriction:** n≥0.

ne must hold the number of matrix entries being input by the user. **Restriction:** ne≥1.

row and col are rank-1 arrays of size ne. Each diagonal entry a_{ii} of A must be represented by row[k]=i and col[k]=i and each pair of off-diagonal entries a_{ij} and a_{ji} must be represented by row[k]=i and col[k]=j or by row[k]=j and col[k]=i. Duplicated entries are summed and out-of-range entries are discarded.

val may be NULL. If it is not NULL, it must be a rank-1 array and val(k) must hold the value of the entry in row(k) and col(k). val must not be NULL if a matching-based elimination ordering is required (control%ordering=7 or 8).

akeep, control, info, order: see Section 2.7.4.

2.7.6 To factorize the matrix and optionally solve AX = B

To factorize the matrix, a call of the following form should be made:

```
void ma97_factor(int matrix_type, const int ptr[], const int row[],
      const pkgtype val[], const void **akeep, void **fkeep,
      const struct ma97_control *control, struct ma97_info, double scale[])
```

If the user wishes to solve at the same time as factorizing the matrix, a call of the following form should be made

```
void ma97_factor_solve(int matrix_type, const int ptr[], const int row[],
      const pkgtype val[], int nrhs, pkgtype x[], int ldx, const void **akeep,
      void **fkeep, const struct ma97_control *control, struct ma97_info,
      double scale[])
```

matrix_type specifies the type of matrix to be factorized. It must be set as follows:

- 3 if A is real, symmetric positive definite
- 4 if A is real, symmetric indefinite
- -3 if A is Hermitian, positive definite
- -4 if A is Hermitian, indefinite
- -5 if A is complex, symmetric indefinite

Restriction: matrix_type = 3,4 (real case), matrix_type = -3,-4,-5 (complex case).

ptr and row may be NULL, otherwise they are rank-1 arrays. They are only accessed, and must not be NULL, if ma97_analyse was called with check set to .false.. In this case, they must be unchanged since that call.

val is a rank-1 array. If ma97_analyse was called, val[k] must hold the value of the entry in row[k]. Otherwise, if ma97_analyse_coord was called, val[k] must hold the value of the entry in row[k] and col[k].

akeep must be unchanged since the call to ma97_analyse or ma97_analyse_coord.

control, info: see Section 2.7.4.

fkeep 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, one of the subroutines ma97_free_fkeep or ma97_finalise must be used to clean up and deallocate this memory after the numerical factorization is no longer required. On the first call to ma97_factor or ma97_factor_solve, *fkeep must be set to NULL. On subsequent calls, *fkeep may contain a value set by a previous call to ma97_factor or ma97_factor_solve, in which case that factorization will be overwritten.

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

x is a rank-2 array with size x[nrhs][ldx]. It must be set so that x[j][i] holds the component of the right-hand side for variable i to the jth system. On exit, x[j][i] holds the solution for variable i to the jth system.

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≥n.

scale may be NULL, otherwise it is a rank-1 array of size n. If control.scaling≤0 and scale is NULL, no scaling is performed; if control.scaling≤0 and scale is not NULL, it must contain the diagonal entries of the scaling matrix S and is unchanged on exit. If control.scaling>0, scaling is performed and if scale is not NULL, on exit it contains the diagonal entries of the scaling matrix S.

2.7.7 To solve linear systems using the computed factors

After the call to ma97_factor (or ma97_factor_solve), one or more calls of the following form may be made to solve $AX = B$. Both partial and full solutions are available depending on the value of job.

```
void ma97_solve(int job, int nrhs, pkgtype x[], int ldx, void **akeep, void **fkeep,
      const struct ma97_control *control, struct ma97_info *info)
```

job specifies the system to be solved. In the positive-definite case, the Cholesky factorization that has been computed may be expressed in the form

$$SAS = (PL)(PL)^\dagger$$

where P is a permutation matrix and L is lower triangular. In the indefinite case, the factorization that has been computed may be expressed in the form

$$SAS = (PL)D(PL)^\dagger$$

where P is a permutation matrix, L is unit lower triangular, and D is block diagonal with blocks of order 1 and 2. S is a diagonal scaling matrix (S is equal to the identity, if control.scaling==0 and scale is not present on the last call to ma97_factor or ma97_factor_solve). A partial solution may be computed by setting job to have one of the following values:

- 0 for solving $AX = B$
- 1 for solving $PLX = SB$
- 2 for solving $DX = B$ (indefinite case only)
- 3 for solving $(PL)^\dagger S^{-1}X = B$
- 4 for solving $D(PL)^\dagger S^{-1}X = B$ (indefinite case only)

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

nrhs, x, ldx, akeep: see Section 2.7.6.

fkeep must be unchanged since the last call to ma97_factor or ma97_factor_solve.

control, info: see Section 2.7.4.

2.7.8 The finalisation and free subroutines

Once all other calls are complete for a problem or after an error return that does not allow the computation to continue, a call should be made to free memory allocated by HSL_MA97 and associated with the structures akeep and/or fkeep using calls to ma97_free_akeep and ma97_free_fkeep respectively.

The ma97_finalise call is provided as a convenient shortcut to call both ma97_free_akeep and ma97_free_fkeep.

```
void ma97_free_akeep(void **akeep)
void ma97_free_fkeep(void **fkeep)
void ma97_finalise(void **akeep, void **fkeep)
```

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

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

2.7.9 To obtain information on the factorization (positive-definite case)

After a successful call to ma97_factor or to ma97_factor_solve with matrix_type=3 or -3 and prior to a call to ma97_finalise, information on the pivots may be obtained using a call of the form

```
void ma97_enquire_posdef(const void **akeep, const void **fkeep,
      const struct ma97_control *control, struct ma97_info *info, double d[])
```

akeep, fkeep: see Section 2.7.7.

control, info: see Section 2.7.4.

d is a rank-1 array of size n. The i-th pivot will be placed in d[i], i = 0,2,...,n-1.

