Science & Technology
Facilities Council

HSL_MI20: an efficient AMG preconditioner

J. Boyle, M. D. Mihajlović and J. A. Scott

December 2007

RAL-TR-2007-021

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

HSL_MI20: an efficient AMG preconditioner[1]

by

J. Boyle[2,3], M. D. Mihajlović[4] and J. A. Scott[5,6]

Abstract

Algebraic multigrid (AMG) is an efficient multilevel method for solving large sparse linear systems obtained from the discretisation of scalar elliptic partial differential equations. AMG can be used to compute powerful preconditioners for use with Krylov subspace methods. We report on the design and development of an efficient, robust and portable implementation of AMG that is available as package HSL_MI20 within the HSL mathematical software library. HSL_MI20 implements the classical (Ruge-Stüben) AMG method and, although it can be used as a "black-box" preconditioner, it offers the user a large number of options and parameters that may be tuned to enhance its performance for specific applications. The performance of HSL_MI20 is illustrated using finite element discretisations of diffusion and convection-diffusion problems in three dimensions.

Keywords: large sparse linear systems, Krylov subspace methods, algebraic multigrid, preconditioning, Fortran 95.

---

[1] Current reports available from "http://www.numerical.rl.ac.uk/reports/reports.html".

[2] School of Mathematics, University of Manchester,
Alan Turing Building, Manchester M13 9PL, England, UK.
Email: j.boyle@manchester.ac.uk

[3] Boyle was supported by the EPSRC grant EP/C000528/1.

[4] School of Computer Science, University of Manchester,
Kilburn Building, Manchester M13 9PL, England, UK.
Email: milan@cs.man.ac.uk

[5] Computational Science and Engineering Department, Rutherford Appleton Laboratory,
Chilton, Oxfordshire, OX11 0QX, England, UK.
Email: j.a.scott@rl.ac.uk

[6] Scott was supported by the EPSRC grant GR/S42170.

Computational Science and Engineering Department,
Atlas Centre, Rutherford Appleton Laboratory,
Oxon OX11 0QX, England.

December 17, 2007.

Contents

1 Introduction to AMG

Multigrid methods are attractive for solving certain classes of large-scale sparse linear systems of equations since they aim to solve such systems of order n in $\mathcal{O}(n)$ time, thus exhibiting a (near-optimal) scaling. This near optimality is achieved by employing two complementary processes: smoothing and coarse-grid correction. Smoothing involves applying a smoother (also called a relaxation method), which is generally a few steps of a simple iterative method such as Gauss-Seidel or damped Jacobi. Coarse-grid correction involves transferring information about the approximate solution residual to a coarser grid through restriction, solving a coarse-grid system of equations, then transferring the correction back to the fine grid through interpolation (also called prolongation). This is generally followed by additional post smoothing steps. In a classical geometric multigrid method, smoothing reduces high frequency (or oscillatory) components of the error, while the low frequency (or smooth) components are tackled using an auxiliary lower-resolution version of the problem (coarse-grid). This geometric interpretation of multigrid was critical to its early development (see, for example, [2]) but limits the problems it can be used to solve to those that are discretized on a sequence of nested grids. For problems on complex domain geometries it is difficult to construct such grid hierarchies. Moreover, the book-keeping of the geometric information associated with such a grid hierarchy is memory-consuming and requires sophisticated data structures. This can adversely impact the performance of software that implements them. Algebraic multigrid (AMG) was introduced as an approach to solving linear systems based on multigrid principles, but in a way that does not explicitly use the geometry of grids (and allows it to be used for linear systems that are not associated with an underlying grid).

Although our main interest is in developing an AMG preconditioner, 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. Thus, we start by considering the linear system

$$Ax = b \tag{1.1}$$

where A is an $n \times n$ sparse matrix, $b \in \mathcal{R}^n$ is the given right-hand side vector, and $x \in \mathcal{R}^n$ is the solution vector. At the heart of AMG is a series of progressively coarser (smaller) representations of the matrix A. Given an approximation $\hat{x}$ to the solution x, consider solving the residual equation $Ae = r$ to find the error e, where r is the residual vector $r = b - A\hat{x}$. A fundamental principle of AMG is that of *algebraically smooth error*. This principle is designed to mimic the notion of low-frequency error components introduced in geometric multigrid. An algebraically smooth error is an error that cannot be reduced effectively by the application of simple fixed-point iteration methods (smoothers) [4, p. 138]. A plot of an algebraically smooth error may not look smooth. To reduce the algebraically smooth errors further, they need to be represented by a smaller defect equation (coarse grid residual equation) $A_c e_c = r_c$, which is cheaper to solve. As in geometric multigrid, coarse error correction is the process of approximately solving the coarse-level defect equation and then interpolating the correction back to the fine level. The interpolation back to the fine level introduces parasite error components. Fortunately, these can be eliminated efficiently by a few post-smoothing steps. The whole process can be applied recursively, producing a hierarchy of coarse levels and a series of coarse error corrections to the solution. The coarsest (smallest) level may be solved using either a direct method or a simple iterative scheme such as Gauss-Seidel, and the coarse level correction to the solution must then be successively 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.

Clearly, the AMG method requires some means of constructing the $n_c \times n_c$ coarse level coefficient matrix A_c, together with a method of transferring the residual and error vectors between the coarse and fine levels. A restriction operator maps the fine level to the coarse level and may be represented by an $n_c \times n$ matrix $I_{fc} : \mathcal{R}^n \to \mathcal{R}^{n_c}$. Similarly, an interpolation (or prolongation) operator represented by an $n \times n_c$ matrix $I_{cf} : \mathcal{R}^{n_c} \to \mathcal{R}^n$ maps the coarse level to the fine level. Setting $A_f = A$ and $n_f = n$, we list the main components an AMG algorithm in Table 1.1. Assuming these components have been constructed,

Table 1.1: Components of an AMG algorithm

Fine level vertices:	$\Omega_f = 1 : n_f$
Splitting of vertices:	$\Omega_f \rightarrow C \cup F$ $(C \cap F = \emptyset)$
Coarse level vertices:	$\Omega_c = C$
Smoothers:	$\mathcal{S}$ simple stationary iterative methods
Restriction and interpolation operators:	I_{fc} and I_{cf}
Coarse level matrix:	$A_c := I_{fc} A_f I_{cf}$
Coarse level solve:	Direct or iterative solver

the basic two-level AMG correction scheme proceeds as in Figure 1.1 and this cycle is repeated until convergence.

In practice, the solve at Step 5 is itself replaced by a two-level procedure, leading to the recursive definition of a V-cycle multigrid method. In Step 1, $\mathcal{S}_{pre}^{\nu_1}(x_f, A_f, b_f)$ denotes the application of ν_1 iterations of the chosen pre-smoother to the linear system $A_f x_f = b_f$. Similarly, $\mathcal{S}_{post}^{\nu_2}(x_f, A_f, b_f)$ is the application of ν_2 iterations of the chosen post-smoother. In the symmetric case, if Gauss-Seidel smoothing is used, it is important that the sweep direction is reversed for the post-smoothing; this ensures symmetry of the preconditioning operator is retained. In this paper, we follow the standard approach of always defining the restriction operator to be the transpose of the interpolation operator,

$$I_{fc} = I_{cf}^T.$$

If $A = A_f$ is symmetric positive definite, it then immediately follows that the Galerkin matrix $A_c = I_{cf}^T A_f I_{cf}$ is also symmetric positive definite, independent of the choice of I_{cf} (provided it is of full rank).

Figure 1.1: One cycle of a two-level AMG correction scheme

1. Pre-smooth on the fine level $x_f \leftarrow \mathcal{S}_{pre}^{\nu_1}(x_f, A_f, b_f)$
2. Calculate the residual $r_f = b_f - A_f x_f$
3. Restrict r_f to the coarse level $r_c = I_{cf} r_f$
4. Compute the coarse level matrix $A_c = I_{fc} A_f I_{cf}$
5. Solve the coarse level residual equation $A_c e_c = r_c$
6. Prolong the coarse level correction e_c onto the fine level $e_f = I_{fc} e_c$
7. Update the solution $x_f \leftarrow x_f + e_f$
8. Post-smooth on the fine level $x_f \leftarrow \mathcal{S}_{post}^{\nu_2}(x_f, A_f, b_f)$

The AMG method can be shown theoretically to converge for systems for which the coefficient matrix A is an **M-matrix** (two-level convergence results in this case can be found in [18, p. 444]). A symmetric positive definite matrix $A = \{a_{ij}\}$ is an M-matrix if $a_{ii} > 0$ for all i and $a_{ij} \leq 0$ for all i, j, that is, the diagonal entries are all strictly positive and the off-diagonal entries are non-positive. In addition, an inverse of an M-matrix is a strictly positive matrix (see [16, p. 28]). An example is the Laplacian operator in two dimensions discretised using linear finite elements on a uniform triangular grid, provided that no triangle has an internal angle larger than 90°, or using bi-linear quadrilateral finite elements, provided that the grid is not too stretched[1]. For more general linear systems that do not violate the M-matrix properties too strongly (such as arise from the discretisation of a perturbed elliptic operator), reasonably good performance may be expected. Alternatively, the AMG algorithm can be used to compute a preconditioner

[1]In particular, if the ratio of the sides of a rectangular element is greater than $\sqrt{2}$, positive off-diagonal elements will appear.

for use with an iterative solver, such as the conjugate gradient (CG) method or GMRES (see, for example, [16, Ch. 6]). Importantly, an AMG preconditioner can be used as a black-box preconditioner, that is, without tuning parameters. When AMG is used as a preconditioner, each time an application of the preconditioner is required by the iterative method, a small, fixed number (typically 1 or 2) V-cycles is performed. In this case, the right hand side vector is the current residual $r^{(k)} = b - Ax^{(k)}$, where $x^{(k)}$ is the value of current approximation to the solution x (see, for example, [16, p. 262]).

