Science and Technology Facilities Council

HSL_MI13

1 SUMMARY

Given a **block symmetric matrix**

$$\mathbf{K}_H = \begin{pmatrix} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{C} \end{pmatrix},$$

where **H** has n rows and columns and **A** has m rows and n columns, this package constructs **preconditioners** of the form

$$\mathbf{K}_G = \begin{pmatrix} \mathbf{G} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{C} \end{pmatrix}. \tag{1.1}$$

Here, the leading block matrix **G** is a suitably chosen approximation to **H**; it may either be prescribed **explicitly**, in which case a symmetric indefinite factorization of $\mathbf{K}_G$ will be formed using `HSL_MA57`, or **implicitly**. In the latter case, $\mathbf{K}_G$ will be ordered to the form

$$\mathbf{K}_G = \mathbf{P} \begin{pmatrix} \mathbf{G}_{11} & \mathbf{G}_{21}^T & \mathbf{A}_1^T \\ \mathbf{G}_{21} & \mathbf{G}_{22} & \mathbf{A}_2^T \\ \mathbf{A}_1 & \mathbf{A}_2 & -\mathbf{C} \end{pmatrix} \mathbf{P}^T \tag{1.2}$$

where **P** is a permutation and $\mathbf{A}_1$ is an invertible sub-block ("basis") of the columns of **A**; the selection and factorization of $\mathbf{A}_1$ uses `HSL_MA48` - any dependent rows in **A** are removed at this stage. Once the preconditioner has been constructed, solutions to the preconditioning system

$$\begin{pmatrix} \mathbf{G} & \mathbf{A}^T \\ \mathbf{A} & -\mathbf{C} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix} \tag{1.3}$$

may be computed. The essential ideas are described in detail in

H. S. Dollar, N. I. M. Gould and A. J. Wathen. "On implicit-factorization constraint preconditioners". In Large Scale Nonlinear Optimization (G. Di Pillo and M. Roma, eds.) Springer Series on Nonconvex Optimization and Its Applications, Vol. 83, Springer Verlag (2006) 61–82

and

H. S. Dollar, N. I. M. Gould, W. H. A. Schilders and A. J. Wathen "On iterative methods and implicit-factorization preconditioners for regularized saddle-point systems". SIAM Journal on Matrix Analysis and Applications, **28(1)** (2006) 170–189.

Full advantage is taken of any zero coefficients in the matrices **H**, **A** and **C**.

ATTRIBUTES — Version: 1.2.0 (10 April 2013). **Types:** Real (single,double). **Uses:** `KB07`, `MC59`, `HSL_ZD11`, `HSL_MA57`, `HSL_MA48`, `BLAS` routine `_GEMV`, `LAPACK` routines `_POTRF` and `_POTRS`. **Date:** July 2007. **Origin:** H. S. Dollar and N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 2003 subset(F95 + TR15581). **Remark:** The development of this package was supported by EPSRC grant GR/S42170.

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

Access to the package requires a `USE` statement of the form

Single precision version

```
        USE HSL_MI13_single
```

Double precision version

```
USE HSL_MI13_double
```

In HSL_MI13_single, all reals are default reals. In HSL_MI13_double, all reals are double precision reals. In both modules, all integers are default integers.

If it is required to use both modules at the same time, then the derived types ZD11_type, MI13_control_type (§2.3), MI13_inform_type (§2.4) and MI13_data_type (§2.5) and the subroutines MI13_setup, MI13_initialize, MI13_form_and_factorize, MI13_solve and MI13_terminate (§2.6) must be renamed on one of the USE statements.

The following subroutines are available to the user:

1. The subroutine MI13_initialize is used to set default values and initialize private components of data before solving one or more problems with the same sparsity and bound structure.
2. The subroutine MI13_form_and_factorize is called to form and factorize the preconditioner.
3. The subroutine MI13_solve is called to apply the preconditioner, that is to solve a linear system of the form (1.3).
4. The subroutine MI13_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by MI13_form_and_factorize at the end of the solution process.

2.2 The derived data type for holding matrices

The derived data type ZD11_TYPE is used to hold the matrices **H**, **A** and **C**. The components of ZD11_TYPE used here are:

m is a scalar component of type default INTEGER that holds the number of rows in the matrix.

n is a scalar component of type default INTEGER that holds the number of columns in the matrix.

type is a rank-one array of type default CHARACTER that is used to indicate the storage scheme used. If the dense storage scheme (see §2.2.1) is used, then the first five components of type must contain the string DENSE. For the sparse co-ordinate scheme (see §2.2.2), the first ten components of type must contain the string COORDINATE, for the sparse row-wise storage scheme (see §2.2.3), the first fourteen components of type must contain the string SPARSE_BY_ROWS, and for the diagonal storage scheme (see §2.2.4), the first eight components of type must contain the string DIAGONAL. It is also permissible to set the first four components of type to the string ZERO in the case of matrix **C** to indicate that $\mathbf{C} = 0$.

