SAND2001–8028
Unlimited Release
Printed December 2000

ML 2.0 Smoothed Aggregation User's Guide[1]

Jonathan Hu
Computational Math & Algorithms
Sandia National Laboratories
Albuquerque, NM 87185-9214
jhu@ca.sandia.gov

Charles Tong
Lawrence Livermore Laboratory
Livermore, CA 94550
chtong@appleton.llnl.gov

Ray S. Tuminaro
Computational Math & Algorithms
Sandia National Laboratories
Albuquerque, NM 87185-9214
tuminaro@ca.sandia.gov

Abstract

ML is a multigrid preconditioning package intended to solve linear systems of equations $Ax = b$ where A is a user supplied $n \times n$ sparse matrix, b is a user supplied vector of length n and x is a vector of length n to be computed. **ML** should be used on large sparse linear systems arising from partial differential equation (PDE) discretizations. While technically any linear system can be considered, **ML** should be used on linear systems that correspond to things that work well with multigrid methods (e.g. elliptic PDEs). **ML** can be used as a stand-alone package or to generate preconditioners for a traditional iterative solver package (e.g. Krylov methods). We have supplied support for working with the **Aztec** iterative

[1]This work was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was partially performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-94AL85000.

package [5]. However, other solvers can be used by supplying a few functions. This document describes using one specific algebraic multigrid method: smoothed aggregation. Other methods exist within **ML** but are not described in this document. Examples are given illustrating the problem definition and exercising multigrid options.

Contents

1 Overview

This guide describes the use of an algebraic multigrid method within the **ML** package. The algebraic multigrid method can be used to solve systems

$$Ax = b$$

where A is a user supplied $n \times n$ sparse matrix, b is a user supplied vector of length n and x is a vector of length n to be computed. **ML** is intended to be used on large sparse linear systems arising from partial differential equation (PDE) discretizations. While technically any linear system can be considered, **ML** should be used on linear systems that correspond to things that work well with multigrid methods (e.g. elliptic PDEs).

The **ML** package is used by creating an ml object and then associating a matrix, A, and a set of multigrid parameters which describe the specifics of the solver. Once, created and initialized, the **ML** object can be used to solve linear systems. This manual proceeds by giving a brief multigrid introduction, followed by a high level user application, and specific examples for creating matrices within **ML** . The remaining sections discuss different multigrid options and some details associated with using **ML** in conjunction with other solver packages.

2 Multigrid background

A brief multigrid description is given (see [1], [2], or [3] for more information). A multigrid solver tries to approximate the original PDE problem of interest on a hierarchy of grids and use 'solutions' from coarse grids to accelerate the convergence on the finest grid. A simple multilevel iteration is illustrated in Figure 1. In the above method, the $S_k^1()$'s and $S_k^2()$'s are

```
/* Solve A_k u = b (k is current grid level)          */
proc multilevel(A_k, b, u, k)
     u = S_k^1(A_k, b, u);
     if ( k ≠ Nlevel − 1)
          P_k = determine_interpolant( A_k );
          r̂ = P_k^T (b − A_k u) ;
          Â_{k+1} = P_k^T A_k P_k;   v = 0;
          multilevel(Â_{k+1}, r̂, v, k + 1);
          u = u + P_k v;
          u = S_k^2(A_k, b, u);
```

Figure 1: High level multigrid V cycle consisting of 'Nlevel' grids to solve $A_0 u = b$.

approximate solvers corresponding to pre and post smoothing respectively. These smoothers are discussed in Section 6. For now, it suffices to view them as basic iterative methods (e.g. Gauss-Seidel) which effectively smooth out the error associated with the current approximate solution. The P_k's (interpolation operators that transfer solutions from coarse grids to finer grids) are the key ingredient that are determined automatically by the algebraic multigrid method.[2] For the purposes of this guide, it is important to understand that when the multigrid

[2]The P_k's are usually determined as a preprocessing step and not computed within the iteration.

method is used, a hierarchy of grids, grid transfer operators (P_k), and coarse grid discretizations (A_k) are created. To complete the specification of the multigrid method, smoothers must be supplied on each level. There are several smoothers within **ML** or an iterative solver package can be used, or users can write their own smoother (see Section 6).

3 A sample ML program

A brief sample program is given in Figure 2. The function `ML_Create` creates a multilevel

```
ML_Create         (&ml_object, N_grids);

ML_Init_Amatrix      (ml_object, 0,  nlocal, nlocal,(void *) A_data);
ML_Set_Amatrix_Getrow(ml_object, 0,  user_getrow, NULL, nlocal_allcolumns);
ML_Set_Amatrix_Matvec(ml_object, 0,  user_matvec);

N_levels = ML_Gen_MGHierarchy_UsingAggregation(ml_object, 0,
                                               ML_INCREASING, NULL);
ML_Gen_Smoother_Jacobi(ml_object, ML_ALL_LEVELS, ML_PRESMOOTHER, 1,
                       ML_DEFAULT);
ML_Gen_Solver    (ml_object, ML_MGV, 0, N_levels-1);
ML_Iterate(ml_object, sol, rhs);
ML_Destroy(&ml_object);
```

Figure 2: High level multigrid sample code.

solver object that is used to define the preconditioner. It requires the maximum number of multigrid levels be specified. In almost all cases, `N_grids`= 20 is more than adequate. The three 'Amatrix' statements are used to define the discretization matrix, A, that is solved. This is discussed in greater detail in Section 4. The multigrid hierarchy is generated via `ML_Gen_MGHierarchy_UsingAggregation`. Controlling the behavior of this function is discussed in Section 7. For now, it is important to understand that this function takes the matrix A and sets up relevant multigrid operators corresponding to the smoothed aggregation multigrid method [8] [7]. In particular, it generates a graph associated with A, coarsens this graph, builds functions to transfer vector data between the original graph and the coarsened graph, and then builds an approximation to A on the coarser graph. Once this second multigrid level is completed, the same operations are repeated to the second level approximation to A generating a third level. This process continues until the current graph is sufficiently coarse. The function `ML_Gen_Smoother_Jacobi` indicates that a Jacobi smoother should be used on all levels. Smoothers are discussed further in Section 6. Finally, `ML_Gen_Solver` is invoked when the multigrid preconditioner is fully specified. This function performs any needed initialization and checks for inconsistent options. After `ML_Gen_Solver` completes `ML_Iterate` can be used to solve the problem with an initial guess of `sol` (which will be overwritten with the solution) and a right hand side of `rhs`. At the present time, the external interface to vectors are just arrays. That is, `rhs` and `sol` are simple one-dimensional arrays of the same length as the number of rows in A. In addition to `ML_Iterate`, the function `ML_Solve_MGV` can be used to perform one multigrid 'V' cycle as a preconditioner.

4 Creating a matrix: single processor

Matrices are created by defining some size information, a matrix-vector product and a getrow function (which is used to extract matrix information). We note that **Aztec** users do not need to read this (or the next) section as there are special functions to convert **Aztec** matrices to **ML** matrices (see Section 8). Further, functions for some common matrix storage formats (CSR & MSR) already exist within ML and do not need to be rewritten[3].

