HSL_MI20

PACKAGE SPECIFICATION HSL 2011

1 SUMMARY

Given an $n \times n$ sparse matrix $\mathbf{A}$ and an $n-$vector $\mathbf{z}$, `HSL_MI20` computes the vector $\mathbf{x} = \mathbf{Mz}$, where $\mathbf{M}$ is an algebraic multigrid (AMG) v-cycle preconditioner for $\mathbf{A}$. A classical AMG method is used, as described in [1] (see also Section 5 below for a brief description of the algorithm). The matrix $\mathbf{A}$ must have positive diagonal entries and (most of) the off-diagonal entries must be negative (the diagonal should be large compared to the sum of the off-diagonals). During the multigrid coarsening process, positive off-diagonal entries are ignored and, when calculating the interpolation weights, positive off-diagonal entries are added to the diagonal.

Reference

[1] K. Stüben. *An Introduction to Algebraic Multigrid.* In U. Trottenberg, C. Oosterlee, A. Schüller, eds, 'Multigrid', Academic Press, 2001, pp 413-532.

ATTRIBUTES — Version: 1.5.1 (8 November 2012) **Types:** Real (single, double). **Precision:** At least 8-byte arithmetic is recommended. **Uses:** `HSL_MA48`, `HSL_MC65`, `HSL_ZD11`, and the `LAPACK` routines `_GETRF` and `_GETRS`. **Original date:** September 2006. **Origin:** J. W. Boyle, University of Manchester and J. A. Scott, Rutherford Appleton Laboratory. **Language:** Fortran 95, plus allocatable dummy arguments and allocatable components of derived types. **Remark:** The development of `HSL_MI20` was funded by EPSRC grants EP/C000528/1 and GR/S42170.

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

Access to the package requires a `USE` statement:

Single precision version

```
USE HSL_MI20_single
```

Double precision version

```
USE HSL_MI20_double
```

In `HSL_MI20_single`, all reals are default reals. In `HSL_MI20_double`, all reals are double precision reals.

If it is required to use both modules at the same time, the derived types (Section 2.2) must be renamed in one of the `USE` statements.

The following procedures are available to the user:

`MI20_setup` takes the matrix $\mathbf{A}$ and generates data that is required by the AMG preconditioner.

`MI20_precondition` performs the preconditioning operation $\mathbf{x} = \mathbf{Mz}$, where $\mathbf{M}$ is the AMG preconditioner and $\mathbf{z}$ is a user-supplied vector.

`MI20_finalize` should be called after all other calls are complete for a problem to deallocate components of the derived data types.

2.2 The derived data types

For each problem, the user must employ the derived types defined by the module to declare scalars of the types `MI20_keep`, `MI20_control`, `MI20_info`, and `ZD11_type`, and an array of type `MI20_data`. The following pseudocode illustrates this.

```
use HSL_MI20_double
...
type (MI20_keep) :: keep
type (MI20_control) :: control
type (MI20_info) :: info
type (ZD11_type) :: matrix
type (MI20_data), allocatable :: coarse_data(:)
...
```

The components of `MI20_keep` are used to pass data between the subroutines of the package and must not be altered by the user. The components of the other derived types are explained in Sections 2.2.1 to 2.2.4.

2.2.1 The derived data type `MI20_control` for holding control parameters

The derived data type `MI20_control` is used to hold controlling data. The components, which are automatically given default values in the definition of the type, are as follows.

Controls used by `MI20_setup` (in alphabetical order)

`aggressive` is a scalar of type default `INTEGER` that controls the coarsening used. If `aggressive=1`, normal (non-aggressive) coarsening is used. For values greater than `1`, aggressive coarsening is used, and the value determines the number of coarsening steps that are applied between levels (see Section 5.1.2). The default is `1`. **Restriction:** `aggressive`$\geq$ `1`.

`c_fail` is a scalar of type default `INTEGER` that controls the coarsening failure criteria. A value of `1` indicates that coarsening terminates if **any** row in a coarse level matrix has at least one strictly positive entry but no negative off-diagonal entries. A value of `2` indicates that coarsening terminates if **all** the rows in a coarse level matrix have at least one strictly positive entry and no negative off-diagonal entries or if the lack of negative negative off-diagonals causes coarsening to fail. The default is `1`. **Restriction:** `c_fail` = `1` or `2`.

`max_levels` is a scalar of type default `INTEGER` that holds the maximum number of coarse levels in the multigrid structure that is generated by `MI20_setup`. The default is `100`. **Restriction:** `max_levels`$\geq$ `1`.

`max_points` is a scalar of type default `INTEGER`. Coarsening terminates if either the number of coarse levels is `max_levels` or the number of points in a coarse level is less than or equal to `max_points`. The default is `1`. **Restriction:** `max_points`$>$ `0`.