For convenience, the procedure ZD11_put may be used to allocate sufficient space and insert the required keyword into type. For example, if H is of derived type ZD11_type and we wish to use the co-ordinate storage scheme, we may simply

```
CALL ZD11_put(H%type,'COORDINATE',stat=stat)
```

See the documentation for the HSL package ZD11 for further details on the use of ZD11_put.

ne is a scalar variable of type default INTEGER that holds the number of matrix entries.

val is a rank-one allocatable array of type default REAL and dimension at least ne, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix **H** is represented as a single entry (see §2.2.1–2.2.3); the same applies to **C**. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed. If the matrix is stored using the diagonal scheme (see §2.2.4), val should be of length n, and the value of the i-th diagonal stored in val(i).

`row` is a rank-one allocatable array of type default `INTEGER` and dimension at least `ne`, that may hold the row indices of the entries. (see §2.2.2).

`col` is a rank-one allocatable array of type default `INTEGER` and dimension at least `ne`, that may hold the column indices of the entries (see §2.2.2–2.2.3).

`ptr` is a rank-one allocatable array of type default `INTEGER` and dimension at least `m + 1`, that may hold the allocatables to the first entry in each row (see §2.2.3).

Each of the input matrices **H**, **A** and **C** may be stored in a variety of input formats.

2.2.1 Dense storage format

The matrix **A** is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n*(i-1)+j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1,\ldots,m$, $j = 1,\ldots,n$. Since **H** and **C** are symmetric, only the lower triangular parts (that is the part h_{ij} for $1 \le j \le i \le n$ and c_{ij} for $1 \le j \le i \le m$) need be held. In these cases the lower triangle will be stored by rows, that is component $i*(i-1)/2+j$ of the storage array `H%val` will hold the value h_{ij} (and, by symmetry, h_{ji}) for $1 \le j \le i \le n$. Similarly component $i*(i-1)/2+j$ of the storage array `C%val` will hold the value c_{ij} (and, by symmetry, c_{ji}) for $1 \le j \le i \le m$.

2.2.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **A**, its row index i, column index j and value a_{ij} are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to **H** and **C** (thus, for **H**, requiring integer arrays `H%row`, `H%col`, a real array `H%val` and an integer value `H%ne`), except that only the entries in the lower triangle need be stored.

2.2.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of **A**, the i-th component of a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr` $(m+1)$ holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i-th row are stored in components $l =$ `A%ptr`$(i), \ldots,$`A%ptr` $(i+1)-1$ of the integer array `A%col`, and real array `A%val`, respectively. The same scheme is applicable to **H** and **C** (thus, for **H**, requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle need be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2.4 Diagonal storage format

If **H** is diagonal (i.e., $h_{ij} = 0$ for all $1 \le i \ne j \le n$) only the diagonals entries h_{ii}, $1 \le i \le n$, need be stored, and the first n components of the array `H%val` may be used for the purpose. The same applies to **C**, but there is no sensible equivalent for the non-square **A**.

2.2.5 Zero storage format

If **C** is a zero matrix (i.e., $\mathbf{C} = 0$), then no entries need be stored. There is not a sensible equivalent for **A** or **H**.

2.3 The derived data type for holding control parameters

The derived data type `MI13_control_type` is used to control the action. The user must declare a structure of type `MI13_control`. Components of this derived type are automatically given their default values in the definition of the type: the user does not need to set them unless values other than the defaults are required. The following components are employed:

`error` is a scalar variable of type default `INTEGER` that holds the stream number for error and warning messages. Printing of error and warning messages in `MI13_solve` and `MI13_terminate` is suppressed if `error` < 0. The default is `error=6`.

`out` is a scalar variable of type default `INTEGER` that holds the stream number for diagnostic printing. Diagnostic printing is suppressed if `out<0`. The default is `out=6`.

`print_level` is a scalar variable of type default `INTEGER` that is used to control the amount of diagnostic printing required. No no diagnostic printing will occur if `print_level`$\leq$`0`. If `print_level=1`, a single line of output will be produced for each iteration of the process. If `print_level`$\geq$`2`, this output will be increased to provide significant detail of each iteration. The default is `print_level=0`.

`new_h` is a scalar variable of type default `INTEGER` that is used to indicate how **H** has changed (if at all) since the previous call to `MI13_form_and_factorize`. Possible values are:

- `0` **H** is unchanged
- `1` the values in **H** have changed, but its nonzero structure is as before.
- `2` both the values and structure of **H** have changed.

The default is `new_h=2`.

`new_a` is a scalar variable of type default `INTEGER` that is used to indicate how **A** has changed (if at all) since the previous call to `MI13_form_and_factorize`. Possible values are:

- `0` **A** is unchanged
- `1` the values in **A** have changed, but its nonzero structure is as before.
- `2` both the values and structure of **A** have changed.

The default is `new_a=2`.

`new_c` is a scalar variable of type default `INTEGER` that is used to indicate how **C** has changed (if at all) since the previous call to `MI13_form_and_factorize`. Possible values are:

- `0` **C** is unchanged
- `1` the values in **C** have changed, but its nonzero structure is as before.
- `2` both the values and structure of **C** have changed.

The default is `new_c=2`.