Size information is indicated via `ML_Init_Amatrix`. The third parameter in the Figure 2 invocation indicates that a matrix with `nlocal` rows is being defined. The fourth parameter gives the vector length of vectors that can be multiplied with this matrix. Additionally, a data pointer, `A_data`, is associated with the matrix. This pointer is passed back into the matrix-vector product and getrow functions that are supplied by the user. Finally, the number '0' indicates at what level within the multigrid hierarchy the matrix is to be stored. For discussions within this document, this is always '0'. It should be noted that there appears to be some redundant information. In particular, the number of rows and the vector length in `ML_Init_Amatrix` should be the same number as the discretization matrices are square. Cases where these 'apparently' redundant parameters might be set differently are not discussed in this document.

The function `ML_Set_Amatrix_Matvec` associates a matrix-vector product with the discretization matrix. The invocation in Figure 2 indicates that the matrix-vector product function `user_matvec` is associated with the matrix located at level '`0`' of the multigrid hierarchy. The signature of `user_matvec` is

```
int user_matvec(void *A_data, int in_length, double p[], int out_length,
                double ap[])
```

where `A_data` is the user-defined data pointer specified in the `ML_Init_Amatrix`, `p` is the vector to apply to the matrix, `in_length` is the length of this vector, and `ap` is the result after multiplying the discretization matrix by the vector `p` and `out_length` is the length of `ap`.

Finally, `ML_Set_Amatrix_Getrow` associates a getrow function with the discretization matrix. This getrow function returns nonzero information corresponding to specific rows. The invocation in Figure 2 indicates that a user supplied function `user_getrow` is associated with the matrix located at level '`0`' of the multigrid hierarchy and that this matrix contains `nlocal_allcolumns` columns and that no communication (`NULL`) is used (discussed in the next section). It again appears that some redundant information is being asked as the number of columns was already given. However, when running in parallel this number will include ghost node information and is usually different from the number of rows. The signature of `user_getrow` is

```
int user_getrow(void *A_data, int N_requested_rows, int requested_rows[],
   int allocated_space, int columns[], double values[], int row_lengths[])
```

where `A_data` is the user-defined data pointer in `ML_Init_Amatrix`, `N_requested_rows` is the number of matrix rows for which information is returned, `requested_rows` are the specific rows for which information will be returned, `allocated_space` indicates how much space has been allocated in `columns` and `values` for nonzero information. On return, the user's function

[3]The functions `CSR_matvec`, `CSR_getrows`, `MSR_matvec` and `MSR_getrows` can be used.

should take each row in order within `requested_rows` and place the column numbers and the values corresponding to nonzeros in the arrays `columns` and `values`. The length of the ith requested row should appear in `row_lengths[i]`. If there is not enough allocated space in `columns` or `values`, this routine simply returns a '0', otherwise it returns a '1'.

To clarify, these functions, one concrete example is given corresponding to the matrix:

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}. \tag{1}$$

To implement this matrix, the following functions are defined:

```
int Poisson_getrow(void *A_data, int N_requested_rows, int requested_rows[],
   int allocated_space, int columns[], double values[], int row_lengths[])
{
   int count = 0, i, start, row;


   for (i = 0; i < N_requested_rows; i++) {
      if (allocated_space < count+3) return(0);
      start = count;
      row = requested_rows[i];
      if ( (row >= 0) || (row <= 4) ) {
         columns[count] = row; values[count++] = 2.;
         if (row != 0) { columns[count] = row-1; values[count++] = -1.; }
         if (row != 4) { columns[count] = row+1; values[count++] = -1.; }
      }
      row_lengths[i] = count - start;
   }
   return(1);
}
```

and

```
int Poisson_matvec(void *A_data, int in_length, double p[], int out_length,
                   double ap[])
{
   int i;

   for (i = 0; i < 5; i++ ) {
      ap[i] = 2*p[i];
      if (i != 0) ap[i] -= p[i-1];
      if (i != 4) ap[i] -= p[i+1];
   }
   return 0;
}
```

Finally, these matrix functions along with size information are associated with the fine grid discretization matrix via

```
ML_Init_Amatrix      (ml_object, 0,  5, 5, NULL);
ML_Set_Amatrix_Getrow(ml_object, 0,  Poisson_getrow, NULL, 5);
ML_Set_Amatrix_Matvec(ml_object, 0,  Poisson_matvec);
```

Notice that in these simple examples `A_data` was not used. In the next section we give a parallel example which makes use of `A_data`. The complete sample program can be found in the file `mlguide.c` within the **ML** code distribution.

5 Creating a matrix: multiple processors

Creating matrices in parallel requires a bit more work. In this section local versus global indexing as well as communication are discussed. In the description, we reconsider the previous example (1) partitioned over two processors. The matrix row indices (ranging from 0 to 4) are referred to as global indices and are independent of the number of processors being used. On distributed memory machines, the matrix is subdivided into pieces that are assigned to individual processors. **ML** requires matrices be partitioned by rows (i.e. each row is assigned to a processor which holds the entire data for that row). These matrix pieces are stored on each processor as smaller local matrices. Thus, global indices in the original matrix get mapped to local indices on each processor. In our example, we will assign global rows 0 and 4 to processor 0 and store them locally as rows 1 and 0 respectively. Global columns 0, 1, 3, and 4 are stored locally as columns 1, 3, 2, and 0. This induces the local matrix

$$\begin{pmatrix} 2 & & -1 & \\ & 2 & & -1 \end{pmatrix}.$$

Likewise, processor 1 is assigned global rows 1, 2, and 3 which are stored locally as rows 0, 1, and 2 respectively. Global columns 0 - 4 are stored locally as columns 3, 0, 1, 2, and 4 inducing the local matrix

$$\begin{pmatrix} 2 & -1 & & & -1 \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \end{pmatrix}.$$

At the present time, there are some restrictions as to what type of mappings can be used. In particular, all global rows stored on a processor must be mapped from 0 to $k-1$ where k is the number of rows assigned to this processor. This row mapping induces a partial column mapping. Any additional columns must be mapped with consecutive increasing numbers starting from k.