`one_pass_coarsen` is a scalar of type default `LOGICAL` with default value `.false.`. If set to `.true.`, one pass coarsening is used. This reduces the time required at each level to construct the coarse and fine points (and can significantly reduce the time required to compute the preconditioner) but it may result in a poorer quality preconditioner.

`reduction` is a scalar of type `REAL`. If two successive levels have n_c and n_f points, respectively, coarsening continues while $n_c \leq n_f$`*reduction`. `reduction` must be at least `0.5` and at most one `1`. The default value is `0.8`.

`st_method` is a scalar of type default `INTEGER` that controls the method used to find strong transpose connections (see Section 5.1.1). If `st_method = 1`, they are found as they are required; if `st_method = 2`, they are found before coarsening starts and stored. If the matrix has an unsymmetric sparsity pattern, method `2` is always used. The default is `2`. **Restriction:** `st_method` = `1` or `2`.

`st_parameter` is a scalar of type `REAL` that is used in determining whether connections are strong or weak (see Section 5.1.1 for details). The default is `0.25` but for some applications (especially in 3D), it can be advantageous to use a larger value. **Restriction:** $0.0 \leq$`st_parameter`≤ 1.0.

`testing` is a scalar of type default `INTEGER` that controls whether or not the user-supplied matrix data is tested for errors. If `testing` = `0`, no testing is performed; if `testing` = `1`, the data is tested for duplicates and out-of-range entries (which are not allowed). Testing involves a small overhead. The default is `1`. **Restriction:** `testing` = `0` or `1`.

`trunc_parameter` is a scalar of type `REAL` that controls truncation of the interpolation weights. The default is 0.0 (interpolation weights are not truncated). **Restriction:** $0.0 \leq$`control%trunc_parameter`< 1.0.

Controls used by `MI20_precondition` (in alphabetical order)

`coarse_solver` is a scalar of type default `INTEGER` that controls which solver is used on the coarsest level. Possible values are:

`1`: damped Jacobi (with damping factor `damping`)
`2`: Gauss-Seidel
`3`: sparse direct solver `HSL_MA48`
`4`: `LAPACK` dense direct solver `_GETRF`

The default is `3` but note that it may be faster to use an iterative solver (`coarse_solver` = `1` or `2`). **Restriction:** `coarse_solver` = `1`, `2`, `3` or `4`.

`coarse_solver_its` is a scalar of type default `INTEGER`. `coarse_solver_its` controls the number of iterations used by the iterative solver on the coarsest level (`control%coarse_solver = 1` or `2` only). If `control%coarse_solver = 2`, one iteration comprises a forward and a backward Gauss-Seidel sweep. The default is `10`. **Restriction:** `coarse_solver_its`> 0.

`damping` is a scalar of type `REAL`. If damped Jacobi is used (`control%smoother` = `1`), it holds the damping factor. The default is `0.8`. **Restriction:** $0.0 <$`damping`≤ 1.0.

`err_tol` is a scalar of type `REAL` that determines the failure criterion for `MI20_precondition`. If $\|\mathbf{x}\|_2 >$ `control%err_tol` $*\|\mathbf{z}\|_2$ where $\mathbf{x} = \mathbf{M}\mathbf{z}$, an error is returned (see error return `-14` in Section 3.3). The default is `1.0e10`. **Restriction:** `err_tol`> 0.

`levels` is a scalar of type default `INTEGER` that controls the maximum number of coarse levels used before the coarse level solve is performed. A value $<$ `0` indicates that the maximum number of available coarse levels should be used (that is, the value of `info%clevels` returned by `MI20_setup`, see Section 2.2.2). The default is `-1`.

`ma48` is a scalar of type default `INTEGER` that is used if `control%coarse_solver` = `3` to control the use of `HSL_MA48`. If `control%ma48 = 0`, `HSL_MA48` is used with its default settings and no action is required by the user. Other values of `control%ma48` allow the use to change the default settings and/or to run the analyse and factorization phases of `HSL_MA48` separately. Possible values are:

`1`: `MI20_precondition` will return once the `HSL_MA48` control derived type `ma48_cntrl` (see Section 2.3.2) has been initialised with default settings. At this point, the user can reset one of more of the components of `ma48_cntrl` and then recall `MI20_precondition` with `control%ma48` = `2`, `3`, or `4`.

2: `MI20_precondition` will return after the analyse phase of `HSL_MA48` is complete. At this point, the user can use the information contained in `info%ma48_ainfo` to decide whether or not to continue with the factorization phase. If the user decides to continue, `MI20_precondition` should be recalled with `control%ma48` = `3` or `4`.

3: `MI20_precondition` will return after the factorisation phase of `HSL_MA48` is complete. At this point, the user can use the information contained in `info%ma48_finfo` to decide whether or not to continue to use `HSL_MA48`. If the user decides to continue, `MI20_precondition` should be recalled with `control%ma48` = `4`.

4: `MI20_precondition` will complete any phases of `HSL_MA48` that have not yet been performed and will then perform the preconditioning operation.

