Yale University
Department of Computer Science

DAMG: An Abstract Multilevel Solver

Craig C. Douglas

YALEU/DCS/TR-950
February 9, 1993

This work was supported in part by the Office of Naval Research (grant N00014-91-J-1576), Yale University, and the Research Division of International Business Machines.

DAMG: AN ABSTRACT MULTILEVEL SOLVER*

CRAIG C. DOUGLAS†

Abstract. A fast solver (DAMG) for linear algebra problems or partial differential equations, based on multgrid methods, is presented. DAMG can be used with boundary value problems defined on uniform, tensor product, or arbitrary grids in any number of dimensions. The calling sequence is described in detail. What the subroutine library does and returns is also described.

Key words. multigrid, partial differential equations, linear algebra

AMS(MOS) subject classifications. 65N20, 65F10, 65F05.

1. Introduction. Differential equations provide mathematical descriptions of numerous physical phenomena. This field can be divided into numerous categories, the main ones being ordinary (only one variable is differentiated) and partial (more than one variable is differentiated).

Ordinary differential equations arise in the study of electrical circuits and oscillating mechanical systems,

$$ay''(x) + by'(x) + cy(x) = f(x),$$

and cable suspension,

$$y''(x) = \frac{w}{H}\sqrt{1 + (y'(x))^2}.$$

They also arise when the technique of separation of variables is applied to a partial differential equation:

$$\frac{d}{dx}\left(a(x)\frac{dy}{dx}\right) + b(x)y = f(x).$$

Partial differential equations can be characterized by

$$\text{(1)} \qquad \begin{cases} \mathcal{L}u(x) &= f(x), \\ \mathcal{B}u(x) &= g(x), \end{cases}$$

where x is a vector. In (1), $\mathcal{L}u(x) = f(x)$ represents the problem to be solved, subject to the boundary and/or initial conditions $\mathcal{B}u(x) = g(x)$.

* This work was supported in part by the Office of Naval Research, grant N00014-91-J-1576. Yale University Department of Computer Science Research Report YALEU/DCS/TR-950, February, 1993.

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

Some common partial differential equations are Poisson's equation,

$$u_{xx} + u_{yy} = f(x, y)$$

with the special case of $f(x, y) = 0$ (Laplace's equation), the heat or diffusion equation,

$$u_t - a^2 u_{xx} = 0.$$

and the wave equation,

$$u_{tt} - a^2 u_{xx} = 0,$$

These three equations are examples of elliptic, parabolic, and hyperbolic partial differential equations respectively.

There are three common classes of boundary conditions. The first is known as a Dirichlet condition: the value of $u(x)$ is specified along the boundary. The second is known as a Neumann condition: the value of the normal derivative du/dn is specified along the boundary. The third is known as a mixed condition: $\gamma u(x) + \psi du/dn$. More complicated conditions exist.

These problems are converted into finite dimensional ones using finite element or difference schemes, collocation, or box schemes (sometimes referred to as a finite volume schemes). Numerous books exist on how to do this, when the problems are well posed, and how to determine when the discretization is stable and consistent (see [2], [5], [6], [9], and [11]).

2. Introduction to multigrid methods. Once a linear differential or integral equation is discretized, we must solve

$$Ax = b, \quad x \in \mathcal{M},$$

where $\mathcal{M}$ is a vector space. We will solve this using an abstract multilevel (or multigrid) iteration. An auxiliary set of equations are used which each approximate the original one:

$$A_j x_j = b_j, \quad levelf \leq j \leq levelc, \quad x_j \in \mathcal{M}_j,$$

where $levelc$ and $levelf$ be the coarsest and finest levels, respectively, $A_{levelf} = A$, $x_{levelf} = x$, and $b_{levelf} = b$. Neither symmetry nor definiteness (positive or negative) is required in the A_j's.

Multigrid solvers frequently use particular features of an elliptic boundary value problem and the domain. There are similar procedures, known as aggregation-disaggregation methods, when A is not derived from partial differential equations; this routine can be used with either of these procedures. The term multigrid is usually applied only to problems based on grids, whereas the term multilevel is applied to problems which may or may not be grid based. We want to apply this method independent of the properties of the grid, domain, discretization, and differential equation.

Multilevel methods combine scaled iterative methods (called *smoothers* or *roughers*) with iterative residual correction on coarser levels to reduce the error on a given level. Iteration i on some level $j > levelc$ consists of a smoothing step (introducing an operator $S_j^{(i)}$), a correction step, and another smoothing step (introducing another operator $T_j^{(i)}$). There are μ_j of these iterations. On level $j = levelc$, just smoothing occurs (say, $S_j^{(i)}$); this may be an iterative or a direct procedure like (sparse) Gaussian elimination.

Common smoothers $S_j^{(i)}$ and $T_j^{(i)}$ are relaxation methods (e.g., Gauss-Seidel, SOR, line or plane methods), preconditioned conjugate direction methods (e.g., conjugate gradients, minimum residuals, Orthomin, CGS, CG–STAB), and the identity operator.

The correction step involves a two way transfer of information between levels. This is accomplished using mappings between the solution spaces:

$$R_j : \mathcal{M}_j \rightarrow \mathcal{M}_{j+1} \quad \text{and} \quad P_{j+1} : \mathcal{M}_{j+1} \rightarrow \mathcal{M}_j.$$

These are referred to as *restriction* and *prolongation* operators in the multigrid literature. Typically, P_{j+1} is a standard interpolation operator and R_j is its transpose.

There are two basic linear multilevel algorithms: correction ones and nested iteration ones. Correction multilevel algorithms start on a fine grid and use the coarser levels solely to correct the approximate solution on finer levels (we will define two such algorithms shortly, namely, MGC and MGFAS). Nested iteration multilevel algorithms start on a coarse level and work their way to some finer level, using the approximate solution on coarser levels to produce initial guesses and corrections on the finer levels (we will define two such algorithms shortly, namely, NIC and NIFAS).

Define a k-level correction multigrid algorithm by

ALGORITHM MGC($k, \{\mu_\ell\}, x_k, f_k$)
- (1) If $k = levelc$, then solve $A_k x_k = f_k$ exactly or iteratively
- (2) If $k \neq levelc$, then repeat $i = 1, \cdots, \mu_k$:
  - (2a) Smoothing: $x_k \leftarrow S_k^{(i)}(x_k, f_k)$
  - (2b) Residual Correction: $x_k \leftarrow x_k +$
    $P_{k+1}($ MGC$(k+1, \{\mu_\ell\}, 0, R_k(A_k x_k + f_k)))$
  - (2c) Smoothing: $x_k \leftarrow T_k^{(i)}(x_k, f_k)$
- (3) Return x_k

This definition requires that $\mu_{levelc} = 1$. Examples of the flow of control between levels for both of the correction algorithms are contained in Figure 1.

Define a k-level (*standard*) full approximation multigrid scheme by

$\text{ALGORITHM MGFAS}(k, \{\mu_\ell\}, x_k, f_k)$

(1) If $k = levelc$, then solve $A_k x_k = f_k$ exactly or iteratively

(2) If $k \neq levelc$, then repeat $i = 1, \cdots, \mu_k$:

(2a) Smoothing: $x_k \leftarrow S_k^{(i)}(x_k, f_k)$

(2b) Residual Correction:

$$x_k \leftarrow x_k + P_{k+1}(\ \text{MGFAS}(k+1, \{\mu_\ell\}, 0, R_k(A_k x_k + f_k) - A_{k+1} R_k x_k) - R_k x_k)$$

(2c) Smoothing: $x_k \leftarrow T_k^{(i)}(x_k, f_k)$

(3) Return x_k

This definition requires that $\mu_{levelc} = 1$. Examples are contained in Figure 1. Algorithm MGFAS can be used to solve nonlinear problems by using nonlinear smoothers. For linear problems, it is equivalent to Algorithm MGC.

Define a k-level nested iteration multigrid algorithm by

$\text{ALGORITHM NIC}(k, \{\mu_\ell, \psi_\ell\}, x_{levelc}, \{f_\ell\})$

(1) For $j = levelc, levelc - 1, \cdots, k$, do

(1a) If $j \neq levelc$, then $x_j \leftarrow P_{j+1} x_{j+1}$

(1b) Set $\mu \leftarrow \mu_j$ and then $\mu_j \leftarrow \psi_j$.

(1c) $x_j \leftarrow \text{MGC}(j, \{\mu_\ell\}, x_j, f_j)$

(1d) Restore $\mu_j \leftarrow \mu$.

(2) Return x_k

Alternatively, MGFAS can be substituted for MGC.

$\text{ALGORITHM NIFAS}(k, \{\mu_\ell, \psi_\ell\}, x_{levelc}, \{f_\ell\})$

(1) For $j = levelc, levelc - 1, \cdots, k$, do

(1a) If $j \neq levelc$, then $x_j \leftarrow P_{j+1} x_{j+1}$

(1b) Set $\mu \leftarrow \mu_j$ and then $\mu_j \leftarrow \psi_j$.

(1c) $x_j \leftarrow \text{MGFAS}(j, \{\mu_\ell\}, x_j, f_j)$

(1d) Restore $\mu_j \leftarrow \mu$.

(2) Return x_k

An example of the flow of control between levels for both of the nested iteration algorithms is contained in Figure 2.

3. Storage formats. Any collection of subroutines to solve partial differential equations or integral equations must support a variety of matrix storage formats. Discretizations of partial differential equations usually lead to large, sparse systems of equations. This will result in diagonal matrices, ones with a nearly constant number of nonzeros per row, or ones with a highly varying number of nonzeros per row. In all cases, the nonzero structure of the resulting matrix is usually nearly symmetric even when the matrix is not. On the other hand, integral equations and spectral discretizations of partial differential equations usually lead to dense systems of equations.