ML has no notion of global indices and uses only the local indices. In most cases, another package or application already mapped the global indices to local indices and so **ML** works with the existing local indices. Specifically, the parallel version of `user_getrow` and `user_matvec` should correspond to each processor's local matrix. This means that when giving the column information with `ML_Set_Amatrix_Getrow`, the total number of columns in the local matrix should be given and that when row k is requested, `user_getrow` should return the k^{th} local row using local column indices. Likewise, the matrix-vector product takes a local input vector and multiplies it by the local matrix. It is important to note that this local input vector does not contain ghost node data (i.e. the input vector is of length `nlocal` where `nlocal` is the number of matrix rows). Thus, `user_matvec` must perform the necessary communication to update ghost variables. When invoking `ML_Init_Amatrix`, the local number of rows should be given for the number of rows and the vector length[4]. A specific communication function must

[4] In contrast to `ML_Set_Amatrix_Getrow` in which the number of local columns are given (including those that correspond to ghost variables), `ML_Init_Amatrix` does not include ghost variables and so both size parameters should be the number of local rows.

also be passed into **ML** when supplying the getrow function so that **ML** can determine how local matrices on different processors are 'glued' together. The signature of the communication function is

`int user_comm(double x[], void *Adata)`

where `A_data` is the user-defined data pointer specified in the `ML_Init_Amatrix` and `x` is a vector of length `nlocal_allcolumns` specified in `ML_Set_Amatrix_Getrow`. This parameter should be set to the total number of matrix columns stored on this processor. On input, only the first `nlocal` elements of `x` are filled with data where `nlocal` is the number of rows/columns specified in `ML_Init_Amatrix`. On output, the ghost elements are updated to their current values (defined on other processors). Thus, after this function a local matrix-vector product could be properly performed using `x`. To make all this clear, we give the new functions corresponding to our two processor example.

```
int Poisson_getrow(void *A_data, int N_requested_rows, int requested_rows[],
   int allocated_space, int cols[], double values[], int row_lengths[])
{
   int m = 0, i, row, proc, *itemp, start;

   itemp = (int *) A_data;
   proc  = *itemp;

   for (i = 0; i < N_requested_rows; i++) {
      row = requested_rows[i];
      if (allocated_space < m+3) return(0);
      values[m] = 2; values[m+1] = -1; values[m+2] = -1;
      start = m;
      if (proc == 0) {
         if (row == 0) {cols[m++] = 0; cols[m++] = 2;               }
         if (row == 1) {cols[m++] = 1; cols[m++] = 3;}
      }
      if (proc == 1) {
         if (row == 0) {cols[m++] = 0; cols[m++] = 1; cols[m++] = 4;}
         if (row == 1) {cols[m++] = 1; cols[m++] = 0; cols[m++] = 2;}
         if (row == 2) {cols[m++] = 2; cols[m++] = 1; cols[m++] = 3;}
      }
      row_lengths[i] = m - start;
   }
   return(1);
}

int Poisson_matvec(void *A_data, int in_length, double p[], int out_length,
                   double ap[])
{
   int i, proc, *itemp;
   double new_p[5];

   itemp = (int *) A_data;
   proc  = *itemp;

   for (i = 0; i < in_length; i++) new_p[i] = p[i];
   Poisson_comm(new_p, A_data);

   for (i = 0; i < out_length; i++) ap[i] = 2.*new_p[i];
```

```
   if (proc == 0) {
      ap[0] -= new_p[2];
      ap[1] -= new_p[3];
   }
   if (proc == 1) {
      ap[0] -= new_p[1]; ap[0] -= new_p[4];
      ap[1] -= new_p[2]; ap[1] -= new_p[0];
      ap[2] -= new_p[3]; ap[2] -= new_p[1];
   }
   return 0;
}
```

and

```
int Poisson_comm(double x[], void *A_data)
{
   int    proc, neighbor, length, *itemp;
   double send_buffer[2], recv_buffer[2];


   itemp = (int *) A_data;
   proc  = *itemp;

   length = 2;
   if (proc == 0) {
      neighbor = 1;
      send_buffer[0] = x[0]; send_buffer[1] = x[1];
      send_msg(send_buffer,  length, neighbor);
      recv_msg(recv_buffer,  length, neighbor);
      x[2] = recv_buffer[1]; x[3] = recv_buffer[0];
   }
   else {
      neighbor = 0;
      send_buffer[0] = x[0]; send_buffer[1] = x[2];
      send_msg(send_buffer,  length, neighbor);
      recv_msg(recv_buffer,  length, neighbor);
      x[3] = recv_buffer[1]; x[4] = recv_buffer[0];
   }
   return 0;
}
```

Finally, these matrix functions along with size information are associated with the fine grid discretization matrix via

```
if      (proc == 0) {nlocal = 2; nlocal_allcolumns = 4;}
else if (proc == 1) {nlocal = 3; nlocal_allcolumns = 5;}
else                {nlocal = 0; nlocal_allcolumns = 0;}

ML_Init_Amatrix      (ml_object, 0,  nlocal, nlocal, &proc);
ML_Set_Amatrix_Getrow(ml_object, 0,  Poisson_getrow, Poisson_comm,
                      nlocal_allcolumns);
ML_Set_Amatrix_Matvec(ml_object, 0,  Poisson_matvec);
```

6 Multigrid & Smoothing Options

Several options can be set to tune the multigrid behavior. In this section, smoothing and high level multigrid choices are discussed. In the next section, the more specialized topic of the grid transfer operator is considered. The details of the functions described in these next two sections are given in the Appendix.

For most applications, smoothing choices are important to the overall performance of the multigrid method. Unfortunately, there is no simple advice as to what smoother will be best and systematic experimentation is often necessary. **ML** offers a variety of standard smoothers. Additionally, user-defined smoothers can be supplied and it is possible to use **Aztec** as a smoother. A list of **ML** functions that can be invoked to use built-in smoothers are given below along with a few general comments.

ML_Gen_Smoother_Jacobi	Typically, not the fastest smoother. Should be used with damping. For Poisson problems, the recommended damping values are $\frac{2}{3}$ (1D), $\frac{4}{5}$ (2D), and $\frac{5}{7}$ (3D). In general, smaller damping numbers are more conservative.
ML_Gen_Smoother_GaussSeidel	Probably the most popular smoother. Typically, faster than Jacobi and damping is often not necessary nor advantageous.
ML_Gen_Smoother_SymGaussSeidel	Symmetric version of Gauss Seidel. When using multigrid preconditioned conjugate gradient, the multigrid operator must be symmetrizable. This can be achieved by using a symmetric smoother with the same number of pre and post sweeps on each level.
ML_Gen_Smoother_BlockGaussSeidel	Block Gauss-Seidel with a fixed block size. Often used for PDE systems where the block size is the number of degrees of freedom (DOFs) per grid point.
ML_Gen_Smoother_VBlockJacobi	Variable block Jacobi smoother. This allows users to specify unknowns to be grouped into different blocks when doing block Jacobi.
ML_Gen_Smoother_VBlockSymGaussSeidel	Symmetric variable block Gauss-Seidel smoothing. This allows users to specify unknowns to be grouped into different blocks when doing symmetric block Gauss-Seidel.

It should be noted that the parallel Gauss-Seidel smoothers are not true Gauss-Seidel. In particular, each processor does a Gauss-Seidel iteration using off-processor information from the previous iteration.