The default is `0`. **Restriction:** `ma48` = `0`, `1`, `2`, `3`, or `4`.

`pre_smoothing` is a scalar of type default `INTEGER` that holds the number of pre-smoothing iterations that are performed during each v-cycle (see Section 5.2). The default is `2`. **Restriction:** `pre_smoothing`$\geq$ 0 and `pre_smoothing` + `post_smoothing`$\neq$ 0.

`post_smoothing` is a scalar of type default `INTEGER` that holds the number of post smoothing iterations that are performed (see Section 5.2). If `control%smoother=2`, the Gauss-Seidel sweep direction is reversed for the post smoothing and, in this case, if **A** is symmetric, `post_smoothing` should be set to be equal to `pre_smoothing`. The default is `2`. **Restriction:** `post_smoothing`$\geq$ 0 and `pre_smoothing` + `post_smoothing`$\neq$ 0.

`smoother` is a scalar of type default `INTEGER` that controls which smoother is used during each v-cycle. If `smoother=1`, damped Jacobi is used; if `smoother=2`, symmetric Gauss-Seidel is used (that is, the Gauss-Seidel sweep direct is reversed on the post smoothing iterations). The default is `2`. **Restriction:** `smoother` = `1` or `2`.

`v_iterations` is a scalar of type default `INTEGER` that controls the number of v-cycle iterations to be performed. The default is `1`. **Restriction:** `v_iterations`$\geq$ 1.

Printing controls

`error` is a scalar of type default `INTEGER` that holds the unit number for the printing of error and warning messages. Printing is suppressed if `error < 0`. The default is `6`.

`print` is a scalar of type default `INTEGER` that holds the unit number for diagnostic printing. Printing is suppressed if `print < 0`. The default is `6`.

`print_level` is a scalar of type default `INTEGER` that controls the amount of printing. Possible values are:

0: no printing

1: printing of errors and warnings only

2: as `1` plus additional diagnostic printing

The default is `1`. **Restriction:** `print_level` = `0`, `1` or `2`.

2.2.2 The derived data type `MI20_info` for holding information

The components of the derived data type `MI20_info` are used to provide information about the progress of the algorithm. The components of `MI20_info` are:

`flag` is a scalar of type default `INTEGER` that is used as a error and warning flag. See Section 3 for details.

`clevels` is a scalar of type default `INTEGER` that, after a call to `MI20_setup`, contains the number of coarse levels generated.

`cpoints` is a scalar of type default `INTEGER` that, after a call to `MI20_setup`, contains the order of the matrix on the coarsest level.

`cnnz` is a scalar of type default `INTEGER` that, after a call to `MI20_setup`, contains the number of nonzeros in the matrix on the coarsest level.

`ma48_ainfo` is a scalar of derived type `MA48_AINFO`. The components of `ma48_ainfo` are only set if `control%coarse_solver = 3`. In this case, `ma48_ainfo` is the derived type used by the solver `HSL_MA48` to hold information from `MA48_ANALYSE`. `ma48_ainfo%flag` holds the error flag for `MA48_ANALYSE`. Full details on the derived type `MA48_AINFO` are given in the user documentation for `HSL_MA48`.

`ma48_finfo` is a scalar of derived type `MA48_FINFO`. The components of `ma48_finfo` are only set if `control%coarse_solver = 3`. In this case, `ma48_finfo` is the derived type used by the solver `HSL_MA48` to hold information from `MA48_FACTORIZE`. `ma48_finfo%flag` holds the error flag for `MA48_FACTORIZE`. Full details on the derived type `MA48_FINFO` are given in the user documentation for `HSL_MA48`.

`ma48_sinfo` is a scalar of derived type `MA48_SINFO`. The components of `ma48_sinfo` are only set if `control%coarse_solver = 3`. In this case, `ma48_sinfo` is the derived type used by the solver `HSL_MA48` to hold information from `MA48_SOLVE`. `ma48_sinfo%flag` holds the error flag for `MA48_SOLVE`. Full details on the derived type `MA48_SINFO` are given in the user documentation for `HSL_MA48`.

`getrf_info` is a scalar of type default `INTEGER`. If the LAPACK solver `_GETRF` is used `control%coarse_solver = 4`), `getrf_info` holds the error flag returned by `_GETRF`,

`stat` is a scalar of type default `INTEGER`. On successful exit, `info%stat` is set to `0`. In the event of an allocation or deallocation error, if the Fortran `stat` parameter is available, it is returned in `info%stat`, otherwise `info%stat` is set to `-99`.

2.2.3 Derived data type `ZD11_type` for holding sparse matrices

The derived data type `ZD11_type` must be used to hold the sparse matrix **A**, and is also used by the derived data type `MI20_data` (see Section 2.2.4) to hold matrices at each multigrid level. The matrices are stored in compressed row storage (CRS) format.

The following components of `ZD11_type` are used within `HSL_MI20`.

