

MADPACK: A FAMILY OF ABSTRACT MULTIGRID OR MULTILEVEL SOLVERS

Craig C. Douglas*

Abstract: Multigrid can be formulated as an algorithm for an abstract problem that is independent of the partial differential equation, domain, discretization method, and number of processors. In such an abstract setting, problems not arising from partial differential equations can be treated also (c.f. aggregation-disaggregation methods). Quite general theory exists for linear problems, e.g., C. C. Douglas and J. Douglas, SIAM J. Numer. Anal., 30 (1993), pp. 136–158.

The general theory was motivated by a series of abstract solvers (Madpack). The latest version (5) was motivated by the theory and user feedback. Madpack now provides a wide variety of iterative and direct solvers, preconditioners, and interpolation and projection schemes, including user call back ones.

It allows for matrices to be stored in sparse, dense, and stencil formats. The matrices can be symmetric or nonsymmetric with real, complex, or user defined data types.

The type of solvers and design decisions (including language, data structures, external library support, and call backs) are discussed here. Based on the author's experiences with two versions of Madpack, a better approach is proposed here. This is based on a mixed language formulation (C and Fortran+preprocessor). Reasons for not just using Fortran, C, or C++ are given.

Key words: multigrid, iterative methods, solvers, computer languages

1 Introduction

The term *abstract multigrid* was coined in [6]. This refers to theory which is quasi-independent of the elliptic boundary value problem. The dependence is introduced by assuming that the (discretized) problem satisfies a very small number of hypotheses which contribute simple expressions to the convergence rate formula. The theory in [6] is general enough to apply to nonnested solution spaces and includes example boundary value problems on general domains, with variable coefficients, and finite difference and finite element discretizations.

*IBM Research Division, T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, USA and Department of Computer Science, Yale University, P. O. Box 208285, New Haven, CT 06520-8285, USA. E-mail: na.cdouglas@na-net.ornl.gov.

The concept of abstract multigrid was pushed to the extreme in [13], where a general theory for linear problems is presented with virtually no constraints on the origin of the problems.

Abstract multigrid is defined in §2. Two implementations of abstract multilevel methods (see [11] and [12]) are discussed in §3. A discussion of what might be the right set of languages to implement abstract multilevel methods is in §4. The current, object oriented implementation is discussed in §5. Finally, some conclusions are drawn in §6.

2 Abstract Multigrid

Assume we are solving some problem, possibly derived from a partial differential equation, possibly not. Assume further that by various means a sequence of (linear) problems

$$A_j x_j = b_j, \quad 1 \leq j \leq k, \quad (2.1)$$

are formed which approximate the *real problem*

$$A_k x_k = b_k, \quad (2.2)$$

where $x_j, b_j \in \mathcal{M}_j$, $1 \leq j \leq k$. Typically, \mathcal{M}_j is a real or complex vector space when actually computing the solution to the problem. Frequently,

$$\dim(\mathcal{M}_j) \approx C \dim(\mathcal{M}_{j-1}), \quad C > 1.$$

In practice, $C \in \{4, 8\}$ normally.

There are typically three mappings between the neighboring solution spaces.

$$\begin{cases} \mathcal{R}_j, \mathcal{Q}_j : \mathcal{M}_j \rightarrow \mathcal{M}_{j-1}, & 2 \leq j \leq k, \\ \mathcal{P}_j : \mathcal{M}_j \rightarrow \mathcal{M}_{j+1}, & 1 \leq j \leq k-1. \end{cases}$$

The \mathcal{R}_j and \mathcal{Q}_j are *restriction* (or projection) matrices and the \mathcal{P}_j are *prolongation* (or interpolation) matrices. Frequently,

$$\mathcal{P}_j = c \mathcal{R}_{j-1}^T, \quad \text{where } c \in \mathbb{R}. \quad (2.3)$$

In practice, $c \in \{0.5, 1, 2\}$ normally. The matrices A_j and A_{j-1} are typically related through the relation

$$A_{j-1} = \mathcal{Q}_j A_j \mathcal{P}_{j-1}, \quad 2 \leq j \leq k.$$

The Galerkin form of multigrid requires that $\mathcal{Q}_j = \mathcal{P}_{j-1}^T$. The \mathcal{Q}_j are frequently injection matrices when a finite difference discretization is applied to a partial differential equation.

A multilevel correction algorithm is simply defined by

Algorithm MGC ($lev, \{A_j, x_j, b_j\}_{j=1}^k, \{\mathcal{P}_j\}_{j=1}^{k-1}, \{\mathcal{R}_j\}_{j=2}^k$)

1. $x_{lev} \leftarrow \text{Solver}_{lev}(A_{lev}, x_{lev}, b_{lev})$
2. If $lev > 1$, then repeat 2a–2d until some condition is met:
 - 2a. $x_{lev-1} \leftarrow 0$ and $b_{lev-1} \leftarrow \mathcal{R}_{lev}(b_{lev} - A_{lev}x_{lev})$
 - 2b. MGC ($lev - 1, \{A_j, x_j, b_j\}_{j=1}^k, \{\mathcal{P}_j\}_{j=1}^{k-1}, \{\mathcal{R}_j\}_{j=2}^k$)
 - 2c. $x_{lev} \leftarrow x_{lev} + \mathcal{P}_{lev-1}x_{lev-1}$
 - 2d. $x_{lev} \leftarrow \text{Solver}_{lev}(A_{lev}, x_{lev}, b_{lev})$

A common condition in step 2 is to do steps 2a–2d some specified number of times (e.g., 0 for one way multigrid, 1 for a V Cycle, or 2 for a W Cycle).