2.7.10 To obtain information on the factorization (indefinite case)

After a successful call to ma97_factor or to ma97_factor_solve with matrix_type=4, -4 or -5 and prior to a call to ma97_finalise, information on the pivot sequence and the matrix D^{-1} may be obtained using a call of the form

```
void ma97_enquire_indef(const void **akeep, const void **fkeep,
      const struct ma97_control *control, struct ma97_info *info,
      int piv_order[], pkgtype d[])
```

akeep, fkeep: see Section 2.7.7.

control, info: see Section 2.7.4.

piv_order may be NULL, otherwise it is a rank-1 array of size n. If not NULL, then if i is used to index a variable, its position in the pivot sequence will be placed in piv_order[i], with its sign negative if it is part of a 2×2 pivot.

d may be NULL, otherwise it is a rank-2 array shape d[n][2]. If not NULL, the diagonal entries of D^{-1} will be placed in d[i][0], i = 0,1,...,n-1, the off-diagonal entries of D^{-1} will be placed in d[i][1], i = 0,1,...,n-2, and d[n-1][2] will be set to zero.

2.7.11 To alter D^{-1}

After a successful call to ma97_factor or to ma97_factor_solve with matrix_type=4, -4 or -5 and prior to a call to ma97_finalise, the matrix D^{-1} may be altered using a call of the form

```
void ma97_alter(const pkgtype d[], const void **akeep, void **fkeep,
      const struct ma97_control *control, struct ma97_info *info)
```

d is a rank-2 array of shape d[n][2]. The diagonal entries of D^{-1} will be altered to d[i][0], i = 0,1,...,n-1, and the off-diagonal entries will be altered to d[i][1], i = 0,1,...,n-2 (and the $PLD(PL)^{\dagger}$ factorization of A will no longer be available).

akeep, fkeep: see Section 2.7.7.

control, info: see Section 2.7.4.

2.7.12 To solve linear systems in the indefinite singular case

In the indefinite case, after the call to ma97_factor or ma97_factor_solve, one or more calls of the following form may be made. If A is non-singular, the computed solution is the same as that obtained using ma97_solve but, if A is singular, the user may request that the computed X_i corresponding to the $i-$th right-hand side B_i will satisfy either $AX_i = B_i$ or $AX_i = 0$ and $X_i^\dagger B_i \neq 0$ (Fredholm alternative).

```
void ma97_solve_fredholm(int nrhs, int flag_out[], pkgtype x[], int ldx,
      void **akeep, void **fkeep, const struct ma97_control *control,
      struct ma97_info *info);
```

nrhs, ldx, akeep, fkeep, control, info: see Section 2.7.7.

flag_out is a rank-1 array of size nrhs. On exit, flag_out(j) is set to 1 (i.e. true) if the $j-$th system is consistent and to 0 (i.e. false) otherwise.

x is a rank-2 array of size x[2*nrhs][ldx]. It must be set so that, for j $= 0, 1, ...,$ nrhs $- 1$, x[j][i] holds the component of the right-hand side for variable i to the jth system. On exit, x[1:nrhs][1:n] holds the same solution as is returned by ma97_solve and, if flag_out[j]=0 (i.e. false), x[nrhs+j][1:n] holds the Fredholm alternative solution for the $j-$th system.

2.7.13 To form a matrix-vector or matrix-matrix product with $S^{-1}PL$ or $(S^{-1}PL)^\dagger$

In the indefinite case, after the call to ma97_factor, one or more calls of the following form may be made to calculate $Y = S^{-1}PLX$ or $Y = (S^{-1}PL)^\dagger X$.

```
void ma97_lmultiply(int trans, int k, const pkgtype x[], int ldx,
      pkgtype y[], int ldy, void **akeep, void **fkeep,
      const struct ma97_control *control, struct ma97_info *info);
```

trans specifies the operation to perform. If trans!=0 (i.e. evaluates to .true.), the operation $Y = (S^{-1}PL)^\dagger X$ is performed. Otherwise, if trans==0 (i.e. false), the operation $Y = S^{-1}PLX$ is performed.

k holds the number columns in the matrices X and Y. **Restriction:** k$\geq$1.

x is a rank-2 array of shape x[k][ldx]. It must be set to contain the matrix X.

ldx must be set to the first extent of the array x. **Restriction:** ldx$\geq$n.

y is a rank-2 array of shape y[k][ldy]. On exit, it will be set to the requested matrix-matrix product Y.

ldy must be set to the first extent of the array y. **Restriction:** ldy$\geq$n.

akeep, fkeep, control, info: see Section 2.7.7.