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

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

`col` is an allocatable array of type default `INTEGER` of rank one that contains the column indices of the entries of the matrix, ordered by rows, with the entries in row 1 preceding those in row 2, and so on.

`val` is an allocatable array of type `REAL` of rank one that contains the entries of matrix in the same order as in `col`.

`ptr` is an allocatable array of type default `INTEGER` of rank one. It holds the starting position of the rows in `col` and `val`, so that for row `i` the column indices are stored in `col(ptr(i):ptr(i+1)-1)` and the corresponding matrix entries in `val(ptr(i):ptr(i+1)-1)`. `ptr(m+1)` is set to be one more than the number of non zero entries in the matrix.

`type` is an allocatable array of rank one and type default `CHARACTER(1)` that is used to indicate matrix properties. Within `HSL_MI20`, this is always set to `general`.

2.2.4 Derived data type `MI20_data` for preconditioning data

The data used to perform the AMG preconditioning is generated by `MI20_setup` and is stored in an array of derived type `MI20_data`. A suitable array called `coarse_data` may be declared as follows:

```
type(MI20_data), allocatable :: coarse_data(:)
```

On exit from `MI20_setup`, `coarse_data(j)` contains two `ZD11_type` scalars, `A_mat` and `I_mat`, that hold the coefficient and interpolation matrices on level `j`, respectively (see Section 5). So, for example, the order of the matrix on the second level is `coarse_data(2)%A_mat%m`. Since memory in `A_mat` and `I_mat` is allocated only when needed, the memory cost of unused entries in the `MI20_data` array is small and we therefore suggest that, if the required number of coarse levels is unknown, the user should choose a large value of `control%max_levels`.

2.3 Argument lists and calling sequences

We use square brackets [] to indicate optional arguments. 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**.

2.3.1 The AMG setup phase

To set up the AMG preconditioner, a call of the following form must be made:

```
call MI20_setup(A, coarse_data, keep, control, info)
```

`A` a scalar of `INTENT(INOUT)` argument of type `ZD11_type`. The user must set the components `m`, `col`, `val`, and `ptr` (see Section 2.2.3). Duplicated and out-of-range entries are not allowed. The diagonal must be present and all diagonal entries must be strictly positive. If **A** is symmetric, the entries in the upper and lower triangular parts must be entered. The components `n` and `type` are set internally by `MI20_setup`, and any existing values are overwritten. On exit, the order of the entries within each row of **A** is changed so the diagonal is the first entry. **Restriction:** `A%m` $\geq$ `1`.

`coarse_data` an allocatable `INTENT(OUT)` argument of type `MI20_data`. On successful exit, it is allocated with size `control%max_levels` and the first `info%clevels` entries contain data for the coarse levels (see Section 2.2.4).

`keep` is a scalar `INTENT(OUT)` argument of type `MI20_keep`. It is used to hold data about the preconditioner and must be passed unchanged to the other subroutines.

`control` a scalar `INTENT(IN)` argument of type `MI20_control` (see Section 2.2.1).

`info` a scalar `INTENT(OUT)` argument of type `MI20_info`. On successful exit, `info%flag` is set to `0` and the other components of `info` contain information about the coarsening (see Section 2.2.2). Note that, although a warning may indicate that coarsening has terminated before the requested number of levels have been computed (see Section 3.4), the data generated on previous coarse levels may be suitable for preconditioning. `info%clevels` is the number of levels that were successfully produced before failure occurred and, provided `info%clevels>0`, the user may continue the computation by making subsequent calls to `MI20_precondition`.

2.3.2 Applying the AMG preconditioner

The AMG v-cycle preconditioner may be applied by making a call as follows.

```
call MI20_precondition(A, coarse_data, z, x, keep, control, info [,ma48_cntl])}
```

`A` is a scalar `INTENT(IN)` argument of type `ZD11_type` that must be unchanged since the call to `MI20_setup`.

`coarse_data` is an array `INTENT(IN)` argument of type `MI20_data` that must be unchanged since the call to `MI20_setup`.

`z` is an array `INTENT(IN)` argument of type `REAL` and size at least the order of **A**. It must be set by the user to hold the vector **z** to which the AMG v-cycle preconditioner **M** is to be applied.

`x` is an array of `INTENT(OUT)` argument of type `REAL` and size at least the order of **A**. On exit, `x` contains **Mz**, where **M** is the AMG v-cycle preconditioner.

`keep` is a scalar `INTENT(INOUT)` argument of type `MI20_keep` that must be passed unchanged by the user.

`control` a scalar `INTENT(IN)` argument of type `MI20_control` (see Section 2.2.1).

`info` a scalar `INTENT(OUT)` argument of type `MI20_info`. On exit, it contains information (see Section 2.2.2).