Currently, one standard dense storage format is supported:

- "general matrix"

One standard sparse storage format plus one nonstandard one are supported:

- "storage by rows"
- "stencil storage mode"

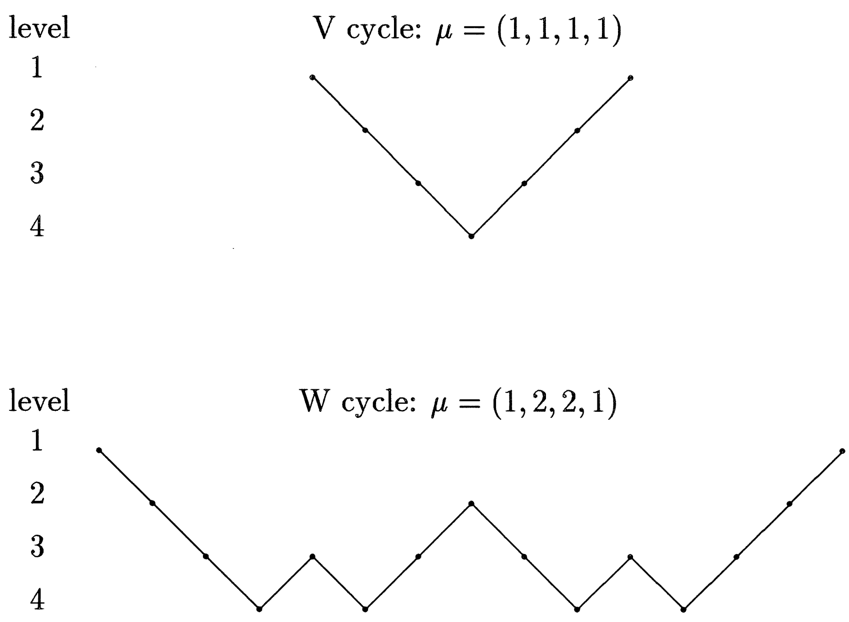


FIG. 1. *Correction Algorithms (MGC and MGFAS) V and W cycles*

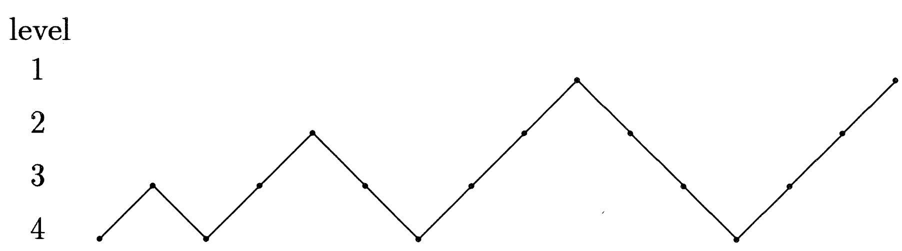


FIG. 2. *Nested Iteration Algorithms (NIC and NIFAS) V cycle*

More formats should be supported at some point in the future.

3.1. General dense matrix. General dense matrices are stored in column major form (i.e., Fortran's standard method, not C's). For a given dense matrix M, only one matrix DM is required to store the elements.

- DM, a long precision matrix of dimension (ndm, n) with $ndm \geq m$, contains the element M_{ij} in $DM(i,j)$.

Consider the following as an example of a 5×5 general dense matrix M:

$$M = \begin{bmatrix} 11 & 0 & 13 & 0 & 0 \\ 21 & 22 & 0 & 0 & 25 \\ 0 & 0 & 33 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 51 & 52 & 0 & 0 & 55 \end{bmatrix} \tag{2}$$

which can be stored as a matrix as

$$DM_{matrix} = M$$

or as a vector as

$$\begin{aligned} DM_{vector} = (\ & 11, \ 21, \ 0, \ 0, \ 51, \\ & 0, \ 22, \ 0, \ 0, \ 52, \\ & 13, \ 0, \ 33, \ 0, \ 0, \\ & 0, \ 0, \ 0, \ 0, \ 0, \\ & 0, \ 25, \ 0, \ 0, \ 55 \ ) \end{aligned}$$

For a symmetric matrix, obviously half of the storage could be eliminated. At this time, this is not supported (stay tuned).

3.2. Storage by rows. For a general sparse matrix M, storage by rows uses three vectors to define the matrix: IM, JM, and DM. Given the $m \times n$ sparse matrix M having ne nonzero elements, the vectors are set up as follows:

- DM, a long precision vector of length at least ne, contains the ne nonzero elements of the sparse matrix M stored contiguously. The rows of M are stored in ascending order. The elements of each row in M are stored in any order, but ascending order should be used if possible.
  WARNING: Some of the iterative solvers actually require the rows to be stored in ascending order.
- IM, an integer vector of length at least $m + 1$, contains the relative starting position of each row of matrix M in vector DM. Hence, row i of M begins at $DM(IM(i))$ and ends at $DM(IM(i + 1) - 1)$. If row j is all zero (i.e., an empty row), $IM(j) = IM(j + 1)$. The last element, $IM(m + 1)$, indicates the position after the last element in vector DM, which is $ne + 1$.
- JM, an integer vector of length at least ne, contains the corresponding column numbers of each nonzero element M_{ij} in matrix M.

Consider the example matrix M in (2) cast as a 5×5 general sparse matrix. This can be stored as

$$DM = (11, 13, 21, 22, 25, 33, 51, 52, 55)$$

$$IM = (1, 3, 6, 7, 7, 10)$$

$$JM = (1, 3, 1, 2, 5, 3, 1, 2, 5)$$

For a symmetric matrix, obviously half of the storage could be eliminated. At this time, this is not supported (stay tuned).

3.3. Stencil storage mode. Interpolation between similar grids is an important feature of multigrid algorithms when solving partial differential equations (see §2). Suppose we have two grids G_1 and G_2, with N_1 and N_2 grid points, respectively, and $N_1 \gg N_2$. For a d dimensional problem, $N_1 \approx 2^d N_2$ is common.

Suppose an $N_2 \times N_1$ matrix R_1 is defined based on the grids G_1 and G_2, i.e.,

$$R_1 : \mathbb{R}^{N_1} \rightarrow \mathbb{R}^{N_2}.$$

Then we can transfer information from G_1 to G_2 or vice versa using the following (sparse) matrix–vector multiplications:

$$y = Rx \quad \text{or} \quad x = R^T y, \quad x \in \mathbb{R}^{N_1},\ y \in \mathbb{R}^{N_2}.$$

Frequently, the rows of R_1 are similar to many other rows in R_1 except that the first nonzero is in a different column. The matrix R_1 is really a collection of weighted sums of values of some function or vector at grid points and a few of the values at neighboring grid points.

We introduce a *stencil storage mode* which is very space efficient for regular grids. Information about each stencil is stored in two vectors: one integer (fullword) and the other real (long precision). The integer one is dimensioned N_2 larger than the real one.

This storage format is designed only to do the sparse–matrix vector multiplies using little memory while still being fast. We calculate $y = R_1 x$ using the following algorithm:

(1) $j = 1$.

(2) For $i = 1, \cdots, N_2$ do:

(2a) Let p be the stencil associated with y_i.

(2b) $y_i = \sum_{\ell \in Stencil_p} r_\ell \sum_{k \in Offset_{p,\ell}} x_{j+k}$.

(2c) $j = j + Increment_p$.

Each stencil p has a set of multipliers ($\{r_\ell\}$). Associated with each r_ℓ is a set of offsets ($\{Offset_{p,\ell}\}$) and an increment ($Increment_p$). We will define all of these terms more concretely shortly.

Suppose rows and i and $i+1$ of some example R_1 are the following:

$$\begin{array}{ccccccccccccc} 0 & \cdots & 0 & 1 & 2 & 4 & 2 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & 1 & 2 & 4 & 2 & 1 & 0 & \cdots & 0 \end{array}.$$

Let stencil p encapsulate row i's stencil (1,2,4,2,1). This is stored as part of vectors R and JR:

R	JR	Description	What
1.0	2	2 entries to multiply by 1.0	$Offset_{p,1}$
	0	offset $= 0$	
	4	offset $= 4$	
2.0	2	2 entries to multiply by 2.0	$Offset_{p,2}$
	1	offset $= 1$	
	3	offset $= 3$	
4.0	1	1 entry to multiply by 4.0	$Offset_{p,3}$
	2	offset $= 2$	
	0	End of stencil indicator	$Increment_p$
	2	add 2 to j in (2c)	

The blank entries in R are never referenced, so they can be anything (zero is a safe value, however). In the algorithm for computing $y = R_1 x$, there is an outer loop (the i loop) in which y_i is computed, one row at a time. However, the starting index for x increases by a variable amount (which is rarely 1 in practice) in line (2c). Under unusual circumstances, the increment can actually be zero or negative. This is what is stored in $Increment_p$.

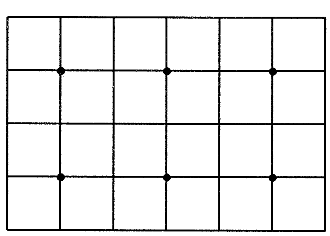
In this example, row $i+1$'s stencil is the same stencil as that for row i if and only if the increment is identical for the two rows. This anomaly results in very similar stencils differing only in the increment.

Stencil storage mode matrices have three distinct components: two pointer sections and the stencils' section. The term *pointer* is used here to refer to an index into a FORTRAN-77 style vector. Formally,

Index	R	JR
1	*	Pointer to stencil pointers
2 to K	Stencils	
$K+1$ to $K+N_2$		Pointers to stencils to compute y_i

Note that the R and JR vectors are of length K and $K+N_2$, respectively, in the description above.

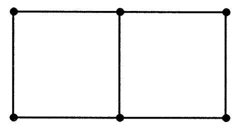
This section is concluded with two real examples: both are for transferring data from a grid (in two or three dimensions) to another. Let G_1 be a rectangular, uniform grid with $N_1 = (2N_x+1) \times (2N_y+1)$ points:



$2N_x + 1 = 7$

$2N_y + 1 = 5$

Let $G_2 \subset G_1$ correspond to the $N_2 = N_x \times N_y$ points singled out above:



$N_x = 3$

$N_y = 2$

Suppose we use a stencil that is a nine point weighting of nearby neighbors, centered at the points where $G_1 \cap G_2$, of the form

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

The 1 entry corresponds to the the center of the stencil. Then $R_1 : G_1 \rightarrow G_2$ will have two nearly identical stencils. The first will have an increment of 2, the second will have an increment of $N_y + 3$. Each stencil has length 14, so

$$JR(1) = 30.$$

Stencil 1 is stored in R and JR as

Index	R	JR	Description
2	.25	4	$Offset_{1,1}$
3		0	
4		2	
5		$2N_y$	
6		$2N_y + 2$	
7	.5	4	$Offset_{1,2}$
8		1	
9		N_y	
10		$N_y + 2$	
11		$2N_y + 1$	
12	1.0	1	$Offset_{1,3}$
13		$N_y + 1$	
14		0	$Increment_1$
15		2	

Stencil 2 can be copied from stencil 1 using the following FORTRAN-77 code fragment:

```
DO I = 2,14
    R(I+14) = R(I)
    JR(I+14) = JR(I)
ENDDO
JR(29) = Ny + 3                ← The 1 change
```

Finally, we need to generate pointers to the correct stencils. This can be done using the following FORTRAN-77 code fragment:

```
M = 30
DO J = 1,N_x
    DO I = 1,N_y - 1
        JR(M) = 2            ← stencil 1
        M = M + 1
    ENDDO
    JR(M) = 16               ← stencil 2
    M = M + 1
ENDDO
```

This completes the two dimensional example.