2.7.14 To solve $PLX = SB$ for sparse B

After the call to ma97_factor (or ma97_factor_solve), one or more calls of the following form may be made to solve $PLX = SB$ for a **single sparse right-hand side**

```
void ma97_sparse_fwd_solve(int nbi, const int bindex[], const pkgtype b[],
     const int order[], int *nxi, int xindex[], pkgtype x[], void **akeep,
     void **fkeep, const struct ma97_control *control, struct ma97_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 at least nbi. The first nbi entries must hold the indices of the nonzero entries in the right-hand side.

b is a rank-1 array of size 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 n. It must be unchanged since the call to ma97_analyse.

nxi holds, on exit, the number of nonzero entries in the solution.

xindex is a of size nxi (that is at most n). On exit, the first nxi entries hold the indices of the nonzero entries in the solution.

x is a rank-1 array of size n. On entry, it must be set by the user to zero. On exit, if xindex[i]=k, x[k] holds the k-th nonzero component of the solution; all other entries of x are zero.

akeep, fkeep, control, info: see Section 2.7.7.

2.8 The derived types

2.8.1 The derived data type for holding control parameters

The derived data type ma97_control is used to hold controlling data. The members, which may be given default values through a call to ma97_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 print_level is used to controls the level of printing. The different levels are:

- < 0 No printing.
- $= 0$ Error and warning messages only.
- $= 1$ As 0, plus basic diagnostic printing.
- > 1 As 1, plus some additional diagnostic printing.

The default is print_level=0.

int unit_diagnostics holds the Fortran unit number for diagnostic printing. If unit_diagnostics< 0, printing is suppressed. The default is unit_diagnostics=6.

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.

int unit_warning 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 ma97_analyse **and** ma97_analyse_coord

int ordering controls the ordering method used. If set to 0, the user must supply an elimination order in order; otherwise an elimination order will be computed by ma97_analyse or ma97_analyse_coord. The options are:

- 0 User-supplied ordering is used.
- 1 An approximate minimum degree (AMD) ordering is used.
- 2 A minimum degree ordering is used.
- 3 METIS ordering with default settings is used. Note that the user needs to supply the METIS library. If METIS is not supplied and this option is requested, the routine will return immediately with an error.
- 4 MA47 ordering for indefinite matrices is used.
- 5 A heuristic choice is made between AMD and METIS orderings. assuming the factorization is to be run in **parallel**. If METIS is not available, AMD is used. The actual ordering chosen is indicated by the value of info.ordering on return from ma97_analyse or ma97_analyse_coord.
- 6 As 5 but assuming the factorization is to be run in **serial**.
- 7 A matching-based elimination ordering is computed using HSL_MC80. AMD is used on the compressed matrix. This option should only be chosen for indefinite systems. A scaling is also computed that may be passed to ma97_factor or ma97_factor_solve (see control.scaling below).
- 8 As 7 but METIS is used on the compressed matrix.

The default is order=5. **Restriction:** order=0, 1,..., 8.

int nemin controls node amalgamation. Two neighbours in the elimination tree are merged if they both involve fewer than nemin eliminations. The default is nemin=8. The default is used if nemin<1.

Controls used by ma97_factor

int scaling controls the use of scaling. The available options are:

- $\leq$ 0 No scaling (scale argument is NULL), or user-supplied scaling (scale is not NULL).
- = 1 Generate a scaling using a weighted bipartite matching using the package MC64.
- = 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.
- = 3 A matching-based ordering has been generated during the analyse phase using control.ordering = 7 or 8. Use the scaling generated as a side-effect of this process. The scaling will be the same as that generated with control.scaling = 1 if matrix values have not changed. This option will generate an error if a matching-based ordering was not used.
- $\geq$ 4 Generate a scaling by minimising the absolute sum of log values in the scaled matrix using the package MC30.

The default is scaling=0.

Controls used by ma97_factor **with** matrix_type=4, -4 **or** -5 **(*A* indefinite)**

int action controls behaviour when a matrix is singular. If the matrix is found to be singular (has rank less than the number of non-empty rows), the computation continues after issuing a warning if action evaluates to true or terminates with an error otherwise The default is action=1 (true).

long factor_min controls the use of parallelism within the factorization. Parallelism is only used if the predicted number of floating point operations (info.num_flops) is greater than or equal to factor_min. The default is factor_min= 2×10^7.

double multiplier controls memory usage. To allow for delayed pivots, the arrays that store the factors and associated index lists are allocated to accommodate a matrix of order $s \times max(1,$multiplier$)$, where s is the expected size of the factors without delays. If, during the factorization, this space is found to be too small, additional memory will be allocated dynamically. The default is multiplier= 1.1.

double small is a scalar of type. Any pivot whose modulus is less than small is treated as zero. The default in the double and double complex versions is small= 10^{-20}, and in the single and single complex versions is small= 10^{-12}.

double u holds the relative pivot tolerance u. The default in the double and double complex versions is u=0.01, and in the single and single complex versions is u=0.1. Values outside the range $[0, 0.5]$ are treated as the default.

Controls used by ma97_solve **and/or** ma97_solve_fredholm

double consist_tol is a scalar of type REAL that holds the tolerance used to determine if a system is inconsistent in ma97_solve_fredholm. The default is consist_tol=epsilon(), the smallest quantity for the package type such that $1+\varepsilon \neq 1$.

int solve_mf controls the algorithm used for the solve. If solve_mf evaluates to true, a multifronal-style forward solve is used. Otherwise a supernodal-style solve is used. The supernodal solve does not use parallelism during the forward solve and typically performs best on small problems, with the multifrontal solve performing best on large problems. If the user wishes to make more than one call to ma97_solve, we recommend comparing the solve time with both schemes. The default value is solve_mf=0 (false).

int solve_blas3 controls whether level 2 (solve_blas3 evaluates to false) or level 3 (solve_blas3 evaluates to true) BLAS are used in the case of a single right-hand side solution. On larger problems the level 3 BLAS can often out perform the level 2 BLAS. The default is solve_blas3=0 (false).

long solve_min controls the use of parallelism within the solve. Parallelism is only used if the number of entries in L (info.num_factor) is greater than or equal to solve_min. The default is solve_min=100000.

2.8.2 The derived data type for holding information