`ma48_cntl` is an optional argument of type `MA48_CONTROL` and `INTENT(INOUT` that need only be present if `control%coarse_solver = 3` **and** `control%ma48 = 1, 2, 3,` or `4`. If `control%ma48 = 1`, `ma48_cntl` need not be set by the user; on exit, it will have been initialised (by a call to `MA48_INITALIZE`) to the default settings for `HSL_MA48`. If `control%ma48 = 2, 3,` or `4`, the components of `ma48_cntl` must be set by the user or may be passed unchanged from a call with `control%ma48 = 1`; in these cases, `ma48_cntl` is unchanged by the routine. For full details of the derived type `MA48_CONTROL`, the user should refer to the documentation for `HSL_MA48`.

2.3.3 The finalisation subroutine

A call of the following form should be made after all other calls are complete for a problem (including after an error return that does not allow the computation to continue) to deallocate components of the derived data types.

```
call MI20_finalize(coarse_data, keep, control, info)
```

`coarse_data` is an array `INTENT(INOUT)` argument of type `MI20_data`. On successful exit, allocatable components will have been deallocated.

`keep` is a scalar `INTENT(INOUT)` argument of type `MI20_keep` that must be passed unchanged. On exit, allocatable components will have been deallocated.

`control` a scalar `INTENT(IN)` argument of type `MI20_control` (see Section 2.2.1).

`info` a scalar `INTENT(OUT)` argument of type `MI20_info`. On exit, it contains information (see Section 2.2.2).

3 Error Diagnostics

A successful return from a subroutine in the package is indicated by `info%flag` having the value zero. A negative (respectively, positive) value is associated with an error (respectively, warning) message that by default will be output on unit `control%error`. Possible non-zero values are listed below.

3.1 Errors associated with testing the user-supplied matrix

−1 Out-of-range entries found in `A%col`.

−2 One or more diagonal entry is missing.

−3 One or more diagonal entry is ≤ 0.

−4 Either `A%ptr` is not allocated or is allocated with `size(A%ptr)` too small.

−5 Either `A%col` is not allocated or is allocated with `size(A%col) < A%ptr(m+1)-1`.

−6 Either `A%val` is not allocated or is allocated with `size(A%val) < A%ptr(m+1)-1`.

−7 Out-of-range entries found in `A%ptr`.

−8 Duplicate entries found in `A%col` (that is, one or more rows of A has duplicated column indices).

−9 `A%m < 1`.

3.2 Errors associated with out-of-range control parameters

−100 `testing` out-of-range.

−101 `st_parameter` out-of-range.

−102 `err_tol` out-of-range.

−103 `max_points` out-of-range.

−104 `st_method` out-of-range.

−105 `aggressive` out-of-range.

−106 `c_fail` out-of-range.

−107 `v_iterations` out-of-range.

−108 `smoother` out-of-range.

−109 `pre_smoothing` out-of-range.

−110 `post_smoothing` out-of-range.

−111 `pre_smoothing` + `post_smoothing` = 0.

−112 `coarse_solver` out-of-range.

−113 `coarse_solver_its` out-of-range.

−114 `print_level` out-of-range.

−115 `damping` out-of-range.

−116 `max_levels` out-of-range.

−117 `ma48` out-of-range.

−118 `trunc_parameter` out-of-range.

−119 `reduction` out-of-range.

3.3 Other possible error returns

-10 Allocation error at first level of multigrid process.

-11 Deallocation error at first level of multigrid process.

-12 The coarsening has failed. This is because one or more rows of the user-supplied matrix has at least one strictly positive entry and no negative off-diagonal entries (`control%c_fail = 1`) or all the rows have at least one strictly positive entry and no negative off-diagonal entries (`control%c_fail = 2`). The coarsening may also fail if there are one or more rows with negative off-diagonal entries (that is, rows with strong connections) that are connected to rows with no negative off-diagonals (that is, to rows with no connections).

-14 The action of the AMG preconditioner on the user-supplied vector `z` caused an increase in the 2-norm of the vector greater than `control%err_tol` (`MI20_precondition` only).

-15 Call to `MI20_precondition` follows an unsuccessful call to `MI20_setup`.

-16 Error return from `MI20_precondition` because `size(x)` and/or `size(z)` is less than the order of **A**.

-17 Error return from `_GETRF` (see `info%getrf_info`).

-18 Error return from `HSL_MA48` (a negative value for `info%ainfo%flag`, `info%finfo%flag`, or `info%sinfo%flag` indicates which phase of the solver returned the error; see Section 2.2.2).

-19 Error return from `MI20_precondition` because `control%coarse_solver` = `3` and `control%ma48` = `1`, `2`, `3` or `4` but `ma48_cntl` is not present.

3.4 Warnings

If a warning `info%flag = 10`, `11`, `12`, or `13` is issued by `MI20_setup`, a preconditioner has been computed but coarsening terminated prematurely; the number of coarse levels is `info%clevels`. After a warning has been issued, the user may continue the computation by calling `MI20_precondition`. The following warnings may be issued by `MI20_setup`:

1 Method used to find strong transpose connections changed from method `1` to method `2` (see `control%st_method` in Section 2.2.1).