The three dimensional example is similar. In this case, grid G_1 has $N_1 = (2N_x + 1) \times (2N_y + 1) \times (2N_z + 1)$ points and grid G_2 has $N_1 = N_x \times N_y \times N_z$ points where a point $(x_i, y_j, z_k) \in G_2$ is the point $(x_{2i}, y_{2j}, z_{2k}) \in G_1$. The weighting used to construct R_1 has 27 entries (3^3) in it. The stencil is a weighting of the 9 nearest neighbors on the plane the stencil is centered on,

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix},$$

and the 9 nearest neighbors on the planes directly above and below (using the same weighting on each),

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \end{bmatrix}.$$

In this case, there are 3 stencils: ones with an increment of 2, N_y+3, and $N_xN_y+2N_y+3$. Each stencil has four sets of offsets, one for each of the $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, and 1 multipliers. The offsets are as follows:

$\frac{1}{8}$	$0,\ 2,\ 2N_y,\ 2N_y+2,\ 2N_xN_y,\ 2N_xN_y+2,$ $2(N_xN_y+N_y),\ 2(N_xN_y+N_y)+2$
$\frac{1}{4}$	$1,\ N_y,\ N_y+2,\ 2N_y+1,\ N_xN_y,\ N_xN_y+2,$ $N_xN_y+2N_y,\ N_xN_y+2N_y+2,\ 2N_xN_y+1,\ 2N_xN_y+N_y,$ $2N_xN_y+N_y+2,\ 2N_xN_y+2N_y+1$
$\frac{1}{2}$	$N_y+1,\ N_xN_y+1,\ N_xN_y+N_y,\ N_xN_y+N_y+2,$ $N_xN_y+2N_y+1,\ 2N_xN_y+N_y+1$
1	$N_xN_y+N_y+1$

Each stencil is of length 33, so

$$JR(1) = 101.$$

The second and third stencils can be copied from the first, similarly to the two dimensional example before. Finally, we need to generate the pointers to the correct stencils. This can be done using the following FORTRAN-77 code fragment:

```
M = 101
DO K = 1,N_z
     DO J = 1,N_x - 1
          DO I = 1,N_y - 1
               JR(M) = 2              ← stencil 1
               M = M + 1
          ENDDO
          JR(M) = 35                  ← stencil 2
          M = M + 1
     ENDDO
     DO J = 1,N_y - 1
          JR(M) = 2                   ← stencil 1
          M = M + 1
     ENDDO
     JR(M) = 68                       ← stencil 3
     M = M + 1
ENDDO
```

This completes the three dimensional example.

We conclude this section by summarizing the savings in memory by using the stencil storage mode instead of standard sparse matrix storage schemes. In the two and three dimensional examples, we require the following amount of long precision real and integer memory locations:

	Stencil storage mode		Storage by rows	
Example	Real	Integer	Real	Integer
2D	30	$N_xN_y + 30$	$9N_xN_y$	$10N_xN_y + 1$
3D	100	$N_xN_yN_z + 100$	$27N_xN_yN_z$	$28N_xN_yN_z + 1$

To say that using stencil storage mode is a savings for this example is an understatement.

4. Subroutine DAMG. Algorithms MGC, MGFAS, NIC, and NIFAS have all been encapsulated in the long precision subroutine DAMG. It calls the correct multilevel algorithm subroutine(s), which in turn calls the appropriate matrix–vector multiplication routines, direct or iterative solvers, and possibly user supplied routines.

4.1. Syntax of DAMG. DAMG can be called from either FORTRAN or C using the following convention:

FORTRAN	$CALL\ \ DAMG\ ($	$subchl, subpre, subsmr, infalg, infm,$ $b, x, dm, im, jm, iparm, resid, aux, naux)$
C	$damg\ ($	$subchl, subpre, subsmr, infalg, infm,$ $b, x, dm, im, jm, iparm, resid, aux, \&naux)$

4.2. On entry to DAMG. The arguments to DAMG have the following meaning:

subchl

is an external subroutine for changing levels. This is used instead of a call to the (sparse) matrix-vector multiply routine. It will occur automatically if the entries in $infm$ corresponding to R_j, P_{j+1}, and NIP_{j+1} are zero for some level j. A routine DAMGN which generates a "not implemented" message and then terminates is provided in the library. For details, see §5.2.
Specified as: the name of a subroutine that is declared as EXTERNAL in your calling program. It can be whatever name you choose.

subpre

is an external subroutine to be used as a preconditioner in the smoothing routines, where applicable. A routine DAMGN which just returns is provided in the library. For details, see §5.3.
Specified as: the name of a subroutine that is declared as EXTERNAL in your calling program. It can be whatever name you choose.

subsmr

is a user supplied solver subroutine (and usually an iterative one). A routine DAMGN which just returns is provided in the library. For details, see 5.4.
Specified as: the name of a subroutine that is declared as EXTERNAL in your calling program. It can be whatever name you choose.

infalg

is a 2 dimensional array which contains information about each level. It is dimensioned $(12, L)$, where L is at least as great as the number of levels. The second index refers to a level (which will be denoted by j in this description).

- $infalg(1, j) = Solver$ is which solver (see Table 1) to use on level j. See §4.5 for a description of the defaults.
- $infalg(2, j) = SolverIters$ is how many iterations to do of the solver on level j each time smoothing is requested by a multilevel algorithm. The default is 2, but any value in the 1–4 range is typical.
- $infalg(3, j) = Precond$ is which preconditioner (see Table 2) to use on level j. The default is 0 (no preconditioner).
- $infalg(4, j) = MGIters$ is μ_j in multilevel algorithms. The default is 1, but either 1 or 2 is typical.
- $infalg(5, j) = NIIters$ is ψ_j in NIC or NIFAS. The default is 1.
- $infalg(6, j) = IdxXB$ is where x_j and b_j start in the x and b vectors. This is a FORTRAN-77 style vector index. See §4.5 for a description of how to dimension b and x and how to stack the b_j and x_j inside of b and x.
- $infalg(7, j) = NXB$ is the length of x_j and b_j. See §4.5 for a description of this.
- $infalg(8, j) = Colors$ is the number of colors for the Gauss-Seidel red-black solver, where $0 \leq Colors \leq NXB(j)$. The default is 2.
- $infalg(9 - 12, j)$ are reserved.

A summary of *infalg* is in Table 3.

TABLE 1
DAMG Solver Information

Solver	Symbolic name	Definition
0	NoSolver	no solver used on this level
1	User	user supplied routine *subsmr*
2	DSFactor	Gaussian elimination factorization
3	DSSolve	Gaussian elimination solution
4	SGS	symmetric Gauss-Seidel
5	GSNat	Gauss-Seidel, natural ordering
6	GSRedBlack	Gauss-Seidel, red-black ordering
7	CG	conjugate gradients
8	MR	minimum residuals
9	CG-Squared	conjugate gradients squared
10	CG-STAB	variant of CG-squared
11	GMRES	generalized minimum residuals

TABLE 2
DAMG Preconditioner Information

Preconditioner	Symbolic name	Definition
0	NoPrecond	no preconditioner used on this level
1	User	user supplied routine *subpre*
2	ILU	incomplete factorization
3	Diag	diagonal preconditioner
4	SGS	symmetric Gauss-Seidel
5	SSOR	successive over relaxation

TABLE 3
Summary of $infalg$

$infalg(i,j)$ on level j		
i	Symbolic name	Definition
1	$Solver$	Which solution method
2	$SolverIters$	Iterations of $Solver$
3	$Precond$	Which preconditioning method
4	$MGIters$	Iterations of Algorithm MGC or MGFAS
5	$NIIters$	Iterations of Algorithm NIC or NIFAS
6	$IdxXB$	Index of first element of b_j or x_j in b or x
7	NXB	Number of elements in b_j and x_j
8	$Colors$	Number of colors in a multicolor ordering
9-12	reserved	

$infm$

is a 3 dimensional array which contains information about each level. This is the mother of all arguments. No program should be without one. It is dimensioned $(10, l2infm, L)$, where L is at least as great as the number of levels (see *iparm* for a description of $l2infm$). The third index refers to a level (which will be denoted by j in this description).

This array contains information about all five types of matrices that can be associated with each level. In all likelihood, only two matrices will be associated with any given level, however. The five possible matrices for level j are as follows:

A_j The coefficient matrix used in solving a linear system on level j.

R_j The restriction matrix used to transfer data to level $j+1$. The transpose of R_j may be used to transfer data from level $j+1$, too.

P_j The prolongation matrix used to transfer data to level $j-1$. The transpose of R_j may be used to transfer data from level $j-1$, too.

NIP_j The prolongation matrix used only in Algorithm NIC or NIFAS to transfer data from level $j-1$. Normally, P_j is used instead.

$FASR_j$ The injection or projection matrix used in Algorithm MGFAS to transfer the approximate solution x_j onto level $j+1$ as the initial guess to the approximate solution x_{j+1}.

All of these are optional.

$infm$ is a very simple data structure actually: Consider $infm(i,k,j)$, where j is the level. The symbolic names are in Table 4. The definitions of these variables are highly dependent on the tables above. Instead of defining all of these variable separately, we define them one row at a time, substituting a ? for A_j, R_j, P_j, NIP_j, and $FASR_j$:

TABLE 4
Symbolic names for infm entries

i/k	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	$IdxJA$	$IdxJR$	$IdxJP$	$IdxJNIP$	$IdxJFASR$
9			*reserved*		
10			*reserved*		

$?Type$ The matrix type. See Table 5.

$?Cols$ The number of columns in the matrix.

$?Rows$ The number of rows in the matrix.

$?Dim1$ The first dimension of the matrix stored in dm, as it would be logically be defined in a dimension statement in FORTRAN.

$?Dim2$ The second dimension of the matrix stored in dm, as it would be logically be defined in a dimension statement in FORTRAN. This is ignored unless $?Type = MatrixDense$.

$Idx?$ A FORTRAN-77 style 1 based index in dm for the real part of the matrix. This is ignored if $?Type = MatrixNone$ or $MatrixUser$ (see Table 6).

$IdxI?$ A FORTRAN-77 style 1 based index in im for one of the integer descriptions of the matrix. This is ignored unless $?Type = MatrixByRow$ $MatrixStencil$ (see Table 6).

$IdxJ?$ A FORTRAN-77 style 1 based index in im for one of the integer descriptions of the matrix. This is ignored unless $?Type = MatrixByRow$ (see Table 6).

b

is a vector containing the right hand sides b_j, stacked one after the next. For a given b_j, it starts at the location referenced by $infalg(IdxXB_j)$. See §4.5 for a description of how to dimension b.

Specified as: a vector of long precision real numbers.