The derived data type ma97_info is used to hold parameters that give information about the progress and needs of the algorithm. The members of struct ma97_info (in alphabetical order) are:

int flag gives the exit status of the algorithm (details in Section 2.9).

int flag68 holds, on exit from ma97_analyse or ma97_analyse_coord, the error flag from HSL_MC68.

int flag77 holds, on exit from ma97_factor or ma97_factor_solve, the error flag from MC77.

int matrix_dup holds, on exit from ma97_analyse with check set to .true. or from ma97_analyse_coord, the number of duplicate entries that were found and summed.

int matrix_missing_diag holds, on exit from ma97_analyse with check!=0 (true) or from ma97_analyse_coord, the number of diagonal entries without an explicitly provided value.

int matrix_outrange holds, on exit from ma97_analyse with check!=0 (true) or from ma97_analyse_coord, the number of out-of-range entries that were found and discarded.

int matrix_rank holds, on exit from ma97_factor or ma97_factor_solve, the computed rank of the factorized matrix.

int maxdepth holds, on exit from ma97_analyse or ma97_analyse_coord, the maximum depth of the assembly tree.

int maxfront holds, on exit from ma97_analyse or ma97_analyse_coord, the maximum front size in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from ma97_factor or ma97_factor_solve, it holds the maximum front size.

int num_delay holds, on exit from ma97_factor or ma97_factor_solve, the number of eliminations that were delayed, that is, the total number of fully-summed variables that were passed to the father node because of stability considerations. If a variable is passed further up the tree, it will be counted again.

long num_factor holds, on exit from ma97_analyse or ma97_analyse_coord, the number of entries that will be in the factor L in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from ma97_factor or ma97_factor_solve, it holds the actual number of entries in the factor L. In the indefinite case, 2n entries of D^{-1} are also held.

long num_flops holds, on exit from ma97_analyse or ma97_analyse_coord, the number of floating-point operations that will be needed to perform the factorization in the positive-definite case (or in the indefinite case with the same pivot sequence). On exit from ma97_factor or ma97_factor_solve, it holds the number of floating-point operations performed.

int num_neg holds, on exit from ma97_factor or ma97_factor_solve, the number of negative eigenvalues of the matrix D.

int num_sup holds, on exit from ma97_analyse or ma97_analyse_coord, the number of supernodes in the problem.

int num_two holds, on exit from ma97_factor and ma97_factor_solve, the number of 2×2 pivots used by the factorization, that is, the number of 2×2 blocks in D.

int ordering indicates, on exit from ma97_analyse or ma97_analyse_coord, the ordering method chosen. Values have the same meanings as in the context of control.ordering.

int stat holds, in the event of an allocation or deallocation error, the Fortran stat parameter if it is available (and is set to 0 otherwise).

2.9 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 An error has been made in the sequence of calls (this includes calling a subroutine after an error that cannot be recovered from).

−2 Returned by ma97_analyse and ma97_analyse_coord if n<0. Also returned by ma97_analyse_coord if ne<1.

−3 Returned by ma97_analyse if there is an error in ptr.

−4 Returned by ma97_analyse if all the variable indices in one or more columns are out-of-range. Also returned by ma97_analyse_coord if all entries are out-of-range.

−5 Returned by ma97_factor and ma97_factor_solve if matrix_type is out-of-range. The user may reset matrix_type and recall ma97_factor or ma97_factor_solve.

−6 Returned by ma97_factor and ma97_factor_solve if matrix_type=-3 or -4 (A is Hermitian) and one or more of the diagonal entries is not real.

−7 Returned by ma97_factor and ma97_factor_solve if matrix_type=4, -4 or -5 and control.action == 0 (false) when the matrix is found to be singular. The user may reset the matrix values in val and recall ma97_factor or ma97_factor_solve.

−8 Returned by ma97_factor and ma97_factor_solve if matrix_type=3 or -3 and the matrix is found to be not positive definite. This may be because the scaling MC64 found the matrix to be singular. The user may reset the matrix values in val and recall ma97_factor or ma97_factor_solve.

−9 Returned by ma97_factor if IEEE infinities found in the reduced matrix, probably caused by control.small or control.u having too small a value. The user may reset control.small and/or control.u or may reset the matrix values in val and recall ma97_factor or ma97_factor_solve.

−10 Returned by ma97_factor and ma97_factor_solve if ma97_analyse was called with check==0 (false) but ptr and/or row are NULL.

−11 Returned by ma97_analyse and ma97_analyse_coord if control.ordering is out-of-range, or control.ordering=0 and the user has either failed to provide an elimination order or an error has been found in the user-supplied elimination order (held in order).

−12 Returned by ma97_factor_solve and ma97_solve if there is an error in the size of array x (that is, ldx<n or nrhs<1). The user may reset ldx and/or nrhs and recall ma97_factor_solve or ma97_solve.

−13 Returned by ma97_solve if job is out-of-range. The user may reset job and recall ma97_solve.

−14 Returned by ma97_enquire_posdef if matrix_type=4, -4 or -5 on the last call to ma97_factor or ma97_factor_solve.

−15 Returned by ma97_enquire_indef if matrix_type=3 or -3 on the last call to ma97_factor or ma97_factor_solve.

−16 Allocation error. If available, the stat parameter is returned in info.stat. The user may wish to try the more memory conservative codes HSL_MA86 or HSL_MA77.

−17 Returned by ma97_analyse and ma97_analyse_coord if METIS ordering was requested but METIS is not available.

−18 Returned by ma97_analyse and ma97_analyse_coord if there is an unexpected error from HSL_MC68. The user is advised to ensure that if ma97_analyse was called, check!=0 (true). Further information may be provided by info.flag68.

−19 Returned by ma97_factor and ma97_factor_solve if there is an unexpected error from MC77. The user is advised to ensure that if ma97_analyse was called, check!=0 (true). Further information may be provided by info.flag77.

−20 Returned by ma97_analyse and ma97_analyse_coord if control.ordering=7 or 8 but val is NULL.

−21 Returned by ma97_factorise if control.scaling≥3 but a matching based ordering was not used during the call to ma97_analyse or ma97_analyse_coord (i.e. was called with control.ordering≠7 or 8)

−22 Returned by ma97_sparse_fwd_solve if nbi is out of range. The user may reset nbi and recall.

A positive value of info.flag is used to warn the user that the input matrix data may be faulty or that the subroutine cannot guarantee the solution obtained. Possible values are:

+1 Returned by ma97_analyse and ma97_analyse_coord if out-of-range variable indices found. Any such entries are ignored and the computation continues. info.matrix_outrange is set to the number of such entries.

+2 Returned by ma97_analyse and ma97_analyse_coord if duplicated indices found. Duplicates are recorded and the corresponding entries are summed. info.matrix_dup is set to the number of such entries.

+3 Returned by ma97_analyse and ma97_analyse_coord if both out-of-range and duplicated variable indices found.