On the coarsest level, $lev = 1$, the solver is frequently some flavor of Gaussian elimination (e.g., a sparse one). Common solvers on the other levels include relaxation methods (e.g., point, line, plane, or zebra Gauss–Seidel) and conjugate direction methods (e.g., conjugate gradients or residuals, CGS, GMRES, or Orthomin). The latter class of iterative methods is most effective on highly nonuniform meshes with a significant difference between the largest and smallest mesh spacing or diameter on a level.

A general algorithm that provides very good initial guesses is the nested iteration one:

Algorithm NIC ($k, \{A_j, x_j, b_j\}_{j=1}^k, \{\mathcal{P}_j\}_{j=1}^{k-1}, \{\mathcal{R}_j\}_{j=2}^k$)

1. MGC ($1, \{A_j, x_j, b_j\}_{j=1}^k, \{\mathcal{P}_j\}_{j=1}^{k-1}, \{\mathcal{R}_j\}_{j=2}^k$)
2. Do steps 2a–2b with $lev = 2, \dots, k$:
 - 2a. $x_{lev} \leftarrow \mathcal{P}_{lev-1}x_{lev-1}$
 - 2b. MGC ($lev, \{A_j, x_j, b_j\}_{j=1}^k, \{\mathcal{P}_j\}_{j=1}^{k-1}, \{\mathcal{R}_j\}_{j=2}^k$)

A one way multilevel algorithm means that Algorithm MGC never performs any portion of its step 2 as part of its use by Algorithm NIC. Most complexity arguments showing that multigrid is of optimal order are based on Algorithm NIC, not Algorithm MGC.

For nonlinear problems, there are two standard approaches: the Full Approximation Scheme (FAS) and damped Newton multilevel. FAS is similar to Algorithm MGC, but changes two lines:

- 2a. $x_{lev-1} \leftarrow \mathcal{R}_{lev}^{(FAS)}x_{lev}$ and
 $b_{lev-1} \leftarrow \mathcal{R}_{lev}(b_{lev} - A_{lev}x_{lev}) - A_{lev-1}x_{lev-1}$
- 2c. $x_{lev} \leftarrow x_{lev} + \mathcal{P}_{lev-1}(x_{lev-1} - \mathcal{R}_{lev}^{(FAS)}x_{lev})$

Note that in many situations $\mathcal{R}_{lev}^{(FAS)} = \mathcal{R}_{lev}$ or $\mathcal{R}_{lev}^{(FAS)} = \mathcal{Q}_{lev}$. Also, the operator A_j is not linear anymore, but involves function evaluations.

The damped Newton algorithm is a modification of Algorithm NIC. Before each reference to Algorithm MGC, a Jacobian is formed and a damped Newton step is performed. The last Jacobian on a level is saved for use in subsequent multilevel correction steps.

The difference between these two nonlinear approaches is easy to categorize. FAS uses a nonlinear iterative method (e.g., nonlinear Gauss–Seidel) while damped Newton uses standard linear solvers (e.g., Gauss–Seidel or a conjugate direction method). When evaluating the nonlinear function is inexpensive, FAS usually produces an approximate solution faster than the damped Newton multilevel method. However, when the function evaluations are expensive, the damped Newton multilevel method usually produces an approximate solution faster than FAS.

Note that in Algorithms MGC and NIC, there are only two obvious components per level: the solver and the methods for passing information between levels. There are other components hidden by this formulation: a possible set of preconditioners for use by the solvers, a method for computing a matrix–vector product for some set of storage formats, and a set of discretization methods in the partial differential equation case.

For problems not arising from partial differential equations, the only components in Algorithm MGC that can be optimized are the solvers and the restriction matrices \mathcal{Q}_j and \mathcal{R}_j . Both theory and practical experience demonstrate rather conclusively that finding better \mathcal{Q}_j matrices is far superior to trying to find an optimal iterative method as the solver (e.g., see [3]).

For partial differential equation problems, using better discretization methods usually makes a bigger impact on the convergence rate than searching for a slightly better interpolation scheme or iterative solver. There are exceptions to this for trivial problems (e.g., Laplace’s equation on a square with uniform grids) and for quite difficult nonlinear problems. The interpolation method in a damped Newton multilevel iteration should be good enough to keep the initial guess on a level within the domain of convergence of the Newton iteration (otherwise, multiple Jacobians will be calculated).

3 Madpack

The term Madpack is a mnemonic for *multigrid–aggregation–disaggregation package* (multilevel can be substituted for multigrid). It started as a compact set of subroutines for solving problems of the form (2.1)–(2.2). The first two versions were released in 1986 and the fourth in 1992. All versions have been written using numerous macros to hide data structures and improve the readability. Currently, version 2 is available through MGNet (see [4] and [9] for a description of MGNet). Version 2 is in the public domain. Version 4 is not really compatible with version 2 and is also owned by IBM. It is available through IBM’s Internet anonymous ftp server and MGNet. All announcements and bug fixes for version 4 are distributed through MGNet.

Version 2 is discussed in §3.1. Version 4 is discussed in §3.2. A number of issues that these two versions raise are discussed in §4.

3.1 Madpack, Version 2

Version 2 [7] was originally written in an extended flavor of Ratfor. A translator converted this to Fortran-77. This, in turn, is compiled by whatever compiler is available on a given machine. After determining that on some machines (e.g., SUN workstations in 1986) a C version of the subroutines ran up to 40% faster than the Fortran-77 equivalents, the entire code was ported to C. Including comments, there are only 1500–1600 lines in each language version. All three language versions are distributed.