Aztec user's [5] can invoke `ML_Gen_SmootherAztec` to use either **Aztec** solvers or **Aztec** preconditioners as smoothers on any grid level. Thus, for example, it is possible to use preconditioned conjugate-gradient (where the preconditioner might be an incomplete Cholesky factorization) as a smoother within the multigrid method. Using Krylov smoothers as a preconditioner could potentially be more robust than using the simpler schemes provided directly by **ML** . However, one must be careful when multigrid is a preconditioner to an outer Krylov iteration. Embedding an inner Krylov method within a preconditioner to an outer Krylov method may not converge due to the fact that the preconditioner can no longer be represented by a simple matrix. Finally, it is possible to pass user-defined smoothing functions into **ML** via `ML_Set_Smoother`. The signature of the user defined smoother function is

```
int user_smoothing(void *data, int x_length, double x[],
                   int rhs_length, double rhs[])
```

where `data` is a pointer given with the `ML_Set_Smoother` invocation, `x` is a vector (of length `x_length`) that corresponds to the initial guess on input and is the improved solution estimate on output, and `rhs` is the right hand side vector of length `rhs_length`. A simple (and suboptimal) damped Jacobi smoother for the finest grid of our example is given below:

```
int user_smoothing(void *data, int x_length, double x[], int rhs_length, double rhs[])
{
   int i;
   double ap[5], omega = .5; /* temp vector and damping factor */

   Poisson_matvec(data, x_length, x, rhs_length, ap);
   for (i = 0; i < x_length; i++) x[i] = x[i] + omega*(rhs[i] - ap[i])/2.;

   return 0;
}
```

A more complete smoothing example that operates on all multigrid levels is given in the file `mlguide.c`. This routine uses the functions `ML_Operator_Apply`, `ML_Operator_Get_Diag`, and `ML_Get_Amatrix` to access coarse grid matrices constructed during the algebraic multigrid process. By writing these user-defined smoothers, it is possible to tailor smoothers to a particular application or to use methods provided by other packages. In fact, the **Aztec** methods within **ML** have been implemented by writing wrappers to existing **Aztec** functions and passing them into **ML** via `ML_Set_Smoother`.

At the present time there are only a few supported general parameters that may be altered by users. However, we expect that this list will grow in the future. When using `ML_Iterate`, the convergence tolerance (`ML_Set_Tolerance`) and the frequency with which residual information is output (`ML_Set_ResidualOutputFrequency`) can both be set. Additionally, the level of diagnostic output from either `ML_Iterate` or `ML_Solve_MGV` can be set via `ML_Set_OutputLevel`. The maximum number of multigrid levels can be set via `ML_Create` or `ML_Set_MaxLevels`. Otherwise, **ML** continues coarsening until the coarsest grid is less than or equal to a specified size (by default 10 degrees of freedom). This size can be set via `ML_Aggregate_Set_MaxCoarseSize`.

7 Smoothed Aggregation Options

When performing smooth aggregation, the matrix graph is first coarsened (actually vertices are aggregated together) and then a grid transfer operator is constructed. A number of parameters can be altered to change the behavior of these phases.

7.1 Aggregation Options

A graph of the matrix is usually constructed by associating a vertex with each equation and adding an edge between two vertices i and j if there is a nonzero in the $(i,j)^{th}$ or $(j,i)^{th}$ entry. It is this matrix graph whose vertices are aggregated together that effectively determines the next coarser mesh. The above graph generation procedure can be altered in two ways. First, a block matrix graph can be constructed instead of a point matrix graph. In particular, all the degrees of freedom (DOFs) at a grid point can be collapsed into a single vertex of the matrix graph. This situation arises when a PDE system is being solved where each grid point has the same number of DOFs. The resulting block matrix graph is significantly smaller than the point matrix graph and by aggregating the block matrix graph, all unknowns at a grid point are kept together. This usually results in better convergence rates (and the coarsening is actually less expensive to compute). To indicate the number of DOFs per node, the function `ML_Aggregate_Set_NullSpace` is used. The second way in which the graph matrix can be altered is by ignoring small values. In particular, it is often preferential to ignore weak coupling during coarsening. The error between weakly coupled points is generally hard to smooth and so it is best not to coarsen in this direction. For example, when applying a Gauss-Seidel smoother to a standard discretization of

$$u_{xx} + \epsilon u_{yy} = f$$

(with $0 \leq \epsilon \leq 10^{-6}$) , there is almost no coupling in the y direction. Consequently, simple smoothers like Gauss-Seidel do not effectively smooth the error in this direction. If we apply a standard coarsening algorithm, convergence rates suffer due to this lack of y-direction smoothing. There are two principal ways to fix this: use a more sophisticated smoother or coarsen the graph only in the x direction. By ignoring the y-direction coupling in the matrix graph, the aggregation phase effectively coarsens in only the x-direction (the direction for which the errors are smooth) yielding significantly better multigrid convergence rates. In general, a drop tolerance, tol_d, can be set such that an individual matrix entry, $A(i,j)$ is dropped in the *coarsening phase* if

$$|A(i,j)| \leq tol_d * \sqrt{|A(i,i)A(j,j)|}.$$

This drop tolerance (whose default value is zero) is set by `ML_Aggregate_Set_Threshold`.

There are 3 different graph coarsening algorithms along with a hybrid scheme. A description of the three schemes (uncoupled aggregation, coupled aggregation, and MIS aggregation) along with some numerical results are given in [6]. As the default, the hybrid scheme is used which does uncoupled aggregation on finer grids and switches to the more expensive MIS aggregation on coarser grids. Poorly done aggregation can adversely affect the multigrid convergence and the time per iteration. In particular, if the scheme coarsens too rapidly multigrid convergence may suffer. However, if coarsening is too slow, the number of multigrid levels increases and the number of nonzeros per row in the coarse grid discretization matrix may grow rapidly. We refer the reader to the above paper and indicate that users might try experimenting with the different schemes via `ML_Aggregate_Set_CoarsenScheme_Uncoupled`, `ML_Aggregate_Set_CoarsenScheme_Coupled`, and `ML_Aggregate_Set_CoarsenScheme_MIS`.

7.2 Interpolation Options

An interpolation operator is built using coarsening information, seed vectors, and a damping factor. We refer the reader to [7] for details on the algorithm and the theory. In this section, we explain a few essential features to help users direct the interpolation process.

Coarsening or aggregation information is first used to create a tentative interpolation operator. This process takes a seed vector or seed vectors and builds a grid transfer operator. The details of this process are not discussed in this document. It is, however, important to understand that only a few seed vectors are needed (often but not always equal to the number of DOFs at each grid point) and that these seed vectors should correspond to components that are difficult to smooth. The tentative interpolation that results from these seed vectors will interpolate the seed vectors perfectly. It does this by ensuring that all seed vectors are in the range of the interpolation operator. This means that each seed vector can be recovered by interpolating the appropriate coarse grid vector. The general idea of smoothed aggregation (actually all multigrid methods) is that errors not eliminated by the smoother must be removed by the coarse grid solution process. If the error after several smoothing iterations was known, it would be possible to pick this error vector as the seed vector. However, since this is not the case, we look at vectors associated with small eigenvalues (or singular values in the nonsymmetric case) of the discretization operator. Errors in the direction of these eigenvectors are typically difficult to smooth as they appear much smaller in the residual ($r = Ae$ where r is the residual, A is discretization matrix, and e is the error). For most scalar PDEs, a single seed vector is sufficient and so we seek some approximation to the eigenvector associated with the lowest eigenvalue. It is well known that a scalar Poisson operator with Neumann boundary conditions is singular and that the null space is the constant vector. Thus, when applying smoothed aggregation to Poisson operators, it is quite natural to choose the constant vector as the seed vector. In many cases, this constant vector is a good choice as all spatial derivatives within the operator are zero and so it is often associated with small singular values. Within **ML** the default is to choose the number of seed vectors to be equal to the number of DOFs at each node (given via `ML_Aggregate_Set_NullSpace`). Each seed vector corresponds to a constant vector for that DOF component. Specifically, if we have a PDE system with two DOFs per node. Then one seed vector is one at the first DOF and zero at the other DOF throughout the graph. The second seed vector is zero at the first DOF and one at the other DOF throughout the graph. In some cases, however, information is known as to what components will be difficult for the smoother or what null space is associated with an operator. In elasticity, for example, it is well known that a floating structure has six rigid body modes (three translational vectors and three rotation vectors) that correspond to the null space of the operator. In this case, the logical choice is to take these six vectors as the seed vectors in smoothed aggregation. When this type of information is known, it should be given to **ML** via the command `ML_Aggregate_Set_NullSpace`.