The purpose of this paper is discuss the design of a new software package `HSL_MI20` that computes and applies a black-box AMG preconditioner. `HSL_MI20` has been developed for the mathematical software library HSL [10], and may optionally be used with the iterative solvers that are available within HSL. We describe the overall design of `HSL_MI20`, highlighting the options that it offers, and then illustrate its performance when solving the large sparse linear systems that arise from the finite element discretisation of diffusion and convection-diffusion problems in three dimensions.

We remark that, since the original introduction of the method, a wide variety of AMG algorithms have been developed that target different problem classes and have different robustness and efficiency properties [3],[12],[17]. Our new software package, `HSL_MI20`, is designed to efficently implement the classical AMG method (also refered to as the Ruge-Stüben method), as described, for example, in [15] and [18].

2 Design of `HSL_MI20`

Given a real $n \times n$ sparse matrix A and an $n-$vector z, `HSL_MI20` computes the vector $y = Mz$, where M is an AMG V-cycle preconditioner for A. The matrix A (which may be symmetric or unsymmetric) must be "close" to an M-matrix, that is, it must have positive diagonal entries and ideally (most of) the off-diagonal entries should be negative (the diagonal should not be small compared to the sum of the modulii of the off-diagonals). In this section, we discuss our choice of programming language for `HSL_MI20`, briefly explain the user interface, and then outline how the main steps in the AMG algorithm are implemented within `HSL_MI20`.

2.1 Language

HSL is a Fortran library and includes Fortran 77 and Fortran 95 packages. A key design decision was to choose to write `HSL_MI20` in Fortran 95. We have adhered to the Fortran 95 standard except that we use allocatable structure components and dummy arguments. These are part of the official extension that is defined by Technical Report TR 15581(E), [11] and is included in Fortran 2003. It allows arrays to be of dynamic size without the computing overheads and memory-leakage dangers of pointers.

Addressing is less efficient in code that implements pointer arrays since it has to allow for the possibility that the array is associated with an array section, such as `a(i,:)`, that is not a contiguous part of its parent. Furthermore, optimization of a loop that involves a pointer may be inhibited by the possibility that its target is also accessed in another way in the loop.

The principal version of `HSL_MI20` uses `double precision` reals, which we expect to occupy 64 bits. We also provide a version using `default` reals; we anticipate that this will be useful on a computer with 64-bit addressing so that all reals and integers occupy 64 bits.

One of the main reasons for choosing Fortran 95 rather than Fortran 77 was because it allows the user interface to be simplified. In particular, all work arrays needed by the AMG algorithm are allocated (and deallocated) within the package. We have also found it useful to make extensive use of derived types. These are discussed in Section 2.2. Furthermore, `HSL_MI20` uses recursion (which was not available in Fortran 77) as a convenient and efficient way of checking and updating the coarse and fine level vertices; this is outlined in Section 2.3.1.

2.2 The user interface and derived data types

The following procedures are available to the user:

`MI20_setup` takes the matrix A, optionally checks it for errors (duplicates and out-of-range entries are not allowed), and then generates all the data that is required by the AMG preconditioner. That is, it selects the coarse levels, builds the interpolation operators and constructs the coarse level matrices. A must be supplied in compressed sparse row format (CSR), that is, the non zero entries in A must be stored row by row.

`MI20_precondition` performs the preconditioning operation $y = Mz$, where M is the AMG preconditioner and z is a user-supplied vector. `MI20_precondition` may be called repeatedly after a single call to `MI20_setup`.

`MI20_finalize` should be called after all other calls to `MI20_setup` and `MI20_precondition` are complete for a problem. It deallocates all array components of the derived data types set up by `HSL_MI20`.

The following derived types are used by `HSL_MI20`:

`MI20_control` is used to hold controlling data. The components, which are automatically given default values in the definition of the type, are used to control the action. In `MI20_setup`, controls allow the user to choose the maximum number of coarse levels and the maximum size of the coarsest level, and to control the construction of the coarse levels. In `MI20_precondition`, there are controls to select the solver that is used on the coarsest level, as well as the pre- and post-smoothers ($\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$), the numbers ν_1 and ν_2 of pre- and post-smoothing steps, and the number of V-cycles. In addition, the user can specify the amount of diagnostic printing that is required. The default settings for the control parameters have been chosen to make the code robust and, in general, efficient. If these settings are used, `HSL_MI20` provides the user with a black-box preconditioner but, for some practical applications, it may be worthwhile for the user to experiment with different choices of the control parameters. This is illustrated in Section 3.

`MI20_info` is used to provide information about the progress of the algorithm. After the call to `MI20_setup`, it includes data on the number of coarse levels computed and the order of and the number of nonzeros in the matrix on the coarsest level. After the call to `MI20_precondition`, it contains information on the coarse grid solver. In addition, the component `flag` is used throughout the computation as an error and warning flag.

The components of the derived type `MI20_keep` are private and are used to pass data between the subroutines of the package.

The data used to perform the AMG preconditioning that is generated by `MI20_setup` is stored in an array of derived type `MI20_data`. Entry j of this array has two components: one holds the coefficient matrix on level j and the other stores the interpolation matrix on level j. Both matrices are held in CSR format.

2.3 `MI20_setup`

During the setup phase, the coarsening procedure is applied recursively with the aim of producing a sequence of coarse-level problems of progressively smaller size. Constructing each level of the AMG hierarchy comprises the following steps:

- Splitting the vertices into F and C vertices;
- Constructing the interpolation matrix I_{cf} to transfer information between the fine and coarse levels;

- Constructing the coarse level matrix.

We briefly outline each of these steps. We will assume that we are starting with a fine level matrix A_f and we wish to construct the next coarse level A_c. To describe AMG coarsening, we use the adjacency graph of the matrix $A_f = \{a_{ij}\}$ with n_f vertices and consider connections between vertices. Let Ω_f be the set of vertices 1 to n_f. Vertex $i \in \Omega_f$ is said to be **connected** to vertex $j \in \Omega_f$ if $a_{ij} < 0$ and N_i is defined to be the set of vertices that are connected to i, that is,

$$N_i = \{j \in \Omega_f : a_{ij} < 0\}. \tag{2.2}$$

2.3.1 Finding F and C vertices

In Section 1, we introduced the concept of algebraically smooth errors. The other key concept in the AMG method is the *strength of dependence principle*. The non-zero elements in the i-th equation of system (1.1) reveal which unknowns x_j affect the solution x_i. If the coefficient a_{ij}, which multiplies x_j in the i-th equation, is relatively large compared to a_{ik} $(k \neq i)$, then x_j strongly influences x_i. If vertex i has been selected to be a fine level vertex (an F vertex), the coarse level correction needs to be evaluated at i (and at all other F vertices). This can be done by interpolating the value at i from the values at its neighbouring coarse level (C) vertices, which are computed by solving the coarse level residual equation at Step 5 in Figure 1.1. The best candidates for C vertices adjacent to vertex i are those that strongly influence i, that is, the vertices j for which the coefficients a_{ij} are relatively large. The two key concepts of algebraically smooth errors and the strength of dependence principle lie at the heart of the two-grid correction scheme and hence of the AMG algorithm.

To generate A_c, the set Ω_f of vertices on the fine level are split into disjoint sets of C vertices (those vertices which will exist on the coarse level), F vertices (which must interpolate their values from the C vertices), and unconnected vertices. This division is based on a strength of dependence measure that is employed to determine if a given vertex j has a strong influence on the solution at a connected vertex i. Let i be connected to j. If for some given threshold θ, called the **strength of dependence threshold**, $(0 < \theta \leq 1)$

$$|a_{ij}| \geq \theta \max\{|a_{ik}| : a_{ik} < 0\}, \tag{2.3}$$

i is said to have a **strong connection** to j; otherwise, i is considered to be **weakly** connected to j. In other words, the strength of the connection between a vertex i and its neighbours is measured relative to the largest off-diagonal negative entry in row i. Note that positive off-diagonal connections are considered to be weak connections and are ignored. We denote by S_i the set of vertices j that i has a strong connections to, that is,

$$S_i = \{j \in N_i : i \text{ is strongly connected to } j\}. \tag{2.4}$$

Since the relationship between strongly connected vertices is not, in general, symmetric (if i has a strong connection to j, it is possible that j is only weakly connected to i, even if A is symmetric), we also need the concept of strong transpose connections. The set of **strong transpose connections** of i consists of all vertices j that are strongly connected to i, that is,

$$S_i^T = \{j \in \Omega_f : i \in S_j\}.$$

In `MI20_setup`, θ corresponds to the control parameter `control%st_parameter`. Although the precise choice of θ is often not crucial, it will effect the speed of the coarsening, the memory required and quality of the resulting preconditioner. In `MI20_setup`, the default value is 0.25. This value is frequently cited in the literature (see, for example, [18]). In Section 3, we report results for a range of values of θ.

