SAND2004–2195
Unlimited Release
Printed May 2004

ML 3.0 Smoothed Aggregation User's Guide

Marzio Sala
Computational Math & Algorithms
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Jonathan J. Hu and Ray S. Tuminaro
Computational Math & Algorithms
Sandia National Laboratories
P.O. Box 0969
Livermore, CA 94551-0969

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 2.1 and AZTECOO iterative package [15]. However, other solvers can be used by supplying a few functions.

This document describes one specific algebraic multigrid approach: smoothed aggregation. This approach is used within several specialized multigrid methods: one for the eddy current formulation for Maxwell's equations, and a multilevel and domain decomposition method for symmetric and non-symmetric systems of equations (like elliptic equations, or compressible and incompressible fluid dynamics problems). Other methods exist within **ML** but are not described in this document. Examples are given illustrating the problem definition and exercising multigrid options.

(page intentionally left blank)

Contents

1 Notational Conventions **7**

2 Overview **7**

3 Multigrid Background **7**

4 Configuring and Building ML **8**
4.1 Building in Standalone Mode 9
4.2 Building with AZTEC 2.1 Support 9
4.3 Building with TRILINOS Support (RECOMMENDED) 9
4.3.1 Enabling Third Party Library Support 10
4.3.2 Enabling Profiling 11

5 ML and Epetra: Getting Started with the MultiLevelPreconditioner Class **12**
5.1 Example 1: ml_example_epetra_preconditioner.cpp 12
5.2 Example 2: ml_example_epetra_preconditioner_2level.cpp 14

6 Parameters for the ML_Epetra::MultiLevelPreconditioner Class **15**
6.1 Setting Options on a Specific Level 16
6.2 General Usage of the Parameter List 16
6.3 Default Parameter Settings for Common Problem Types 17
6.4 Commonly Used Parameters 19
6.5 List of All Parameters for MultiLevelPreconditioner Class 19
6.5.1 General Options 19
6.5.2 Aggregation Parameters 21
6.5.3 Smoothing Parameters 21
6.5.4 Coarsest Grid Parameters 22

7 Advanced Usage of ML **23**

8 Multigrid & Smoothing Options **24**

9 Smoothed Aggregation Options **26**
9.1 Aggregation Options 26
9.2 Interpolation Options 27

10 Advanced Usage of ML and Epetra **28**

11 Using ML without Epetra **29**
11.1 Creating a **ML** matrix: Single Processor 29
11.2 Creating a **ML** matrix: Multiple Processors 31

12 Visualization Capabilities **34**

13 ML Functions **36**
AZ_ML_Set_Amat . . . 36
AZ_set_ML_preconditioner . . . 36
ML_Aggregate_Create . . . 37
ML_Aggregate_Destroy . . . 37
ML_Aggregate_Set_CoarsenScheme_Coupled . . . 38
ML_Aggregate_Set_CoarsenScheme_MIS . . . 38
ML_Aggregate_Set_CoarsenScheme_Uncoupled . . . 39
ML_Aggregate_Set_CoarsenScheme_METIS . . . 39
ML_Aggregate_Set_CoarsenScheme_ParMETIS . . . 40
ML_Aggregate_Set_DampingFactor . . . 40
ML_Aggregate_Set_MaxCoarseSize . . . 41
ML_Aggregate_Set_NullSpace . . . 41
ML_Aggregate_Set_SpectralNormScheme_Calc . . . 42
ML_Aggregate_Set_SpectralNormScheme_Anorm . . . 42
ML_Aggregate_Set_Threshold . . . 43
ML_Create . . . 43
ML_Destroy . . . 44
ML_Gen_Blocks_Aggregates . . . 44
ML_Gen_Blocks_Metis . . . 45
ML_Gen_CoarseSolverSuperLU . . . 45
ML_Gen_MGHierarchy_UsingAggregation . . . 46
ML_Gen_SmootherAmesos . . . 47
ML_Gen_SmootherAztec . . . 47
ML_Gen_Smoother_BlockGaussSeidel . . . 48
ML_Gen_Smoother_GaussSeidel . . . 49
ML_Gen_Smoother_Jacobi . . . 50
ML_Gen_Smoother_SymGaussSeidel . . . 51
ML_Gen_Smoother_VBlockJacobi . . . 51
ML_Gen_Smoother_VBlockSymGaussSeidel . . . 52
ML_Gen_Solver . . . 53
ML_Get_Amatrix . . . 54
ML_Init_Amatrix . . . 54
ML_Iterate . . . 55
ML_Operator_Apply . . . 56
ML_Operator_Get_Diag . . . 56
ML_Operator_Getrow . . . 57
ML_Set_Amatrix_Getrow . . . 58
ML_Set_Amatrix_Matvec . . . 58
ML_Set_ResidualOutputFrequency . . . 59
ML_Set_Smoother . . . 59
ML_Set_Tolerance . . . 60
ML_Solve_MGV . . . 61