`preconditioner` is a scalar variable of type default `INTEGER` that specifies the preferred preconditioner to be computed; positive values correspond to explicit-factorization preconditioners while negative values indicate implicit-factorization ones. If the chioce of preconditioner is unsuitabble for the structure of the problem, then a different preconditioner may be computed: `info%preconditioner` contains the value of the preconditioner used. Possible values are:

`0` the preconditioner is chosen automatically on the basis of which option looks most likely to be the most efficient.

`1` $\mathbf{G}$ is chosen to be the identity matrix.

`2` $\mathbf{G}$ is chosen to be $\mathbf{H}$

`3` $\mathbf{G}$ is chosen to be the diagonal matrix whose diagonals are the larger of those of $\mathbf{H}$ and a positive constant (see `min_diagonal` below).

`4` $\mathbf{G}$ is chosen to be the band matrix of given semi-bandwidth whose entries coincide with those of $\mathbf{H}$ within the band. (see `semi_bandwidth` below).

`11` $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$ and $\mathbf{G}_{22} = \mathbf{H}_{22}$.

`12` $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = \mathbf{H}_{21}$ and $\mathbf{G}_{22} = \mathbf{H}_{22}$.

`-1` for the special case when $\mathbf{C} = 0$, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22}$ is the identity matrix, and the preconditioner is computed implicitly. If $\mathbf{C} \neq 0$, then the subroutine proceeds as if `preconditioner = -3`.

`-2` for the special case when $\mathbf{C} = 0$, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22} = \mathbf{H}_{22}$ and the preconditioner is computed implicitly. If $\mathbf{C} \neq 0$, then the subroutine proceeds as if `preconditioner = -4`.

`-3` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22}$ is the identity matrix, and the preconditioner is computed implicitly.

`-4` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = 0$, $\mathbf{G}_{22} = \mathbf{H}_{22}$ and the preconditioner is computed implicitly.

`-5` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = \mathbf{H}_{21}$, $\mathbf{G}_{22} = \mathbf{I} + \mathbf{H}_{21}\mathbf{A}_1^{-1}\mathbf{C}\mathbf{A}_1^{-T}\mathbf{H}_{21}^T + \mathbf{H}_{21}\mathbf{A}_1^{-1}\mathbf{A}_2 + \mathbf{A}_2^T\mathbf{A}_1^{-T}\mathbf{H}_{21}^T$ and the preconditioner is computed implicitly.

`-6` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = 0$, $\mathbf{G}_{21} = \mathbf{H}_{21}$, $\mathbf{G}_{22} = \mathbf{H}_{22} + \mathbf{H}_{21}\mathbf{A}_1^{-1}\mathbf{C}\mathbf{A}_1^{-T}\mathbf{H}_{21}^T + \mathbf{H}_{21}\mathbf{A}_1^{-1}\mathbf{A}_2 + \mathbf{A}_2^T\mathbf{A}_1^{-T}\mathbf{H}_{21}^T$ and the preconditioner is computed implicitly.

`-7` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = \alpha\mathbf{A}_1^T\mathbf{A}_1$, $\mathbf{G}_{21} = \alpha\mathbf{A}_2^T\mathbf{A}_1$, $\mathbf{G}_{22} = \mathbf{I} + \alpha\mathbf{A}_2^T\mathbf{A}_2$ and the preconditioner is computed implicitly. The value of α may be given by the user or determined automatically. See `alpha` below.

`-8` for the general case when $\mathbf{C}$ is symmetric, $\mathbf{G}$ is chosen so that $\mathbf{G}_{11} = \alpha\mathbf{A}_1^T\mathbf{A}_1$, $\mathbf{G}_{21} = \alpha\mathbf{A}_2^T\mathbf{A}_1$, $\mathbf{G}_{22} = \mathbf{H}_{22} + \alpha\mathbf{A}_2^T\mathbf{A}_2$ and the preconditioner is computed implicitly. The value of α may be given by the user or determined automatically. See `alpha` below.

Other values may be added in future. The default is `preconditioner=0`.

`semi_bandwidth` is a scalar variable of type default `INTEGER` that specifies the semi-bandwidth of the band preconditioner when `preconditioner=4`, if appropriate. The default is `semi_bandwidth=5`.

`factorization` is a scalar variable of type default `INTEGER` that specifies which factorization of the preconditioner should be used if `preconditioner`$\geq$`1`. Possible values are:

`0` the factorization is chosen automatically on the basis of which option looks likely to be the most efficient.

`1` if $\mathbf{G}$ is diagonal and non-singular, then a Schur-complement factorization (see §4), involving factors of $\mathbf{G}$ and $\mathbf{A}\mathbf{G}^{-1}\mathbf{A}^T$, will be used. Otherwise, and augmented-system factorization, involving factors of $\mathbf{K}_G$, will be used.

`2` an augmented-system factorization, involving factors of $\mathbf{K}_G$, will be used.

`3` a null-space factorization (see §4) will be used provided that $\mathbf{C} = \mathbf{0}$. If $\mathbf{C} \neq \mathbf{0}$, then the subroutine will proceed as if `factorization = 1`.