TABLE 5
Matrix Type Information

Type	Symbolic name	Definition
0	*MatrixNone*	No matrix specified
1	*MatrixUser*	user supplied function used instead of matrix
2	*MatrixByRow*	"storage by rows" sparse matrix
3	*MatrixStencil*	stencil mode sparse matrix
4	*MatrixDense*	dense matrix

TABLE 6
Matrix Data Structure Correlation

Matrix	DAMG's 3 vectors		
Type	*dm*	*im*	*jm*
MatrixNone	–	–	–
MatrixUser	–	–	–
MatrixByRow	A(LNA)	IA(LNA)	JA(N+1)
MatrixStencil	A(JA(1)-1)	JA(JA(1)+N-1)	–
MatrixDense	A(LDA,N)	–	–

Term	Definition
N	Number of rows
LNA, LDA	Leading dimension

x

is a vector containing the approximate solutions or corrections x_j, stacked one after the next. For a given x_j, it starts at the location referenced by $infalg(IdxXB_j)$. See §4.5 for a description of how to dimension x.
Specified as: a vector of long precision real numbers.

dm

is a vector containing the matrices (A_j, R_j, P_j, NIP_j, and $FASR_j$) stacked one after the next. For a given A_j, it starts at the location referenced by $infm(IdxA, 1, j)$ (and similarly for the remaining matrix types). Each matrix is stored in one of the matrix formats (see §3). Its size is specified by $lndm$ and the last element in use is specified by $lastdm$ (see $iparm$).
Specified as: a vector of long precision real numbers.

im

is a vector containing the vectors im_j, stacked one after the next. For a given IA_j, it starts at the location referenced by $infalg(IdxIA, 1, j)$. Its size is specified by $lnim$ and the last element in use is specified by $lastim$ (see $iparm$).
Specified as: a vector of integers.

jm

is a vector containing the matrices JA_j, stacked one after the next. For a given JA_j, it starts at the location referenced by $infm(IdxJA, 1, j)$. Its size is specified by $lnjm$ and the last element in use is specified by $lastjm$ (see $iparm$).
Specified as: a vector of integers.

iparm

is a vector of integer arguments.

- $iparm(1) = mgfn$ determines which of the multilevel algorithms to use:
  1 MGC
  2 MGFAS
  3 NIC
  4 NIFAS
- $iparm(2) = l2infm$ is the second dimension of $infm$. This must be positive.
- $iparm(3) = bxsize$ is the size of b and x vectors. See §4.5 for a description of how to dimension b and x.
- $iparm(4) = lndm$ is the size of dm vector. See §4.5 for a description of how to dimension dm.
- $iparm(5) = lnim$ is the size of im vector. See §4.5 for a description of how to dimension im.
- $iparm(6) = lnjm$ is the size of jm vector. See §4.5 for a description of how to dimension jm.
- $iparm(7) = levelf$ is the finest level number, where $levelf \leq levelc$.
- $iparm(8) = levelc$ is the coarsest level number, where $levelc \geq levelf$.
- $iparm(9) = startl$ is the index of the starting level in the multigrid algorithm, where $levelf \leq startl \leq levelc$. The default is $levelf$ for Algorithms MGC and MGFAS. The default is $levelc$ for Algorithms NIC and NIFAS.
- $iparm(10) = presva$ is whether or not to preserve the matrices on the coarsest level. If $iparm(10) = 1$, then the coarsest level's A, IA, and JA entries in dm, im, and jm are destroyed during the direct solve phase of the computation. Otherwise, these are preserved at the expense of copying the relevant parts of the vectors to the end of their respective vectors. The default is 0 (preserve).
- $iparm(11) = lastdm$ is the one based index of the last element in dm which is used. So, the last $lndm - lastdm + 1$ elements can be used by DAMG.
- $iparm(12) = lastim$ is the one based index of the last element in im which is used. So, the last $lnim - lastim + 1$ elements can be used by DAMG.

- $iparm(13) = lastjm$ is the one based index of the last element in jm which is used. So, the last $lnjm - lastjm + 1$ elements can be used by DAMG.
- $iparm(14) = info$ controls how much information is printed during a computation.
  0 Print nothing.
  1] Print flow control.
  2 Print vectors as well as flow control.
- $iparm(15) = restart$ is used to communicate to DAMG that this is a continuation of a previous call or not. If this is 1, then DAMG can assume that it has been called before. This should be used with care since it is not well tested.
- $iparm(16 - 19)$ are reserved for future use and should be initialized to zero by the caller.
- $iparm(20) = assist$ is for when all else fails. If this is 5551212, then additional information will written to unit 9.

See Table 7 for a summary.
Specified as: a vector of integers of length at least 20.

resid

is a vector where the residuals are stored. Its size is at least as large as the maximum number of unknowns on the finest level.
Specified as: a vector of long precision real numbers.

aux

is the storage work area used by this subroutine. If $restart = 1$ (see *iparm*), this must be the exact same work area that was used before. Its size is specified by *naux*.
Specified as: a vector of long precision real numbers of length *naux*.

naux

is the size of the floating point scratch storage.
WARNING: Do not pass a constant; use a variable.
Specified as: an integer.

4.3. On return from DAMG. The following arguments to DAMG may change before it returns:

infalg

The one entry that specifies that the coarsest level should be factored, is changed to indicate that this has been done and only a solve need be done on this level.

TABLE 7
Summary of iparm

$iparm(i)$		
i	Symbolic name	Definition
1	$mgfn$	Which multilevel algorithm
2	$l2infm$	Second dimension of $infm$ array
3	$bxsize$	Length of b and x arrays
4	$lndm$	Length of dm array
5	$lnim$	Length of im array
6	$lnjm$	Length of jm array
7	$levelf$	Index of the finest level
8	$levelc$	Index of the coarsest level
9	$startl$	Index of the starting level
10	$presva$	Preserve coarsest level's matrices or not
11	$lastdm$	Index of last element in dm in use
12	$lastim$	Index of last element in im in use
13	$lastjm$	Index of last element in jm in use
14	$info$	Control of debugging information
15	$restart$	Continued computation indicator
16–19	reserved	
20	$assist$	When all else fails

b

contains the right hand side for level $levelf$ and is destroyed on the other levels.

x

contains the approximate solution for level $levelf$ and is destroyed on the other levels.

dm

If $Solver(levelc)$ specifies a direct solve, then the factorization of the coarsest level's matrix will be returned. Additionally, the coarsest level matrix, see $IdxA(levelc)$, will have been destroyed if $presva = 0$. See $iparm$ and $infm$.

im

If $Solver(levelc)$ specifies a direct solve, then the factorization of the coarsest level's matrix will be returned. Additionally, the coarsest level matrix, see $IdxIA(levelc)$, will have been destroyed if $presva = 0$. See $iparm$ and $infm$.

jm

If $Solver(levelc)$ specifies a direct solve, then the factorization of the coarsest level's matrix will be returned. Additionally, the coarsest level matrix, see $IdxJA(levelc)$, will have been destroyed if $presva = 0$. See $iparm$ and $infm$.

$resid$

is a vector of length $NXB(levelf)$ where the residuals for level $levelf$ are stored.

aux

is destroyed. If you plan on restarting DAMG later ($restart = 1$, see $iparm$), this must not be changed between calls to DAMG.

$naux$

is the estimate for what $naux$ ought to have been if the value supplied in the call to DAMG is too small.

4.4. Errors associated with DAMG. There are three classes of errors: input, input or computational, and computational ones.

4.4.1. Input errors.

1. $Solver(j)$ is not in $0 - 11$ range.
2. $SolverIters(j) < 0$.
3. $Precond(j)$ is not in $0 - 5$ range.
4. Matrix type for level j, $Precond(j)$ and $Solver(j)$ are incompatible.
5. $MGIters(j) < 0$.
6. $mgfn = 3$ or 4, but $NIIters(j)$ is not positive.
7. $IdxXB(j)$ is out of range.
8. $NXB(j)$ is not positive.
9. Choice of $Solver(j)$ requires that $ACols(j) = ARows(j)$.
10. Matrix type is not 0, 1, 2, 3, or 4.
11. Number of columns in matrix is not positive.
12. Number of rows in matrix is not positive.
13. First dimension of matrix should be positive, but it is not.
14. Second dimension of matrix should be positive, but it is not.
15. Index into dm is out of range.
16. Index into im is out of range.
17. Index into jm is out of range.
18. $l2infm$ is not positive.
19. $mgfn$ is not 1, 2, 3, or 4.
20. $bxsize$ is not positive.
21. At least one of $lndm$, $lnim$, or $lnjm$ is not positive.
22. $levelf$ is negative or $levelf > 50$.
23. $levelc < levelf$ or $levelc > 50$.
24. $presva$ is not 0 or 1 or must be 1 and it is not.
25. At least one of $lndm$, $lnim$, or $lnjm$ is too small to factor coarse level matrix.

26. At least one of *lastdm*, *lastim*, or *lastjm* is not positive or is greater than *lndm*, *lnim*, or *lnjm*, respectively.
27. *info* is not 0, 1, or 2.
28. $startl < 0$, $startl > levelc$, or $startl < levelf$.
29. $Colors(j)$ is not positive or $Colors(j) > NXB(j)$.

4.4.2. Input or Computational Errors.

1. *naux* is not large enough.

4.4.3. Computational Errors.

1. Diagonal entry of coefficient matrix for level j is zero.
2. Coefficient matrix is stored by rows, but column indices are not stored in ascending order in some row.
3. Error occurred in Minimum Residual solver due to an inappropriate coefficient matrix. Re-run DAMG using one of the CGS, GMRES, or CGSTAB solvers instead.
4. An error occurred in *subchl*, *subpre*, or *subsmr*.

4.5. Notes about DAMG.

1. The philosophy behind subroutine DAMG is found in [3] and [4].
2. A number of iterative procedures are supported as smoothers (see Table 1). For a description of these procedures, see [1] for SGS, CG, MR, see [11] for GSNat and GSRedBlack, see [8] for CG-squared, see [10] for CG-STAB, and see [7] for GMRES. Solvers 2 and 3 are direct solvers which make no use of symmetry. Solvers 4 to 7 require the matrix A_j to be symmetric. Solvers 8 to 11 are used when the matrix A_j is nonsymmetric. Solver 1 has to be used when there is no matrix A_j. Solver 6 is really a multicolored Gauss-Seidel iteration. The number of colors is determined by the *Colors* entry in *infalg*. The default is 2 which is just the standard red-black ordering. The maximum allowed per level is the number of unknowns, $nxb(j)$, which corresponds to a reverse natural ordering.
3. If A_j is not present, the user must provide an external subroutine (see *subsmr*) to do solves.
4. Normally, an iterative procedure is used as a smoother on all levels except the coarsest one, where a direct solver may be substituted. However, smoothing will be skipped on a level if NoSolver is specified as the solver for a level. This corresponds to the Identity operator as a smoother in the definitions in §2.

5. The solvers, preconditioners, and matrix types must be compatible with each other. The exact set of acceptable combinations is as follows:

Solver	*Preconditioner*					
	None	*User*	*ILU*	*Diag*	*SGS*	*SSOR*
NoSolver	*	*	*	*	*	*
User	*any*	*any*	*	*	*	*
Factor	*RD*	*	*	*	*	*
Solve	*RD*	*	*	*	*	*
SGS	*R*	*	*	*	*	*
GS	*RSD*	*	*	*	*	*
GSRB	*RSD*	*	*	*	*	*
CG	*RSD*	*RSD*	*R*	*R*	*R*	*R*
MR	*RSD*	*RSD*	*	*	*R*	*
CGS	*R*	*	*R*	*R*	*	*R*
CGSTAB	*R*	*	*R*	*R*	*	*R*
GMRES	*R*	*	*R*	*R*	*	*R*

$$\begin{array}{lcl} * & = & Error \\ R & = & MatrixByRow \\ S & = & MatrixStencil \\ D & = & MatrixDense \\ any & = & any\ format \end{array}$$

The *NoSolver* case is invariably a mistake.