1 Notational Conventions

In this guide, we show typed commands in this font:

```
% a_really_long_command
```

The character % indicates any shell prompt[1]. Function names are shown as ML_Gen_Solver. Names of packages or libraries as reported in small caps, as EPETRA. Mathematical entities are shown in italics.

2 Overview

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

$$Ax = b \tag{1}$$

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 (distributed) 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 a **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 is structured as follows. Multigrid and multilevel methods are briefly recalled in Section 3. The process of configuring and building **ML** is outlined in Section 4. Section 5 shows the basic usage of **ML** as a black-box preconditioner for EPETRA matrices. The definition of (parallel) preconditioners using ML_Epetra::MultiLevelPreconditioner is detailed. This class only requires the linear system matrix, and a list of options. Available parameters for ML_Epetra::MultiLevelPreconditioner are reported in Section 6. More advanced uses of **ML** are presented in Section 7. Here, we present how to define and fine-tune smoothers, coarse grid solver, and the multilevel hierarchy. Multigrid options are reported in Section 8. Smoothing options are reported in Section 9, where we also present how to construct a user's defined smoother. Advanced usage of **ML** with EPETRA objects is reported in Section 10. Section 11 reports how to define matrices in **ML** format without depending on EPETRA. Section 12 detailes the (limited) visualization capabilities of **ML**.

3 Multigrid Background

A brief multigrid description is given (see [1], [6], or [7] 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

[1] For simplicity, commands are shown as they would be issued in a Linux or Unix environment. Note, however, that **ML** has and can be built successfully in a Windows environment.

```
/* 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 (1), with $A_0 = A$.

are approximate solvers corresponding to k steps of pre and post smoothing, respectively. These smoothers are discussed in Section 8. 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 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 8).

4 Configuring and Building ML

ML is configured and built using the GNU autoconf [4] and automake [5] tools. It can be configured and build as a standalone package without or with AZTEC 2.1 support (as detailed in Section 4.1 and 4.2), or as a part of the TRILINOS framework [8] (as described in Section 4.3). Even though **ML** can be compiled and used as a standalone package, the recommended approach is to build **ML** as part of the TRILINOS framework, as a richer set of features are then available.

ML has been configured and built successfully on a wide variety of operating systems, and with a variety of compilers (as reported in Table 1).

Operating System	Compilers(s)
Linux	GNU and Intel
IRIX N32, IRIX 64, HPUX, Solaris, DEC	Native
ASCI Red	Native and Portland Group
CPlant	Native
Windows	Microsoft

Table 1: Main operating systems and relative compilers supported by **ML**.

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

Although it is possible to configure directly in the **ML** home directory, we strongly advise against this. Instead, we suggest working in an independent directory and configuring and building there.

4.1 Building in Standalone Mode

To configure and build **ML** as a standalone package without any AZTEC support, do the following. It's assumed that the shell variable `$ML_HOME` identifies the **ML** directory.