The default is factorization=0.

max_col is a scalar variable of type default INTEGER that specifies the maximum number of nonzeros in a column of **A** which is permitted by the Schur-complement factorization. The default is max_col=35.

indmin is a scalar variable of type default INTEGER that specifies an initial estimate as to the amount of integer workspace required by the factorization package MA57. The default is indmin=1000.

valmin is a scalar variable of type default INTEGER that specifies an initial estimate as to the amount of real workspace required by the factorization package MA57. The default is valmin=1000.

len_ma48min is a scalar variable of type default INTEGER that specifies an initial estimate as to the amount of workspace required by the factorization package MA48. The default is len_ma48min=1000.

itref_max is a scalar variable of type default INTEGER that specifies the maximum number of iterative refinements allowed with each application of the preconditioner. The default is itref_max=1.

basis_scale is a scalar variable of type default INTEGER that specifies whether the columns of **A** should be scaled or reordered before **A** is analyzed to find the invertible sub-block $\mathbf{A}_1$. Any scaling is removed before operating with $\mathbf{A}_1$ or $\mathbf{A}_2$. If the diagonal entries of **H** differ by several orders of magnitude, then it is recommended that basis_scale>0. Possible values are:

0 no scaling or initial reordering used.

1 For $i = 1,\dots,n$: the ith column of **A** is scaled by the inverse of the ith diagonal entry of **H**.

2 For $i = 1,\dots,n$: the ith column of **A** is scaled by the inverse of the square root of the ith diagonal entry of **H**.

3 If find_basis_by_transpose = .false. then the columns of **A** are initially reordered such that the diagonal entries of **H** are monotonically increasing when correspondingly reordered; otherwise, no scaling or initial reordering used.

The default is basis_scale=2.

alpha is a scalar variable of type default REAL that specifies the value of α for preconditioner=-7 or preconditioner = -8. If alpha is non-positive, then the value of α is chosen automatically such that $\mathbf{C}+\alpha\mathbf{I}$ is positive definite. If alpha is positive, then $\alpha =$ alpha is used and $\mathbf{C}+\alpha\mathbf{I}$ must be **positive definite**. The default is alpha=-1.

pivot_tol is a scalar variable of type default REAL that holds the threshold pivot tolerence used by the matrix factorization. See the documentation for the packages MA57 and MA48 for details. The default is pivot_tol=0.01.

pivot_tol_for_basis is a scalar variable of type default REAL that holds the threshold pivot tolerence used by the package MA48 when computing the non-singular basis matrix $\mathbf{A}_1$ for an implicit-factorization preconditioner. Since the calculation of a suitable basis is crucial, it is sensible to pick a larger value of pivot_tol_for_basis than of pivot_tol. The default is pivot_tol_for_basis=0.5.

zero_pivot is a scalar variable of type default REAL that is used to detect singularity. Any pivot encountered during the factorization whose absolute value is less than or equal to zero_pivot will be regarded as zero, and the matrix as singular. The default is zero_pivot=EPSILON(1.0)$^{0.75}$.

min_diagonal is a scalar variable of type default REAL that specifies the smallest permitted diagonal in **G** for some of the preconditioners provided. See preconditioner above. The default is min_diagonal=0.00001.

remove_dependencies is a scalar variable of type default LOGICAL that must be set .TRUE. if linear dependent rows from the second block equation $\mathbf{Ax}-\mathbf{Cy}=\mathbf{b}$ from (1.3) should be removed and .FALSE. otherwise. The default is remove_dependencies=.TRUE..

check_basis is a scalar variable of type default LOGICAL that must be set .TRUE. if the basis matrix $\mathbf{A}_1$ constructed when using an implicit-factorization preconditioner or null-space factorization should be assessed for ill conditioning and corrected if necessary. If these precautions are not thought necessary, check_basis should be set .FALSE.. The default is check_basis=.TRUE..

find_basis_by_transpose is a scalar variable of type default LOGICAL that must be set .TRUE. if the invertible subblock $\mathbf{A}_1$ of the columns of $\mathbf{A}$ is computed by analysing the transpose of $\mathbf{A}$ and .FALSE. if the analysis is based on $\mathbf{A}$ itself. Generally an analysis based on the transpose is faster. The default is find_basis_by_transpose=.TRUE..

use_old_basis is a scalar variable of type default LOGICAL that must be set .TRUE. if new_a = 0 and the user wishes to use the previously computed basis. If new_a > 0 or use_old_basis = .FALSE. the basis will be recomputed. The default is use_old_basis=.FALSE..

affine is a scalar variable of type default LOGICAL that must be set .TRUE. if the component $\mathbf{b}$ of the right-hand side of the required system (1.3) is zero, and .FALSE. otherwise. Computational savings are possible when $\mathbf{b} = 0$. The default is affine=.FALSE..

perturb_to_make_definite is a scalar variable of type default LOGICAL that must be set .TRUE. if the user wants to guarantee that the computed preconditioner is suitable by boosting the diagonal of the requested $\mathbf{G}$ and .FALSE. otherwise. The default is perturb_to_make_definite=.TRUE..

get_norm_residual is a scalar variable of type default LOGICAL that must be set .TRUE. if the user wishes the package to return the value of the norm of the residuals for the computed solution when applying the preconditioner and .FALSE. otherwise. The default is get_norm_residual=.FALSE..

space_critical is a scalar variable of type default LOGICAL that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical=.FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL that must be set .TRUE. if the user wishes to terminate execution if a allocatable deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal=.FALSE..

prefix is a scalar variable of type default CHARACTER and length 30 that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix="".