Before attempting to divide the vertices into F and C vertices, `MI20_setup` performs tests to check that coarsening is possible. Two coarsening problems can occur. Firstly, there may be a row (or more than one row) that has no negative off diagonal entries but has at least one strictly positive entry. The corresponding vertex i has no connections and so is unconnected. The user has control over what action is taken. If the

control parameter `control%c_fail` is set to 1, a flag is set and the coarsening is terminated; otherwise, the unconnected vertices are flagged and take no further part in the coarsening. The second coarsening problem is that there may be one or more rows with negative off-diagonal entries (that is, vertices with strong connections) that are connected to rows with no negative off-diagonals (that is, rows that are treated as unconnected). The corresponding vertices are also treated as unconnected and take no further part in the coarsening. Clearly, if all the vertices are flagged as being unconnected, the coarsening is terminated. Provided the early termination of the coarsening is at the second or a subsequent level, the computation continues but, in this case, the number of coarse levels (which is returned to the user in the information parameter `info%clevels`) will be less than the requested number of levels (`control%max_levels`).

Each vertex i that is not flagged as unconnected is initially put into the set U of undecided vertices and is given a weight λ_i equal to the number of its strong transpose connections, that is, $\lambda_i = |S_i^T|$. An undecided vertex $i \in U$ with maximum weight is moved into C and all undecided j that are strongly connected to i (that is, all $j \in S_i^T \cap U$) are moved into F. For each such new $j \in F$, the weight of each undecided neighbour that is strongly connected to j is increased by one (that is, for each $k \in S_j \cap U$, $\lambda_k \leftarrow \lambda_k + 1$). This process is repeated until there are no undecided vertices with a nonzero weight.

Once the above process (which we refer to as the **first pass**) has terminated, a **second pass** is made to check that all pairs of strongly connected F vertices share a common C vertex (that is, they both have a strong connection to the same C vertex). If a pair of strongly connected F vertices does not share a common C vertex, one of the pair is moved into C. Any remaining $i \in U$ with a strong connection to the new C vertex are then moved into F, whilst also ensuring that pairs of strongly connected F vertices without strong connections to a common C vertex are not created. Note that, at this point in the coarsening, a U vertex has no strong connections to any of the C vertices (otherwise it would already have been moved into F). Moreover, such variables have no strong transpose connections (else they would have been moved into C). Therefore, if $i \in U$ is moved into F, and has a strong connection to another F vertex, this new pair of F vertices cannot share a common C vertex. Also, no new F vertex can become a C vertex since it has no strong transpose connections. Thus, if there is a connection from a new F vertex to an existing F vertex, the existing F vertex must be moved into C. It is then necessary to move any $i \in U$ with a strong transpose connection to the new C vertex into F, and so on. In `MI20_setup`, this process is implemented using a recursive subroutine.

Having completed the second pass, a **third pass** deals with any remaining $i \in U$. Any such i must have a strong connection but, as already noted, it can have no strong connections to any of the C vertices and no strong transpose connections. Thus either it must have a strong connection to an F vertex, in which case we set this F vertex to be a new C vertex and the undecided vertex to be an F vertex, or it has strong connections only to vertices that have already been flagged as unconnected, in which case i is also flagged as unconnected.

To avoid the coarsening process becoming too slow, a control parameter `control%reduction` is used to ensure there is a sufficiently large reduction in the number of points between successive levels. Specifically, the coarsening is considered to have **stagnated** if

$$n_c \geq n_f * \texttt{control\%reduction},$$

where `control%reduction` has default value 0.8 and must lie between 0.5 and 1.0. In this case, the coarsening is terminated, a warning flag is set and the number of levels that have been computed is held in `info%clevels`. Otherwise, coarsening continues until either the requested maximum number of levels (`control%max_levels`) has been reached or the number of vertices has been reduced below a chosen threshold (given by the control parameter `control%max_vertices`). By default, `control%max_levels` is 100 and `control%max_vertices` is 1.

Throughout the coarsening, it is necessary to know the strong connections (for F vertices) and the strong transpose connections (for C vertices). It is easy to test whether connections to a vertex are strong (for vertex 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 vertex 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. Since this is expensive, the strong transpose connections are found before the coarsening starts and then stored for later use. 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. In this case, the user can decide, using the control parameter `control%st_method`, to compute the strong transpose connections whenever they are required, rather than computing and storing them. Note that the former is often faster and has the additional advantage of requiring less storage.

2.3.2 One pass coarsening

`HSL_MI20` includes an option for one pass coarsening, controlled by the parameter `control%one_pass_coarsen`. If set to `.true.`, at each level, only the first pass, where the vertices are first divided into C and F vertices is performed; the second and third passes that (possibly) make additional F vertices into C vertices in order to improve the interpolation are bypassed. Clearly, implementing only the first pass speeds up the coarsening and thus the time required to compute and to subsequently apply the preconditioner. But it may result in a poorer quality preconditioner so that the preconditioned iterative method then needs a larger number of iterations to attain the same accuracy. Thus there is a tradeoff between speed and quality. We have included results in Section 3 that illustrate this feature.

2.3.3 Aggressive coarsening

So far, we have assumed that to generate the coarse level, coarsening is applied to the fine level and then the coarse level matrix computed; the whole process is then repeated until the coarsening terminates. However, for some applications, the memory requirements can be substantially reduced if the coarsening is applied more than once before the next coarse level matrix is stored. This is called **aggressive coarsening** and, in `MI20_setup`, may be controlled by the user with the parameter `control%aggressive`. The default value is 1 (that is, only one coarsening is applied before the coarse level matrix is stored). In our tests, we did not find using `control%aggressive` > 1 worked well, but examples have been reported in the literature that illustrate the gains of using `control%aggressive` > 1 can outweigh the potential disadvantages of a poorer preconditioner (see, for example, [18]).

2.3.4 Interpolation operator

An algebraically smooth error has two important properties that enable the efficient construction of the interpolation operator I_{cf}. The first is that algebraically smooth errors have small residuals [4, p. 139]. This can be expressed as

$$Ae \simeq 0. \tag{2.5}$$

The second key property of an algebraically smooth error is that it changes slowly in the directions of strong dependence. This makes interpolation effective [4, p. 141] because, if i depends strongly on j, the fine level unknown x_i can be accurately approximated from the coarse level unknown x_j.

The (direct) interpolation used within `HSL_MI20` was originally devised for matrices with non-positive off-diagonal entries, but it is known to be valid if positive off-diagonals are sufficiently small [18] . As already discussed in Section 2.3.1, during coarsening positive off-diagonal entries correspond to unconnected vertices and, when calculating interpolation weights, we add any positive off-diagonal entries to the diagonal. From (2.5) it follows that, for each vertex $i \in \Omega_f$,

$$\tilde{a}_{ii}e_i + \sum_{j \in N_i} a_{ij}e_j \simeq 0$$

(the tilde here indicates any positive off-diagonal entries have been added to the diagonal and N_i is the set of vertices connected to i given by (2.2)). Thus, the error e_i at vertex i can be determined approximately from the errors e_j for $j \in N_i$, that is,

$$e_i \simeq -\frac{1}{\tilde{a}_{ii}} \sum_{j \in N_i} a_{ij} e_j.$$

It is also assumed that, when the smoother stagnates, the error varies slowly in the direction of strong connections. This suggests the possibility of efficiently approximating the above equation using a set of interpolation vertices P_i, which are C vertices that vertex i has strong connections to, that is, $P_i = S_i \cap C$ (S_i is given by (2.4)). Specifically, e_i may be approximated by the e_k values, where $k \in P_i$, using

$$e_i \simeq -\left(\frac{\sum_{j \in N_i} a_{ij}}{\sum_{k \in P_i} a_{ik}} \right) \sum_{k \in P_i} \frac{a_{ik}}{\tilde{a}_{ii}} e_k \,.$$

This can be written as

$$e_i \simeq \sum_{k \in P_i} w_{ik} e_k,$$

where

$$w_{ik} = -\left(\frac{a_{ik}}{\tilde{a}_{ii}} \right) \left(\frac{\sum_{j \in N_i} a_{ij}}{\sum_{k \in P_i} a_{ik}} \right)$$

are the entries of the i-th row of the interpolation matrix I_{cf}. Once the interpolation weights have been calculated, I_{cf} is used to form the next coarse level matrix using

$$A_c = I_{cf}^T A_f I_{cf}.$$

A more detailed description of this interpolation, together with a more complete justification, and the class of matrices for which its is valid can be found in Section 4.2 of [18].