```
% cd $ML_HOME
% mkdir standalone
% cd standalone
% $ML_HOME/configure --disable-epetra --disable-aztecoo \
    --prefix=$ML_HOME/standalone
% make
% make install
```

The **ML** library file `libml.a` and the header files will be installed in the directory specified in `--prefix`.

4.2 Building with AZTEC 2.1 Support

To enable the supports for AZTEC 2.1, **ML** must be configured with the options reported in the previous section, plus `--with-ml_aztec2_1` (defaulted to `no`).

All of the AZTEC 2.1 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 11.1 and Section 11.2). 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 8 and Section 13). 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.

4.3 Building with TRILINOS Support (RECOMMENDED)

We recommend to configure and build **ML** as part of the standard TRILINOS build and configure process. In fact, **ML** is built by default if you follow the standard TRILINOS configure and build directions. Please refer to the TRILINOS documentation for information about the configuration and building of other TRILINOS packages.

To configure and build **ML** through TRILINOS, you may need do the following (actual configuration options may vary depending on the specific architecture, installation, and user's need). It's assumed that shell variable `$TRILINOS_HOME` identifies the TRILINOS directory, and, for example, that we are compiling under LINUX and MPI.

```
% cd $TRILINOS_HOME
% mkdir LINUX_MPI
% cd LINUX_MPI
% $TRILINOS_HOME/configure  --with-mpi-compilers \
    --prefix=$TRILINOS_HOME/LINUX_MPI
% make
% make install
```

If required, other TRILINOS and **ML** options can be specified in the configure line. A complete list of **ML** options is given in Section 4.3.1 and 4.3.2. You can also find a complete list and explanations by typing `./configure --help` in the **ML** home directory.

4.3.1 Enabling Third Party Library Support

ML can be configured with the following third party libraries (TPLs): SUPERLU, SUPERLU_DIST, METIS, and PARMETIS. It can take advantage of the following TRILINOS packages: IFPACK, TEUCHOS, TRIUTILS, AMESOS. Through AMESOS, **ML** can interface with the direct solvers KLU, UMFPACK , SUPERLU, SUPERLU_DIST[3], MUMPS. It is assumed that you have already built the appropriate libraries (e.g., `libsuperlu.a`) and have the header files. To configure **ML** with one of the above TPLs, you must enable the particular TPL interface in **ML**. All of the options below are disabled by default.

The same configure options that one uses to enable certain other Trilinos packages also enables the interfaces to those packages within **ML**:

`--enable-epetra`	Enable support for the EPETRA package.
`--enable-aztecoo`	Enable support for the AZTECOO package.
`--enable-amesos`	Enables support for the AMESOS package. AMESOS is an interface with several direct solvers. **ML** supports UMFPACK [2], KLU, SUPERLU_DIST (1.0 and 2.0), MUMPS [14]. This package is used only in function ML_Gen_SmootherAmesos.
`--enable-teuchos`	Enables support for the TEUCHOS package. This package is used only in the definition of class ML_Epetra::MultiLevelPreconditioner (see Section 5). and by the Amesos smoother
`--enable-triutils`	Enables support for the TRIUTILS package. **ML** uses TRIUTILS only in some examples, to create the linear system matrix.
`--enable-ifpack`	Enable support for the IFPACK package [9]. IFPACK is used only to create smoothers via ML_Gen_SmootherIfpack.

[3]Currently, **ML** can support SUPERLU_DIST directly (without AMESOS support), or through AMESOS.

`--enable-anasazi`	Enable support for the ANASAZI package. ANASAZI is a high level interface package for various eigenvalue computations.

The following configure line options enable interfaces in **ML** to certain TPLs.

`--with-ml_metis`	Enables interface for METIS [12].
`--with-ml_parmetis2x`	Enables interface for PARMETIS, version 2.x.
`--with-ml_parmetis3x`	Enables interface for PARMETIS [11], version 3.x.
`--with-ml_superlu`	Enables **ML** interface for serial SUPERLU [3]. The **ML** interface to SUPERLU is deprecated in favor of the AMESOS interface.