Version 2 consists of 9 subroutines:

Routine	Description
klmg	Algorithm MGC
klni	Algorithm NIC
klax	Matrix–vector multiply
kldsnf	Factor matrices
kldsss	Forward/backward solves
klres	Residual calculation
klsgs	Symmetric Gauss–Seidel
klsgsc	Preconditioned conjugate gradients
klsgsm	Preconditioned Orthomin(1)

The first two subroutines, klmg and klni, are meant to be the only user callable subroutines, but any can be called directly.

Version 2 supports an odd flavor of sparse matrix storage (see [2]) in the solver routines. The matrices A_j are assumed to have a symmetric nonzero structure, independent of whether or not $A_j = A_j^T$. This means that in some cases, a small number of zeroes are actually stored in the sparse matrix representation of A_j . The main diagonal, the nonzero elements of the columns of the upper triangular part of A_j , and the nonzero elements of the rows of the lower triangular part of A_j are stored independently (the lower part only if A_j is nonsymmetric). This allows for only half of the row or column indices to be stored due to the symmetry of the nonzero structure. It also allows for numerous computational simplifications and some tricks in reducing costs in the direct and iterative solvers (see [1]).

For restriction and prolongation matrices, two additional storage formats are supported. A general sparse matrix format, as implemented in the second Yale Sparse Matrix Package (see [16]) is useful on irregular grids. A stencil format is extremely efficient for uniform or tensor product grids. Typically, $r_j + \bar{c}$ storage elements are used, where $r_j = \text{Rows}(\mathcal{R}_j)$ and \bar{c} is a small natural number.

Only Algorithms MGC and NIC are included. There is no support for nonlinear or time dependent problems. Version 2 has been imbedded in other people’s nonlinear and time dependent codes, however. There is also no user call back mechanism, so that if the user has some special solver, preconditioner, or change of level subroutine, the source code for version 2 has to be modified.

Table 1: Solvers and preconditioners

Solver		Preconditioner					
		None	User	ILU	Diag	SGS	SSOR
0	NoSolver	*	*	*	*	*	*
1	User	any	any	*	*	*	*
2	Factor	GD	*	*	*	*	*
3	Solve	GD	*	*	*	*	*
4	Symmetric Gauss–Seidel	G	*	*	*	*	*
5	Gauss–Seidel	GSD	*	*	*	*	*
6	Multi-colored Gauss–Seidel	GSD	*	*	*	*	*
7	Conjugate gradients	GSD	GSD	G	G	G	G
8	Minimum residuals	GSD	GSD	*	*	G	*
9	CGS	G	*	G	G	*	G
10	CGSTAB	G	*	G	G	*	G
11	GMRES	G	*	G	G	*	G

* = Error
 G = General sparse matrices
 S = Stencil matrices
 D = Dense matrices
 any = any format

A 500 line subset of Madpack2 was used to implement a highly efficient multiple coarse space nontelegraphing multilevel solver for problems in two and three dimensions on a variety of parallel computers (see [8], [15], and [14]). The compactness of this package made this easy to do.

3.2 Madpack, Version 4

This is a complete redesign and rewrite of Madpack. It is incompatible with version 2 in numerous ways. This is actually two quite distinct codes, DAMG [11] and DPMG [12]. DAMG is an abstract solver for linear and mildly nonlinear problems (FAS is supported). DPMG is a fast Poisson solver for two and three dimensional problems on simple uniform or tensor product grids.

DAMG supports dense, stencil, and general sparse matrix formats (this time, the more common first Yale Sparse Matrix Package [17] format was used) in the computational kernels. The dense case rarely occurs in solving partial differential equations; it is more common when solving aggregation–disaggregation problems (see [3]). Table 1 contains a summary of the solvers and preconditioners supported in the IBM version.

Unlike version 2, version 4 requires an external library of solvers (there are some solvers provided, but not many). What is distributed by IBM runs only on machines

Table 2: Level independent information data structure

<i>iparm(i)</i>		
<i>i</i>	Symbolic name	Definition
1	<i>mgfn</i>	Which multilevel algorithm
2	<i>l2infm</i>	Second dimension of <i>infm</i> array
3	<i>bysize</i>	Length of <i>b</i> and <i>x</i> arrays
4	<i>lndm</i>	Length of <i>dm</i> array
5	<i>lnim</i>	Length of <i>im</i> array
6	<i>lnjm</i>	Length of <i>jm</i> array
7	<i>levelf</i>	Index of the finest level
8	<i>levelc</i>	Index of the coarsest level
9	<i>startl</i>	Index of the starting level
10	<i>presva</i>	Preserve coarsest level's matrices or not
11	<i>lastdm</i>	Index of last element in <i>dm</i> in use
12	<i>lastim</i>	Index of last element in <i>im</i> in use
13	<i>lastjm</i>	Index of last element in <i>jm</i> in use
14	<i>info</i>	Control of debugging information
15	<i>restart</i>	Continued computation indicator
20	<i>assist</i>	When all else fails

with their proprietary engineering and scientific subroutine library. Currently, this library only runs on IBM mainframes and RISC System/6000 workstations. Since DAMG was originally written on a machine that is not supported by this library, there is obviously a version which uses other libraries, e.g., LAPACK and the first Yale Sparse Matrix Package. Interfacing DAMG to other libraries is now fairly painless.