+4 Returned by ma97_analyse and ma97_analyse_coord if one and more diagonal entries of A is missing.

+5 Returned by ma97_analyse and ma97_analyse_coord if one and more diagonal entries of A is missing and out-of-range and/or duplicated variable indices have been found.

+6 Returned by ma97_analyse and ma97_analyse_coord if A is found be (structurally) singular. This will overwrite any of the above warnings.

+7 Returned by ma97_factor and ma97_factor_solve if control%action is set to .true. and the matrix is found to be (structurally or numerically) singular.

+8 Returned by ma97_factor and ma97_factor_solve if a matching-based ordering was used (i.e. control%ordering=7 or 8) but the associated scaling was not (i.e. control%scaling< 3).

3 GENERAL INFORMATION

Workspace: Provided automatically by the module.

Other routines called directly: MC30, HSL_MC34, HSL_MC64, HSL_MC68 (optionally using METIS), HSL_MC69, MC77, HSL_MC78, HSL_MC80, _axpy, _gemm, _gemv, _nrm2, _potrf, _swap, _syrk, _trmv, _trmm, _trsm, _trsv.

Input/output: Output is provided under the control of control.print_level. In the event of an error, diagnostic messages are printed. The output units for these messages are respectively controlled by control.unit_err, control.unit_warning and control.unit_diagnostics (see Section 2.8.1).

Restrictions: n≥0; ne≥1; nrhs≥1; ldx≥n;
control.ordering=0, 1, 2, 3, 4, 5;
matrix_type = 3, 4, -3, -4, or -5;
job = 1, 2, 3, or 4.

Portability: Fortran 2003 subset (F95 + TR15581 + C interoperability). OpenMP 3.0 or above for (optional) parallel usage.

Changes from Version 1
Version 2 offers the option of computing a matching-based elimination ordering. This requires the user to supply the numerical values of the matrix on the call to the analyse phase so the val argument was added. If the user wishes to factorize another matrix with the same sparsity pattern but different numerical values, it may be necessary to recall the analyse phase. A matching-based elimination ordering may be a good choice for tough indefinite systems.

4 METHOD

ma97_analyse and ma97_analyse_coord
If check!=0 (true) on the call to ma97_analyse or if ma97_analyse_coord is called, the HSL package HSL_MC69 is used to check the matrix data. The cleaned matrix data (duplicates are summed and out-of-range indices discarded) is stored in akeep. The use of checking is optional on a call to ma97_analyse as it incurs both time and memory overheads. Some form of checking is recommended since the behaviour of the other routines in the package is unpredictable if duplicates and/or out-of-range variable indices are entered. Calling the HSL_MC69 routine mc69_verify offers an alternative that can be used for debugging purposes.

If the user has supplied a pivot order it is checked for errors. Otherwise, a pivot order is generated using HSL_MC68, or if a matching-based ordering is requested, HSL_MC80. The pivot order is used to construct the assembly tree using HSL_MC78.

On exit, order is set so that order[i] holds the position at which variable i is eliminated. If a user order was supplied, this order may differ, but will be equivalent in terms of fill-in to that provided.

If a matching-based ordering is requested and scale is present, on exit, scale contains scaling factors computed by MC64. These may be passed unchanged to ma97_factor and ma97_factor_solve.

ma97_factor and ma97_factor_solve
ma97_factor and ma97_factor_solve optionally compute a scaling and then perform the numerical factorization. The user must specify whether or not the matrix is positive definite. If matrix_type is set to 3 or -3, no pivoting is performed. As a result the computation will terminate with an error if a non-positive pivot is encountered.

The factorization uses the assembly tree that was set up by the analyse phase. If running on a single thread (or if there is insufficient work available to justify running in parallel), the nodes of the tree are iterated over in a post-order.

At a node, the contributions from the children relating to those columns that are fully summed at this node are first assembled. A dense partial factorization is then performed on these columns. In the positive-definite case, LAPACK's _potrf (or _herk for Hermitian matrices) is used. In the indefinite case, an algorithm based on the same pivoting algorithm as HSL_MA64 is used.

The generated element is calculated by first forming the outer product of the fully summed columns' uneliminated rows. The contributions from the children are then added, and the stack memory used by the children is freed. As this involves copying from one stacked contribution to another, two separate stacks are used to do this.

If a pivot candidate does not pass pivot tests at a given node, it is delayed to the parent node where additional eliminations may make the pivot feasible. This results in the generation of additional fill-in and floating-point operations, and may result in additional memory allocations being required.

In parallel computation, we exploit two levels of parallelism using OpenMP tasks. In tree-level parallelism, different subtrees are factorized in independent tasks. To ensure results are bit-compatible regardless of the number of threads used, the assembly order of the children is fixed at assembly time. In node-level parallelism, the operations forming the outer-product in both the dense factorization kernel and the calculation of the generated element are broken into multiple tasks. Bit-compatibility is ensured in this case by using a data-parallel approach so each individual sum is effectively calculated in serial.

If ma97_factor_solve is called, the forward substitutions are performed as the factor entries are generated. Once the factorization is complete, the back substitutions are performed by an internal call to ma97_solve with job = 3 (positive-definite case) or job = 4 (indefinite case).

ma97_solve
Having checked the user's data, ma97_solve performs a forward substitution followed by a combined diagonal solve and back substitution (unless only one of these is requested).

In a supernodal solve updates are done directly into the right-hand side vector and do not readily admit bit-compatible parallelism in the forward substitution. In the multifrontal solve updates are passed up the tree utilising a stack. This allows parallelism to be implemented but can be slower than the supernodal solve on small problems. Regardless of whether a supernodal or multifrontal solve is chosen the same backwards solve is used that works directly on the right-hand side vectors. Due to the differing data dependencies from the forward substitution a bit-compatible parallel solve is possible.

The matrix factor 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. (2011). HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems. RAL Technical Report. RAL-TR-2011-024.

5 EXAMPLE OF USE

5.1 First example: sparse column entry

Suppose we wish to factorize the matrix

$$A = \begin{pmatrix} 2. & 1. & & & \\ 1. & 4. & 1. & & 1. \\ & 1. & 3. & 2. & \\ & & 2. & 0. & \\ & 1. & & & 2. \end{pmatrix}$$

and then solve for the right-hand side

$$B = \begin{pmatrix} 4. \\ 12. \\ 10. \\ 4. \\ 4. \end{pmatrix}$$

.

The following code may be used. Note that, in this example, it would be more efficient to pass the right-hand side to ma97_factor_solve; here our aim is to illustrate calling ma97_solve after ma97_factor.