`--with-ml_superlu_dist`	Enables **ML** interface for SUPERLU_DIST [3]. The **ML** interface to SUPERLU_DIST is deprecated in favor of the AMESOS interface.

For METIS, PARMETIS, and the **ML** interface to SUPERLU and SUPERLU_DIST, the user must specify the location of the header files, with the option

`--with-incdirs=include-locations`

(Header files for TRILINOS libraries are automatically located if **ML** is built through the TRILINOS `configure`.) In order to link the **ML** examples, the user must indicate the location of all the enabled packages' libraries[4] , with the option

`--with-ldflags=lib-locations`

The user might find useful the option

`--disable-examples`

which turns off compilation and linking of the examples.

More details about the installation of TRILINOS can be found at the TRILINOS web site,

`http://software.sandia.gov/Trilinos`

and [10, Chapter 1].

4.3.2 Enabling Profiling

All of the options below are disabled by default.

`--enable-ml_timing`	This prints out timing of key **ML** routines.
`--enable-ml_flops`	This enables printing of flop counts.

Timing and flop counts are printed when the associated object is destroyed.

[4] An example of configuration line that enables METIS and PARMETIS might be as follows: `./configure --with-mpi-compilers --enable-ml_metis --enable-ml_parmetis3x --with-cflags="-I$HOME/include" --with-cppflags="-I$HOME/include" --with-ldflags="-L$HOME/lib/LINUX_MPI -lparmetis-3.1 -lmetis-4.0"` .

5 ML and Epetra: Getting Started with the MultiLevelPreconditioner Class

In this Section we show how to use **ML** as a preconditioner to EPETRA and AZTECOO through the MultiLevelPreconditioner class[5] in the ML_Epetra namespace.[6] Although limited to algebraic multilevel preconditioners, this allows the use of **ML** as a black-box preconditioner.

The MultiLevelPreconditioner class automatically constructs all the components of the preconditioner, using the parameters specified in a TEUCHOS parameter list. The constructor of this class takes as input an Epetra_RowMatrix pointer and a TEUCHOS parameter list[7].

In order to compile, it may also be necessary to include the following files: `ml_config.h` (as first **ML** include), `Epetra_ConfigDefs.h` (as first EPETRA include), `Epetra_RowMatrix.h`, `Epetra_MultiVector.h`, `Epetra_LinearProblem.h`, and `AztecOO.h`. Check the EPETRA and AZTECOO documentation for more details. Additionally, the user must include the header file `"ml_epetra_preconditioner.h"`. Also note that the macro `HAVE_CONFIG_H` must be defined either in the user's code or as a compiler flag.

5.1 Example 1: ml_example_epetra_preconditioner.cpp

We now give a very simple fragment of code that uses the MultiLevelPreconditioner. For the complete code, see `$ML_HOME/examples/ml_example_epetra_preconditioner.cpp`. (In order to be effectively compiled, this example requires **ML** to be configured with option `--enable-triutils`; see Section 4.) The linear operator `A` is derived from an Epetra_RowMatrix, `Solver` is an AztecOO object, and `Problem` is an Epetra_LinearProblem object.

```
#include "ml_include.h"
#include "ml_epetra_preconditioner.h"
#include "Teuchos_ParameterList.hpp"

...

  Teuchos::ParameterList MList;

  // set default values for smoothed aggregation in MLList
  ML_Epetra::SetDefaults("SA",MLList);

  // overwrite with user's defined parameters
  MLList.set("max levels",6);
  MLList.set("increasing or decreasing","decreasing");
  MLList.set("aggregation: type", "MIS");
  MLList.set("coarse: type","Amesos-KLU");
```

[5] The MultiLevelPreconditioner class is derived from the Epetra_RowMatrix class.

[6] **ML** does not rely on any particular matrix format or iterative solver. Examples of using of **ML** as a preconditioner for user-defined matrices (i.e., non-Epetra matrices) are reported in Section 11.1 and 11.2.

[7] In order to use the MultiLevelPreconditioner class, **ML** must be configured with options `--enable-epetra --enable-teuchos`.