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. Default is `control%trunc_parameter = 0.0` (no truncation of interpolation weights).

2.4 MI20_precondition

After a successful call to `MI20_setup`, `MI20_precondition` may be called repeatedly by the user to perform the preconditioning operations. There are a number of options available, including a choice of smoother and of coarse level solver, which we discuss in this section. In addition, using the parameter `control%v_iterations`, the user can select the number of V-cycles (the default value is 1). The information returned by `MI20_setup` may be used to select the number of levels used in `MI20_precondition`; this is controlled by `control%levels` and can be chosen to be fewer than the number constructed by `MI20_setup`.

2.4.1 Choice of smoother

The choice of smoother is controlled by the parameter `control%smoother`. If set to 1, damped Jacobi (DJ) is used; if set to 2, Gauss-Seidel (GS) is used [16, p. 103]. The damping factor for the Jacobi method is controlled by `control%damping`, which has default value 0.8. The default smoother is Gauss-Seidel.

This is frequently the method of choice in the literature because it is effective on a wide variety of common model problems but it may well be worthwhile for the user to experiment with both choices for his or her problems and then select the one that performs best. The user can also control the number of pre-smoothing and post-smoothing iterations that are performed on each V-cycle. The default is to perform two pre-smoothing and two post-smoothing iterations. If `control%smoother` = 2, the Gauss-Seidel sweep direction is reversed for the post smoothing and, in this case, if A is symmetric, `control%post_smoothing` should be set to be equal to `control%pre_smoothing`. As already noted, reversing the direction for the post-smoothing ensures a symmetric preconditioner for a symmetric problem thereby allowing it to be used in conjunction with the CG method. Again, the user may wish to experiment with different settings for these parameters to optimise performance.

2.4.2 Coarse level solver

On the coarsest level, if the number of points is greater than 1, `HSL_MI20` offers the following solvers:

- damped Jacobi;
- Gauss-Seidel;
- sparse direct solver `HSL_MA48`;
- `LAPACK` dense direct solver `_GETRF`.

The choice is controlled by the parameter `control%coarse_solver`. For robustness, the default is `HSL_MA48` [7], [8] (which is included within the HSL library) but it may be faster to use an iterative solver or, if the coarse level matrix is almost dense, to use the dense solver `_GETRF` from the `LAPACK` library [1]. For both the Jacobi and Gauss-Seidel solvers, the number of iterations is controlled by `control%coarse_solver_its` (with default value is 10). For Gauss-Seidel, to maintain symmetry in the symmetric case, one iteration comprises a forward and a backward Gauss-Seidel sweep.

`HSL_MA48` is a sophisticated, general-purpose sparse direct solver. Using a variant of Gaussian elimination, it computes an LU factorization and then completes the solution by successively performing two triangular solves. In common with other modern sparse direct solvers, `HSL_MA48` has a number of distinct phases. The subroutine `MA48_initialize` initializes the structure for the L and U factors and sets default values for the components of the control structure. `MA48_analyse` accepts the pattern of the system matrix A_c and chooses pivots for Gaussian elimination using a selection criterion designed to preserve sparsity. `MA48_factorize` factorizes the matrix A_c using the information from the call to `MA48_analyse`. `MA48_solve` then uses the factors to solve the linear system. Repeated calls to `MA48_solve` may follow the call to `MA48_factorize`.

If `HSL_MA48` is chosen as the coarse level solver, the parameter `control%ma48` controls its use within `MI20_precondition`. The choice `control%ma48` = 0 (which is the default), uses `HSL_MA48` with its default settings (although the printing level is set using the `MI20_precondition` printing controls) and no action is required by the user, that is, each phase of `HSL_MA48` is called by `MI20_precondition` as needed. On a second or subsequent call to `MI20_precondition`, the code will recognise that the matrix factors have already been computed, and this will significantly reduce the cost of the coarse level solve since only the (cheap) triangular solves must be performed. Other values of `control%ma48` give the user greater control by allowing values other than the `HSL_MA48` default settings to be used and/or by allowing the analyse and factorize phases of `HSL_MA48` to be called separately. Possible values for `control%ma48` are:

1: `MI20_precondition` will return to the user once `HSL_MA48` has initialised its control derived type `ma48_cntrl`. At this point, if values other than the defaults are wanted for one or more of the controls that are not related to printing, the user of `MI20_precondition` can reset the corresponding 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 the derived type `info%ma48_ainfo` to decide whether or not to continue to the factorization phase (otherwise, the user can select an alternative coarse level solver). If the user decides to continue, `MI20_precondition` must be recalled with `control%ma48` = 3 or 4.

3: MI20_precondition will return after the factorization phase of HSL_MA48 is complete. At this point, the user can use the information contained in the derived type 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, including the solve phase, and will then perform the preconditioning operation.

Full details of the HSL_MA48 control parameters and of the information on the analyse, factorize, and solve phases of HSL_MA48 that is available to the user in the derived types info%ma48_ainfo, info%ma48_finfo and info%ma48_sinfo are given in the documentation for HSL_MA48.

Typically, MI20_precondition will be called a number of times after the call to MI20_setup. We have designed the code to be flexible so that the user does not have to use the same coarse level solver on each call. The user may decide, using the information returned by HSL_MA48, that it would be worthwhile to switch to one of the other solvers. This is possible without recalling MI20_setup and allows the user to experiment to see which solver is most appropriate for his/her problem.

3 Numerical experiments

In this section, we present numerical results obtained using the HSL_MI20 AMG preconditioner with Krylov solvers for two classes of problems: the diffusion problem [9, Ch. 1,2] and the convection-diffusion problem [9, Ch. 3,4]. Our test examples are from finite element discretisations of these problems on non-trivial domains in 3D, which lead to large, sparse linear systems, in which the system matrices have highly irregular sparsity patterns. AMG-preconditioned Krylov methods are often used in these situations in preference to both direct sparse solvers and Krylov methods with general algebraic preconditioners (such as the ILU(0) method [16, p. 287]).

The definition of domain geometries and the finite element mesh generation are performed in FEMLAB [5]. The finite element discretisations use linear tetrahedral elements, with the standard Galerkin finite element method (FEM) for the diffusion problem [9, p. 17], and the streamline upwinding Petrov-Galerkin (SUPG) FEM with an optimal choice of the stabilisation parameter (as suggested in [9, p. 126]) deployed in the convection-diffusion examples.

All experiments are performed on a PC with a Pentium Core Duo processor with 2.66 GHz clock and 4 GB of RAM. The g95 Fortran compiler (see g95.sourceforge.net) with optimisation flag -O3 is used throughout. The reported times are elapsed times, in seconds, for the setup phase and for the total execution (that is, the time for the setup followed by the solution phase) measured using the Fortran intrinsic function dtime. We use the control parameters control%v_iterations = 1 (default), and control%pre_smoothing = control%post_smoothing = 1; otherwise, unless explicitly stated, default values are used for the remaining HSL_MI20 control parameters. RS2 denotes the standard coarsening described in Section 2.3.1 and RS1 denotes one pass coarsening, obtained by setting control%one_pass_coarsen = .true. (see Section 2.3.2). We use the default setting control%c_fail = 1 for the diffusion problems and control%c_fail = 2 for the convection-diffusion problems. The latter choice is because, in the convection-diffusion case, the coarsening can give coarse level matrices that have some rows with negative off-diagonal entries connected to the rows with no negative off-diagonal entries. This causes early termination of the coarsening.

3.1 The Poisson equation in 3D

We first consider the solution of the linear systems that arise in a Galerkin FEM approximation of the Poisson equation

$$-\nabla^2 u = f \quad \text{in } \Omega \subset \mathbb{R}^3\,, \tag{3.6}$$

subject to the boundary conditions

$$u = u_D \quad \text{on } \partial\Omega_D \qquad \text{and} \qquad \frac{\partial u}{\partial \hat{n}} = u_N \quad \text{on } \partial\Omega_N, \tag{3.7}$$

where $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ is the boundary of Ω and $\partial\Omega_D \cap \partial\Omega_N = \emptyset$ ($\partial\Omega_D \neq \emptyset$). Following the discretisation procedure described in [9, Ch. 1], we obtain a linear system

$$Ax = b \tag{3.8}$$

where $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite [9, p. 18]. In this case, the Krylov method of choice is the conjugate gradient (CG) method. In our experiments, we use the implementation of CG from the HSL Library (routine `MI21`) and terminate the computation when

$$\|r^{(k)}\|_2 < \varepsilon, \tag{3.9}$$

where $r^{(k)} = b - Ax^{(k)}$ is the residual vector at the k-th iteration. We use $\varepsilon = 10^{-6}$, which is the standard value used in this context [9, p. 77], and for the computed solution $x^{(k)}$ we check the $\|x^{(k)} - \tilde{x}\|_2$, where $\tilde{x}$ is the exact solution (if known) or the solution obtained by a direct sparse solver.