```
/* hsl_ma97ds.c */
/* Simple code to illustrate entry by columns to hsl_ma97 */

#include <stdio.h>
#include <stdlib.h>
#include "hsl_ma97d.h"

int main(void) {
   typedef double pkgtype;

   void *akeep, *fkeep;
   struct ma97_control control;
   struct ma97_info info;
```

```
int *ptr, *row, *piv_order;
pkgtype *val, *x;

int i,matrix_type,n,ne,check;

/* Read in the order n of the matrix and number of entries in lwr triangle */
scanf("%d %d", &n, &ne);

/* Allocate arrays for matrix data and arrays for hsl_ma97 */
ptr = (int *) malloc((n+1)*sizeof(int));
row = (int *) malloc(ne*sizeof(int));
val = (pkgtype *) malloc(ne*sizeof(pkgtype));
x = (pkgtype *) malloc(n*sizeof(pkgtype));
piv_order = (int *) malloc(n*sizeof(int));

for(i=0; i<n+1; i++) scanf("%d", &(ptr[i]));
for(i=0; i<ne; i++) scanf("%d", &(row[i]));
for(i=0; i<ne; i++) scanf("%lf", &(val[i]));

ma97_default_control(&control);

/* Perform analyse and factorise with data checking */
check = 1; /* true */
ma97_analyse(check,n,ptr,row,NULL,&akeep,&control,&info,NULL);
if (info.flag < 0) {
   ma97_free_akeep(&akeep);
   free(ptr); free(row); free(val); free(x); free(piv_order);
   return 1;
}
matrix_type = 4;
fkeep = NULL; /* important that this is initialised to NULL on first call */
ma97_factor(matrix_type,ptr,row,val,&akeep,&fkeep,&control,&info,NULL);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(piv_order);
   return 1;
}

/* Read in the right-hand side. */
for(i=0; i<n; i++) scanf("%lf", &(x[i]));

/* Solve */
ma97_solve(0,1,x,n,&akeep,&fkeep,&control,&info);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(piv_order);
   return 1;
}
```

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

   /* Determine the pivot order used */
   printf("\nPivot order:");
   ma97_enquire_indef(&akeep,&fkeep,&control,&info,piv_order,NULL);
   for(i=0; i<n; i++) printf(" %d", piv_order[i]);
   printf("\n");

   ma97_finalise(&akeep,&fkeep);

   /* Deallocate all arrays */
   free(ptr); free(row); free(val); free(x); free(piv_order);

   return 0;
}
```

with the following data:

```
  5  8
  0  2  5  7  7  8
  0  1  1  4  2  3  2  4
  2. 1. 4. 1. 1. 2. 3. 2.
  4. 12. 10. 4. 4.
```

This produces the following output:

```
The computed solution is:
 1.000000 2.000000 2.000000 1.000000 1.000000
Pivot order: 2 3 1 4 0
```