2.4 The derived data type for holding informational parameters

The derived data type MI13_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of MI13_inform_type are:

status is a scalar variable of type default INTEGER that gives the exit status of the algorithm. See §2.7 for details.

alloc_status is a scalar variable of type default INTEGER that gives the status of the last attempted array allocation or deallocation. This will be 0 if status=0.

bad_alloc is a scalar variable of type default CHARACTER and length 80 that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status=0.

ma57_analyse_status is a scalar variable of type default INTEGER that reports the return code from the most recent call to MA57_analyse by MI13_form_and_factorize. A non-zero value indicates a warning or an error. See the documentation for the package MA57 for further details.

ma57_factorize_status is a scalar variable of type default INTEGER that reports the return code from the most recent call to MA57_factorize by MI13_form_and_factorize. A non-zero value indicates a warning or an error. See the documentation for the package MA57 for further details.

ma57_solve_status is a scalar variable of type default INTEGER that reports the return code from the most recent call to MA57_solve by MI13_solve. A non-zero value indicates a warning or an error. See the documentation for the package MA57 for further details.

ma48_analyse_status is a scalar variable of type default INTEGER that reports the return code from the most recent call to MA48_analyse by MI13_form_and_factorize. A non-zero value indicates a warning or an error. See the documentation for the package MA48 for further details.

ma48_solve_status is a scalar variable of type default INTEGER reports the return code from the most recent call to MA48_solve by MI13_solve. A non-zero value indicates a warning or an error. See the documentation for the package MA48 for further details.

factorization_integer is a scalar variable of type default INTEGER reports the number of integers required to hold the factorization.

factorization_real is a scalar variable of type default INTEGER reports the number of reals required to hold the factorization.

preconditioner is a scalar variable of type default INTEGER that indicates the preconditioner method used. The range of values returned corresponds to those requested in control%preconditioner, excepting that the requested value may have been altered to a more appropriate one during the factorization. In particular, if the automatic choice control%preconditioner 0 is requested, preconditioner reports the actual choice made.

factorization is a scalar variable of type default INTEGER that indicates the factorization method used. The range of values returned corresponds to those requested in control%factorization, excepting that the requested value may have been altered to a more appropriate one during the factorization. In particular, if the automatic choice control%factorization=0 is requested, factorization reports the actual choice made.

rank is a scalar variable of type default INTEGER that gives the computed rank of **A**.

inform%entries_ignored_a is a scalar variable of type default INTEGER that gives the computed number of entries in **A** that have out-of-range indices.

inform%entries_ignored_c is a scalar variable of type default INTEGER that gives the computed number of entries in **C** that have out-of-range indices.

inform%entries_ignored_h is a scalar variable of type default INTEGER that gives the computed number of entries in **H** that have out-of-range indices.

rank_def is a scalar variable of type default LOGICAL that has the value .TRUE. if MI13_form_and_factorize believes that **A** is rank defficient, and .FALSE. otherwise.

perturbed is a scalar variable of type default LOGICAL that has the value .TRUE. if and only if the original choice of **G** has been perturbed to ensure that $\mathbf{K}_G$ is an appropriate preconditioner to use. This will only happen if control%perturb_to_make_definite has been set .TRUE..

alpha is a scalar variable of type default REAL that holds the value α used when preconditioner=-7 or preconditioner=-8. Otherwise it will have the value -1.0.

norm_residual is a scalar variable of type default REAL that holds the infinity norm of the residual of the system (1.3) after a call to MI13_solve if control%get_norm_residual has been set .TRUE.. Otherwise it will have the value -1.0.

2.5 The derived data type for holding problem data

The derived data type `MI13_data_type` is used to hold all the data for the problem and the factors of its preconditioners between calls of `MI13` procedures. This data should be preserved, untouched, from the initial call to `MI13_initialize` to the final call to `MI13_terminate`.

2.6 Argument lists and calling sequences

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.**

2.6.1 The initialization subroutine

Default values are provided as follows:

```
CALL MI13_initialize(data,control)
```

`data` is a scalar `INTENT(OUT)` argument of type `MI13_data_type` (see §2.5). It is used to hold data about the problem being solved. `MI13_initialize` will ensure that all components that are allocatable arrays are disassociated.

`control` is a scalar `INTENT(OUT)` argument of type `MI13_control_type` (see §2.3). On exit, its components will have been given the default values specified in §2.3.

2.6.2 The subroutine for forming and factorizing the preconditioner

A preconditioner of the form (1.1) is formed and factorized as follows:

```
CALL MI13_form_and_factorize(n,m,H,A,C,data,control,inform)
```

`n` is a scalar `INTENT(IN)` argument of type default `INTEGER` that specifies the number of rows of **H** (and columns of **A**). **Restriction:** $n \geq 1$.

`m` is a scalar `INTENT(IN)` argument of type default `INTEGER` that specifies the number of rows of **A** and **C**. **Restriction:** $0 \leq m \leq n$.

`H` is a scalar `INTENT(IN)` argument of type `ZD11_type` whose components must be set to specify the data defining the matrix **H** (see §2.2).