Example 3.1.1. We start by solving (3.6) on a cylindrical domain $\Omega = \{x^2 + y^2 \leq 1\} \times [0, 5] \subset \mathbb{R}^3$. We assume that the problem has a known analytical solution $u = x^2 + y^2 + z^2$ and set the forcing term f in (3.6) and the non-homogeneous Dirichlet boundary conditions in (3.7) accordingly.

In Table 3.1 we present iteration counts and execution times for the RS1 and RS2 coarsening strategies. For each, we test three values of the strength of dependence threshold θ (defined in (2.3)) and use both the damped Jacobi (DJ) and Gauss-Seidel (GS) smoothers. Our experiments with different values of θ and the comparison of the coarsening strategies are motivated by recent results of [17], where the authors suggest that, in examples such as this, it is advantageous to use coarsening strategies that produce relatively sparse coarse-level operators. One way of achieving this is to increase θ from its default of 0.25.

Table 3.1: The iteration counts and the setup and total execution times for the AMG-preconditioned CG method, as a function of the coarsening type (RS2/RS1), the smoother $\mathcal{S}$, and the strength threshold parameter θ.

n		22,764		69,285		208,965	
$\mathcal{S}$	θ	RS2	RS1	RS2	RS1	RS2	RS1
DJ	0.25	8 (0.30/0.43)	12 (0.09/0.21)	8 (1.23/1.72)	14 (0.32/0.81)	8 (4.92/6.71)	14 (1.16/2.92)
	0.50	8 (0.22/0.36)	13 (0.09/0.23)	9 (0.89/1.42)	14 (0.34/0.85)	9 (3.28/5.24)	17 (1.22/3.47)
	0.67	10 (0.16/0.31)	16 (0.09/0.26)	11 (0.65/1.24)	20 (0.33/1.08)	11 (2.36/4.46)	26 (1.19/4.73)
GS	0.25	7 (0.30/0.49)	10 (0.09/0.24)	7 (1.23/1.93)	12 (0.32/0.94)	7 (4.92/7.52)	12 (1.16/3.38)
	0.50	7 (0.22/0.42)	10 (0.09/0.25)	7 (0.89/1.59)	12 (0.34/0.99)	8 (3.28/6.10)	15 (1.22/4.21)
	0.67	8 (0.16/0.36)	13 (0.09/0.30)	9 (0.65/1.43)	17 (0.33/1.29)	10 (2.36/5.41)	22 (1.19/5.70)

We see that the fastest times are obtained for RS1 coarsening with $\theta = 0.25$ and DJ smoothing. Furthermore, the times increase by a factor of 4 when the problem size n is increased by a factor 3. Ideally, the time and problem size should scale at the same rate but this does not happen because of the non-optimality of the AMG coarsening procedure, which results from the coarse level matrices being much denser than the original matrix A (see Table 3.2). Note that, because the Galerkin projection (Step 4 in Figure 1.1) is used to generate the coarse level matrices, this also implies the interpolation matrices are fairly dense[2]. Also, if GS smoothing is used, its application becomes more expensive at the coarser levels. Denser coarse level matrices adversely effect data caching, which slows the algorithm performance.

[2]In particular, they are denser than the interpolation matrices used in geometric multigrid, which are constructed from the finite element interpolation. For example, in the case of a nested sequence of uniformly refined quadrilateral grids in 2D, and bi-linear finite element discretisation, geometric multigrid interpolation matrices would have at most 4 non-zero elements per row.

Table 3.2: Coarsening statistics as a function of the coarsening type (RS2/RS1), and the strength of dependence threshold θ. *nlevel* is the number of coarse levels, c_G is the grid complexity, c_A is the operator complexity, c_S is the average matrix stencil size across all coarse levels, and $c_S^{(1)}$ is the average stencil size of the original coefficient matrix.

n		22,764		69,285		208,965	
θ		RS2	RS1	RS2	RS1	RS2	RS1
0.25	*nlevel*	11	6	12	7	14	7
	c_G	1.67	1.23	1.68	1.23	1.71	1.24
	c_A	4.62	1.59	5.23	1.57	5.86	1.59
	c_S	64.5	19.5	102	21.1	152	28.0
0.50	*nlevel*	12	8	14	8	15	9
	c_G	1.93	1.34	1.98	1.35	2.02	1.36
	c_A	4.32	1.77	4.69	1.79	4.87	1.79
	c_S	44.2	16.4	58.1	20.4	77.4	23.5
0.67	*nlevel*	13	8	14	9	16	9
	c_G	1.91	1.42	1.99	1.44	2.01	1.45
	c_A	3.34	1.82	3.60	1.85	3.61	1.85
	c_S	25.7	14.9	34.0	17.2	39.0	20.3
$c_S^{(1)}$		11.2		12.0		12.8	

To analyse how the choice of coarsening strategy and the strength of dependence threshold θ effects the coarsening quality and the size of the coarse level operators, in Table 3.2 we report the average stencil size c_S, the grid complexity c_G, and the operator complexity c_A. The average stencil size c_S is defined as

$$c_S = \frac{1}{nlevel} \sum_{i=1}^{nlevel} \frac{nnz(A_i)}{n_i}, \tag{3.10}$$

where *nlevel* denotes the number of levels in the AMG coarsening, A_i is the coefficient matrix at the i-th level (with $A_1 = A$), n_i is the order of A_i and $nnz(A_i)$ is the number of non-zero entries in A_i. This quantity should be compared to the average stencil size of the original matrix $c_S^{(1)} = nnz(A)/n$. Note that, because of small contributions from the coarsest levels, the value of c_S is usually an optimistic estimate. For example, for RS2 coarsening with $\theta = 0.25$ and $n = 208,965$, the average stencil size was $c_S = 152$ (see Table 3.2) while the largest value of $nnz(A_i)/n_i$ was 435 (this was at level 6, with $n_6 = 1976$). The grid complexity is defined to be

$$c_G = \left(\sum_{i=1}^{nlevel} n_i \right) / n_1, \tag{3.11}$$

For comparison, we note that if a 3D problem is discretised on a sequence of nested, uniformly refined grids, and a standard (geometric multigrid type) full coarsening is used, then $c_G = \frac{8}{7}$ [4, p. 154] (because $\sum_{i=1}^{nlevel} \left(\frac{1}{2^3}\right)^i < \sum_{i=1}^{+\infty} \left(\frac{1}{2^3}\right)^i = \frac{8}{7}$). Finally, the operator complexity c_A is defined to be

$$c_A = \left(\sum_{i=1}^{nlevel} nnz(A_i) \right) / nnz(A_1). \tag{3.12}$$

c_A provides an indication of the AMG storage requirements, relative to that needed for the original matrix. Note that the storage for the interpolation matrices I_{cf} is not considered here. For large values of c_A, the I_{cf} are also fairly dense. In the ideal case of a sequence of nested, uniformly refined grids in 3D, and direct FEM discretisation of the underlying PDE problem on each of the coarse grids, all the A_i have the same stencil sizes and hence $c_A \simeq c_G$. However, as already observed, AMG coarsening produces A_i with stencils that are considerably larger than for A. Thus, the ratio c_A/c_G provides a measure of how much denser the A_i are compared to the ideal case (the larger the ratio, the denser the matrices).

These results illustrate that RS1 coarsening produces much sparser coarse grid matrices and, consequently, is faster. The cost of applying an AMG preconditioner is proportional to the number *nlevel* of coarse levels and the size of operator complexity c_A. Thus, the aim is to select the coarsening parameters to reduce *nlevel* and c_A without adversely affecting the quality of the preconditioner. If the coarse level matrices and the associated interpolation matrices are too sparse, the number of iterations needed by the AMG-preconditioned Krylov solver may grow as n is increased. We see this in Table 3.1 for RS1 coarsening with $\theta = 0.50$ and $\theta = 0.67$.

We end the results for this example by presenting the iteration counts and the times for of the CG method using a ILU(0) preconditioner (Table 3.3). The ILU(0) factorisation is performed using the HSL package `MI11`.