10 Coarsening terminated because of an allocation error.

11 Coarsening terminated because of a deallocation error.

12 Coarsening terminated. This is because one or more rows of the coarse level matrix had at least one strictly positive entry but no negative off-diagonal entries (`control%c_fail = 1`) or because all the rows had at least one strictly positive entry but no negative off-diagonal entries (`control%c_fail = 2`). The coarsening may also terminate if there are one or more rows with negative off-diagonal entries (that is, rows with strong connections) that are connected to rows with no negative off-diagonals (that is, to rows with no connections).

13 Coarsening terminated because the requirement that the number of points n_c and n_f on two successive levels should satisfy $n_c \leq n_f$`*reduction` was not met. See Section 5.1.1.

The following warning may be issued by `MI20_precondition`:

20 The number of requested levels `control_levels` is greater than the number of available levels (`info%clevels`). The number of levels used is `info%clevels`.

4 GENERAL INFORMATION

Workspace: Provided automatically by the module.

Other modules used directly: `HSL_MA48`, `HSL_MC65`, `HSL_ZD11`, and `LAPACK` routines `_GETRF` and `_GETRS`.

Input/output: Output is provided under the control of `control%print_level`. In the event of an error or warning, diagnostic messages are printed. The output units for these messages are controlled by `control%print`, and `control%error` (see Section 2.2.1).

Restrictions: `A%m` $\geq$ `1`,
`size(A%col)` $\geq$ `A%ptr(m+1)-1`,
`size(A%val)` $\geq$ `A%ptr(m+1)-1`,
`control%aggressive`$\geq$ 1,
`control%c_fail` = `1` or `2`,
`control%max_levels`$\geq$ 1,
`control%max_points`$>$ 0,
0.5 $\leq$ `control%reduction`$\leq$ 1.0,
`control%st_method` = `1` or `2`,
0.0 $\leq$`control%st_parameter`$\leq$ 1.0,
`control%testing` = `0` or `1`,
`control%coarse_solver` = `1`, `2`, `3`,
`control%coarse_solver_its`$>$ 0,
0.0 $<$`control%damping`$\leq$ 1.0,
0.0 $\leq$`control%trunc_parameter`$<$ 1.0,
`control%err_tol`$>$ 0,
`control%ma48` = `1`, `2`, `3` or `4`,
`control%pre_smoothing`$\geq$ 0, `control%post_smoothing`$\geq$ 0,
`control%pre_smoothing` + `control%post_smoothing`$\neq$ 0,
`control%smoother` = `1` or `2`,
`control%v_iterations`$\geq$ 1,
`control%print_level` = `1`, `2`, or `3`.

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

5 METHOD

The classical AMG algorithm implemented by `HSL_MI20` is described in detail in Section 7 of [1]. This implementation of AMG ignores positive off-diagonal entries during coarsening, adds any positive off-diagonals to the diagonal when calculating interpolation weights, and uses direct interpolation.

The AMG method was originally devised as a linear solver and it is more intuitive to think of AMG as a solver when describing the method. It is also important to note that AMG may provide a good preconditioner even when it fails as a solver (for example, block preconditioning within a larger system).

Consider the linear system $\mathbf{Au} = \mathbf{f}$. At the heart of multigrid is a series of ever coarser representations of the matrix $\mathbf{A}$. Given an approximation $\hat{\mathbf{u}}$ to the solution $\mathbf{u}$, consider solving $\mathbf{Ae} = \mathbf{r}$ to find the error $\mathbf{e}$, where $\mathbf{r}$ is the residual $\mathbf{r} = \mathbf{f} - \mathbf{A}\hat{\mathbf{u}}$. Multigrid applies a 'smoother' (such as Gauss-Seidel or damped Jacobi) to remove high frequency components of the error vector. The problem can then be represented by a smaller (coarser) system $\mathbf{A}_c\mathbf{e}_c = \mathbf{r}_c$, which is cheaper to solve. This idea can be applied recursively, producing a series of coarse levels and coarse error corrections to the solution. The coarsest (smallest) level is solved using a direct method or a simple iterative scheme such as Gauss-Seidel. The coarse level solution must then be prolonged to each of the finer levels. When used as a linear solver, the whole multigrid process is applied iteratively until a solution with the desired tolerance is obtained. The method can also be used to efficiently precondition a linear system. Typically, preconditioning is a single multigrid iteration.

Multigrid requires some means of producing coarse level coefficient matrices $\mathbf{A}_c$, together with a means of transferring the residual and error vectors between the levels; in AMG this is devised algebraically.

5.1 Setup

Coarse level matrices $\mathbf{A}_c$ and the interpolation/restriction matrices $\mathbf{I}_{cf}$ are created in the setup phase which comprises the following steps.