DAMG accepts three external subroutine arguments in case the user wants to use their own solver(s), preconditioner(s), or change of level subroutine(s). In retrospect, there should have been a fourth for matrix-vector multiplies. These features are used extensively in DPMG to avoid storing matrices.

Both DAMG and DPMG are written in the same extended Ratfor as is version 2. Only the Fortran-77 translation is distributed by IBM, however. The codes assume double precision real data. Changing to single precision only requires changing one line of a file included by each of the Ratfor codes.

DAMG can be restarted after it returns. This allows for coarse levels to be removed from the computational flow. It also allows an external adaptive grid refinement procedure to work with DAMG to add finer levels.

Data is passed to and from DAMG in the standard awkward style imposed by Fortran-77's limitations. Matrices and vectors are piled into a set of five (integer and real) vectors. As a substitute for the more natural pointer data type, indices are stored in information data arrays, indexed by the level number (see Tables 2–

Table 3: Level dependent algorithm information data structure

<i>invalg(i, j)</i> on level <i>j</i>		
<i>i</i>	Symbolic name	Definition
1	<i>Solver</i>	Which solution method
2	<i>SolverIters</i>	Iterations of <i>Solver</i>
3	<i>Precond</i>	Which preconditioning method
4	<i>MGIters</i>	Iterations of Algorithm MGC or MGFAS
5	<i>NIIters</i>	Iterations of Algorithm NIC or NIFAS
6	<i>IdxXB</i>	Index of first element of b_j or x_j in b or x
7	<i>NXB</i>	Number of elements in b_j and x_j
8	<i>Colors</i>	Number of colors in a multicolor ordering

Table 4: Matrix information data structure

<i>infm(i, k, j)</i> on level <i>j</i>					
<i>i/k</i>	1	2	3	4	5
1	AType	RType	PType	NIPType	FASRType
2	ACols	RCols	PCols	NIPCols	FASRCols
3	ARows	RRows	PRows	NIPRows	FASRRows
4	ADim1	RDim1	PDim1	NIPDim1	FASRDim1
5	ADim2	RDim2	PDim2	NIPDim2	FASRDim2
6	IdxA	IdxR	IdxP	IdxNIP	IdxFASR
7	IdxIA	IdxIR	IdxIP	IdxINIP	IdxIFASR
8	IdxJA	IdxJR	IdxJP	IdxJNIP	IdxJFASR

4). A language that supports more reasonable data structures, pointers, and dynamic memory allocation and freeing would simplify this.

Table 2 contains information which is level independent. This includes the length and the index of the last used element of certain vectors, which multilevel algorithm to start with, the indices of the finest, coarsest, and starting levels, how much debugging information to print, and whether this is a restart of an earlier computation.

Table 3 contains information relevant to the computational algorithms which is level dependent. This includes the solver and preconditioner pairing, how many iterations of the algorithms to use on this level, the index into the solution and right hand side vectors for x_j and b_j , and their lengths.

When changing levels, it is very rare that \mathcal{R}_j , \mathcal{P}_j , \mathcal{NIP}_j , and $\mathcal{R}_j^{(FAS)}$ will all be defined. \mathcal{NIP}_j corresponds to a special version of \mathcal{P}_j in step 2a in Algorithm NIC (see §2). Usually only one or two of these will be defined. Further, the matrices are typically related to each other in very particular ways mathematically. An effort has

Table 5: How matrices are chosen for changing levels

<i>Wanted</i>	<i>Order of selection</i>
\mathcal{R}_j	$\mathcal{R}_j, \mathcal{P}_{j+1}^T, \text{ and } \mathcal{NIP}_{j+1}^T$
\mathcal{P}_j	$\mathcal{P}_j, \mathcal{R}_{j+1}^T, \text{ and } \mathcal{NIP}_j$
\mathcal{NIP}_j	$\mathcal{NIP}_j, \mathcal{P}_j, \text{ and } \mathcal{R}_{j+1}^T$
$\mathcal{R}_j^{(FAS)}$	$\mathcal{R}_j^{(FAS)}, \mathcal{R}_j, \mathcal{P}_{j+1}^T, \text{ and } \mathcal{NIP}_{j+1}^T$

been made to allow users of DAMG the option of generating only one matrix when it can be re-used or is the transpose of another matrix. DAMG determines which operation is wanted and then determines from information in the (three dimensional) *infm* data structure (see Table 4) how to change levels. Table 5 contains the order of choice, as determined by which matrix is wanted. The user call back for changing levels is the last choice unless the matrix type specifies doing this.

DPMG uses DAMG to do multileveling. Specialized solvers, interpolation, and projection subroutines are used throughout the computations, however. This means that DPMG does not store matrices normally, thus saving enormous amounts of memory which can be used instead for solving much larger problems. DPMG solves

$$\begin{cases} -\Delta u = b \text{ in } \Omega, \\ u = g_0 \text{ on } \partial\Omega_0, \\ u_n = g_1 \text{ on } \partial\Omega_1, \end{cases} \quad (3.1)$$

where $\partial\Omega_0 \cup \partial\Omega_1 = \partial\Omega$ and $\partial\Omega_0 \cap \partial\Omega_1 = \emptyset$.