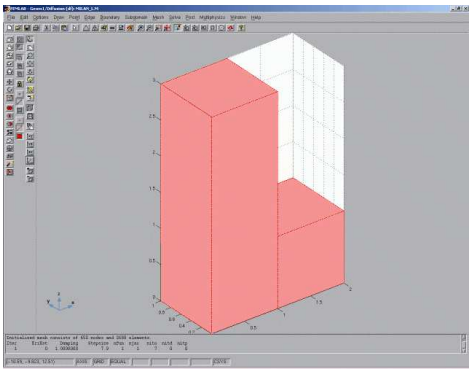
Table 3.3: The iteration counts n_{it} and the setup and total execution times for the ILU(0)-preconditioned CG method.

n	22,764	69,285	208,965
n_{it} (*time*)	49 (4.83/5.02)	71 (45.75/46.80)	99 (445.39/450.84)

We see that the iteration counts and times grow significantly with n (the asymptotic execution time behaves like $O(n^2)$). It should be noted, however, that most of the time goes on computing the incomplete factorisation (the setup time) and the average time of each CG iteration with the ILU(0) preconditioner is less than with the AMG preconditioner. Finally, we comment on the total memory use, as measured by the Linux program `top`. The total memory required is modest in all cases. For example, for $n = 208,965$ with $\theta = 0.25$ and RS2 coarsening, the memory use was 250 MB, while for RS1 coarsening it was 175 MB. For the ILU(0) preconditioner, the total memory required was 125 MB.

Example 3.1.2. This example is motivated by the well-known 2D Poisson problem on an L-shaped domain (see [9, p. 13]). We extend this to 3D by defining the domain geometry to be $\Omega = [1, 2] \times [0, 1]^2$ (see Figure 3.2). We solve equation (3.6) subject to $u_D = 0$ on $\partial\Omega_D$ and $u_N = 0$ on $\partial\Omega_N$, with $\partial\Omega_N = \Omega \cap \{x = 2\}$ and $\partial\Omega_D = \partial\Omega \setminus \partial\Omega_N$.

Figure 3.2: Domain geometry $\Omega = [1, 2] \times [0, 1]^2$



We perform the same tests as for Example 3.1.1. Our findings in terms of iteration counts and the optimal choice of coarsening parameters and the smoother are consistent with those reported in Tables 3.1 and 3.2. Again, the best results are obtained for RS1 coarsening, $\theta = 0.25$ and DJ smoothing and these are reported on in Table 3.4. For comparison purposes, the iteration counts and the times for the ILU(0)-preconditioned CG method are also presented.

We see that, with the AMG preconditioner, the iteration counts remain (almost) constant when n is increased, with the time exhibiting the same asymptotic behaviour as in Example 3.1.1.

Table 3.4: The iteration counts and the setup and total execution times for the AMG-preconditioned CG method and the ILU(0)-preconditioned CG method.

n	18,005	53,431	160,737
AMG	9 (0.07/0.14)	11 (0.24/0.52)	12 (0.84/1.95)
ILU(0)	34 (3.00/3.10)	50 (26.72/27.47)	74 (260.63/263.57)

Example 3.1.3. In this example, we consider the Poisson problem (3.6) on the following highly non-convex domain: $\Omega = \Omega_1 \setminus \Omega_2$, with $\Omega_1 = [-0.5, 0.5]^2 \times [0, -5]$, and $\Omega_2 = [0, 0.25]^2 \times [0, -3]$ (see Figure 3.3 for the outside picture of the domain and Figure 3.4 for the cross-section). Basically, Ω is a box-like domain with a deep cavity within it (note that the cavity is not centered within the domain). We want to verify whether the coarsening should be allowed to generate the coarsest level with only one (or a few) vertices (this is the default option within `HSL_MI20` and was used already in the simpler domains of Examples 3.1.1 and 3.1.2). The concern is whether such coarse-level problems represent suitable approximations of the original problem (the non-convexity of the domain might be lost).

Figure 3.3: Domain geometry (surface) $\Omega = \Omega_1 \setminus \Omega_2$, $\Omega_1 = [-0.5, 0.5]^2 \times [0, -5]$, $\Omega_2 = [0, 0.25]^2 \times [0, -3]$.

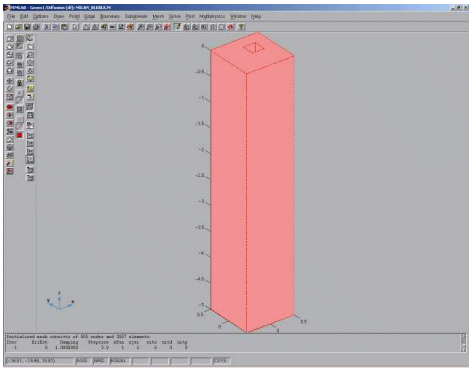
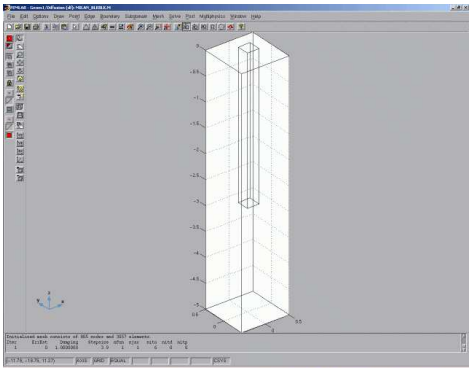


Figure 3.4: Domain geometry (cross-section) $\Omega = \Omega_1 \setminus \Omega_2$, $\Omega_1 = [-0.5, 0.5]^2 \times [0, -5]$, $\Omega_2 = [0, 0.25]^2 \times [0, -3]$.



We again perform the same tests as for the previous two examples and again find very similar results. The best results are for RS1 coarsening, $\theta = 0.25$ and DJ smoothing and these are reported on in Table 3.5.

We conclude that the effectiveness of AMG as a preconditioner is not affected by the shape and convexness of the domain and that it appears to be safe to coarsen as far as possible even in the cases of highly non-convex domains. This supports our default setting within `HSL_MI20` of `control%max_points = 1`.

Table 3.5: The iteration counts and the setup and total execution times for the AMG-preconditioned CG method and the ILU(0)-preconditioned CG method.

n	21,177	62,675	188,636
AMG	10 (0.08/0.16)	11 (0.28/0.60)	13 (0.98/2.41)
ILU(0)	27 (4.11/4.20)	41 (36.73/37.25)	61 (357.97/360.85)

3.2 The convection-diffusion equation in 3D

In this subsection, we examine the effectiveness of an AMG-preconditioned Krylov method for solving the linear systems that arise in the FEM discretisation of the 3D convection-diffusion equation. The convection-diffusion equation is a scalar elliptic PDE that models a host of important processes and phenomena in different areas, including fluid mechanics and electronics. It can occur either as a stand-alone problem, or as a part of more complex systems of PDEs (such as the Navier-Stokes equation, the Boussinesq equation or the drift-diffusion equation).

We consider the linear systems that arise in the streamline upwinding Petrov-Galerkin (SUPG) FEM discretisation of the convection-diffusion equation

$$-\nu\nabla^2 u + \vec{w}\cdot\nabla u = f \qquad \text{in } \Omega \subset \mathbb{R}^3, \tag{3.13}$$

subject to the boundary conditions (3.7). Details of the FEM discretisation can be found in [9, p. 126]. Here $\nu > 0$ is the diffusivity parameter determining the relative contribution of the convection and the diffusion (in most practical cases $\nu \ll 1$), $\vec{w} : \mathbb{R}^3 \mapsto \mathbb{R}^3$ is the vector-valued function referred to as the convective field (or wind) that determines the direction of the convection, and f is the forcing term.

The SUPG discretisation of (3.13) and the boundary conditions (3.7) leads to a linear system

$$\Phi x = b\,, \tag{3.14}$$

where the system matrix $\Phi \in \mathbb{R}^{n\times n}$ can be expressed in the form

$$\Phi = \nu A + C + S\,. \tag{3.15}$$

Here A is the diffusion matrix defined in (3.8), C is the convection matrix and S is the stabilisation matrix. A is symmetric positive-definite; provided the convective field is incompressible ($\nabla\cdot\vec{w} = 0$), C is skew-symmetric (that is, $c_{ij} = -c_{ji}$, $c_{ii} = 0$, $i, j = 1, \ldots, n$); S is symmetric but possibly indefinite. In particular, if the Galerkin FEM is used, $S = 0$. If geometric multigrid is used to precondition (3.14), it is essential that the SUPG discretisation is applied at all the coarse levels (see [9, p. 194]). In the case of an AMG preconditioner, the coarse level matrices are created automatically. However, experience has shown that the performance of the AMG preconditioner is more robust if the original problem is discretised using the SUPG discretisation, rather than the standard Galerkin FEM.

Although the original design of the classical AMG coarsening procedure is based on the properties of A, in practice it adapts well to the system matrix Φ given by (3.15). In particular, analysis of the geometric positions of the coarse-grid vertices shows that AMG performs a version of semi-coarsening in the characteristic directions (these are the directions of strongly coupled unknowns) [4, p. 119]. This feature makes AMG a robust preconditioner for convection dominated problems, even though it uses simple point smoothers. By contrast, robust performance of geometric multigrid (which uses full coarsening), requires

not only SUPG discretisation at all coarse levels, but also fairly sophisticated smoothing techniques, based on the directional ordering of vertices [9, p. 195].