Once the tentative prolongator is created, it is smoothed via a damped Jacobi iteration. The reasons for this smoothing are related to the theory where the interpolation basis functions must have a certain degree of smoothness (see [7]). However, the smoothing stage can be omitted by setting the damping to zero using the function `ML_Aggregate_Set_DampingFactor`. Though theoretically poorer, unsmoothed aggregation can have considerably less set up time and less cost per iteration than smoothed aggregation. When smoothing, **ML** has two ways to determine the Jacobi damping parameter and each require some estimate of the largest eigenvalue of the discretization operator. The current default is to use a few iterations of a conjugate-gradient method to estimate this value. However, if the matrix is nonsymmetric, the infinity norm of the matrix should be used instead via `ML_Aggregate_Set_SpectralNormScheme_Anorm`. There are several other internal parameters that have not been discussed in this document. In the future, it is anticipated that some of these will be made available to users.

8 ML and other packages

Currently, the primary package which can be used with **ML** is **Aztec** [5]. **ML** must be compiled with a "-DAZTEC" as described in the install notes to use this capability. All of the **Aztec** functionality that **ML** accesses is contained in the file `ml_aztec_utils.c`. In principal by creating a similar file, other solver packages could work with **ML** in the same way. For the **Aztec** users there are essentially three functions that are important. The first is `AZ_ML_Set_Amat` which converts **Aztec** matrices into **ML** matrices by making appropriate **ML** calls (see Section 4 and Section 5). It is important to note that when creating **ML** matrices from **Aztec** matrices information is not copied. Instead, wrapper functions are made so that **ML** can access the same information as **Aztec** . The second is `ML_Gen_SmootherAztec` that is used for defining **Aztec** iterative methods as smoothers (discussed in Section 6 and in the Appendix). The third function, `AZ_set_ML_preconditioner`, can be invoked to set the **Aztec** preconditioner to use the multilevel 'V' cycle constructed in **ML** . Thus, it is possible to invoke several instances of **Aztec** within one solve: smoother on different multigrid levels and/or outer iterative solve. In addition, to **Aztec** it is possible to use both the serial and distributed memory versions of SuperLU as a coarse grid solver. **ML** must be compiled with a -DSUPERLU or -DDSUPERLU respectively and the function `ML_Gen_CoarseSolverSuperLU` must be invoked. Sample programs can be found within the **ML** distribution that illustrate the use of **Aztec** and SuperLU.

9 ML Functions

Prototype

```
int AZ_ML_Set_Amat(ML *ml_object, int k, int isize, int osize, AZ_MATRIX *Amat,
                   int *proc_config)
```

Description

Create an **ML** matrix view of an existing **Aztec** matrix and store it within the 'ml_object' context.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the discretization matrix of level k is the same as given by Amat.
k	On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels†-1).
isize	On input, the number of local rows in the submatrix stored on this processor.
osize	On input, the number of columns in the local submatrix stored on this processor not including any columns associated with ghost unknowns.
Amat	On input, an **Aztec** data structure representing a matrix. See the **Aztec** User's Guide.
proc_config	On input, an **Aztec** data structure representing processor information. See the **Aztec** User's Guide.

Prototype

```
void AZ_set_ML_preconditioner(AZ_PRECOND **Precond, AZ_MATRIX *Amat,
                              ML *ml_object, int options[])
```

Description

Associate the multigrid V cycle method defined in ml_object with an **Aztec** preconditioner. Thus, when Precond and options are passed into the **Aztec** iterative solver, it will invoke the V cycle multigrid algorithm described by ml_object.

Parameters

Precond	On input, an **Aztec** data structure representing a preconditioner. On output, the multigrid V cycle method described by ml_object will be associated with this preconditioner. See the **Aztec** User's Guide.
Amat	On input, an **Aztec** data structure representing a matrix. See the **Aztec** User's Guide.
ml_object	On input, **ML** object pointer (see ML_Create) representing a V cycle multigrid method.
options	On input, an **Aztec** data structure representing user chosen options. On output, set appropriately for multigrid V cycle preconditioner.

Prototype

```
int ML_Aggregate_Create(ML_Aggregate **agg_object)
```

Description

Create an aggregate context (or handle). This instance will be used in all subsequent function invocations that set aggregation options.

Parameters

agg_object	On input, a pointer to a noninitialized ML_Aggregate object pointer. On output, points to an initialized ML_Aggregate object pointer.

Prototype

```
int ML_Aggregate_Destroy(ML_Aggregate **agg_object)
```

Description

Destroy the aggregate context, agg_object, and delete all memory allocated by **ML** in building and setting the aggregation options.

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, all memory allocated by **ML** and associated with this context is freed.

Prototype

```
int ML_Aggregate_Set_CoarsenScheme_Coupled(ML_Aggregate *agg_object)
```

Description

Set the aggregate coarsening scheme to be used as 'coupled' (see Section 7).

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, the 'coupled' aggregation will be used for automatic coarsening.

Prototype

```
int ML_Aggregate_Set_CoarsenScheme_MIS(ML_Aggregate *agg_object)
```

Description

Set the aggregate coarsening scheme to be used as 'MIS' (see Section 7).

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, the 'MIS' aggregation will be used for automatic coarsening.

Prototype

```
int ML_Aggregate_Set_CoarsenScheme_Uncoupled(ML_Aggregate *agg_object)
```

Description

Set the aggregate coarsening scheme to be used as 'uncoupled' (see Section 7).

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, the 'uncoupled' aggregation will be used for automatic coarsening.

Prototype

```
int ML_Aggregate_Set_DampingFactor( ML_Aggregate *ag, double factor)
```

Description

Set the damping factor used within smoothed aggregation. In particular, the interpolation operator will be generated by

$$P = (I - \frac{\omega}{\tilde{\rho}} A) P_t$$

where A is the discretation matrix, ω is the damping factor (default is $\frac{4}{3}$), ρ is an estimate of the spectral radius of A, and P_t are the seed vectors (tentative prolongator).