```
// create the preconditioner
ML_Epetra::MultiLevelPreconditioner * MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(A, MLList, true);

// create an AztecOO solver
AztecOO Solver(Problem)

// set preconditioner and solve
Solver.SetPrecOperator(MLPrec);
Solver.SetAztecOption(AZ_solver, AZ_gmres);
Solver.Iterate(Niters, 1e-12);

...

delete MLPrec;
```

We now detail the general procedure to define the MultiLevelPreconditioner. First, the user defines a TEUCHOS parameter list[8]. Table 2 briefly reports the most important methods of this class.

`set(Name,Value)`	Add entry `Name` with value and type specified by `Value`. Any C++ type (like int, double, a pointer, etc.) is valid.
`get(Name,DefValue)`	Get value (whose type is automatically specified by `DefValue`). If not present, return `DefValue`.
`subList(Name)`	Get a reference to sublist `List`. If not present, create the sublist.

Table 2: Some methods of Teuchos::ParameterList class.

Input parameters are set via method `set(Name,Value)`, where `Name` is a string defining the parameter, and `Value` is the specified parameter, that can be any C++ object or pointer. A complete list of parameters available for class MultiLevelPreconditioner is reported in Section 6.

The parameter list is passed to the constructor, together with a pointer to the matrix, and a boolean flag. If this flag is set to `false`, the constructor will not create the multilevel hierarchy until when `MLPrec->ComputePreconditioner()` is called. The hierarchy can be destroyed using `MLPrec->Destroy()`[9]. For instance, the user may define a code like:

```
// A is still not filled with numerical values
ML_Epetra::MultiLevelPreconditioner * MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(A, MLList, false);

// compute the elements of A
...
// now compute the preconditioner
```

[8] See the TEUCHOS documentation for a detailed overview of this class.

[9] We suggest to create the preconditioning object with `new` and to free memory with `delete`. Some MPI calls occur in `Destroy()`, so the user should not call `MPI_Finalize()` or delete the communicator used by **ML** before the preconditioning object is destroyed.

```
MLPrec->ComputePreconditioner();

// solve the linear system
...
// destroy the previously define preconditioner, and build a new one
MLPrec->Destroy();

// re-compute the elements of A
// now re-compute (if required) the preconditioner
MLPrec->ComputePreconditioner();

// re-solve the linear system
```

In this fragment of code, the user defines the **ML** preconditioner, but the preconditioner is created only with the call `ComputePreconditioner()`. This may be useful, for example, when **ML** is used in conjunction with nonlinear solvers (like NOX [13]).

5.2 Example 2: ml_example_epetra_preconditioner_2level.cpp

As a second example, here we explain with some details the construction of a 2-level domain decomposition preconditioner, with a coarse space defined using aggregation.

File `$ML_HOME/examples/ml_example_epetra_preconditioner_2level.cpp` reports the entire code. In the example, the linear system matrix `A`, coded as an Epetra_CrsMatrix, corresponds to the discretization of a 2D Laplacian on a Cartesian grid. `x` and `b` are the solution vector and the right-hand side, respectively.
The AztecOO linear problem is defined as

```
Epetra_LinearProblem problem(&A, &x, &b);
AztecOO solver(problem);
```

We create the TEUCHOS parameter list as follows:

```
ParameterList MLList;
ML_Epetra::SetDefaults("DD", MLList);
MLList.set("max levels",2);
MLList.set("increasing or decreasing","increasing");

MLList.set("aggregation: type", "METIS");
MLList.set("aggregation: nodes per aggregate", 16);
MLList.set("smoother: pre or post", "both");
MLList.set("coarse: type","Amesos-KLU");
MLList.set("smoother: type", "Aztec");
```

The last option tells **ML** to use the AZTEC preconditioning function as a smoother. All AZTEC preconditioning options can be used as **ML** smoothers. AZTEC requires an integer vector `options` and a double vector `params`. Those can be defined as follows:

```
int options[AZ_OPTIONS_SIZE];
double params[AZ_PARAMS_SIZE];
```

```
AZ_defaults(options,params);
options[AZ_precond] = AZ_dom_decomp;
options[AZ_subdomain_solve] = AZ_icc;
MLList.set("smoother: Aztec options", options);
MLList.set("smoother: Aztec params", params);
```

The last two commands set the pointer to `options` and `params` in the parameter list[10].

The **ML** preconditioner is created as in the previous example,

```
ML_Epetra::MultiLevelPreconditioner * MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(A, MLList, true);
```

and we can check that no options have been mispelled, using

```
MLPrec->PrintUnused();
```

The AztecOO solver is called using, for instance,

```
solver.SetPrecOperator(MLPrec);

solver.SetAztecOption(AZ_solver, AZ_cg_condnum);

solver.SetAztecOption(AZ_kspace, 160);

solver.Iterate(1550, 1e-12);
```

Finally, some (limited) information about the preconditioning phase are obtained using

```
cout << MLPrec->GetOutputList();
```

Note that the input parameter list is *copied* in the construction phase, hence later changes to `MLList` will not affect the preconditioner. Should the user need to modify parameters in the `MLPrec`'s internally stored parameter list, he can get a reference to the internally stored list:

```
ParameterList & List = MLPrec->GetList();
```

and then directly modify `List`.

6 Parameters for the ML_Epetra::MultiLevelPreconditioner Class

In this section we give general guidelines for using the MultiLevelPreconditioner class effectively. The complete list of input parameters is also reported.

[10] Only the *pointer* is copied in the parameter list, not the array itself. Therefore, `options` and `params` should not go out of scope before the destruction of the preconditioner.

6.1 Setting Options on a Specific Level

Some of the parameters that affect MultiLevelPreconditioner can in principle be different from level to level. By default, the set method for the MultiLevelPreconditioner class affects all levels in the multigrid hierarchy. In order to change a setting on a particular level (say, `d`), the string "`(level d)`" is appended to the option string (note that a space must separate the option and the level specification). For instance, assuming decreasing levels starting from 4, one could set the aggregation schemes as follows:

```
MLList.set("aggregation: type","Uncoupled");
MLList.set("aggregation: type (level 1)","METIS");
MLList.set("aggregation: type (level 3)","MIS");
```

If the finest level is 0, and one has 5 levels, the code will use `Uncoupled` for level 0, `METIS` for levels 1 and 2, then `MIS` for levels 3 and 4.

In §6.5, parameters that can be set differently on individual levels are denoted with the symbol $\star$ (that is not part of the parameter name). Note that some parameters (e.g., `Uncoupled-MIS` aggregation) correspond to quantities that must be the same at all levels.

6.2 General Usage of the Parameter List

All **ML** options can have a common prefix, specified by the user in the construction phase. For example, suppose that we require `ML:` (in this case with a trailing space) to be the prefix. The constructor will be

```
char Prefix[] = "ML: ";
ML_Epetra::MultiLevelPreconditioner * MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(*A, MLList, true, Prefix);
```

A generic parameter, say `aggregation: type`, will now be defined as

```
MLIst.set("ML: aggregation: type", "METIS");
```

It is important to point out that some options can be effectively used only if **ML** has been properly configured. In particular:

- METIS aggregation scheme requires `--with-ml_metis`, or otherwise the code will include all nodes in the calling processor in a unique aggregate;
- PARMETIS aggregation scheme required `--with-ml_metis --enable-epetra` and `--with-ml_parmetis2x` or `--with-ml_parmetis3x`.
- AMESOS coarse solvers require `--enable-amesos`. Moreover, AMESOS must have been configure to support the requested coarse solver. Please refer to the AMESOS documentation for more details;
- IFPACK smoother requires `--enable-ifpack`.

6.3 Default Parameter Settings for Common Problem Types

The MultiLevelPreconditioner class provides default values for four different preconditioner types:

1. Linear elasticity
2. Classical 2-level domain decomposition for the advection diffusion operator
3. 3-level algebraic domain decomposition for the advection diffusion operator
4. Eddy current formulation of Maxwell's equations

Default values are listed in Table 3. In the table, `SA` refers to "classical" smoothed aggregation (with small aggregates and relative large number of levels), `DD` and `DD-ML` to domain decomposition methods (whose coarse matrix is defined using aggressive coarsening and limited number of levels). `Maxwell` refers to the solution of Maxwell's equations.

Default values for the parameter list can be set by `ML_Epetra::SetDefaults()`. The user can easily put the desired default values in a given parameter list as follows:

```
Teuchos::ParameterList MLList;
ML_Epetra::SetDefaults(ProblemType, MLList);
```

or as

```
Teuchos::ParameterList MLList;
ML_Epetra::SetDefaults(ProblemType, MLList, Prefix);
```

`Prefix` (defaulted to an empty string) is the prefix to assign to each entry in the parameter list.

For `DD` and `DD-ML`, the default smoother is `Aztec`, with an incomplete factorization ILUT, and minimal overlap. Memory for the two AZTEC vectors is allocated using `new`, and the user is responsible to free this memory, for instance as follows:

```
int * options;
options = MLList.get("smoother: Aztec options", options);
double * params;
params = MLList.get("smoother: Aztec params", params);
.
.
.
// Make sure solve is completed before deleting options & params!!
delete [] options;
delete [] params;
```

The rational behind this is that the parameter list stores a *pointer* to those vectors, not the content itself. (As a general rule, the vectors stored in the parameter list should not be prematurely destroyed or permitted to go out of scope.)

Option Name	Type	SA	DD	DD-ML	maxwell
max levels	int	16	2	3	5
output	int	8	8	8	10
increasing or decreasing	string	increasing	increasing	increasing	decreasing
PDE equations	int	1	1	1	–
null space dimension	int	1	1	1	–
null space vectors	double *	NULL	NULL	NULL	NULL
aggregation: type	string	Uncoupled	METIS	METIS	Uncoupled-MIS
aggregation: type (level 1)	string	–	–	ParMETIS	–
aggregation: type (level 8)	string	MIS	–	–	–
aggregation: local aggregates	int	–	1	–	–
aggregation: nodes per aggregate	int	–	–	512	–
aggregation: damping factor	double	4/3	4/3	4/3	0.0
eigen-analysis: type	string	Anorm	Anorm	Anorm	Anorm
coarse: max size	int	128	128	128	128
aggregation: threshold	double	0.0	0.0	0.0	0.0
aggregation: next-level aggregates per process	int	–	–	128	–
smoother: sweeps	int	2	2	2	2
smoother: damping factor	double	0.67	–	–	0.67
smoother: pre or post	string	both	both	both	both
smoother: type	string	Gauss-Seidel	Aztec	Aztec	–
smoother: Aztec as solver	bool	–	false	false	–
smoother: MLS polynomial order	int	–	–	–	3
smoother: MLS alpha	double	–	–	–	30.0
coarse: type	string	Amesos_KLU	Amesos_KLU	Amesos_KLU	SuperLU
coarse: sweeps	int	1	1	1	1
coarse: damping factor	double	1.0	1.0	1.0	1.0
coarse: max processes	int	16	16	16	–
print unused	int	0	0	0	0

Table 3: Default values for ML_Epetra::MultiLevelPreconditioner for the 4 currently supported problem types `SA`, `DD`, `DD-ML`, `Maxwell`. "–" means not set.

`Uncoupled`	Attempts to construct aggregates of optimal size (3^d nodes in d dimensions). Each process works independently, and aggregates cannot span processes.
`Coupled`	As `Uncoupled`, but aggregates can span processes (deprecated).
`MIS`	Uses a maximal independent set technique to define the aggregates. Aggregates can span processes. May provide better quality aggregates than either `Coupled` or `uncoupled`. Computationally more expensive than either because it requires matrix-matrix product.
`Uncoupled-MIS`	Uses `Uncoupled` for all levels until there is 1 aggregate per processor. Then switches over to `MIS`. The coarsening scheme on a given level cannot be specified with this option.
`METIS`	Use a graph partitioning algorithm to creates the aggregates, working process-wise. The number of nodes in each aggregate is specified with the option `aggregation: nodes per aggregate`. Requires **ML** to be configured with `--with-ml_metis`.