6. The right hand sides $\{b_j\}$ and approximate solutions $\{x_j\}$ are stored in the b and x vectors. Each b_j and x_j is stored in locations $IdxXB(j)$ to $IdxXB(j) + NXB(j) - 1$ (see *infalg*). Suppose there are three levels with $levelf = 1$ and $levelc = 3$ (see *iparm*). Then

Level (j)	*NXB*	*IdxXB*	*Locations in b and x for* b_j *and* x_j
1	289	1	$1 - 289$
2	81	290	$290 - 370$
3	25	371	$371 - 395$

The minimum for *bxsize* is 395 in this example. Each x_j begins in x at the same location as the corresponding b_j in b.

7. When changing levels, it is very rare that R_j, P_j, NIP_j, and $FASR_j$ will all be defined. Usually only one or two these will be defined. These matrices are typically related to each other in very particular ways mathematically. An effort has 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 $infm$ data structure how to change levels. The order of choice is determined by which matrix is wanted:

Wanted	*Order of selection*
R_j	$R_j,\ P_{j+1}^T,\ and\ NIP_{j+1}^T$
P_j	$P_j,\ R_{j+1}^T,\ and\ NIP_j$
NIP_j	$NIP_j,\ P_j,\ and\ R_{j+1}^T$
$FASR_j$	$FASR_j,\ R_j,\ P_{j+1}^T,\ and\ NIP_{j+1}^T.$

The external subroutine (*subchl*) is the last choice unless the matrix type is $MatrixUser$ (see $infm$).

8. Algorithm MGFAS must inject or project the approximate solution x_j as the initial guess to x_{j+1}. This is usually done by a restriction operator that is different from the one used to project the residual onto the b_{j+1}. For a typical application, this should be a matrix stored by rows with a single entry of 1 in each row which just maps elements of level j onto the elements of level $j+1$ (referred to as injection in the literature).
9. The coarse level coefficient matrix should be stored after all other matrices if it is to be factored and not preserved (*presva* $= 0$ in *iparm*; see also $infm$). This is becasue the call to direct solver to factor the coarse level matrix will overwrite the matrix and space after the *dm* and *jm* parts of them. Hence, space must be provided at the end of the *dm*, *im*, and *jm* vectors for the factorization and possibly a copy of the matrix.
   For Algorithms MGFAS and NIFAS, *presva* $= 1$ must be assumed if a direct solve is used on the coarsest level.
   When *presva* $= 1$, the coarse level matrix is copied to end of the active part of the *dm*, *im*, and/or *jm* vectors (depending on matrix storage type). DAMG uses *lastdm*, *lastim*, and *lastjm* (see *iparm*) to determine the end of the active areas. DAMG will use the remaining parts of these vectors for use with the coarsest level's computations.
10. The index variables (see $infalg$ and $infm$) can be checked for simple "out of range" errors. These include an index which is less than one or where the end of the vector goes beyond the end of the storage area. No effort is made to check for overlapping vectors inside of the storage areas (see *dm*, *im*, and *jm*).
11. Should DAMG abnormally end, *iparm*, $infm$, and $infalg$ might be changed from what the user expects.
12. In very special cases, the starting level (*startl* in *iparm*) cannot be either $levelf$ or *levelc*, but a level in between, namely, $levelf \leq startl \leq levelc$. An example of this is when the multilevel solver is being used with an adaptive grid refinement procedure: there is computing, followed by grid refinement to produce a finer level, followed by more computing.
   The default for *startl* is as follows:
   NIC, NIFAS: $startl = levelc$
   MGC, MGFAS: $startl = levelc$

WARNING: For both Algorithms NIC and NIFAS, if $startl < levelc$, the first part of the algorithm will just prolong the approximate solution on level $startl+1$, $x_{startl+1}$, onto x_{startl} as the initial guess to the solution on level $startl$. This is fundamentally different from both Algorithms MGC and MGFAS, which will simply start computing on level $startl$ and will end computation on level $levelf$.

13. This is a sufficiently complicated subroutine, that help must be provided directly to the user. Setting $info$ to either 1 or 2 provides this capability. DAMG will run at full speed only when $info = 0$, however. $info = 1$ provides information on what a multilevel algorithm is about to do, e.g.,

    a SMOOTH ? ITERATIONS ON LEVEL ? b
    a PROLONG FROM LEVEL ? TO LEVEL ?
    a RESTRICT FROM LEVEL ? TO LEVEL ?

    where

    ? is a number,
    a is one of MGC, MGFAS, NIC, or NIFAS,
    b is the name of a solver.

    $info = 2$ provides additional information. Certain vectors are printed on unit 6 after major operations:

Operation	*Vectors printed*
Smooth on level j	x_j, *resid*
Prolong to level j	x_j
Restrict to level j	x_j, b_j

    In addition, the values of various vectors will be printed on unit 9 when $iparm(20) = 5551212$. Setting $info = 2$ for large problems will requires a lot of disk space.

14. Calculating $naux$ is complex since it is dependent on the storage requirements of the solver used on each level. Certain solver-preconditioner pairs introduce a machine dependence to calculating $naux$.
    The solvers used by DAMG are partly home grown and partly from IBM's proprietary ESSL. The $naux$ requirements per level are determined from the

following table, where a blank entry means none:

Solver	*Preconditioner*					
	None	*User*	*ILU*	*Diag*	*SGS*	*SSOR*
NoPrecond						
User						
Factor	*a*					
Solve	*a*					
SGS	*b*					
GS	*c*					
GSRB	*c*					
CG	*d*	*d*	*h*	*i*	*e*	*h*
MR	*f*	*f*			*g*	
CGS	*j*		*k*	*l*		*k*
CGSTAB	*j*		*k*	*l*		*k*
GMRES	*m*		*n*	*o*		*n*

For level i, $NZ(i)$ is the number of nonzeros in a matrix A_i stored by rows.
(a) Space must be left for the Gaussian elimination routines. For dense matrices, $NXB(i)/2$. For matrices stored by rows, this is unpredictable.
(b) $(3/2)NXB(i)$
(c) $NXB(i)$
(d) $3NXB(i)$
(e) $(9/2)NXB(i)$
(f) $4NXB(i)$
(g) $(11/2)NXB(i)$
(h) $3NZ(i) + (23/2)NXB(i) + 61$. See DSRIS for details.
(i) $(3NZ(i) + 15NXB(i))/2 + 31$. See DSRIS for details.
(j) $(3NZ(i) + 19NXB(i))/2 + 31$. See DSRIS for details.
(k) $3NZ(i) + 16NXB(i) + 61$. See DSRIS for details.
(l) $(3NZ(i) + 21NXB(i))/2 + 31$. See DSRIS for details.
(m) For $k = SolverIters(i)$, $(3NZ(i) + 5NXB(i))/2 + k(k+4) + (k+2)NXB(i) + 32$. See DSRIS for details.
(n) For $k = SolverIters(i)$, $(3NZ(i)+7NXB(i))/2+k(k+4)+(k+2)NXB(i)+62$. See DSRIS for details.
(o) For $k = SolverIters(i)$, $(3NZ(i)+7NXB(i))/2+k(k+4)+(k+2)NXB(i)+32$. See DSRIS for details.
So, *naux* is the sum over each level of the requirements from the above table. Obviously, it is easier to set $naux = 1$ and get an error message back from DAMG.
WARNING: A number of these formulas $(a, h - o)$ are based on ones in the IBM ESSL manual. Some of these formulas do not require enough memory to actually get IBM's subroutine DSRIS to run. In these cases you need to modify mga1.f in the neighborhood of lines 1210–1250. You should only have

to modify at most 2 lines of the code. Then send the author e–mail explaining that you had a problem so that this can be fixed.

5. Programming considerations for external subroutine arguments. Three of the arguments on entry to DAMG are for user defined subroutines. They can be called whatever the user pleases, must be declared EXTERNAL in the user's program which calls DAMG, and must have a particular set of arguments themselves. A default subroutine is defined in the library.

An example of a program which uses this feature of DAMG is the companion program DPMG for solving Poisson's equation in two or three dimensions. DPMG provides its own smoothers and change level subroutines since it does not store any matrices normally.

5.1. DAMGN: a stub-routine. Not everyone needs their own subroutine to change levels, to act as a preconditioner, or be a smoother (arguments 1 to 3 of DAMG). A subroutine is provided which terminates if it is ever called by DAMG, but alleviates the user from having to code up to three dummy subroutines to use in the calling sequence of DAMG.

To use this subroutine with subroutine DAMG, use the following FORTRAN-77 code fragment:

```
EXTERNAL DAMGN
.
.
.
CALL DAMG ( DAMGN, ... )
```

This can be used in any combination of the first 3 arguments of DAMG, e.g.,

```
EXTERNAL DAMGN, OOPS
.
.
.
CALL DAMG ( DAMGN, OOPS, DAMGN, ... )
```

5.2. SUBCHL: changing levels. Normally, the grid transfers in the multigrid algorithms occur by a call to a (sparse) matrix–vector multiply routine which computes one of

$$b_{j+1} = R_j r_j, \quad x_j = P_{j+1} c_{j+1}, \quad \text{or} \quad x_j = x_j + P_{j+1} c_{j+1}. \tag{3}$$

There are numerous reasons sometimes why doing this in this fashion is highly inefficient, e.g., the grids are highly nonuniform. The user can bypass making a matrix R_j by supplying a subroutine to change between levels j and $j+1$ (both directions should be handled).

To use this subroutine with subroutine DAMG, use the following FORTRAN-77 code fragment:

```
EXTERNAL MYCHL
.
.
.
CALL DAMG ( MYCHL, ... )
.
.
.
SUBROUTINE MYCHL ( ... )
ERR = 1
RETURN
END
```

This declares the subroutine MYCHL to be an external address that is passed to DAMG like an ordinary variable. If DAMG calls MYCHL, it will report an error occurred. Normally, there is much more to MYCHL than this (and ERR=0).

DAMG will call SUBCHL assuming the following prologue:

FORTRAN	$CALL\ \ SUBCHL\ (\ \ mgfn, dir, add, lev1, lev2, x1, x2, bf, n1, n2, err)$
C	$subchl\ (\ \ mgfn, dir, add, lev1, lev2, x1, x2, bf, n1, n2, err)$

A subroutine DAMGN which generates a "not implemented" error message and then terminates is included in the library.

Unpredictable results will occur if the user changes any of the arguments except $x2$ and err. The user is responsible for providing any workspaces needed. Under no circumstances should the aux area, passed to DAMG, be touched.