Parameters

agg_object On input, aggregate object pointer (see ML_Aggregate_Create). On output, the damping factor is set to factor.

factor On input, damping factor that will be associated with this aggregation object.

Prototype

```
int ML_Aggregate_Set_MaxCoarseSize( ML_Aggregate *agg_object, int size )
```

Description

Set the maximum coarsest mesh to 'size'. No further coarsening is performed if the total number of matrix equations is less than this 'size' (see Section 6).

Parameters

agg_object On input, aggregate object pointer (see ML_Aggregate_Create). On output, the coarsest mesh size will be set.

size On input, size indicating the maximum coarsest mesh size.

Prototype

```
int ML_Aggregate_Set_NullSpace(ML_Aggregate *agg_object, int num_PDE_eqns, int null_dim,
                               double *null_vect, int leng)
```

Description

Set the seed vectors (rigid body mode vectors) to be used in smoothed aggregation. Also indicate the number of degrees of freedom (DOF) per node so that the aggregation algorithm can group them together.

Parameters

agg_object	On input, an ML_Aggregate object pointer created by invoking ML_Aggregate_Create. On output, the seed vectors and DOFs per node are set to null_vect and num_PDE_eqns respectively.
num_PDE_eqns	On input, indicates number of equations that should be grouped in blocks when performing the aggregation. This guarantees that different DOFs at a grid point remain within the same aggregate.
null_dim	On input, number of seed vectors that will be used when creating the smoothed aggregation grid transfer operator.
null_vect	On input, the seed vectors are given in sequence. Each processor gives only the local components residing on the processor. If null, default seed vectors are used.
leng	On input, the length of each seed vector.

Prototype

```
int ML_Aggregate_Set_SpectralNormScheme_Calc( ML_Aggregate *ag )
```

Description

Set the method to be used for estimating the spectral radius of A (the discretization matrix) to be conjugate gradient. This spectral radius estimate is used when smoothing the initial prolongation operator (see ML_Aggregate_Set_DampingFactor).

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, the spectral radius estimate will be determined by a conjugate gradient routine.

Prototype

```
int ML_Aggregate_Set_SpectralNormScheme_Anorm( ML_Aggregate *ag)
```

Description

Set the method to be used for estimating the spectral radius of A (the discretization matrix) to be the infinity norm. This spectral radius estimate is used when smoothing the initial prolongation operator (see ML_Aggregate_Set_DampingFactor).

Parameters

agg_object	On input, aggregate object pointer (see ML_Aggregate_Create). On output, the spectral radius estimate will be taken as the infinity norm of the matrix.

Prototype

```
int ML_Aggregate_Set_Threshold(ML_Aggregate *agg_object, double tolerance)
```

Description

Set the drop tolerance used when creating the matrix graph for aggregation. Entries in the matrix A are dropped when $|A(i,j)| \le tol\_d * \sqrt{|A(i,i)A(j,j)|}$. See Section 7 for more details.

Parameters

agg_object	On input, an ML_Aggregate object pointer created by invoking ML_Aggregate_Create. On output, drop tolerance for creating the matrix graph is set.
tolerance	On input, value to be used for dropping matrix entries.

Prototype

```
int ML_Create(ML **ml_object, int Nlevels)
```

Description

Create an **ML** solver context (or handle). This **ML** instance will be used in all subsequent **ML** function invocations. The **ML** object has a notation of levels where different multigrid operators corresponding to different grid levels are stored.

Parameters

ml_object	On input, a pointer to a noninitialized **ML** object pointer. On output, points to an initialized **ML** object pointer.
Nlevels	Maximum number of multigrid levels within this **ML** object.

Prototype

```
int ML_Destroy(ML **ml_object)
```

Description

Destroy the **ML** solver context, ml_object, and delete all memory allocated by **ML** in building and setting options.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, all memory allocated by **ML** and associated with this context is freed.

Prototype

```
int ML_Gen_Blocks_Aggregates(ML_Aggregate *agg_object, int k, int *nblocks, int
**block_list)
```

Description

Use aggregates to partition submatrix residing on local processor into blocks. These blocks can then be used within smoothers (see for example ML_Gen_Smoother_VBlockJacobi or ML_Gen_Smoother_VBlockSymGaussSeidel).

Parameters

ml_object	On input, **ML** object pointer (see ML_Create).
k	On input, indicates level within ml_object hierarchy where the aggregate information is found that defines partitioning.
nblocks	On output, indicates the number of partitions.
block_list	On output, equation i resides in the block_list[i]th partition.

Prototype

```
int ML_Gen_Blocks_Metis(ML *ml_object, int k, int *nblocks, int **block_list)
```

Description

Use Metis to partition submatrix residing on local processor into blocks. These blocks can then be used within smoothers (see for example ML_Gen_Smoother_VBlockJacobi or ML_Gen_Smoother_VBlockSymGaussSeidel).

Parameters

ml_object	On input, **ML** object pointer (see ML_Create).
k	On input, indicates level within ml_object hierarchy where the discretization matrix is found that will be partitioned.

nblocks	On input, indicates number of partitions desired on each processor. On output, indicates the number of partitions obtained.
block_list	On output, equation i resides in the block_list[i]th partition.

Prototype

```
int ML_Gen_CoarseSolverSuperLU(ML *ml_object, int k)
```

Description

Use SuperLU for the multigrid coarse grid solver on level k within ml_object and perform any initialization that is necessary.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the coarse grid solver of level k is set to use SuperLU.
k	On input, indicates level within ml_object hierarchy (must be the coarsest level in the multigrid hierarchy).

Prototype

```
int ML_Gen_MGHierarchy_UsingAggregation(ML *ml_object, int start, int inc_or_dec,
                                        ML_Aggregate *agg_object)
```

Description

Generate a multigrid hierarchy via the method of smoothed aggregation. This hierarchy includes a series of grid transfer operators as well as coarse grid approximations to the fine grid discretization operator. On completion, return the total number of multigrid levels in the newly created hiearchy.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, coarse levels are filled with grid transfer operators and coarse grid discretizations corresponding to a multigrid hierarchy.
start	On input, indicates multigrid level within ml_object where the fine grid discretization is stored.
inc_or_dec	On input, ML_INCREMENT or ML_DECREMENT. Normally, set to ML_INCREMENT meaning that the newly created multigrid operators should be stored in the multigrid levels: start, start+1, start+2, start+3, etc. If Set to ML_DECREMENT, multigrid operators are stored in start, start-1, start-2, etc.
agg_object	On input, an initialized aggregation object defining options to the generation of grid transfer operators. If set to NULL, default values are used for all aggregation options. See ML_Aggregate_Create.