`A` is a scalar `INTENT(IN)` argument of type `ZD11_type` whose components must be set to specify the data defining the matrix **A** (see §2.2).

`C` is a scalar `INTENT(IN)` argument of type `ZD11_type` whose components must be set to specify the data defining the matrix **C** (see §2.2).

`data` is a scalar `INTENT(INOUT)` argument of type `MI13_data_type` (see §2.5). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to `MI13_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `MI13_control_type` (see §2.3). Default values may be assigned by calling `MI13_initialize` prior to the first call to `MI13_solve`.

`inform` is a scalar `INTENT(OUT)` argument of type `MI13_inform_type` (see §2.4). A successful call to `MI13_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.7.

2.6.3 The subroutine for applying the preconditioner

The preconditioner may be applied to solve a system of the form (1.3) as follows:

```
CALL MI13_solve(n,m,H,A,C,data,control,inform,SOL)
```

Components `n`, `m`, `H A`, `C`, `data` and `control` are exactly as described for `MI13_form_and_factorize` and must not have been altered in the interim.

`inform` is a scalar `INTENT(OUT)` argument of type `MI13_inform_type` (see §2.4), that should be passed unaltered since the last call to `MI13_form_and_factorize` or `MI13_solve`. A successful call to `MI13_solve` is indicated when the component `status` has the value 0. For other return values of `status`, see §2.7.

`SOL` is a rank-one `INTENT(INOUT)` array of type default `REAL` and length at least `n+m`, that must be set on entry to hold the composite vector $(\mathbf{a}^T\ \ \mathbf{b}^T)^T$. In particular `SOL(`$i$`)`, $i = 1, \ldots$ `n` should be set to a_i, and `SOL(n+`j`)`, $j = 1, \ldots,$ `m` should be set to b_j. On successful exit, `SOL` will contain the solution $(\mathbf{x}^T\ \ \mathbf{y}^T)^T$ to (1.3), that is `SOL(`i`)`, $i = 1, \ldots,$ `n` will give x_i, and `SOL(n+`j`)`, $j = 1, \ldots,$ `m` will contain y_j.