This is discretized on grids

$$\bar{\Omega} = \Omega \cup \partial\Omega_0 \cup \partial\Omega_1.$$

In essence, linear systems of the form (2.1)–(2.2) are solved approximately for a sequence of grids $\bar{\Omega}_j$. The vectors x_j and b_j can be thought of as “grid functions” on $\bar{\Omega}_j$. The values of b , g_0 , and g_1 on $\bar{\Omega}_j$ are stored in b_j (multiplied by the square of the mesh spacing when a uniform mesh is used). The values of g_0 on $\partial\Omega_0$ and an initial guess to the solution u in $\Omega \cup \partial\Omega_1$ are stored in x_j before the call to DPMG. DPMG uses a central difference discretization of Poisson’s equation, even at Neumann boundary points. Dirichlet boundary points are not eliminated a priori.

Interpolation is either bilinear, trilinear, or a fourth order method based on (3.1). The latter uses the difference operator, similar to a Gauss–Seidel iteration with a three

color ordering and a rotated operator, to improve the order of the interpolation (see [18]).

Three restriction methods are based on stencils. These are described in detail in [5]. Two second order methods and a fourth order one are provided.

Only Algorithms MGC and NIC are options. The solvers are sparse Gaussian elimination and Gauss–Seidel with either the natural or red–black orderings.

DPMG was designed to run very fast on four quite different architectures:

1. IBM mainframes with vector units.
2. Conventional vector machines.
3. Nonvector machines with multiply-add hardware chaining.
4. Nonvector machines with no fancy hardware.

An example of 2 above is a Cray, an example of 3 is an IBM RISC System/6000 workstation, and an example of 4 is a SUN workstation or a PC.

The Gauss–Seidel with the natural ordering subroutines were rewritten in IBM mainframe vector assembler. These routines are always faster than the Fortran equivalents no matter what size vectors are used. As an interesting aside, a version was produced that completely vectorizes by using an odd re-interpretation of how to compute the updates based on the trailing vector elements that normally do not vectorize. This is described in [10]. The trick does not work in Fortran, C, or C++ unfortunately.

The usual philosophy for vectorizing Gauss–Seidel is to use a red–black ordering. In addition, this allows the interpolation subroutines to ignore half of the fine grid points. However, the red–black ordering has an unfavorable feature. The right hand side and approximate solution vectors pass through cache twice per iteration. Only if a solver is written in a blocked by the cache size manner can this be alleviated. Due to the boundary conditions in (3.1) and the fact that the matrices are not stored in DPMG, this makes things overly complicated to program. Hence, DPMG uses a traditional implementation for the red–black subroutines.

While the multilevel convergence properties of red–black Gauss–Seidel are better than the naturally ordered one, both solvers provide about equal performance when using Algorithm NIC and a V Cycle.

4 Language Issues

In this section, advantages and disadvantages of Fortran, C, and C++ will be discussed in the context of an abstract multilevel solver. A mixed solution will be proposed.

4.1 Fortran

In §3.2, the disadvantages of Fortran-77 in terms of data structures was discussed. There is no conceivable way to get around this. Even using macros or Ratfor only help

Table 6: Matrix Structure

```
struct Matrix {
  int   MatrixType;      /* the matrix type */
  int   MatrixCols;     /* number of columns */
  int   MatrixRows;     /* number of rows */
  int   MatrixLDim;     /* leading dimension for dense matrices */
  void  *MatrixCoeffs;  /* Pointer to matrix elements */
  int   *MatrixIA;      /* Pointer to IA elements */
  int   *MatrixJA;      /* Pointer to JA elements */
};
typedef struct Matrix Matrix; /* Simplify Matrix declarations */
```

so much. The real problem is that users of the package still have to initialize the data structures. They are not likely to use either my macros or Ratfor.

DAMG uses scratch storage in its solvers. Predicting the amount needed for each (solver, preconditioner) pair is an art which no user should ever have to master. Worse, the formulas given for some popular sparse matrix iterative solvers are wrong (predicting less memory than is required). For all of the solvers used in §3, the amount of scratch storage can be written in terms of N (the number of rows or columns), NZ (the number of nonzeros in A_j), and a constant:

$$N_{scr} = C_n \cdot N + C_{NZ} \cdot NZ + C_{extra}. \quad (4.1)$$

While default values can be used, the user should be able to override these.

However, there are some areas where Fortran shines. For one, real and complex data types of various word lengths are part of the language. So, by using a simple preprocessor (e.g., m4 or GNU m4) that is available on most computer systems used by people who do scientific computation, one source code can be maintained, even if multiple subroutine names are generated, one per data type supported. For example, in the Ratfor source code for DAMG, subroutine `mgal` is referenced by `NameIt(mga1)`. `NameIt` prepends the letter *d* (double real), *s* (single real), *z* (double complex), or *c* (single complex) depending on the definition of a macro, `FLOAT`.

Another area where Fortran does well is in optimizing code for certain classes of machines, particularly ones with vector units. I naively assumed vector machines would go like the dinosaurs with the advent of superscalar, very fast workstations. Unfortunately (or fortunately depending on your point of view), vector units are being glued onto superscalar workstations by several manufacturers. While some C compilers have made serious inroads on producing very high quality code, Fortran still holds some advantages in this case.

4.2 C

This language has an obvious disadvantage since complex and double complex are not a part of the language. While either of these can be defined as a structure, computing with them is inexcusably awkward. In particular, maintaining a single set of solvers for real and complex data means writing a set of weird macros to do floating point arithmetic. This is unacceptable.