5.2.1. On entry to SUBCHL. The arguments to SUBCHL have the following meaning:

$mgfn$

This determines which of the multilevel algorithms is in use:

$mgfn$	Definition
1	MGC
2	MGFAS
3	NIC
4	NIFAS

Specified as: an integer.

dir

determines which direction to change levels, either a restriction or a prolongation operation.

dir	Definition
0	fine to coarse (level j to $j+1$)
1	coarse to fine (level $j+1$ to j)

Specified as: an integer.

add

determines whether the vector $x1$ is added to $x2$ or not. If $add = 1$, then the previous contents of $x2$ are added to $x1$. If $add = 0$, then the previous contents of $x2$ are ignored.
Specified as: an integer.

*lev*1

is the "from" level.
Specified as: an integer.

*lev*2

is the "to" level.
Specified as: an integer.

$x1$

is the x vector on "from" level.
Specified as: a vector of long precision real numbers of length $n1$.

$x2$

is the target vector on "to" level (see (3)).
Specified as: a vector of long precision real numbers of length $n2$.

bf

is the b vector on "finer" level.
Specified as: a vector of long precision real numbers of length $\max n1, n2$.

$n1$

is the size of $x1$.
Specified as: an integer.

$n2$

is the size of $x2$.
Specified as: an integer.

err

is 0.
Specified as: an integer.

5.2.2. On return from SUBCHL. The following arguments to SUBCHL may change before it returns:

$x2$

is the updated vector on the "to" level (see (3)).

err

is nonzero if any problem arises that cannot be solved by the user subroutine. DAMG terminates if this is nonzero; *err* is printed as part of the message on your screen.

5.3. SUBPRE: preconditioning. Most of the time, the user will want to use the iterative procedures included in the library. Occasionally, a user will be sophisticated enough to want to use a very specific preconditioner (that is not included in the library) to a specific iterative procedure. The SUBPRE subroutine allows the caller this flexibility with certain of the smoothers.

Preconditioners typically come in one of three flavors: ones that modify the residual, ones that modify the approximate solution, and ones that do both. Currently, DAMG only supports preconditioners that modify the residual.

The external subroutine is provided with the level number (here referred to as j), the matrix A_j, x_j, b_j, and the residual $b_j - A_j x_j$. Any of these can be modified. Modifying A_j can cause unexpected errors in the multigrid subroutines.

The user should remember that they are accelerating the solution to $A_j x_j = b_j$ with their subroutine.

To use this subroutine with subroutine DAMG, use the following FORTRAN-77 code fragment:

```
EXTERNAL MYPRE
.
.
.
CALL DAMG ( ..., MYPRE, ... )
.
.
.
SUBROUTINE MYPRE ( ... )
ERR = 1
RETURN
END
```

This declares the subroutine MYPRE to be an external address that is passed to DAMG like an ordinary variable. If DAMG calls MYPRE, it will report an error occurred. Normally, there is much more to MYPRE than this (and ERR=0).

DAMG will call SUBPRE assuming the following prologue:

FORTRAN	*CALL SUBPRE* (	$lev, atype, acols, arows, adim1, adim2,$ $a, ia, ja, x, b, resid, updat, err)$
C	*subpre* (	$lev, atype, acols, arows, adim1, adim2,$ $a, ia, ja, x, b, resid, updat, err)$

A subroutine DAMGN which generates a "not implemented" error message and then terminates is included in the library.

The user is responsible for providing any workspaces needed. Under no circumstances should the *aux* area, passed to DAMG, be touched.

5.3.1. On entry to SUBPRE. The arguments to SUBPRE have the following meaning:

lev

is the level one of the multilevel algorithms is currently computing on.
Specified as: an integer.

atype

is the matrix storage type (see Table 5).
Specified as: an integer.

acols

is the number of columns in A_j.
Specified as: an integer.

arows

is the number of rows in A_j.
Specified as: an integer.

*adim*1

is used in the dimensions of A_j. It may also be used in dimensioning IA_j and JA_j; see Table 6.
Specified as: an integer.

*adim*2

is used in the dimensions of A_j. It may also be used in dimensioning IA_j and JA_j; see Table 6.
Specified as: an integer.

a

is A_j, stored in some format determined by *atype*. Its dimensions involve *adim*1 and *adim*2 according to Table 6.
Specified as: a vector or matrix of long precision real numbers.

ia

is IA_j, stored in some format.
Specified as: a vector or matrix of integers.

ja

is JA_j, stored in some format.
Specified as: a vector or matrix of integers.

x

is the approximate solution x_j.
Specified as: a vector of long precision real numbers of length *acols*.

b

is the right hand side b_j.
Specified as: a vector of long precision real numbers of length *arows*.

resid

is the residual $b_j - A_j x_j$.
Specified as: a vector of long precision real numbers of length *arows*.

updat

is 0.
Specified as: an integer.

err

is 0.
Specified as: an integer.

5.3.2. On return from SUBPRE. The following arguments to SUBPRE may change before it returns:

resid

is the modified residual.

updat

is what changed during the call to SUBPRE.

Value	What changed
0	nothing
1	the residual

If it is 0, an error will be presumed to have occurred. Failure to set this correctly will cause serious problems inside the iterative solvers DAMG uses that can call SUBPRE.

err

is nonzero if any problem arises that cannot be solved by the user subroutine. DAMG terminates if this is nonzero; *err* is printed as part of the nasty message on your screen.

5.4. SUBSMR: a smoother or rougher. Most of the time, the user will want to use the iterative procedures included in the library. Occasionally, a user will be sophisticated enough to want to use a very specific iterative procedure that is not included in the library. This can be combined with a user supplied preconditioner (see §5.3). The SUBSMR subroutine allows the caller this flexibility.

The external subroutine is provided with the level number (here referred to as j), the matrix A_j, x_j, and b_j. Only x_j should be modified. Modifying A_j or b_j can cause unexpected errors in the multigrid subroutines.

Due to the fact that a matrix A_j may not actually exist, it is required that the user compute the residual, $b_j - A_j x_j$ (even if A_j is only symbolic here) before returning.

The user should remember that they are trying to solve $A_j x_j = b_j$ with their subroutine.

To use this subroutine with subroutine DAMG, use the following FORTRAN-77 code fragment:

```
EXTERNAL MYSMR
.
```

```
.
.
CALL DAMG ( ..., MYSMR, ... )
.
.
.
SUBROUTINE MYSMR ( ... )
ERR = 1
RETURN
END
```

This declares the subroutine MYSMR to be an external address that is passed to DAMG like an ordinary variable. If DAMG calls MYSMR, it will report an error occurred. Normally, there is much more to MYSMR than this (and ERR=0).

DAMG will call SUBSMR assuming the following prologue:

FORTRAN	*CALL SUBSMR (*	$subpre, iters, lev, atype, acols, arows, adim1, adim2, a, ia, ja, x, b, resid, updat, err)$
C	*subsmr (*	$subpre, iters, lev, atype, acols, arows, adim1, adim2, a, ia, ja, x, b, resid, updat, err)$

A subroutine DAMGN which generates a "not implemented" error message and then terminates is included in the library.

The user is responsible for providing their own workspaces themselves. Under no circumstances should the *aux* area, passed to DAMG, be touched.

5.4.1. On entry to SUBSMR. The arguments to SUBSMR have the following meaning:

subpre
: is an external subroutine to be used as a preconditioner in the smoothing routines, where applicable. For details, see §5.3. A routine DAMGN which just returns is provided in the library.
Specified as: the name of a subroutine that is declared as EXTERNAL in your calling program. It can be whatever name you choose.

iters
: is the maximum number of iterations.
Specified as: an integer.

lev
: is the current computational level in one of the multilevel algorithms.
Specified as: an integer.

atype
: is the matrix storage type (see Table 5).
Specified as: an integer.

acols

is the number of columns in A_j.
Specified as: an integer.

arows

is the number of rows in A_j.
Specified as: an integer.

*adim*1

is used in the dimensions of A_j. It may also be used in dimensioning IA_j and JA_j; see Table 6.
Specified as: an integer.

*adim*2

is used in the dimensions of A_j. It may also be used in dimensioning IA_j and JA_j; see Table 6.
Specified as: an integer.

a

is A_j, stored in some format determined by *atype*. Its dimensions involve *adim*1 and *adim*2 according to Table 6.
Specified as: a vector or matrix of long precision real numbers.

ia

is IA_j, stored in some format.
Specified as: a vector or matrix of integers.

ja

is JA_j, stored in some format.
Specified as: a vector or matrix of integers.

x

is the approximate solution x_j.
Specified as: a vector of long precision real numbers of length *acols*.

b

is the right hand side b_j.
Specified as: a vector of long precision real numbers of length *arows*.

resid

is the residual $b_j - A_j x_j$.
Specified as: a vector of long precision real numbers of length *arows*.

updat

is 0.
Specified as: an integer.

err

is 0.
Specified as: an integer.

5.4.2. On return from SUBSMR. The following arguments to SUBSMR may change before it returns:

x

is the approximate solution x_j.

resid

is the residual $b_j - A_j x_j$.

updat

is what changed during the call to SUBSMR.

Value	What changed
0	nothing
1	the residual
2	x_j
3	both x_j and the residual
4	b_j

If it is 0, an error will be presumed to have occurred. Failure to set this correctly will cause serious problems inside DAMG. The normal return value for *updat* is 3.

err

is nonzero if any problem arises that cannot be solved by the user subroutine. DAMG terminates if this is nonzero; *err* is printed as part of the nasty message on your screen.

REFERENCES

[1] R. E. Bank and C. C. Douglas, *An efficient implementation of the* SSOR *and* ILU *preconditionings*, Appl. Numer. Math., 1 (1985), pp. 489–492.

[2] G. Dhatt and G. Touzot, *The Finite Element Method Displayed*, John Wiley and Sons, Inc., New York, 1984.

[3] C. C. Douglas, *Multi-grid algorithms with applications to elliptic boundary-value problems*, SIAM J. Numer. Anal., 21 (1984), pp. 236–254.

[4] C. C. Douglas and J. Douglas, *A unified convergence theory for abstract multigrid or multilevel algorithms, serial and parallel*, SIAM J. Numer. Anal., 30 (1993), pp. 136–158.

[5] G. E. Forsythe and W. R. Wasow, *Finite-Difference Methods for Partial Differential Equations*, John Wiley and Sons, Inc., New York, 1960.

[6] C. Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge University Press, New York, 1987.

[7] Y. Saad and M. H. Schultz, *Gmres: a generalized minimum residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 7 (1986), pp. 856–869.

[8] P. Sonneveld, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 10 (1989), pp. 36–52.

[9] G. Strang and G. J. Fix, *An Analysis of the Finite Element Method*, Prentice–Hall, New York, 1973.

