1 SUMMARY

Given an $n \times n$ sparse matrix $\mathbf{A}$ and an $n-$vector $\mathbf{z}$, HSL_MI20 computes the vector $\mathbf{x} = \mathbf{M}\mathbf{z}$, 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 2003 subset (F95 + TR15581 + C interoperability) **Remark:** The development of HSL_MI20 was funded by EPSRC grants EP/C000528/1 and GR/S42170.

2 HOW TO USE THE PACKAGE

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 can 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 is 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.

2.1 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"
```

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

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
```

The following procedures are available to the user:

`mi20_setup` takes the matrix **A** and generates data that is required by the AMG preconditioner.

`mi20_precondition` performs the preconditioning operation $\mathbf{x} = \mathbf{Mz}$, where **M** is the AMG preconditioner and **z** is a user-supplied vector.

`mi20_finalize` should be called after all other calls are complete for a problem to free memory associated with the `keep` variable.

2.2 The derived data types

For each problem, the user must employ the structures defined in the header file to declare scalars of the types `mi20_control`, `mi20_info` and a `void *` pointer for `keep`. The following pseudocode illustrates this.

```
#include "hsl_mi20d.h"
...
void *keep;
struct mi20_control control;
struct mi20_info info;
...
```

The members of `mi20_control` and `mi20_info` are explained in Sections 2.2.1 and 2.2.2. The `void *` pointer is used to pass data between the subroutines of the package and must not be altered by the user.

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 members, which may be given default values through a call to `mi20_default_control`, are as follows.

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` and `col` is assumed. Otherwise, if `f_arrays=0` (i.e. evaluates to false), 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).

Controls used by `mi20_setup` (in alphabetical order)

`int aggressive` 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`.

`int c_fail` 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`.

int max_levels 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.

int max_points controls termination of coarsening. 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.

int one_pass_coarsen indicates whether one pass coarsening is used. If one_pass_carsen!=0 (i.e. evaluates 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. The default is one_pass_coarsen=0 (false).

pkgtype reduction controls reduction in coarsening. 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.

int st_method 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.

pkgtype st_parameter 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$\leq$ 1.0.

int testing 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.

pkgtype trunc_parameter 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)**

int coarse_solver 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.

int 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.

pkgtype damping specifies the damping factor used by the damped Jacobi smoother (control.smoother = 1). The default is 0.8. **Restriction:** 0.0 $<$damping$\leq$ 1.0.

pkgtype err_tol 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.

int levels 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.

int pre_smoothing 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≥ 0 and pre_smoothing + post_smoothing$\neq 0$.

int post_smoothing 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≥ 0 and pre_smoothing + post_smoothing$\neq 0$.

int smoother 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.

int v_iterations controls the number of v-cycle iterations to be performed. The default is 1. **Restriction:** v_iterations≥ 1.

Printing controls

int error holds the Fortran unit number for the printing of error and warning messages. Printing is suppressed if error < 0. The default is 6.

int print holds the Fortran unit number for diagnostic printing. Printing is suppressed if print < 0. The default is 6.

int print_level 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 derived data type mi20_info is used to provide information about the progress of the algorithm. The members of mi20_info are:

int flag is used as a error and warning flag. See Section 3 for details.

int clevels contains the number of coarse levels generated after a call to mi20_setup,

int cpoints contains the order of the matrix on the coarsest level after a call to mi20_setup,

int cnnz contains the number of nonzeros in the matrix on the coarsest level after a call to mi20_setup,

int getrf_info holds the error flag returned by the LAPACK solver _GETRF if it is used (control.coarse_solver = 4).

int stat holds, on a relevant error, the Fortran stat parameter for a failed allocation or deallocation; in the event that the stat parameter is not available, it is set to -99. On a successful exit, stat is set to 0.

2.2.3 The AMG setup phase

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

```
void mi20_setup(int n, const int ptr[], const int col[], const pkgtype val[],
      void **keep, const struct mi20_control *control, struct mi20_info *info);
```

`n` must hold the number of rows of A. **Restriction:** `n` $\geq$ `1`.