The system matrix Φ is non symmetric and this influences the choice of Krylov solver. We will use right-preconditioned GMRES. The HSL package `MI24` implements a restarted variant of the algorithm GMRES(m) [16, p. 172] but GMRES with no restarting can be obtained by setting the truncation parameter m equal to maximal allowed number of iterations and this is the setting used in our tests. As in the diffusion tests, we use the stopping criterion (3.9) and monitor the norm $\|x^{(k)} - \tilde{x}\|_2$ ($x^{(k)}$ is the computed solution on the k-th iteration and $\tilde{x}$ is computed using a sparse direct solver).

Example 3.2.1. We first solve (3.13) on a cylindrical domain $\Omega = \{x^2 + y^2 \leq 1\} \times [0, 5] \subset \mathbb{R}^3$, with convective field $\vec{w} = (0, 0, 1)$ (constant uni-directional field), forcing term $f = 0$, and Dirichlet boundary conditions $u_D = 1$ on $z = 0$ (bottom lid) and $u_D = 0$ elsewhere on $\partial\Omega_D$ and Neumann boundary conditions $u_N = 0$ on $z = 5$ (top lid). This is a problem of flow of a viscous fluid through the channel at constant speed $\vec{w}$.

In Table 3.6, we present results for the RS1 and RS2 coarsening strategies (with the default strength of dependence threshold $\theta = 0.25$) and a range of values of the diffusivity parameter ν. Gauss-Seidel (GS) smoothing is used. Note that other choices of θ were tried but gave either similar or poorer results. In the remainder of this section, all results are for the default setting $\theta = 0.25$.

Table 3.6: Iteration counts and the setup and total execution times for the AMG-preconditioned GMRES method, as a function of the coarsening type (RS2/RS1), the diffusivity parameter ν. GS smoothing is used.

n	22,764		69,285		208,965	
	RS2	RS1	RS2	RS1	RS2	RS1
$\nu = 0.02$	8 (0.30/0.58)	11 (0.09/0.26)	7 (1.24/2.11)	12 (0.33/0.94)	7 (5.02/8.18)	12 (1.20/3.34)
$\nu = 0.004$	10 (0.28/0.63)	14 (0.10/0.34)	9 (1.18/2.31)	13 (0.38/1.10)	8 (4.50/8.12)	13 (1.32/3.84)
$\nu = 0.001$	12 (0.28/0.70)	17 (0.11/0.40)	11 (1.17/2.54)	17 (0.40/1.38)	11 (4.45/9.32)	17 (1.43/4.88)

Again, we find the RS1 strategy is faster than RS2, but requires more iterations. We observe that the iteration counts are slightly higher than in the diffusion case (see Table 3.1), but remain (almost) constant as n increases. There is a moderate increase in the iteration counts for a fixed n and decreasing ν. It should be emphasised that, in all our tests, including the highly convective cases (small values of ν), the `HSL_MI20` package and resulting AMG preconditioner proved extremely robust.

In the diffusion case, if damped Jacobi smoothing is used, the optimal damping parameter (which in `HSL_MI20` is the control parameter `control%damping`) can be determined analytically. For uniform triangular grids and linear finite element approximation in 2D the optimal value is 0.8 and for uniform quadrilateral grids and bilinear finite element approximation it is 8/9 [9, p. 100]. For this reason, we chose the default setting to be `control%damping` = 0.8. However, this may not be the best choice for convection-diffusion problems. In Table 3.7, we report results for damped Jacobi (DJ) smoothing for `control%damping` = 1, 0.8 and 0.5. The results suggest that 0.8 is again the best choice, with the convergence rate deteriorating significantly for `control%damping` = 1 and very small values of ν.

In Table 3.8, we report the coarsening statistics for $\nu = 0.001$. Comparing these with the corresponding statistics in Table 3.2, we conclude that the number of coarse levels *nlevel* is much larger in the convection-dominated case. This is because of the semi-coarsening (rather than the approximately full coarsening performed by AMG in the diffusion case). This inevitably leads to larger grid complexities c_G and larger values of the operator complexities c_A.

Table 3.7: Iteration counts for the AMG-preconditioned GMRES method, using damped Jacobi smoothing with a range of values of `damping`, and of the diffusivity parameter ν.

n		22,764		69,285		208,965	
ν	`damping`	RS2	RS1	RS2	RS1	RS2	RS1
0.02	1	11	14	10	15	10	16
	0.8	9	15	8	16	8	17
	0.5	12	20	11	22	11	23
0.004	1	17	24	15	22	13	20
	0.8	12	19	11	19	10	19
	0.5	14	25	13	25	12	25
0.001	1	36	51	46	69	74	65
	0.8	15	25	14	27	14	27
	0.5	16	29	16	31	15	32

Table 3.8: Coarsening statistics for $\nu = 0.001$. *nlevel* is the number of coarse levels, c_G is the grid complexity, c_A is the operator complexity, c_S is the average matrix stencil size across all coarse levels, and $c_S^{(1)}$ is the average stencil size of the original coefficient matrix.

n	22,764		69,285		208,965	
	RS2	RS1	RS2	RS1	RS2	RS1
nlevel	15	8	18	9	20	10
c_G	2.26	1.47	2.34	1.49	2.38	1.49
c_A	5.72	2.13	6.39	2.17	6.77	2.15
c_S	65.4	24.5	82.7	27.6	110	32.1
$c_S^{(1)}$	11.3		12.2		12.9	

Finally, for comparison, in Table 3.9 we present results for the GMRES solver preconditioned by ILU(0). Again, more iterations are needed than for the AMG preconditioner but, interestingly, in contrast to the AMG preconditioner, for fixed n, the iteration count reduces when ν is reduced.

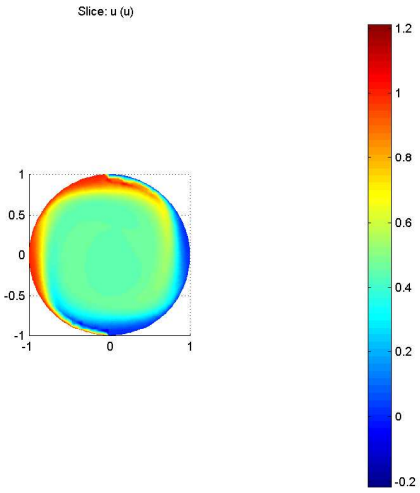
Table 3.9: Iteration counts and the setup and total execution times for the ILU(0)-preconditioned GMRES method.

n	22,764	69,285	208,965
$\nu = 0.02$	35 (4.85/5.03)	53 (45.5/46.6)	83 (446/453)
$\nu = 0.004$	31 (4.85/5.01)	42 (45.5/46.3)	60 (446/451)
$\nu = 0.001$	32 (4.85/5.01)	42 (45.5/46.3)	55 (446/450)

Example 3.2.2. We now solve problem (3.13) on a cylinder domain $\Omega = \{x^2+y^2 \leq 1\}\times[0,5] \subset \mathbb{R}^3$. We set the convective field to be circular $\vec{w} = (w_x, w_y, w_z) = \left(2y(1-x^2), -2x(1-y^2), \sqrt{x^2+y^2}\sin\frac{x}{\sqrt{x^2+y^2}}\right)$. To avioid singularity of the wind function at the points $(0,0,z)$ we set $w_z = 0$. The forcing term is chosen to be $f = 0$ and we use Dirichlet boundary conditions $u_D = 0$ for $z = 0$ and $z = 5$ (the bottom and the top lids of the cylinder), and $u_D = 0$ for $x^2 + y^2 = 1$ and $x \leq 0$, $u_D = 1$ for $x^2 + y^2 = 1$ and $x > 0$. In Figure 3.5, we illustrate the solution for $\nu = 0.004$ at $z = 2.5$ (the horizontal mid-plane).

This example was chosen because circular convective fields are known to be difficult to solve by iterative solvers. In particular, performance of a geometric multigrid preconditioner depends crucially on the choice

Figure 3.5: Computed solution for Example 3.2.2, with $\nu = 0.004$. Mid-plane cross-section $\Omega \setminus \{z = 2.5\}$.



of smoother (see [9, p. 202]), with simple point smoothers proving to be a poor choice, especially in highly convective examples. As in Example 3.2.1, we use the SUPG FEM[3]. The experiments reported on in Tables 3.10 to 3.13 are analogous to those in Tables 3.6 to 3.8.