[10] H. van der Vorst and P. Sonneveld, *Cgstab: A more smoothly convergent variant of cg-s*, tech. report, Delft University of Technology, Delft, The Netherlands, 1990.

[11] R. S. Varga, *Matrix Iterative Analysis*, Prentice–Hall, New York, 1962.

A. Examples of DAMG Usage. All runs in this section were on an IBM RISC SYSTEM/6000™. In §A.1, a simple one dimensional problem is explored in depth. Provided with the code are one, two, and three dimensional example problems. Rather than duplicate the very lengthy set of comments at the beginning of DAMG, we refer the reader to the code.

It is implicitly assumed that you have already made a working version of DAMG. If you do not know how to do this, please see Appendix B.

A.1. Example 1: a one dimensional problem. The first example solves

$$\begin{cases} U_{xx} & = F \quad \textit{in the unit line } (0,1), \\ U(x) & = 0 \quad \textit{for } x \in \{0,1\}.. \end{cases}$$

F is chosen so that the solution is

$$U(x) = x\sin(x\pi)$$

or

$$F(x) = -2\pi\cos(x\pi) + x\pi^2\sin(x\pi).$$

Algorithm NIC is used with the Gauss-Seidel smoother on all levels except the coarsest where a direct solver is used. A uniform mesh, a central difference discretization, and linear interpolation are used.

A sample FORTRAN main program is presented here. The declarations section is simply,

```
      parameter          ( IAUX1 = 1000 )
      parameter          ( IDM1 = 1000 )
      parameter          ( IIM1 = 1000 )
      parameter          ( IJM1 = 1000 )
      parameter          ( IXB1 = 100 )
      parameter          ( NFINE = 7 )
      parameter          ( INFM2 = 2 )
      parameter          ( LEVELS = 2 )
      external           damgn
      integer            infalg(12,LEVELS), infm(10,INFM2,LEVELS),
     *                   im(IIM1), iparm(20), jm(IJM1)
      double precision   aux(IAUX1), b(IXB1), dm(IDM1), resid(IXB1),
     *                   res2, x(IXB1)
      integer            i, idm, iim, ijm, ibx, incdm, incim,
     *                   j, k, lev, n, naux
```

DAMG's integer data structures are initialized to 0. Many of these entries have default values if 0 is passed to DAMG. However, it never hurts to set all of the entries to exactly what you want (which is done later in this example).

```
do j = 1,LEVELS
      do i = 1,12
            infalg(i,j) = 0
      enddo
enddo
do k = 1,LEVELS
      do j = 1,INFM2
            do i = 1,10
                  infm(i,j,k) = 0
            enddo
      enddo
enddo
do i = 1,20
      iparm(i) = 0
enddo
```

Generate A_{lev}, x_{lev}, and b_{lev} for 2 levels, starting with the fine grid problem. The variable *idm*, ..., *ibx* are indices into *dm*, ..., b/x where the next vector can be stored.

```
idm = 1
iim = 1
ijm = 1
ibx = 1
n = NFINE
do lev = 1,LEVELS
```

First, information about A_{lev} is filled in, then gena is called to actually generate A_{lev} (using a storage by rows format), then the indices into DM, IM, and JM are changed.

```
            infm(1,1,lev) = 2                     % Storage by rows
            infm(2,1,lev) = n                     % Rows
            infm(3,1,lev) = n                     % Columns
            infm(6,1,lev) = idm                   % First location in dm
            infm(7,1,lev) = iim                   % First location in im
            infm(8,1,lev) = ijm                   % First location in jm
            call gena ( n, IDM1 - idm + 1, IIM1 - iim + 1,
*                       IJM1 - ijm + 1, dm(idm), im(iim), jm(ijm),
*                       infm(4,1,lev) )
            iim = iim + n + 1
            ijm = ijm + infm(4,1,lev)
            idm = idm + infm(4,1,lev)
```

Next, information about each level's solver is filled in. Then a call to genbx produces a right hand side b_{lev} and initial guess x_{lev}. Finally, the index into b and x is is incremented.

```
        if ( lev .eq. LEVELS ) then
             infalg(1,lev) = 2              Sparse Gaussian elimination
             infalg(2,lev) = 1              1 iteration of this
        else
             infalg(1,lev) = 4              Symmetric Gauss Seidel
             infalg(2,lev) = 2              2 iterations of this
        endif
        infalg(3,lev) = 0                   No preconditioner
        infalg(4,lev) = 2                   2 iterations of algorithm MGC
        infalg(5,lev) = 1                   1 iteration of algorithm NIC
        infalg(6,lev) = ibx                 First location in b and x
        infalg(7,lev) = n                   Number of unknowns
        call genbx ( n, IXB1 - ibx + 1, b(ibx), x(ibx), incbx )
        ibx = ibx + incbx
```

The loop is completed by generating the restriction matrix R_{lev} when lev<LEVELS. To do this, the next coarser level's number of unknowns n must be calculated in advance. After calling genr, the indices into *dm* and *im* are incremented.

```
        n = (n - 1) / 2
        if ( lev .ne. LEVELS ) then
             infm(1,2,lev) = 3                   Stencil storage
             infm(2,2,lev) = n                   Rows
             infm(3,2,lev) = infm(3,1,lev)       Columns
             infm(4,2,lev) = 1                   Unused actually
             infm(6,2,lev) = idm                 First location in dm
             infm(7,2,lev) = iim                 First location in im
             call genr ( infm(2,2,lev), infm(3,2,lev),
     *                   IDM1 - idm + 1, IIM1 - iim + 1,
     *                   dm(idm), im(iim), incdm, incim )
             iim = iim + incim
             idm = idm + incdm
         endif
     enddo
```

Next *iparm* is filled in.

```
iparm( 1) = 3            Algorithm NIC
iparm( 2) = INFM2        Second dimension of infm
iparm( 3) = IXB1         Length of b and x vectors
iparm( 4) = IDM1         Length of dm vector
iparm( 5) = IIM1         Length of im vector
iparm( 6) = IJM1         Length of jm vector
iparm( 7) = 1            Index of finest level
iparm( 8) = LEVELS       Index of coarsest level
iparm( 9) = 0            Index of starting level (default)
iparm(10) = 1            Overwrite A_coarsest
iparm(11) = idm - 1      Last element of dm in use
iparm(12) = iim - 1      Last element of im in use
iparm(13) = ijm - 1      Last element of jm in use
iparm(14) = 0            Minimum normal information
iparm(20) = 5551212      Assistance requested
```

Finally, DAMG can be called and the 2 norm of the residual is printed afterwards.

```
      naux = IAUX1
      call damg( damgn, damgn, damgn, infalg, infm, b, x, dm, im, jm,
     *           iparm, resid, aux, naux )
      res2 = ddot( NFINE, resid, 1, resid, 1 )
      res2 = sqrt( res2 ) / NFINE
      write (*,*) '2 Norm of residual = ', res2
      end
```

There are three subroutines used in the preceding main program:

Subroutine	Description
gena	Generate A_{lev}
genbx	Generate b_{lev} and initial x_{lev}
genr	Generate R_{lev}

Each of these routines demonstrates a slightly different technique. Gena is an example of the storage by rows format. Genr is an example of the stencil storage format.

Gena produces a tridiagonal matrix of the form

$$A_{lev} = \begin{bmatrix} 2.0 & -1.0 & & & & & \\ -1.0 & 2.0 & -1.0 & & & & \\ & -1.0 & 2.0 & -1.0 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & -1.0 & 2.0 & -1.0 \\ & & & & & -1.0 & 2.0 \end{bmatrix}.$$

The program declaration section is straight forward. The variable nzelts is the number of nonzeroes in A_{lev} and is a return value.

```
      subroutine gena ( n, lendm, lenim, lenjm, dm, im, jm, nzelts )
      integer                 lendm, lenim, lenjm, n, nzelts
      integer                 im(*), jm(*)
      double precision        dm(*)
      integer                 irow
```

First, a check is made to ensure that enough space still remains in the *dm*, *im*, and *jm* vectors to store the matrix.

```
      nzelts = 0
      if ( lenim .le. n ) return
      irow = 3 * n - 2
      if ( lendm .lt. irow .or. lenjm .lt. irow ) return
```

Then a tridiagonal matrix using storage by rows is generated.

```
      do irow = 1,n
         nzelts = nzelts + 1
         im(irow) = nzelts
         if ( irow .gt. 1 ) then
            jm(nzelts) = irow - 1
            dm(nzelts) = -1.0d0
            nzelts = nzelts + 1
         endif
         jm(nzelts) = irow
         dm(nzelts) = 2.0d0
         if ( irow .lt. n ) then
            nzelts = nzelts + 1
            jm(nzelts) = irow + 1
            dm(nzelts) = -1.0d0
         endif
      enddo
```

Finally, the last entry in the *im* vector is filled in with the index of of the last entry in *dm* plus one, and then gena returns.

```
      im(n+1) = nzelts + 1
      return
      end
```

Subroutine genbx sets the initial guess for the solution to 0 uniformly. The right hand side is scaled by the square of the mesh spacing due to the discretization method used in gena. The variable incbx is the number of nonzeroes in b_{lev} and is a return value.

```
      subroutine genbx ( n, lenbx, b, x, incbx )
      integer                 incbx, lenbx, n
      double precision        b(*), x(*)
      integer                 irow
      double precision        h, h2, pi, pi2, t, tpi
```

First, a check is made to ensure that enough space still remains in the *b* and *x*vectors.

```
incbx = 0
if ( lenbx .lt. n ) return
```

Finally, the heart of the code is quite simple.

```
call dcopy ( n, 0.0d0, 0, x, 1 )
h = 1.0d0 / (n + 1)
h2 = h * h
pi = 4.0d0 * datan( 1.0d0 )
tpi = 2.0d0 * pi
pi2 = pi * pi
do irow = 1,n
     t = irow * h
     b(irow) = h2 * ( t * pi2 * sin(t*pi) - tpi * cos(t*pi) )
enddo
incbx = n
return
end
```

Subroutine genr generates R_{lev}, which has n rows and $2n+1$ columns. Each row has 3 nonzeroes:

$$\begin{bmatrix} 0.5 & 1.0 & 0.5 & & & & & \\ & & 0.5 & 1.0 & 0.5 & & & \\ & & & & 0.5 & 1.0 & 0.5 & \\ & & & & & & & \cdots \end{bmatrix}.$$

Notice how the stencil always shifts by 2 columns. Hence, there is only one stencil to worry about. Its form is given by

Index	IM	DM	Description
1	9		Index of first stencil pointer
2	2	0.5	2 entries to multiply by 0.5
3	0		Offset 0
4	2		Offset 2
5	1	1.0	1 entry to multiply by 1.0
6	1		Offset 1
7	0	0.0	End of stecil
8	2		Increment value is 2
9	2		Index of stencil for point p-8
.	.		
.	.		
.	.		
8+n	2		

Notice that where DM is blank, any entry can be there since those elements are never referenced. However, it is a good idea to set them to zero (as is done in this example).