`ptr` is a rank-1 array of size `n+1`. `ptr[j]` must be set by the user so that `ptr[j]` is the position in `col` of the first entry of row `j` (`0`$\leq$`j`$\leq$`n-1`) and `ptr[n]` must be set to the number of matrix entries being input by the user. This argument is not altered by the subroutine.

`col` is a rank-1 array of size `ptr[n]`. It must hold the column indices of the entries of A with the column indices for the entries in row 0 preceding those for row 1, and so on (within each row, the column indices may be in arbitrary order). It should contain no duplicates or out-of-range entries. 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. This argument is not altered by the subroutine.

`keep` 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 preconditioner. It must be passed unchanged to the other subroutines. To avoid a memory leak, the subroutine `mi20_finalize` must be used to clean up and deallocate this memory when the preconditioner is no longer required.

`control` is used to control the actions of the package, see Section 2.2.1.

`info` is used to return information about the execution of the package as explained in Section 2.2.2. On successful exit, `info.flag` is set to `0`, otherwise it indicates the error condition encountered. 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`.

2.2.4 Applying the AMG preconditioner

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

```
void mi20_precondition(const pkgtype rhs[], pkgtype solution[],
      void **keep, const struct mi20_control *control, struct mi20_info *info);
```

`rhs` is a rank-1 array with size `n`. It must be set by the user to hold the vector **z** to which the AMG v-cycle preconditioner **M** is to be applied.

`solution` is a rank-1 array with size `n`. On exit, `x` contains **Mz**, where **M** is the AMG v-cycle preconditioner.

`keep` must be unchanged since the call to `mi20_setup`. It is not altered by this subroutine.

`control`, `info`: see Section 2.2.3.

2.2.5 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 free memory associated with `keep`.

```
void mi20_finalize(void **keep, const struct mi20_control *control,
      struct mi20_info *info);
```

`keep` must be unchanged since the call to `mi20_setup`. On exit, `*keep` will be set to `NULL`.

`control`, `info`: see Section 2.2.3.

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 Fortran 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 `col`.

-2 One or more diagonal entry is missing.

-3 One or more diagonal entry is ≤ 0.

-7 Out-of-range entries found in `ptr`.

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

-9 `n < 1`.

3.2 Errors associated with out-of-range control parameters

-100 `control.testing` out-of-range.

-101 `control.st_parameter` out-of-range.

-102 `control.err_tol` out-of-range.

-103 `control.max_points` out-of-range.

-104 `control.st_method` out-of-range.

-105 `control.aggressive` out-of-range.

-106 `control.c_fail` out-of-range.

-107 `control.v_iterations` out-of-range.

-108 `control.smoother` out-of-range.

-109 `control.pre_smoothing` out-of-range.

-110 `control.post_smoothing` out-of-range.

−111 control.pre_smoothing + post_smoothing = 0.

−112 control.coarse_solver out-of-range.

−113 control.coarse_solver_its out-of-range.

−114 control.print_level out-of-range.

−115 control.damping out-of-range.

−116 control.max_levels out-of-range.

−118 control.trunc_parameter out-of-range.

−119 control.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.

−17 Error return from _GETRF (see info.getrf_info).

−18 Error return from HSL_MA48.

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 $n_c \leq n_f * \texttt{reduction}$ 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: `n` $\geq$ 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.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 2003 subset (F95 + TR15581 + C interoperability)