5.2 Second example: coordinate entry, refactorization, factor_solve

Suppose we wish to factorize the matrix

$$A = \begin{pmatrix} 1. & -3. & & 1. & \\ -3. & -5. & 6. & & 4. \\ & 6. & & 2. & \\ 1. & & 2. & 3. & \\ & 4. & & & 1. \end{pmatrix}$$

and then solve for the right-hand sides

$$B = \begin{pmatrix} -1. & -5. \\ 25. & -40. \\ 20. & 8. \\ 19. & -8. \\ 13. & -1. \end{pmatrix}$$

. Suppose we then wish to solve the following system with the same pattern in a single call:

$$\begin{pmatrix} 2. & 1. & & 7. & \\ 1. & 1. & 8. & & 2. \\ & 8. & & 1. & \\ 7. & & 1. & 8. & \\ & 2. & & & 8. \end{pmatrix} \mathbf{x} = \begin{pmatrix} 8. \\ 89. \\ 40. \\ 37. \\ 42. \end{pmatrix}$$

The following code may be used.

```
/* hsl_ma97ds1.c */
/* Simple code to illustrate coordinate entry for hsl_ma97 */

#include <stdio.h>
#include <stdlib.h>
#include "hsl_ma97d.h"

int main(void) {
   typedef double pkgtype;

   void *akeep, *fkeep;
   struct ma97_control control;
   struct ma97_info info;

   int *row, *col;
   pkgtype *val, *x;

   int i,matrix_type,n,ne;

   /* Read in the order n of the matrix and number of entries in lwr triangle */
   scanf("%d %d", &n, &ne);

   /* Allocate arrays for matrix data and arrays for hsl_ma97 */
   row = (int *) malloc(ne*sizeof(int));
   col = (int *) malloc(ne*sizeof(int));
   val = (pkgtype *) malloc(ne*sizeof(pkgtype));
   x = (pkgtype *) malloc(2*n*sizeof(pkgtype));

   for(i=0; i<ne; i++) scanf("%d", &(row[i]));
   for(i=0; i<ne; i++) scanf("%d", &(col[i]));
   for(i=0; i<ne; i++) scanf("%lf", &(val[i]));

   ma97_default_control(&control);

   /* Perform analyse and factorise with data checking */
   ma97_analyse_coord(n,ne,row,col,NULL,&akeep,&control,&info,NULL);
   if (info.flag < 0) {
      ma97_free_akeep(&akeep);
      free(row); free(col); free(val); free(x);
      return 1;
```

```
   }
   matrix_type = 4; /* Real, symmetric indefinite */
   fkeep = NULL; /* important that this is initialised to NULL on first call */
   ma97_factor(matrix_type,NULL,NULL,val,&akeep,&fkeep,&control,&info,NULL);
   if (info.flag < 0) {
      ma97_finalise(&akeep,&fkeep);
      free(row); free(col); free(val); free(x);
      return 1;
   }

   /* Read in the right-hand sides. */
   for(i=0; i<2*n; i++) scanf("%lf", &(x[i]));

   /* Solve */
   ma97_solve(0,2,x,n,&akeep,&fkeep,&control,&info);
   if (info.flag < 0) {
      ma97_finalise(&akeep,&fkeep);
      free(row); free(col); free(val); free(x);
      return 1;
   }
   printf("\nThe computed solution is:\n");
   for(i=0; i<n; i++) printf(" %lf", x[i]);
   printf("\n");
   for(i=0; i<n; i++) printf(" %lf", x[n+i]);

   /* Read second matrix with same pattern */
   for(i=0; i<ne; i++) scanf("%lf", &(val[i]));

   /* Read another right hand side */
   for(i=0; i<n; i++) scanf("%lf", &(x[i]));

   /* Perform combined factor and solve */
   /* Note: no need to set fkeep to NULL, we will overwrite existing factors */
   ma97_factor_solve(matrix_type,NULL,NULL,val,1,x,n,&akeep,&fkeep,&control,
         &info,NULL);
   printf("\nNext solution is:\n");
   for(i=0; i<n; i++) printf(" %lf", x[i]);
   printf("\n");

   ma97_finalise(&akeep,&fkeep);

   /* Deallocate all arrays */
   free(row); free(col); free(val); free(x);

   return 0;
}
```

with the following data:

```
5 9
```

```
0   1   0   1   2   4   3   3   4
0   0   3   1   1   1   3   2   4
1. -3.  1. -5.  6.  4.  3.  2.  1.
-1.  25.  20.  19.  13.
-5. -40.   8.  -8.  -1.
2.  1.  7.  1.  8.  2.  8.  1.  8.
16.5 89.0 40.5 41.0 42.0
```

This produces the following output:

```
 Warning from ma97_analyse_coord. Warning flag =   4
 one or more diagonal entries is missing

The computed solution is:
 1.000000 2.000000 3.000000 4.000000 5.000000
 -3.000000 1.000000 -4.000000 1.000000 -5.000000
Next solution is:
 4.000000 5.000000 9.000000 0.500000 4.000000
```

Note that the warning is entirely innocous and is merely due to the absence of a non-zero in the diagonal $(3,3)$ position.