The program declaration section is straight forward. The variables incdm and incim are the amount of space used in *dm* and *im* to store R_{lev}. Both are return values.

```
      subroutine genr ( nrows, ncols, lendm, lenim, dm, im,
     *                 incdm, incim )
      integer             incdm, incim, lendm, lenim, ncols, nrows
      integer             im(*)
      double precision    dm(*)
      integer             irow
      double precision    desc4(7)
      data                desc4 / .50, 0., 0.,
     *                             1., 0.,
     *                             0., 0. /
```

First, a check is made to ensure that enough space still remains in the *dm* and *im* vectors to store the matrix.

```
      if ( lendm .lt. 8 ) return
      if ( lenim .lt. 8 + nrows ) return
```

Next, the real part of R_{lev} is generated.

```
      dm(1) = 0.
      do i = 1,7
         dm(1+i) = desc4(i)
      enddo
      incdm = 8
```

The integer part of R_{lev} is generated in five easy pieces.

```
      im(1) = 9                  (1) Index to stencil indices
      im(2) = 2                  (2) Stencil 1: part for coefficient 0.5
      im(3) =     0
      im(4) =     2
      im(5) = 1                  (3) Stencil 1: part for coefficient 1.0
      im(6) =     1
      im(7) = 0                  (4) Stencil 1: part for increment
      im(8) = 2
      incim = 8                (5) Stencil indices
      do i = 1,nrows
         incim = incim + 1
         im(incim) = 2
      enddo
      return
      end
```

Compiling, linking, and executing this set of programs results in output of the form.

```
-------------------------------------------------------------------

DAMG INPUT ARGUMENTS


NAUX =
         1000
L2INFM =
```

```
          2
IPARM =
          3,           2,          100,         1000,         1000,
       1000,           1,            2,            0,            1,
         34,          23,           26,            0,            0,
          0,           0,            0,            0,      5551212
INFALG =
      4,       2
      2,       1
      0,       0
      2,       2
      1,       1
      1,       8
      7,       3
      0,       0
      0,       0
      0,       0
      0,       0
      0,       0
INFM =
SECOND INDEX =1, A:
      2,       2
      7,       3
      7,       3
     19,       7
      0,       0
      1,      28
      1,      20
      1,      20
      0,       0
      0,       0
SECOND INDEX =2, R:
      3,       0
      3,       0
      7,       0
      1,       0
      0,       0
     20,       0
      9,       0
      0,       0
      0,       0
      0,       0
DM =
```

```
 2.0000D+00, -1.0000D+00, -1.0000D+00,  2.0000D+00, -1.0000D+00,
-1.0000D+00,  2.0000D+00, -1.0000D+00, -1.0000D+00,  2.0000D+00,
-1.0000D+00, -1.0000D+00,  2.0000D+00, -1.0000D+00, -1.0000D+00,
 2.0000D+00, -1.0000D+00, -1.0000D+00,  2.0000D+00,  0.0000D+00,
 5.0000D-01,  0.0000D+00,  0.0000D+00,  1.0000D+00,  0.0000D+00,
 0.0000D+00,  0.0000D+00,  2.0000D+00, -1.0000D+00, -1.0000D+00,
 2.0000D+00, -1.0000D+00, -1.0000D+00,  2.0000D+00,  2.0000D+00,
-1.0000D+00, -1.0000D+00,  2.0000D+00, -1.0000D+00, -1.0000D+00,
 2.0000D+00
IM =
         1,          3,          6,          9,         12,
        15,         18,         20,          9,          2,
         0,          2,          1,          1,          0,
         2,          2,          2,          2,          1,
         3,          6,          8,          1,          3,
         6,          8
JM =
         1,          2,          1,          2,          3,
         2,          3,          4,          3,          4,
         5,          4,          5,          6,          5,
         6,          7,          6,          7,          1,
         2,          1,          2,          3,          2,
         3,          1,          2,          1,          2,
         3,          2,          3
X =
 0.0000D+00,  0.0000D+00,  0.0000D+00,  0.0000D+00,  0.0000D+00,
 0.0000D+00,  0.0000D+00,  0.0000D+00,  0.0000D+00,  0.0000D+00
B =
-8.3325D-02, -4.2159D-02,  1.5858D-02,  7.7106D-02,  1.2662D-01,
 1.5120D-01,  1.4234D-01, -1.6864D-01,  3.0843D-01,  6.0481D-01
-----------------------------------------------------------------
-----------------------------------------------------------------
DAMG OUTPUT ARGUMENTS

X =
 4.6857D-02,  1.7704D-01,  3.4932D-01,  5.0570D-01,  5.8503D-01,
 5.3799D-01,  3.4001D-01, -7.7707D-05,  7.3776D-04,  1.9584D-03
RESID =
 0.0000D+00, -5.8075D-05, -4.9091D-05,  6.1883D-05,  2.4495D-04,
 2.6801D-04,  3.0220D-04
-----------------------------------------------------------------
2 Norm of residual =    6.89231D-05
```

B. Making DAMG. The source code for DAMG can be found on the Internet. Two possible anonymous ftp sites are the following:

Machine name	IP address
software.watson.ibm.com	
casper.cs.yale.edu	128.36.12.1

There are other machines with copies at this point, but these should do. Do not attempt to get the files or unpack them directly on a mainframe unless it is running UNIX; use a workstation initially if your target is a mainframe.

Before all else, make a new directory and change to it:

```
mkdir madpack4
cd madpack4
```

To retrieve information, from your Internet connected machine, run the ftp program with one of the machine names as its argument, e.g.,

```
% ftp software.watson.ibm.com
```

where % is the prompt assuming you are using the c-shell. You will be prompted for an account name and password: use the account name *anonymous* and use your e–mail address as the password. Then change directory to one with the software and look at the directory (the prompt for the ftp program is "ftp> "):

```
ftp> cd pub/pdes
ftp> dir
```

You should see something like the following:

```
total 888
drwxr-xr-x   512      Jul 22 1992    .
drwxr-xr-x   512      Dec 11 08:50   ..
-rw-r–r–     3691     Apr 28 1992    AGREE.damg
-rw-r–r–     3691     Apr 28 1992    AGREE.dpmg
-rw-r–r–     7229     Jul 16 1992    README.damg
-rw-r–r–     7997     Jun 03 1992    README.dpmg
-rw-r–r–     182136   Jul 22 1992    damg.tar.Z
-rw-r–r–     236555   Jul 16 1992    dpmg.tar.Z
```

WARNING: you may find the codes in the directory mgnet/madpack4 instead of pub/pdes.

You should get all of the Ascii files first:

```
ftp> prompt
ftp> mget *.damg
```

Read the AGREE.damg file since this is your software license for DAMG (a copy of DAMG's license is Appendix C). Assuming there is nothing in the license that you find objectionable, then get the software package and quit:

```
ftp> binary
ftp> get damg.tar.Z
ftp> quit
```

Now you are ready to unpack the files into their own directory:

% mkdir damg
% cd damg
% zcat ../damg.tar | tar xvf –

Both zcat and tar are standard utilities on workstations.

You are now in the damg directory. On a workstation, to make some examples using DAMG, you merely have to run the command

% make

A number of library files (ones with an extension of ".a") will be produced:

File	Contains
libamg.a	Abstract multilevel solver
libdriv.a	Common routines used by the examples

Also, four executables will be made (b1d, m1d, m2d, and m3d).

You can type the examples from §A into your computer yourself or you can get them from MGNet as part of mgnet/madpack4/doc.tar.Z. To unpack the document, use the commands

% cd ..
% zcat doc.tar | tar xvf –

To compile and link the first example, use the commands

% cd doc
% xlf -c damg-ex1.f
% xlf -o damg-ex1 damg-ex1.o -ldamg -lessl

To execute the program,

% damg-ex1

To make all of the examples, use the command *make*.

C. DAMG's software license. DAMG was written while the author was an IBM employee. As such, IBM owns the software. Be that as it may, IBM has a program for releasing software to the public with very few strings attached. After the author filled out a 17 page form (all answers to the really important questions were "not applicable") and collected signatures (only three), the software was made available to Internet users in July, 1992. The author is indebted to Shmuel Winograd, Ashok Chandra, and Larry Carter for signing this form and to Jim McGroddy for not killing this program.

Note that part of the license specifies that updates and bug notifications will be provided through MGNet. The full text of the license agreement is the remainder of this appendix.

–
– See the file USERAGREEMENT distributed with this software for full
– terms and conditions of use.

1. COPYRIGHT
   Program Name: DAMG
   (C) Copyright International Business Machines Corporation 1992. All Rights Reserved.
2. RESEARCH SOFTWARE DISCLAIMER
   As experimental, research software, this program is provided free of charge on an "as is" basis without warranty of any kind, either expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. IBM does not warrant that the functions contained in this program will meet the user's requirements or that the operation of this program will be uninterrupted or error-free. Acceptance and use of this program constitutes the user's understanding that he will have no recourse to IBM for any actual or consequential damages, including, but not limited to, lost profits or savings, arising out of the use or inability to use this program. Even if the user informs IBM of the possibility of such damages, IBM expects the user of this program to accept the risk of any harm arising out of the use of this program, or the user shall not attempt to use this program for any purpose.

3. USER AGREEMENT

BY ACCEPTANCE AND USE OF THIS EXPERIMENTAL PROGRAM THE USER AGREES TO THE FOLLOWING:

a. This program is provided for the user's personal, non-commercial, experimental use and the user is granted permission to copy this program to the extent reasonably required for such use.

b. All title, ownership and rights to this program and any copies remain with IBM, irrespective of the ownership of the media on which the program resides.

c. The user is permitted to create derivative works to this program. However, all copies of the program and its derivative works must contain the IBM copyright notice, the EXPERIMENTAL SOFTWARE DISCLAIMER and this USER AGREEMENT.

d. By furnishing this program to the user, IBM does NOT grant either directly or by implication, estoppel, or otherwise any license under any patents, patent applications, trademarks, copyrights or other rights belonging to IBM or to any third party, except as expressly provided herein.

e. The user understands and agrees that this program and any derivative works are to be used solely for experimental uses and are not to be sold, distributed to a commercial organization, or be commercially exploited in any manner.

f. IBM requests that the user supply to IBM a copy of any changes, enhancements, or derivative works which the user may create. The user grants IBM and its subsidiaries an irrevocable, nonexclusive, worldwide and royalty-free license to use, execute, reproduce, display, perform, prepare derivative works based upon, and distribute, (INTERNALLY AND EXTERNALLY) copies of any and all such materials and derivative works thereof, and to sublicense others to do any, some, or all of the foregoing, (including supporting documentation).

   Copies of these modifications should be sent to:

   software@yktvmv.bitnet or
   na.cdouglas@na-net.ornl.gov

Announcements of updates will be made through the MGNet (multigrid network) mailing list. To join MGNet, send a request to mgnet-requests@cs.yale.edu.