However, not all of DAMG's or DPMG's subroutines are solvers. In fact, the multi-level algorithm or choose which solver to call subroutines are really doing bookkeeping, not floating point arithmetic. For these subroutines, C provides all of the necessary features to dramatically simplify the entire calling sequence and these subroutines. Just being able to dynamically allocate and free memory would reduce the user's frustration level with trying to guess how much memory to pass to DAMG for scratch storage.

C can easily save addresses of objects, e.g., of subroutines or data objects, in complicated data structures. Hence, routines can be called incrementally to pass very complex data objects to an implementation of an abstract multilevel algorithm without any one call being very complicated. This reduces the aggravation of using a complex program considerably.

4.3 C++

Many of the positive comments about C apply directly to C++. Classes can be constructed instead of structures. Further, C++ usually comes with a complex class (but not necessarily in both single and double precision), alleviating C's worst drawback.

One of C++'s strongest design features is the ability to design classes abstractly. At run time, the correct version of some virtual routine is accessed. This feature, while useful, is overkill in the context of abstract multigrid solvers. The data type *void** in C, a pointer to any data type, is sufficient to overcome many of the reasons why C++ would be useful in this context (see §4.4).

A drawback to using C++ is that there is frequently a lot of overhead hidden from the user. This makes C++ programs run unnecessarily slower than the equivalent C or Fortran programs. Due to the manner most C++ compilers change the name of functions, interfacing C++ programs to C or Fortran calling programs is extremely challenging, too.

A more serious drawback is that C++ has not yet been standardized. It is evolving with major new versions coming out yearly. This would not be so bad except that features are sometimes dropped or changed in incompatible ways in newer versions of the language. For someone who wants to write a code once and then never have to touch it again, this is not a good point in C++'s favor.

Table 7: External subroutine information structure

```
struct ExternSubr {
    int      (*Subr)(); /* Pointer to integer function */
    int      *IParms; /* Pointer to integer parameters */
    void     *FParms; /* Pointer to floating point parameters */
    double   CN;      /* See (4.1) */
    double   CNZ;     /* See (4.1) */
    double   Cextra;  /* See (4.1) */
    int      SaveScr  /* Save scratch areas between calls? */
    void     **Scrs   /* Vector of pointers to scratch areas */
    int      *NScrs   /* Vector of lengths of scratch areas */
};
typedef struct ExternSubr ExternSubr; /* Simplify ExternSubr declarations */
```

4.4 C and Fortran: Mixed Language Programming

My personal belief is that mixing Fortran+preprocessor and C is the best choice now. The worst disadvantage is that to compile a mixed language library, some knowledge is required of how the local compilers treat subroutine names. There are three common methods in use for Fortran (e.g., convert to all lower case, convert to all upper case, and convert to all lower case and append an underscore). On many platforms this can be determined automatically. On a very small number of machines, Fortran and C programs cannot be mixed conveniently or at all; these machines will be ignored by this author.

Algorithms MGC and NIC should be implemented in C and the computational solvers should be implemented in FORTRAN+m4. Numerous people who compute only know one language well and are not comfortable normally with a mixed language set of programs. An interface is described in §5 which let these people use what is proposed.

Suppose that we make no assumption about the language of a solver or preconditioning routine, other than it really can be called from C. Then we do not know if it can dynamically allocate memory. Hence, some mechanism must be defined for passing a block of memory. One way is to define a structure for externally called subroutines, e.g., Table 7. The subroutine is expected to return some indication of whether or not it worked or produced an error. The IParms and FParms are integer and floating point vectors containing information that the specific subroutine actually needs. Setting CN=CNZ=0 and Cextra=10 signifies “use the defaults.” Note that only one ExternSubr structure has to be created per subroutine. In this definition, Subr is a pointer (or external reference) to an integer valued function with a fixed set of arguments. By providing an include file with an abstract solver, a set of default ExternSubr structures

Table 8: Level Information Structure

```

struct LevInfo {
  ExternSubr *solver;      /* Pointer to how to call solver */
  ExternSubr *precond;     /* Pointer to how to call preconditioner */
  ExternSubr *matrix_vec; /* Pointer to how to call matrix*vector */
  ExternSubr *change_lev; /* Pointer to how to call level changer */
  int SolverIters;        /* Number of iterations in solver() */
  double SolverRNorm;    /* How much to reduce residual norm */
  int MGIters;           /* Number of iterators of MGC */
  int NIIters;          /* Number of iterators of NIC */
  void *Xj;             /* Pointer to  $x_j$  */
  void *Bj;             /* Pointer to  $b_j$  */
  int NXj;              /* Length of  $x_j$  */
  int NBj;              /* Length of  $b_j$  */
  int NZAj;            /* Number of nonzeros in  $A_j$  */
  Matrix *Aj;           /* Pointer to  $A_j$  representation */
  Matrix *Rj;           /* Pointer to  $\mathcal{R}_j$  representation */
  Matrix *Pj;           /* Pointer to  $\mathcal{P}_j$  representation */
  Matrix *NIPj;         /* Pointer to  $\mathcal{NIP}_j$  representation */
  Matrix *FASRj;        /* Pointer to  $\mathcal{R}_j^{(FAS)}$  representation */
};
typedef struct LevInfo LevInfo; /* Simplify LevInfo declarations */

```

can be given to the user (see Table 1).

Consider Table 4. A single structure can be defined that defines everything in a column of Table 4, so that information about matrices can be made easier to define. Also, pointers to the actual floating point and integer vectors or matrices can be defined (instead of indices into a messy vector), placing all of the relevant information in one place (see Table 6).