Prototype

```
void ML_Gen_SmootherAztec(ML *ml_object, int k, int options[], double params[],
                          int proc_config[], double status[], int N_iterations,
                          int pre_or_post, void (*prec_fun)(double *, int *, int *,
                          double *, AZ_MATRIX *, AZ_PRECOND *))
```

Description

Set the smoother (either pre or post as indicated by pre_or_post) at level k within the multigrid solver context to invoke **Aztec** . The specific **Aztec** scheme is given by the **Aztec** arrays: options, params, proc_config, and status and **Aztec** preconditioning function: prec_function.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, a smoother function is associated within ml_object at level k.
k	On input, indicates where the smoother function pointer will be stored within the multigrid hierarchy.

options, params *proc_config, status*	On input, **Aztec** arrays that determine the **Aztec** scheme and are used for **Aztec** to return information. See the **Aztec** User's Guide.
N_iterations	On input, maximum **Aztec** iterations within a single smoother invocation. When set to `AZ_ONLY_PRECONDITIONER`, only one iteration of the preconditioner is used without an outer Krylov method.
pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the smoother should be performed before or after the coarse grid correction.
prec_fun	On input, **Aztec** preconditioning function indicating what preconditioner will be used within **Aztec** . Normally, this is set to AZ_precondition. See the **Aztec** User's Guide.

Prototype

```
int ML_Gen_Smoother_BlockGaussSeidel(ML *ml_object, int k, int pre_or_post, int ntimes,
                                     double omega, int blocksize)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary. When using block Gauss Seidel, the total number of equations must be a multiple of blocksize. Each consecutive group of blocksize unknowns is grouped into a block and a block Gauss Seidel algorithm is applied.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to block Gauss Seidel.
k	On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels†-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.
pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.
ntimes	On input, sets the number of block Gauss Seidel iterations that will be performed.

omega On input, sets the damping parameter to be used during this block Gauss Seidel smoothing.

blocksize On input, sets the size of the blocks to be used during block Gauss Seidel smoothing.

Prototype

```
int ML_Gen_Smoother_GaussSeidel(ML *ml_object, int k, int pre_or_post, int ntimes,
                                double omega)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary.

Parameters

ml_object On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to Gauss Seidel.

k On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels†-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.

pre_or_post On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.

ntimes On input, sets the number of Gauss Seidel iterations that will be performed.

omega On input, sets the damping parameter to be used during this Gauss Seidel smoothing.

Prototype

```
int ML_Gen_Smoother_Jacobi(ML *ml_object, int k, int pre_or_post, int ntimes,
                           double omega)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to Jacobi.
k	On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels[†]-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.
pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.
ntimes	On input, sets the number of Jacobi iterations that will be performed.
omega	On input, sets the damping parameter to be used during this Jacobi smoothing. ML_DEFAULT sets it to .5

Prototype

```
int ML_Gen_Smoother_SymGaussSeidel(ML *ml_object, int k, int pre_or_post, int ntimes,
                                   double omega)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to symmetric Gauss Seidel.
k	On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels[†]-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.

pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.
ntimes	On input, sets the number of symmetric Gauss Seidel iterations that will be performed.
omega	On input, sets the damping parameter to be used during this symmetric Gauss Seidel smoothing.

Prototype

```
int ML_Gen_Smoother_VBlockJacobi(ML *ml_object, int k, int pre_or_post, int ntimes,
                                 double omega, int nBlocks, int *blockIndices)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary. A block Jacobi smoothing algorithm will be used where the size of the blocks can vary and is given by nBlocks and blockIndices (see ML_Gen_Blocks_Aggregates and ML_Gen_Blocks_Metis).

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to variable block Jacobi.
k	On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels†-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.
pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.
ntimes	On input, sets the number of block Jacobi iterations that will be performed.
omega	On input, sets the damping parameter to be used during this block Jacobi smoothing.
nBlocks	On input, indicates the total number of block equations in matrix.

blockIndices On input, blockIndices[i] indicates block to which ith element belongs.

Prototype

```
int ML_Gen_Smoother_VBlockSymGaussSeidel(ML *ml_object, int k, int pre_or_post,
                                         int ntimes, double omega, int nBlocks,
                                         int *blockIndices)
```

Description

Set the multigrid smoother for level k of ml_object and perform any initialization that is necessary. A block Gauss Seidel smoothing algorithm will be used where the size of the blocks can vary and is given by nBlocks and blockIndices (see ML_Gen_Blocks_Aggregates and ML_Gen_Blocks_Metis).

Parameters

ml_object On input, **ML** object pointer (see ML_Create). On output, the pre or post smoother of level k is set to variable block symmetric Gauss Seidel.

k On input, indicates level within ml_object hierarchy (should be between 0 and Nlevels†-1). ML_ALL_LEVELS sets the smoothing on all levels in ml_object.

pre_or_post On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the pre or post smoother is to be set.

ntimes On input, sets the number of block symmetric Gauss Seidel iterations that will be performed.

omega On input, sets the damping parameter to be used during this block symmetric Gauss Seidel smoothing.

nBlocks On input, indicates the total number of block equations in matrix.

blockIndices On input, blockIndices[i] indicates block to which ith element belongs.

Prototype

```
int ML_Gen_Solver(ML *ml_object, int scheme, int finest_level, int coarsest_level)
```

Description

Initialize the **ML** solver context, ml_object, so that it is ready to be used in a solve. ML_Gen_Solver should be called after the multigrid cycle is fully specified but before ML_Iterate or ML_Solve_MGV is invoked.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, all necessary initialization is completed.
scheme	On input, must be set to ML_MGV indicating a multigrid V cycle is used.
finest_level	On input, indicates the location within ml_object where the finest level is stored. Normally, this is '0'.
coarsest_level	On input, indicates location within ml_object where the coarsest grid is stored. When doing smoothed aggregation, this can be determined using the total number of multigrid levels returned by ML_Gen_MGHierarchy_UsingAggregation.

Prototype

```
int ML_Get_Amatrix(ML *ml_object, int k, ML_Operator **matrix)
```

Description

Set *matrix to point to the discretization matrix associated at level k within the multigrid solver context ml_object. This pointer can then be passed into functions like: ML_Operator_Apply, ML_Operator_Get_Diag, and ML_Operator_Getrow.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create).
k	On input, indicates which level within the multigrid hierarchy should be accessed.
matrix	On output, *matrix points to the discretization matrix at level k within the multigrid hierarchy. This pointer can then be passed into the functions ML_Operator_Apply, ML_Operator_Get_Diag, and ML_Operator_Getrow.

Prototype

```
int ML_Init_Amatrix(ML *ml_object, int k, int ilen, int olen, void *data)
```

Description

Set the size information for the discretization matrix associated at level k within ml_object. Additionally, associate a data pointer that can be used when writting matrix-vector product and matrix getrow functions.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, size information is associated with the discretization matrix at level k.
k	On input, indicates where discretization size information will be stored within the multigrid hierarchy.
ilen	On input, the number of local rows in the submatrix stored on this processor.
olen	On input, the number of columns in the local submatrix stored on this processor not including any columns associated with ghost unknowns.
data	On input, a data pointer that will be associated with the discretization matrix and could be used for matrix-vector product and matrix getrow functions.

Prototype

```
int ML_Iterate(ML *ml_object, double *sol, double *rhs)
```

Description

Iterate until convergence to solve the linear system using the multigrid V cycle defined within ml_object.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create).
sol	On input, a vector containing the initial guess for the linear system contained in ml_object. On output, the solution obtained by performing repeated multigrid V cycles.
rhs	On input, a vector contain the right hand side for the linear system contained in ml_object.

Prototype