5.1.1 Finding F and C points

To describe AMG coarsening, we associate rows of the matrix $\mathbf{A} = \{a_{ij}\}$ with points (so that row i is associated with point i) and consider connections between points. We say that point i is **connected** to point j if $a_{ij} < 0$.

To generate the next coarse level, points are divided into C points (those points which will exist on the next level), F points (which must interpolate their values from the C points), and unconnected points. This division is based upon strong connections. If i is connected to j and $|a_{ij}| \geq \theta \max\{|a_{ik}| : a_{ik} < 0\}$, where $0 < \theta \leq 1$ corresponds to the control parameter `control%st_parameter`, then we say that i has a **strong connection** to j and j has a **strong transpose connection** to i.

After removing unconnected points, each step of the coarsening process proceeds as follows. Each undecided point has a weight that is initially the number of its strong transpose connections. An undecided point with maximum weight is chosen to become a new C point, and points with a strong transpose connection to the new C point become F points. The weights are then increased by the number of strong connections to the new F points. This process is repeated until all points are assigned as either F or C points, or until all remaining undecided points have a weight of zero. If `control%one_pass_coarsen` is equal to `.false.`, further checking is performed that aims to improve the quality of the coarsening by possibly making additional points into C points. If n_c and n_f are the number of C and F points, respectively, the coarsening has **stagnated** if $n_c \geq n_f *$`control%reduction` and, in this case, is terminated. Otherwise, coarsening continues until either the requested maximum number of levels has been reached (`control%max_levels`) or the number of points has been reduced below a chosen threshold (`control%max_points`).

Throughout the coarsening, it is necessary to know the strong connections (for F points) and the strong transpose connections (for C points). It is easy to test whether connections to a point are strong (for point i the data required is contained in row i of the matrix, and this is available since we hold the matrices in compressed row storage (CRS) format). Testing for strong transpose connections is not so straightforward. To find the strong transpose connections to point j, we need to know which rows have a non-zero entry in column j. For a general sparse matrix held in CRS format, we must run through the entire matrix, checking each row. However, if the matrix has a symmetric sparsity pattern, things are simpler, since if row i has an entry in column j, row j has an entry in column i.

To reduce the number of non-zero entries in the coarse level matrices, the interpolation weights may be truncated. If `control%trunc_parameter > 0.0`, interpolation weights will be removed from the interpolation matrix if their value is less than or equal to `control%trunc_parameter` times the largest interpolation weight in their row of the interpolation matrix. After this, remaining weights are scaled so that row sums remains unchanged

For matrices with a symmetric sparsity pattern, `HSL_MI20` offers two methods for finding strong transpose connections. The first, selected by setting `control%st_method = 1`, performs testing as needed; this method is often the fastest. The alternative method (`control%st_method` = `2`) finds strong transpose connections before coarsening and then stores the information; this method is always used when **A** has an unsymmetric sparsity pattern.

5.1.2 Generate interpolation matrix and A_c

The direct interpolation method is used to calculate the interpolation weights, and this is fully described in [1], with positive off diagonals effectively removed by adding them to the diagonal. The coarse level coefficient matrix $\mathbf{A}_c$ is generated from the fine level matrix $\mathbf{A}_f$ and the interpolation matrix $\mathbf{I}_{cf}$ using the Galerkin relation $\mathbf{A}_c = \mathbf{I}_{cf}^T \mathbf{A}_f \mathbf{I}_{cf}$. If `control%aggressive > 1`, more than one coarsening step is performed before $\mathbf{A}_c$ is calculated; this is known as **aggressive** coarsening.

5.2 Preconditioning phase

`MI20_precondition` takes a user-supplied vector $\mathbf{z}$ and returns $\mathbf{x} = \mathbf{Mz}$, where $\mathbf{M}$ is the AMG preconditioner. The preconditioner performs `control%v_iterations` v-cycles. Denoting the interpolation matrix from level k to level $k+1$ by $\mathbf{I}_k^{k+1}$ and the matrix on level k by $\mathbf{A}_k$, the v-cycles are performed within `HSL_MI20` as follows:

```
User-supplied $\mathbf{z}$.
Initialise $\mathbf{x} = 0$;   $\mathbf{A}_1 = \mathbf{A}$;   $its = 0$.
do outer
    if ($its == max\_its$) stop
    $\mathbf{z}_1 = \mathbf{z} - \mathbf{Ax}$
    do $k = 1, ml - 1$
        Initialise $\mathbf{e}_k = 0$
        Pre-smooth $\mathbf{e}_k$ (using damped Jacobi or Gauss-Seidel)
        Compute $\mathbf{r}_k = \mathbf{z}_k - \mathbf{A}_k \mathbf{e}_k$
        Restrict $\mathbf{r}_{k+1} = \mathbf{I}_k^{k+1} \mathbf{r}_k$
        Set $\mathbf{z}_{k+1} = \mathbf{r}_{k+1}$
    end do
    Solve the coarse grid error correction problem $\mathbf{A}_{k+1} \mathbf{e}_{k+1} = \mathbf{z}_{k+1}$
    do $k = ml - 1, 1, -1$
        Prolong and then update the error correction $\mathbf{e}_k \leftarrow \mathbf{e}_k + (\mathbf{I}_k^{k+1})^T \mathbf{e}_{k+1}$
        Post-smooth $\mathbf{e}_k$ (using damped Jacobi or Gauss-Seidel with the sweep direction reversed)
    end do
    Update $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{e}_1$
    $its = its + 1$
end outer
```

Here *max_its* and *ml* are the control parameters `control%v_iterations` and `control%max_levels`, respectively.