2.6.4 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL MI13_terminate(data,control,inform)
```

`data` is a scalar `INTENT(INOUT)` argument of type `MI13_data_type` exactly as for `MI13_solve`, which must not have been altered **by the user** since the last call to `MI13_initialize`. On exit, array components will have been deallocated.

`control` is a scalar `INTENT(IN)` argument of type `MI13_control_type` exactly as for `MI13_solve`.

`inform` is a scalar `INTENT(OUT)` argument of type `MI13_inform_type` exactly as for `MI13_solve`. Only the component `status` will be set on exit, and a successful call to `MI13_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see §2.7.

2.7 Warning and error messages

A negative value of `info%status` on exit from `MI13_form_and_factorize`, `MI13_solve` or `MI13_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

`-1.` An allocation error occured. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

`-2.` A deallocation error occured. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

`-3.` One of the restrictions `n` > 0 or `m` ≥ 0 or requirements that `A%type`, `H%type` and `C%type` contain its relevant string `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` or `'ZERO'` has been violated.

`-4.` An error was reported by `MA57_analyse`. The return status from `MA57_analyse` is given in `inform%ma57_analyse_status`. See the documentation for the HSL package `MA57` for further details.

`-5.` An error was reported by `MA57_factorize`. The return status from `MA57_factorize` is given in `inform%ma57_-factorize_status`. See the documentation for the HSL package `MA57` for further details.

`-6.` An error was reported by `MA57_solve`. The return status from `MA57_solve` is given in `inform%ma57_solve_-status`. See the documentation for the HSL package `MA57` for further details.

`-7.` An error was reported by `MA48_analyse`. The return status from `MA48_analyse` is given in `inform%ma48_analyse_-status`. See the documentation for the HSL package `MA48` for further details.

`-8.` An error was reported by `MA48_solve` when the preconditioner is prescribed explicitly. The return status from `MA48_solve` is given in `inform%ma48_solve_status`. See the documentation for the HSL package `MA48` for further details.

`-9.` The computed preconditioner has the wrong inertia and is thus unsuitable.

`-10.` An error was reported by `MC59` when the preconditioner is prescribed implicitly. The return status from `MC59` is given in `inform%mc59_status`. See the documentation for the HSL package `MC59` for further details.

`-11.` The value of `control%preconditioner` is invalid.

`-12.` The matrix $\mathbf{C}+\alpha\mathbf{I}$ is not positive definite and `control%preconditioner = -7` or `-8`. Hence, the preconditioner is not suitable.

A positive value of `info%status` on exit from `MI13_form_and_factorize` warns of unexpected behaviour. Positive values are summed so that the user can identify all warnings issued by `MI13`, e.g. `info%flag=+3` indicates both warnings +1 and +2 have been issued.

`+1.` The matrix **A** is rank deficient.

`+2.` Some input entries ignored because they were found to be out-of-range. The number of entries found to be out-of-range in **A** is given in `inform%entries_ignored_a`, the number of entries found to be out-of-range in **C** is given in `inform%entries_ignored_c` and the number of entries found to be out-of-range in **H** is given in `inform%entries_ignored_h`.

`+4.` `control%preconditioner`$\neq$`0` and `info%preconditioner`$\neq$`control%preconditioner`.

`+8.` `control%factorization`$\neq$`0` and `info%factorization`$\neq$`control%factorization`.

3 GENERAL INFORMATION

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: `n>0`, `0`$\leq$`m`$\leq$`n`, `H%type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }, `A%type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` }, and `C%type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'`, `'ZERO'` }.

4 METHOD

The method used depends on whether an explicit or implicit factorization is required. In the explicit case, the package is really little more than a wrapper for the symmetric, indefinite linear solver `MA57` in which the system matrix $\mathbf{K}_G$ is assembled from its constituents **A**, **C** and whichever **G** is requested by the user. Implicit-factorization preconditioners are more involved, and there is a large variety of different possibilities. The essential ideas are described in detail in

H. S. Dollar, N. I. M. Gould and A. J. Wathen. "On implicit-factorization constraint preconditioners". In Large Scale Nonlinear Optimization (G. Di Pillo and M. Roma, eds.) Springer Series on Nonconvex Optimization and Its Applications, Vol. 83, Springer Verlag (2006) 61–82

and

H. S. Dollar, N. I. M. Gould, W. H. A. Schilders and A. J. Wathen "On iterative methods and implicit-factorization preconditioners for regularized saddle-point systems". SIAM Journal on Matrix Analysis and Applications, **28(1)** (2006) 170–189.

The Schur-complement factorization is based upon the decomposition

$$\mathbf{K}_G = \begin{pmatrix} \mathbf{G} & \mathbf{0} \\ \mathbf{A} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{G}^{-1} & \mathbf{0} \\ \mathbf{0} & -\mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{G} & \mathbf{A}^T \\ \mathbf{0} & \mathbf{I} \end{pmatrix},$$

where the "Schur complement" $\mathbf{S} = \mathbf{C} + \mathbf{A}\mathbf{G}^{-1}\mathbf{A}^T$. Such a method requires that **S** is easily invertible. This is often the case when **G** is a diagonal matrix, in which case **S** is frequently sparse, or when $m \ll n$ in which case **S** is small and a dense Cholesky factorization may be used.

When $\mathbf{C} = 0$, the null-space factorization is based upon the decomposition

$$\mathbf{K}_G = \mathbf{P} \begin{pmatrix} \mathbf{G}_{11} & \mathbf{0} & \mathbf{I} \\ \mathbf{G}_{21} & \mathbf{I} & \mathbf{A}_2^T\mathbf{A}_1^{-T} \\ \mathbf{A}_1 & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & -\mathbf{G}_{11} \end{pmatrix} \begin{pmatrix} \mathbf{G}_{11} & \mathbf{G}_{21}^T & \mathbf{A}_1^T \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{I} & \mathbf{A}_1^{-1}\mathbf{A}_2 & \mathbf{0} \end{pmatrix} \mathbf{P}^T,$$

where the "reduced Hessian"

$$\mathbf{R} = (-\mathbf{A}_2^T\mathbf{A}_1^{-T} \;\; \mathbf{I}) \begin{pmatrix} \mathbf{G}_{11} & \mathbf{G}_{21}^T \\ \mathbf{G}_{21} & \mathbf{G}_{22} \end{pmatrix} \begin{pmatrix} -\mathbf{A}_1^{-1}\mathbf{A}_2 \\ \mathbf{I} \end{pmatrix}$$

and **P** is a suitably-chosen permutation for which $\mathbf{A}_1$ is invertible. The method is most useful when $m \approx n$ as then the dimension of **R** is small and a dense Cholesky factorization may be used.

5 EXAMPLE OF USE

Suppose we wish to solve the linear system (1.3) with matrix data

$$\mathbf{H} = \begin{pmatrix} 1 & & 4 \\ & 2 & \\ 4 & & 3 \end{pmatrix}, \;\; \mathbf{A} = \begin{pmatrix} 2 & 1 & \\ & 1 & 1 \end{pmatrix} \text{ and } \mathbf{C} = \begin{pmatrix} & 1 \\ 1 & \end{pmatrix}$$

and right-hand sides

$$\mathbf{a} = \begin{pmatrix} 7 \\ 4 \\ 8 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Then storing the matrices in sparse co-ordinate format, we may use the following code:

```
PROGRAM HSL_MI13_EXAMPLE
  USE HSL_MI13_double        ! double precision version
  USE HSL_ZD11_double
  IMPLICIT NONE
  TYPE ( ZD11_type ) :: H, A, C
  DOUBLE PRECISION, ALLOCATABLE, DIMENSION( : ) :: SOL
  TYPE ( MI13_data_type ) :: data
  TYPE ( MI13_control_type ) :: control
  TYPE ( MI13_inform_type ) :: inform
  INTEGER :: stat
  INTEGER :: n = 3, m = 2, h_ne = 4, a_ne = 4, c_ne = 1
  ! start problem data
  ALLOCATE( SOL( n + m ) )
  SOL( 1 : n ) = (/ 7.0, 4.0, 8.0 /)  ! RHS a
  SOL( n + 1 : n + m ) = (/ 2.0, 1.0 /)  ! RHS b
  ! sparse co-ordinate storage format
  CALL ZD11_put( H%type, 'COORDINATE' ,stat=stat)  ! Specify co-ordinate
  CALL ZD11_put( A%type, 'COORDINATE' ,stat=stat)  ! storage for H, A and C
  CALL ZD11_put( C%type, 'COORDINATE' ,stat=stat)
  ALLOCATE( H%val( h_ne ), H%row( h_ne ), H%col( h_ne ) )
  ALLOCATE( A%val( a_ne ), A%row( a_ne ), A%col( a_ne ) )
  ALLOCATE( C%val( c_ne ), C%row( c_ne ), C%col( c_ne ) )
  H%val = (/ 1.0, 2.0, 3.0, 4.0 /)             ! matrix H
  H%row = (/ 1, 2, 3, 3 /)                     ! NB lower triangle
  H%col = (/ 1, 2, 3, 1 /) ; H%ne = h_ne
  A%val = (/ 2.0, 1.0, 1.0 ,1.0 /)             ! matrix A
  A%row = (/ 1, 1, 2, 2 /)
  A%col = (/ 1, 2, 2, 3 /) ; A%ne = a_ne
  C%val = (/ 1.0 /)                            ! matrix C
  C%row = (/ 2 /)                              ! NB lower triangle
  C%col = (/ 1 /) ; C%ne = c_ne
  ! problem data complete
  CALL MI13_initialize( data, control )        ! Initialize control parameters
  control%preconditioner = 2                   ! Exact factorization
  ! factorize matrix
  CALL MI13_form_and_factorize( n, m, H, A, C, data, control, inform )
  IF ( inform%status < 0 ) THEN                ! Unsuccessful call
     WRITE( 6, "( ' MI13_form_and_factorize exit status = ', I6 ) " )          &
          inform%status
     STOP
  END IF
  ! solve system
  CALL MI13_solve( n, m, A, C, data, control, inform, SOL )
  IF ( inform%status == 0 ) THEN                ! Successful return
     WRITE( 6, "( ' MI13: Solution = ', /, ( 5ES12.4 ) )" ) SOL
  ELSE                                          !  Error returns
     WRITE( 6, "( ' MI13_solve exit status = ', I6 ) " ) inform%status
  END IF
  CALL MI13_terminate( data, control, inform )  ! delete internal workspace
END PROGRAM HSL_MI13_EXAMPLE
```

This produces the following output:

```
 MI13: Solution =
  1.0000E+00  1.0000E+00  1.0000E+00  1.0000E+00  1.0000E+00
```

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```
!  sparse co-ordinate storage format
```

```
...
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL ZD11_put( H%type, 'SPARSE_BY_ROWS', stat=stat )  ! Specify sparse-by-rows
   CALL ZD11_put( A%type, 'SPARSE_BY_ROWS', stat=stat )  ! storage for H, A and C
   CALL ZD11_put( C%type, 'SPARSE_BY_ROWS', stat=stat )
   ALLOCATE( H%val( h_ne ), H%col( h_ne ), H%ptr( n + 1 ) )
   ALLOCATE( A%val( a_ne ), A%col( a_ne ), A%ptr( m + 1 ) )
   ALLOCATE( C%val( c_ne ), C%col( c_ne ), C%ptr( m + 1 ) )
   H%val = (/ 1.0, 2.0, 3.0, 4.0 /)             ! matrix H
   H%col = (/ 1, 2, 3, 1 /)                     ! NB lower triangular
   H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row allocatables
   A%val = (/ 2.0, 1.0, 1.0, 1.0 /)             ! matrix A
   A%col = (/ 1, 2, 2, 3 /)
   A%ptr = (/ 1, 3, 5 /)                        ! Set row allocatables
   C%val = (/ 1.0 /)                            ! matrix C
   C%col = (/ 1 /)                              ! NB lower triangular
   C%ptr = (/ 1, 1, 2 /)                        ! Set row allocatables
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL ZD11_put( H%type, 'DENSE', stat=stat )  ! Specify dense
   CALL ZD11_put( A%type, 'DENSE', stat=stat )  ! storage for H, A and C
   CALL ZD11_put( C%type, 'DENSE', stat=stat )
   ALLOCATE( H%val( n * ( n + 1 ) / 2 ) )
   ALLOCATE( A%val( n * m ) )
   ALLOCATE( C%val( m * ( m + 1 ) / 2 ) )
   H%val = (/ 1.0, 0.0, 2.0, 4.0, 0.0, 3.0 /) ! H
   A%val = (/ 2.0, 1.0, 0.0, 0.0, 1.0, 1.0 /) ! A
   C%val = (/ 0.0, 1.0, 0.0 /)               ! C
! problem data complete
```

respectively.

If instead **H** had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for **H**, and in this case we would instead

```
   CALL ZD11_put( H%type, 'DIAGONAL', stat=stat )  ! Specify dense storage for H
   ALLOCATE( H%val( n ) )
   H%val = (/ 1.0, 0.0, 3.0 /) ! Hessian values
```

Notice here that zero diagonal entries are stored.