Information that is in both Tables 3 and 4 can be re-arranged into a single data structure as in Table 8. A NULL pointer can be used to indicate the lack of existence of a matrix. The user, however, should not have to set any of this.

Implementations of Algorithms MGC and NIC now can be reduced to evaluating a series of object oriented functions. These include matrix–vector multiplies and calls to external subroutines. The information in the LevInfo and ExternSubr structures can be used to first allocate scratch space (if necessary) before calling the routines.

Numerous iterative procedures, based primarily on conjugate direction methods, require a user call back routine to calculate matrix–vector products. Also, many iterative procedures allow a stopping criterion based on reducing the (possibly scaled) residual norm by some amount, e.g., SolverRNorm in Table 8.

There is an important issue that must be addressed. There are many people who

Table 9: Data types in Madpack5

<i>dtype</i>	Data	Floating point data description
1	float	single precision real
2	double	double precision real
3	complex	single precision complex
4	dcomplex	double precision complex
<0	user	–value = length in bytes

compute who do not know C, but only Fortran. Using the data structures advocated in §4.2 would preclude these people from using the abstract solvers. Further, requiring these people to learn either C or C++ before using this approach is silly.

5 Madpack, Version 5

Requiring users to fill in all the data structures required to use an abstract multi-level solver is considered by many users too difficult for the end result. A serious effort was undertaken to eliminate the users' ability to even know what that the structures are. Discussions with a number of users of Madpack2 and Madpack4 resulted in a very different approach. For one thing, it became clear that as workstations have become more powerful, more people are trying to solve large three dimensional problems, including ones with complex data, not just real.

Some simple subroutines, callable from Fortran (or many other languages) that build the data structures for the user are included. For example, a Fortran program can call a C program which returns a *data handle* (a small integer):

$$mgh = mgini(levels, dtype)$$

This subroutine allocates space for the structures. The integer argument *dtype* (see Table 9) is used to determine the data type (c.f., the value of FLOAT in §4.1). While this may seem ugly, this simple mechanism allows the C codes to be written in a “typeless” manner. Note that a mechanism is in place for user defined data types as well. Also note that the definition of single or double precision can vary from one computer architecture to the next.

Matrix structures are declared similarly and return a *matrix handle*:

$$mat = mgmat(mgh, type, cols, rows, ldim, coeffs, ia, ja)$$

Matrix handles are coupled to the data handle *mgh*. Hence, if a matrix is used in another call to *mgini*, *mgmat* must be called again.

Subroutines are declared using another routine:

double precision CN, CNZ, Cextra
 external rtn

...
 (set CN, CNZ, and Cextra)
 $isubr = mgsubr(mgh, rtn, iparms, fparms,$
 & CN, CNZ, Cextra, savscr)

Note that only the addresses of *rtn*, *iparms*, and *fparms* are saved by *mgsubr*, not the contents. A *subroutine handle* is returned which is coupled to the data handle *mgh*. Use of the Fortran EXTERNAL declaration allows subroutine addresses to be passed.

Another routine can be called to setup a LevInfo structure for level *j*:

$iret = mglevi(mgh, j, isolver, iprecond, imatv, ichlev,$
 & $nsolviters, rnorm, mgiter, niiter, xj, bj, nxj, nbj,$
 & $nza, mata, matr, matp, matnip, matfas)$

Here, *isolver*, *iprecond*, *imatv*, and *ichlev* are the return values from *mgsubr* or 0 if none is wanted. Also, *mata-matfas* are return valves from *mgmat* or 0 if no corresponding matrix exists. The *x-j* and *b-j* are the addresses of the first elements of *xj* and *bj*. These may be indexed as X(ixb) and B(ixb), respectively, depending on the user's programming style. A nonzero return value means an error occurred.

Finally, the multilevel subroutines can be called:

$iret = mgmeth(mgh, iparm, resid)$

where *iparm* is a simplification of the one in Table 2 (it only needs to contain *mgalg*, *startl*, *levelc*, *levelf*, and *info*, but is extendable). The last argument, *resid*, is an array where the final residual is returned. A nonzero return value means an error occurred.

To free space, a final call can be made:

$iret = mgdone(mgh)$

A nonzero return value means an error occurred. Obviously, this last call is unnecessary if the program immediately ends.

The advantage of this approach is that subroutines can be written in whatever language makes the most sense. Further, people who program in any language will not be penalized by having to construct data structures that only make sense in Fortran.

Madpack5 comes with a number of solvers. These typically have a small C program to do dynamic memory manipulation and to call the correct Fortran+m4 solver based on the data type (this is done using indirect addressing from a table). There is only one Fortran+m4 source file, but the code is compiled up to four different ways for the data types in Table 9. To use one of the solvers, the user must know the generic name (from the C program), not a collection of names based on data type.

The solvers provided are Bi-CGSTAB, point Gauss-Seidel (natural ordering, a parallel domain decomposition style) Symmetric Gauss-Seidel, conjugate gradients (Symmetric Gauss-Seidel preconditioned), Orthomin(1) (Symmetric Gauss-Seidel preconditioned), and sparse Gaussian elimination. Interfaces to some other packages are provided, e.g., LAPACK, the old Yale Sparse Matrix Package, and Scientific Computing Associates' Sparse Matrix Package (SMPAK).

A typical calling program for Madpack5 has the following form:

1. Data declarations.
2. Call to *mgini*.
3. All calls to *mgsubr*.
4. Repeat 4a–4 over the number of levels $j = 1, \dots, k$:
 - 4a. Generate A_j and register it with a call to *mgmat*.
 - 4b. If $j < k$, then generate \mathcal{R}_j and register it with a call to *mgmat*.
 - 4c. Generate x_j and b_j .
 - 4d. Call to *mglevi* to finish registering information.
5. A call to *mgmeth* to solve the problem.