Table 3.10: Iteration counts and the setup and total execution times for the AMG-preconditioned GMRES method for the RS1 and RS2 coarsening strategies and a range of values of the diffusivity parameter ν. GS smoothing is used.

n	22,764		69,285		208,965	
	RS2	RS1	RS2	RS1	RS2	RS1
$\nu = 0.02$	7 (0.32/0.57)	11 (0.09/0.26)	7 (1.32/2.19)	12 (0.33/0.93)	7 (5.13/8.31)	13 (1.18/3.46)
$\nu = 0.004$	9 (0.29/0.60)	15 (0.09/0.34)	8 (1.20/2.22)	15 (0.35/1.17)	7 (4.74/7.96)	15 (1.27/4.07)
$\nu = 0.001$	12 (0.28/0.69)	24 (0.10/0.50)	11 (1.19/2.53)	24 (0.38/1.74)	10 (4.48/8.94)	23 (1.40/6.01)

We observe from Table 3.10 that the iteration counts for RS2 coarsening are marginally better than in the case of uni-directional wind (see Table 3.6). The same holds for RS1 coarsening with $\nu = 0.02$ and $\nu = 0.004$, although the iteration counts in Example 3.2.2 are slightly larger than the corresponding counts for Example 3.2.1. The only notable exception is for RS1 coarsening applied to a highly-convective case ($\nu = 0.001$), for which the iteration counts are considerably higher than in Example 3.2.1 (although even in this case, the counts remain constant as n increases). The asymptotic behaviour of the total execution times exhibits a similar pattern to Example 3.2.1, and is nearly optimal. We examined the error norms $\|x^{(k)} - \tilde{x}\|_2$ for each computed solution and found that (with the same stopping criterion $\|r_k\|_2 < 10^{-6}$) these norms are slightly larger than we found in Example 3.2.1 but, in all cases, were less than 10^{-3}.

The results in Table 3.11 confirm `control%damping` = 0.8 as the optimal choice for the damping parameter in DJ smoothing but for RS2 and $\nu = 0.001$, the smaller value of 0.5 gave similar results. This suggests that for even smaller values of ν, using a value of `control%damping` that is less than the default might be preferable. However, the number of iterations required when using DJ was significantly larger than for GS so that, even though each GS smoothing operation is more expensive, in terms of the total execution time, using DJ was not advantageous here. Comparing the coarsening statistics (Table 3.12) with the corresponding figures for Example 3.2.1 (Table 3.8), we see that they are strikingly similar.

[3]Note from Figure 3.5 that the method does not remove all the spurious oscillations from the solution (the solution values lie between -0.2 and 1.2, although the imposed BCs should imply the solution range [0,1]).

Table 3.11: Iteration counts for the AMG-preconditioned GMRES method, using DJ smoothing, with a range of values of `damping` and of the diffusivity parameter ν.

n		22,764		69,285		208,965	
ν	`damping`	RS2	RS1	RS2	RS1	RS2	RS1
0.02	1	9	14	9	15	10	16
	0.8	8	15	8	16	8	17
	0.5	11	19	10	20	10	22
0.004	1	13	21	11	20	10	20
	0.8	10	21	9	20	9	20
	0.5	12	26	11	27	10	27
0.001	1	32	48	64	58	83	60
	0.8	16	34	14	35	13	34
	0.5	17	41	15	42	13	41

Table 3.12: Coarsening statistics for $\theta = 0.25$ and $\nu = 0.001$. *nlevel* is the number of coarse levels, c_G is the grid complexity, c_A is the operator complexity, c_S is the average matrix stencil size across all coarse levels, and $c_S^{(1)}$ is the average stencil size of the original coefficient matrix.

n	22,764		69,285		208,965	
	RS2	RS1	RS2	RS1	RS2	RS1
nlevel	16	9	17	10	20	11
c_G	2.24	1.46	2.33	1.47	2.37	1.48
c_A	5.56	2.09	6.39	2.13	6.83	2.12
c_S	53.1	21.7	78.6	24.6	100.1	28.8
$c_S^{(1)}$	11.2		12.0		12.8	

Finally, in Table 3.13 results are given for the ILU(0)-preconditioned GMRES method. In contrast to the uni-directional wind example (see Table 3.9) the iteration counts deteriorate quite badly when either n is increased or ν is decreased.

Table 3.13: Iteration counts and the setup and total execution times for the ILU(0)-preconditioned GMRES method.

n	22,764	69,285	208,965
$\nu = 0.02$	81 (4.83/5.41)	131 (45.41/49.27)	202 (445.34/471.68)
$\nu = 0.004$	118 (4.83/5.83)	174 (45.41/51.37)	253 (445.34/483.11)
$\nu = 0.001$	187 (4.83/6.90)	282 (45.41/58.61)	553 (445.34/591.61)

4 Concluding remarks and future directions

In this paper, we have described the design and development of an efficient Fortran 95 AMG code called `HSL_MI20`. We have employed `HSL_MI20` to demonstrate the use of AMG as a preconditioner for the linear systems that arise from scalar elliptic problems in three dimensions. Another important application of AMG is as a building block for the design of efficient block preconditioners (see, for example, [16, p. 337]). Block preconditioning is a technique suitable for the discrete problems obtained from the approximation of systems of PDEs, and/or PDEs in which the unknown functions are vector fields. Such problems arise in modelling multi-physics problems, that is, problems with their constitutive parts coming from different areas (e.g. fluid-solid interaction problems, magnetohydrodynamics, etc.). In such cases, each node in the

mesh is associated with several degrees of freedom corresponding to different physical quantities. A suitable enumeration of the unknowns (whereby the unknowns of the same kind are enumerated consecutively), yields a natural blocking of the underlying coefficient matrix. An effective approach to preconditioning the resulting linear system is to approximately invert the diagonal blocks. AMG is frequently used in this context to approximately invert some of the principal diagonal blocks or the associated Schur complements (which are themselves, in some cases, close to scalar elliptic discrete operators, for which AMG is ideally suited). Examples of effective block preconditioners based on AMG are given in [14] for reservoir simulation, in [13] for linear elasticity, and in [9] for fluid mechanics. Currently, we are using `HSL_MI20` to obtain a block preconditioner for the linear systems obtained from the FEM discretisation of three dimensional fluid mechanics problems; we will report on this work in a separate paper.

A Matlab interface to HSL_MI20 is currently being developed. This will make it very easy for Matlab uses to experiment with the AMG method in a variety of different contexts. A prototype of the Matlab interface has been already successfully used within an iterative method to solve a class of distributed control problems: the AMG method was applied to a system that is a linear combination of a stiffness matrix and a mass matrix [6].

5 Code availability

`HSL_MI20` is available now as part of the 2007 release of the mathematical software library HSL. All use of HSL requires a licence; details of how to obtain a licence and the packages are available at `www.cse.clrc.ac.uk/nag/hsl/`.

6 Acknowledgements

The finite element discretisation of both the diffusion problem and the convection-diffusion problem is performed by the parallel Fortran code `femFluidMechanics`, which is designed for 3D simulations of problems in fluid mechanics. The code was developed by Christopher Smethurst in the School of Computer Science at the University of Manchester.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide (3rd edition)*. SIAM, 1999.

[2] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31:333–390, 1977.

[3] M. Brezina, A.J. Cleary, R.D. Falgout, V.E. Henson, J.E. Jones, T.A. Manteuffel, S.F. McCormick, and J.W. Ruge. Algebraic multigrid based on element interpolation. *SIAM J. Sci. Comput.*, 22(5):1570–1592, 2000.

[4] W.L. Briggs, V. Emden Henson, and S.F. McCormick. *A Multigrid Tutorial (2nd edition)*. SIAM, 2000.

[5] COMSOL. *FEMLAB Version 2.3 Reference Manual*. COMSOL, 2003.

[6] H. S. Dollar. Iterative solution of PDE-constrained optimization problems, 2007. University of Manchester seminar, see ftp://ftp.numerical.rl.ac.uk/pub/talks/hsd.manchester.26X07.pdf.

[7] I.S. Duff and J.K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Report RAL-93-072, Rutherford Appleton Laboratory, 1993.

[8] I.S. Duff and J.K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Mathematical Software*, 22:187–226, 1996.

[9] H. Elman, D. Silvester, and A. Wathen. *Finite Elements and Fast Iterative Solvers*. Oxford University Press, 2005.

[10] HSL. A collection of Fortran codes for large-scale scientific computation, 2007. See http://www.cse.clrc.ac.uk/nag/hsl/.

[11] ISO/IEC. TR 15581(E): Information technology - Programming languages - Fortran - Enhanced data type facilities (second edition), edited by Malcolm Cohen. Technical Report, ISO/IEC, 2001. ISO, Geneva.

[12] J.E. Jones and P.M. Vassilevski. Amge based on element agglomeration. *SIAM J. Sci. Comput.*, 23(1):109–133, 2001.

[13] S. Mijalković and M. Mihajlović. A component decomposition preconditioning for 3d stress analysis problems. *Numer. Linear Algebra Appl.*, 9:567–583, 2002.

[14] C. Powell and D. Silvester. Optimal preconditioning for raviart-thomas mixed formulation of second-order elliptic problems. *SIAM J. Matrix Anal. Appl.*, 25:718–738, 2004.

[15] J.W. Ruge and K. Stuben. Algebraic multigrid. In S.F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, 1987.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems (2nd edition)*. SIAM, 2003.

[17] H. De Sterck, U. Meier Yang, and J.J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J. Matrix Anal. Appl.*, 27:1019–1039, 2006.

[18] K. Stuben. An introduction to algebraic multigrid. In U. Trottenberg, C. Oosterlee, and A. Schuller, editors, *Multigrid*, pages 413–532. Academic Press, 2001.