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 **z** and returns $\mathbf{x} = \mathbf{M}\mathbf{z}$, where **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 **z.**
Initialise $\mathbf{x} = 0$; $\mathbf{A}_1 = \mathbf{A}$; $its = 0$.
do outer
  if $(its == max\_its)$ **stop**
  $\mathbf{z}_1 = \mathbf{z} - \mathbf{A}\mathbf{x}$
  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:

```
#include <stdlib.h>
#include <stdio.h>
#include "hsl_mi20d.h"

/* MI21 Fortran routines (no C interface available) */
/* As these are F77 style codes, we assume that the C binding merely appends
   an underscore to the name of the Fortran routine, and that all data types
   match and are pass by reference. THIS WILL NOT WORK ON ALL COMPILERS. */
void mi21id_(int *icntl, double *cntl, int *isave, double *rsave);
void mi21ad_(int *iact, const int *n, double *w, const int *ldw, int *locy,
      int *locz, double *resid, int *icntl, double *cntl, int *info, int *isave,
      double *rsave);

/* Generate example matrix */
void matrix_gen(const int n, int **ptr, int **col, double **val) {
   int i,nnz,p;

   nnz = n + 2*(n-1);
   *ptr = (int*) malloc((n+1)*sizeof(int));
   *col = (int*) malloc(nnz*sizeof(int));
   *val = (double*) malloc(nnz*sizeof(double));
   p = 0; /* pointer to next empty position */
   for(i=0; i<n; i++) {
     (*ptr)[i] = p;
     if (i==0) {   /* first row */
       (*col)[p] = i;    (*col)[p+1] = i+1;
       (*val)[p] = 2.0;  (*val)[p+1] = -1.0;
       p = p+2;
     } else if (i==n-1) { /* last row */
       (*col)[p] = i-1;  (*col)[p+1] = i;
       (*val)[p] = -1.0; (*val)[p+1] = 2.0;
       p = p+2;
     } else {
       (*col)[p] = i-1;  (*col)[p+1] = i;   (*col)[p+2] = i+1;
       (*val)[p] = -1.0; (*val)[p+1] = 2.0; (*val)[p+2] = -1.0;
       p = p+3;
     }
   }
   (*ptr)[n] = nnz;
}

/* Calculate b = Ax */
void spmv(int n, const int ptr[], const int col[], const double val[],
      double b[], const double x[]) {
```

```
   int i, j;

   for(i=0; i<n; i++) {
      b[i] = 0;
      for(j=ptr[i]; j<ptr[i+1]; j++) {
         b[i] += val[j] * x[col[j]];
      }
   }
}

int main(void) {
   const int n = 10; /* size of system to solve */

   /* matrix data */
   int *ptr, *col;
   double *val;

   /* derived types */
   void *keep;
   struct mi20_control control;
   struct mi20_info info;

   /* Arrays and scalars required by the CG code mi21 */
   double cntl[5], rsave[6];
   int icntl[8],isave[10],info21[4];
   double w[n*4];
   double resid;
   int locy, locz, iact, i;

   /* generate matrix A */
   matrix_gen(n, &ptr, &col, &val);

   /* Prepare to use the CG code mi21 with preconditioning */
   mi21id_(icntl, cntl, isave, rsave);
   icntl[3-1] = 1;

   /* set right hand side to vector of ones */
   for(i=0; i<n; i++) w[i] = 1;

   /* initalize control */
   mi20_default_control(&control);

   /* call mi20_setup */
   mi20_setup(n, ptr, col, val, &keep, &control, &info);
   if (info.flag < 0) {
     printf("Error return from mi20_setup\n");
     return 1;
   }
```

```
   /* solver loop */
   iact = 0;
   while(1) {
      mi21ad_(&iact, &n, w, &n, &locy, &locz, &resid, icntl, cntl, info21,
         isave, rsave);
      if (iact == -1) {
         printf("Error in solver loop\n");
         break;
      } else if (iact == 1) {
         printf(" Convergence in %d iterations\n", info21[2-1]);
         printf(" 2-norm of residual = %e\n", resid);
         break;
      } else if (iact == 2) {
         spmv(n, ptr, col, val, &w[n*(locy-1)], &w[n*(locz-1)]);
      } else if (iact == 3) {
         mi20_precondition(&w[n*(locz-1)], &w[n*(locy-1)], &keep, &control,
            &info);
         if (info.flag < 0) {
            printf("Error return from mi20_precondition\n");
            break;
         }
      }
   }

   /* deallocation */
   mi20_finalize(&keep, &control, &info);
   free(ptr); free(col); free(val);

   return 0; /* sucess */
}
```

This produces the following output:

```
 Convergence in 5 iterations
 2-norm of residual = 5.055712e-10
```