5.3 Third example: advanced features

The following example demonstrates some advanced capabilities of HSL_MA97. First the following singular system is considered,

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 2.0 & 5.0 & & \\ 3.0 & & & \\ 4.0 & & & \end{pmatrix} X = \begin{pmatrix} 7.50 & 7.50 \\ 3.00 & 3.00 \\ 0.75 & 1.00 \\ 1.00 & 1.00 \end{pmatrix},$$

where the first right-hand side is inconsistent and the second is consistent. The routine ma97_solve_fredholm is used to check consistency and produce a Fredholm alternative solution for the first right-hand side. The example then demonstrates multiplying this solution by the factor $S^{-1}PL$, and then solves the following system, that has a sparse right-hand side,

$$S^{-1}PLx = \begin{pmatrix} 0.50 \\ 1.25 \\ 0 \\ 0 \end{pmatrix}.$$

The code is as follows:

```
/* hsl_ma97ds2.c */
/* To illustrate use of advanced features of hsl_ma97 */

#include <stdio.h>
#include <stdlib.h>
#include "hsl_ma97d.h"

int main(void) {
   typedef double pkgtype;
```

```
void *akeep, *fkeep;
struct ma97_control control;
struct ma97_info info;

int *ptr, *row, *order, *flag_out, *bindex, *xindex;
pkgtype *val, *x, *y, *b;

int i,j,matrix_type,n,ne,check,nrhs,nbi,nxi;

/* Read in the order n of the matrix and number of entries in lwr triangle */
scanf("%d %d %d", &n, &ne, &nrhs);

/* Allocate arrays for matrix data and arrays for hsl_ma97 */
ptr = (int *) malloc((n+1)*sizeof(int));
row = (int *) malloc(ne*sizeof(int));
val = (pkgtype *) malloc(ne*sizeof(pkgtype));
x = (pkgtype *) malloc(2*nrhs*n*sizeof(pkgtype));
y = (pkgtype *) malloc(nrhs*n*sizeof(pkgtype));
order = (int *) malloc(n*sizeof(int));
flag_out = (int *) malloc(nrhs*sizeof(int));

for(i=0; i<n+1; i++) scanf("%d", &(ptr[i]));
for(i=0; i<ne; i++) scanf("%d", &(row[i]));
for(i=0; i<ne; i++) scanf("%lf", &(val[i]));

ma97_default_control(&control);

/* Perform analyse and factorise with data checking */
check = 1; /* true */
ma97_analyse(check,n,ptr,row,NULL,&akeep,&control,&info,order);
if (info.flag < 0) {
   ma97_free_akeep(&akeep);
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
   free(order);
   return 1;
}
matrix_type = 4; /* Real, symmetric indefinite */
fkeep = NULL; /* important that this is initialised to NULL on first call */
ma97_factor(matrix_type,ptr,row,val,&akeep,&fkeep,&control,&info,NULL);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
   free(order);
   return 1;
}

/* Read in the right-hand sides. */
for(i=0; i<n*nrhs; i++) scanf("%lf", &(x[i]));
```

```
/* Solve with Fredholm alternative if rhs is inconsistent */
ma97_solve_fredholm(nrhs,flag_out,x,n,&akeep,&fkeep,&control,&info);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
   free(order);
   return 1;
}
for(j=0; j<nrhs; j++) {
   if(flag_out[j]) {
      printf("Right-hand side %d is consistent with solution:\n", j);
      for(i=0; i<n; i++) printf(" %lf", x[i+j*n]);
   } else {
      printf("Right-hand side %d inconsistent. Ax=0, x^Tb/=0 given by:\n",
         j);
      for(i=0; i<n; i++) printf(" %lf", x[nrhs*n+i+j*n]);
   }
   printf("\n");
}

/* Form (S^{-1}PL)X */
ma97_lmultiply(0,2,x,n,y,n,&akeep,&fkeep,&control,&info);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
   free(order);
   return 1;
}
for(j=0; j<nrhs; j++) {
   printf("S^{-1}PLX_%d = ", j);
   for(i=0; i<n; i++) printf(" %lf", y[i+j*n]);
   printf("\n");
}

/* Read sparse right-hand side */
scanf("%d", &nbi);
bindex = (int *) malloc(nbi*sizeof(int));
b = (pkgtype *) malloc(n*sizeof(pkgtype));
xindex = (int *) malloc(n*sizeof(int));
for(i=0; i<nbi; i++) scanf("%d", &(bindex[i]));
for(i=0; i<nbi; i++) scanf("%lf", &(b[bindex[i]]));

/* Perform sparse fwd solve */
ma97_sparse_fwd_solve(nbi, bindex, b, order, &nxi, xindex, x, &akeep,
   &fkeep, &control, &info);
if (info.flag < 0) {
   ma97_finalise(&akeep,&fkeep);
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
```

```
      free(order); free(bindex); free(b); free(xindex);
      return 1;
   }
   printf("Sparse solution has entries:\n");
   for(i=0; i<nxi; i++) printf("%d %lf\n", xindex[i], x[xindex[i]]);

   ma97_finalise(&akeep,&fkeep);

   /* Deallocate all arrays */
   free(ptr); free(row); free(val); free(x); free(y); free(flag_out);
   free(order); free(bindex); free(b); free(xindex);

   return 0;
}
```

Used with the following data

```
4 5 2
0 4 5 5 5
0 1 2 3 1
1.0 2.0 3.0 4.0 5.0
7.50 3.00 0.75 1.00
7.50 3.00 1.00 1.00
2
1 2
0.50 1.25
```

This produces the following output:

```
 Warning from ma97_analyse. Warning flag =   4
 one or more diagonal entries is missing

 Warning from ma97_factor. Warning flag =   7
 Matrix found to be singular
Right-hand side 0 is consistent with solution:
 0.250000 0.500000 0.000000 1.562500
Right-hand side 1 inconsistent. Ax=0, x^Tb/=0 given by:
 0.000000 0.000000 1.000000 -0.750000
S^{-1}PLX_0 =  0.450000 0.500000 4.921875 6.562500
S^{-1}PLX_1 =  0.450000 0.500000 4.003125 5.337500
Sparse solution has entries:
1 0.500000
2 0.078125
0 0.050000
3 0.562500
```