Further details of the method implemented within `HSL_MI20`, together with numerical results are given in [2].

Reference:

[1] K. Stüben. *An introduction to algebraic multigrid.* In U. Trottenberg, C. Oosterlee, A. Schüller, eds, 'Multigrid', Academic Press, 2001, pp 413-532.

[2] J. Boyle, M. D. Mihajlovic and J. A. Scott. `HSL_MI20`: an efficient AMG preconditioner. Rutherford Appleton Technical Report.

6 EXAMPLE OF USE

Suppose we wish to use preconditioned conjugate gradients to solve the linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is the symmetric tridiagonal matrix of order 10 with 2's on the main diagonal and -1's on the off diagonals, and $\mathbf{b}$ is the vector of 1's. Then we may use the following code:

```
program mi20_example
      use hsl_mi20_double
      use hsl_zd11_double
      use hsl_mc65_double

      implicit none
      integer, parameter :: wp = kind(1.0d0)
      integer, parameter :: m = 10 ! size of system to solve

! derived types
      type(zd11_type) :: a
      type(mi20_data), dimension(:), allocatable :: coarse_data
      type(mi20_control) :: control
      type(mi20_info) :: info
      type(mi20_keep) :: keep
      type(ma48_control) :: ma48_cntl

! Arrays and scalars required by the CG code mi21
      real(kind=wp) :: cntl(5),rsave(6)
      integer :: icntl(8),isave(10),info21(4)
      real(kind=wp) :: w(m,4)
      real(kind=wp) :: resid
      integer :: locy, locz, iact

      external mi21id, mi21ad

      integer :: info65  ! mc65 error flag

! generate matrix A
      call matrix_gen(a, m)

! Prepare to use the CG code mi21 with preconditioning
      call mi21id(icntl, cntl, isave, rsave)
      icntl(3) = 1

! set right hand side to vector of ones
      w(:,1) = 1

! call mi20_setup
      call mi20_setup(a, coarse_data, keep, control, info)
      if (info%flag < 0) then
        write(*,*) "Error return from mi20_setup"
        stop
      end if

! solver loop
      iact = 0
      do
        call mi21ad(iact, m, w, m, locy, locz, resid, icntl, cntl, info21, &
          isave, rsave)
        if (iact == -1) then
          write(*,*) "Error in solver loop"
          exit

        else if (iact == 1) then
          write(*,'(a,i3,a)') " Convergence in ", info21(2), " iterations"
          write(*,'(a,es12.4)') " 2-norm of residual =", resid
          exit

        else if (iact == 2) then
          call mc65_matrix_multiply_vector(a, w(:,locz), w(:,locy), info65)

        else if (iact == 3) then
          call mi20_precondition(a, coarse_data, w(:,locz), w(:,locy), keep, &
               control, info, ma48_cntl)
          if (info%flag < 0) then
            write(*,*) "Error return from mi20_precondition"
            exit
          end if

        end if
      end do

! deallocation
      call mi20_finalize(coarse_data, keep, control, info)
      deallocate(a%col,a%val,a%ptr)

contains
      subroutine matrix_gen(a, m)
      integer, intent(in) :: m ! size of matrix
      type(zd11_type), intent(out) :: a
      integer :: i,nnz,p

      nnz = m + 2*(m-1)
      allocate(a%col(nnz),a%val(nnz),a%ptr(m+1))
      a%m = m
      p = 1    ! pointer to next empty position
      do i = 1,m
        a%ptr(i) = p
        if (i==1) then ! first row
          a%col(p) = i;    a%col(p+1) = i+1
          a%val(p) = 2.0;  a%val(p+1) = -1.0
          p = p+2
        else if (i==m) then ! last row
          a%col(p) = i-1;  a%col(p+1) = i
          a%val(p) = -1.0; a%val(p+1) = 2.0
          p = p+2
        else
```

```
          a%col(p) = i-1;  a%col(p+1) = i; a%col(p+2) = i+1
          a%val(p) = -1.0; a%val(p+1) = 2.0; a%val(p+2) = -1.0
          p = p+3
        end if
      end do
      a%ptr(m+1) = nnz+1
      end subroutine matrix_gen

end program mi20_example
```

This produces the following output:

```
 Convergence in   5 iterations
 2-norm of residual =  5.0557E-10
```