In this example, no prolongation matrices are produced and it is assumed to be the transpose of the appropriate restriction matrix (see Table 5).

Suppose we use this template to make a multilevel solver for the boundary–value problem,

$$-\nabla \cdot (a \nabla u) + su = f \text{ in } \Omega \quad ; \quad u = 0 \text{ on } \partial\Omega.$$

If we solve a slightly different problem, namely,

$$-\nabla \cdot (a \nabla u) + su = f \text{ in } \Omega \quad ; \quad u_n + i\omega u = g \text{ on } \partial\Omega,$$

then the following will change:

1. The data declarations section will change from real data to complex.
2. The second argument in the call to *mgini* will increase by 2.
3. The routine to generate A_j will change to reflect the new boundary condition.

Note that the section with the calls to *mgsubr* did not change. In fact, the number of changes to the main program is trivial.

6 Future Directions and Conclusions

Madpack5 currently is an abstract multilevel solver. A collection of programs to solve some example boundary value problems are provided to show how to use the library. An effort is under way to provide several routines for solving general partial differential equations on nontrivial grids using a variety of discretization methods. An adaptive mesh refinement module for piecewise linear triangular meshes is a part of this.

Another effort is under way to provide the same simplicity as described in §5 in a distributed or parallel computing environment. A domain decomposition style of solvers, including ones which work to some degree asynchronously (semi–chaotic solvers), are being added.

Finally, abstract multilevel methods were reviewed in this paper. Three versions of the author's publically distributed multilevel codes (Madpack) were discussed. From the experience of these codes, a model of a better approach using a mixed language approach (C and Fortran+m4) was proposed. Implementing such a system, starting from having already working solvers (e.g., [7], [11], and [12]) was a simple exercise.

The end result is a much more general package (Madpack5) which is easily called from programs written in many common languages, including C, C++, Fortran, and Pascal. It would be straight forward to produce an interface to Mathematica or Matlab as well.

Acknowledgments: This work was supported in part by an ONR grant.

REFERENCES

- [1] R. E. Bank and C. C. Douglas. An efficient implementation of the SSOR and ILU preconditionings. *Appl. Numer. Math.*, 1:489–492, 1985.
- [2] R. E. Bank and R. K. Smith. General sparse elimination requires no permanent integer storage. *SIAM J. Sci. Stat. Comp.*, 8:574–584, 1987.
- [3] F. Chatelin and W. L. Miranker. Acceleration by aggregation of successive approximation methods. *Lin. Alg. Appl.*, 43:17–47, 1982.
- [4] C. C. Douglas. Mgnnet Digests and Code Repository. Monthly digests subscribed to by sending a message to mgnnet-requests@cs.yale.edu and an anonymous ftp site (casper.cs.yale.edu) for codes and papers on multigrid and related topics.
- [5] C. C. Douglas. *Multi-grid algorithms for elliptic boundary-value problems*. PhD thesis, Yale University, May 1982.
- [6] C. C. Douglas. Multi-grid algorithms with applications to elliptic boundary-value problems. *SIAM J. Numer. Anal.*, 21:236–254, 1984.
- [7] C. C. Douglas. Madpack (version 2) users' guide. Technical Report 16169, IBM Research Division, Yorktown Heights, New York, 1990. The most up to date source code is available through anonymous ftp from casper.cs.yale.edu in the directory mgnnet/madpack2.
- [8] C. C. Douglas. A tupleware approach to domain decomposition methods. *Appl. Numer. Math.*, 8:353–373, 1991.
- [9] C. C. Douglas. MGNet: a multigrid and domain decomposition network. *ACM SIGNUM Newsletter*, 27:2–8, 1992.
- [10] C. C. Douglas. Some remarks on completely vectorizing point Gauss–Seidel while using the natural ordering. *Advances in Computational Mathematics*, 2:215–222, 1994.

- [11] C. C. Douglas. DAMG: an abstract multilevel solver. Technical Report YALEU/DCS/TR-950, Department of Computer Science, Yale University, New Haven, 1993.
- [12] C. C. Douglas. DPMG: a multigrid solver for the poisson equation in two and three dimensions. Technical Report YALEU/DCS/TR-951, Department of Computer Science, Yale University, New Haven, 1993.
- [13] C. C. Douglas and J. Douglas. A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel. *SIAM J. Numer. Anal.*, 30:136–158, 1993.
- [14] C. C. Douglas and J. Mandel. A group theoretic approach to the domain reduction method. *Computing*, 48:73–96, 1992.
- [15] C. C. Douglas and W. L. Miranker. Constructive interference in parallel algorithms. *SIAM J. Numer. Anal.*, 25:376–398, 1988.
- [16] S. C. Eisenstat, H. E. Elman, M. H. Schultz, and A. H. Sherman. The (new) Yale sparse matrix package. In A. L. Schoenstadt and G. Birkhoff, editors, *Elliptic Problem Solvers II*, pages 45–52. Academic Press, New York, 1983.
- [17] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package: II. the nonsymmetric codes. Technical Report 114, Department of Computer Science, Yale University, New Haven, 1977.
- [18] J. M. Hyman. Mesh refinement and local inversion of elliptic differential equations. *J. Comp. Phys.*, 23:124–134, 1977.