```
int ML_Operator_Apply(ML_Operator *A, int in_length, double p[], int out_length,
                      double ap[])
```

Description

Invoke a matrix-vector product using the ML_Operator `A`. That is perform $ap = A * p$. Any communication or ghost variables work needed for this operation is also performed.

Parameters

A	On input, an ML_Operator (see ML_Get_Amatrix).
in_length	On input, length of vector p (not including ghost variable space).

p On input, vector which will be multiplied by A.

out_length On input, length of vector ap.

ap On output, vector containing result of $A * p$.

Prototype

```
int ML_Operator_Get_Diag(ML_Operator *A, int length, double **diag)
```

Description

Get the diagonal of the ML_Operator `A` (which is assumed to be square).

Parameters

A On input, an ML_Operator (see ML_Get_Amatrix).

length On input, number of diagonal elements wanted.

diag On output, sets a pointer to an array containing the diagonal elements. NOTE: this is not a copy but in fact a pointer into an ML data structure. Thus, this array should not be freed.

Prototype

```
int ML_Operator_Getrow(ML_Operator *A, int N_requested_rows, int requested_rows[],
                       int allocated_space, int columns[], double values[],
                       int row_lengths[])
```

Description

Get a row (or several rows) from the ML_Operator `A`. If there is not enough space in `columns` and `values` to hold the nonzero information, this routine returns a '0'. Otherwise, a '1' is returned.

Parameters

A	On input, an ML_Operator (see ML_Get_Amatrix).
N_requested_rows	On input, number of matrix rows for which information is returned.
requested_rows	On input, specific rows for which information will be returned.
allocated_space	On input, length of `columns` and `values`.
columns	On output, the column numbers of each nonzero within each row requested in `requested_rows` (where column numbers associated with `requested_rows[i]` appear before column numbers associated with `requested_rows[j]` with `i` < `j`).
values	On output, the nonzero values of each nonzero within each row requested in `requested_rows` (where nonzero values associated with `requested_rows[i]` appear before nonzero values associated with `requested_rows[j]` with `i` < `j`).
row_lengths	On output, `row_lengths[i]` indicates the number of nonzeros in row `i`.

Prototype

```
int ML_Set_Amatrix_Getrow(ML *ml_object, int k, int (*getrow)(void *, int , int* , int,
                          int*, double* , int*), int (*comm )(double *vec, void *data),
                          int comm_vec_leng)
```

Description

Set the matrix getrow function for the discretization matrix associated at level k within the multigrid solver context ml_object.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, matrix getrow function is associated with the discretization matrix at level k.
k	On input, indicates where the matrix getrow function pointer will be stored within the multigrid hierarchy.

getrow	On input, a function pointer to the user-defined matrix getrow function. See Section 4.
comm	On input, a function pointer to the user-defined communication function. See Section 5.

Prototype

```
int ML_Set_Amatrix_Matvec(ML *ml_object, int k, int (*matvec)(void *, int, double *,
                                       int, double *))
```

Description

Set the matrix-vector product function for the discretization matrix associated at level k within the multigrid solver context ml_object.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, matrix-vector product function is associated with the discretization matrix at level k.
k	On input, indicates where the matrix-vector product function pointer is stored within the multigrid hierarchy.
matvec	On input, a function pointer to the user-defined matrix-vector product function. See Section 4

.

Prototype

```
int ML_Set_ResidualOutputFrequency(ML *ml_object, int output_freq)
```

Description

Set the output frequency of residual information. ML_Iterate prints the two norm of the residual every output_freq iterations.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, residual printing frequency is set.
output_freq	On input, value to use for printing frequency.

Prototype

```
int ML_Set_Smoother(ML *ml_object, int k , int pre_or_post, void *data,
                    int (*func)(void *, int, double *, int, double *), char *label)
```

Description

Set the smoother (either pre or post as indicated by pre_or_post) at level k within the multigrid solver context to invoke the user-defined function 'func' and pass in the data pointer 'data'.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, a smoother function is associated within ml_object at level k.
k	On input, indicates where the smoother function pointer will be stored within the multigrid hierarchy.
pre_or_post	On input, ML_PRESMOOTHER or ML_POSTSMOOTHER indicating whether the smoother should be performed before or after the coarse grid correction.
data	On input, a data pointer that will be passed into the user-defined function 'func'.
func	On input, smoothing function to be used at level k when performing a multigrid V cycle. The specific signature and details of this function are given in Section 6.
label	On input, a character string to be associated with Smoother. This string is printed by some routines when identifying the method.

Prototype

```
int ML_Set_Tolerance(ML *ml_object, double tolerance)
```

Description

Set the convergence criteria for ML_Iterate. Convergence is declared when the 2-norm of the residual is reduced by 'tolerance' over the initial residual. This means that if the initial residual is quite small (i.e. the initial guess corresponds quite closely with the true solution), ML_Iterate might continue to iterate without recognizing that the solution can not be improved due to round-off error. Note: the residual is always computed after performing presmoothing on the finest level (as opposed to at the beginning or end of the iteration). Thus, the true residual should be a little bit better than the one used by ML.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create). On output, tolerance is set for convergence of ML_Iterate.
tolerance	On input, value to use for convergence tolerance.

Prototype

```
int ML_Solve_MGV(ML *ml_object, double *din, double *dout)
```

Description

Perform one multigrid V cycle iteration to the solve linear system defined within ml_object.

Parameters

ml_object	On input, **ML** object pointer (see ML_Create).
din	On input, the right hand side vector to be used when performing multigrid.

dout On output, an approximate solution obtained after one multigrid V cycle.

[†]Nlevels refers to the argument given with `ML_Create`.

References

[1] A. Brandt, *Multi-level Adaptive Solutions to Boundary-Value Problems*, Math. Comp., 31 (1977), pp. 333–390.

[2] W. Hackbusch, *Multi-grid Methods and Applications*, Springer-Verlag, Berlin, 1985.

[3] ——, *Iterative Solution of Large Sparse Linear Systems of Equations*, Springer-Verlag, Berlin, 1994.

[4] C. Tong and R. Tuminaro, *ML2.0 Smoothed Aggregation User's Guide*, Tech. Rep. SAND2001-8028, Sandia National Laboratories, Albq, NM, 2000.

[5] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid, *Official Aztec user's guide: Version 2.1*, Tech. Rep. Sand99-8801J, Sandia National Laboratories, Albuquerque NM, 87185, Nov 1999.

[6] R. Tuminaro and C. Tong, *Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines*, in SuperComputing 2000 Proceedings, J. Donnelley, ed., 2000.

[7] P. Vanek, M. Brezina, and J. Mandel, *Convergence of Algebraic Multigrid Based on Smoothed Aggregation*, Tech. Rep. report 126, UCD/CCM, Denver, CO, 1998.

[8] P. Vanek, J. Mandel, and M. Brezina, *Algebraic Multigrid Based on Smoothed Aggregation for Second and Fourth Order Problems*, Computing, 56 (1996), pp. 179–196.