`ParMETIS`	As `METIS`, but partition the global graph. Requires `--with-ml_parmetis2x` or `--with-ml_parmetis3x`. Aggregates can span arbitrary number of processes. Global number of aggregates can be specified with the option `aggregation: global number`.

Table 4: ML_Epetra::MultiLevelPreconditioner: Available coarsening schemes.

6.4 Commonly Used Parameters

Table 4 lists parameter for changing aggregation schemes. Table 5 lists common choices for smoothing options. Table 6 lists common choices affecting the coarse grid solve.
Note that, in the parameters name, spaces are important: Do not include non-required leading or trailing spaces, and separate words by just one space! Mispelled parameters will not be detected. One may find useful to print unused parameters by calling `PrintUnused()` *after* the construction of the multilevel hierarchy.

6.5 List of All Parameters for MultiLevelPreconditioner Class

6.5.1 General Options

`output` — Output level, from 0 to 10 (10 being verbose).

`print unused` — If non-negative, will print all the unused parameter on the specified processor.

`max levels` — Maximum number of levels.

`increasing or decreasing` — If set to `increasing`, level 0 will correspond to the finest level. If set to `decreasing`, `max levels` - 1 will correspond to the finest level.

`Jacobi`	Point-Jacobi. Damping factor is specified using `smoother: dampig factor`, and the number of sweeps with `smoother: sweeps`
`Gauss-Seidel`	Point Gauss-Seidel. Damping factor is specified using `smoother: dampig factor`, and the number of sweeps with `smoother: sweeps`
`Aztec`	Use AztecOO's built-in preconditioning functions as smoothers. Or, if `smoother: Aztec as solver` is `true`, use approximate solutions with AztecOO(with `smoothers: sweeps` iterations as smoothers. The AztecOOvectors `options` and `params` can be set using `smoother: Aztec options` and `smoother: Aztec params`.
`MLS`	Use MLS smoother. The polynomial order is specified by `smoother: MLS polynomial order`, and the alpha value by `smoother: MLS alpha`.

Table 5: ML_Epetra::MultiLevelPreconditioner: Commonly used smoothers.

`Jacobi`	Use `coarse: sweeps` steps of Jacobi (with damping parameter `coarse: damping parameter`) as a solver.
`Gauss-Seidel`	Use `coarse: sweeps` steps of Gauss-Seidel(with damping parameter `coarse: damping parameter`) as a solver.
`Amesos-KLU`	Use KLUthrough Amesos. Coarse grid problem is shipped to proc 0, solved, and solution is broadcast
`Amesos-UMFPACK`	Use UMFPACK through Amesos. Coarse grid problem is shipped to proc 0, solved, and solution is broadcasted.
`Amesos-Superludist`	Use SuperLU_DISTthrough Amesos.
`Amesos-MUMPS`	Use double precision version of MUMPS through Amesos.
`Amesos-ScaLAPACK`	Use double precision version of ScaLAPACK through Amesos.
`SuperLU`	Use **ML** interface to SuperLU.

Table 6: ML_Epetra::MultiLevelPreconditioner: Some of the available coarse matrix solvers. Note: Amesos solvers requires **ML** to be configured with `with-ml_amesos`, and Amesos to be properly configured to support the specified solver.

`PDE equations`	Number of PDE equations for each grid node. This value is not considered for `Epetra_VbrMatrix` objects, as in this case is obtained from the block map used to construct the object. Note that only block maps with constant element size can be considered.
`null space dimension`	Dimension of the null space.
`null space vectors`	Pointer to the null space vectors. If `NULL`, **ML** will use the default null space.

6.5.2 Aggregation Parameters

`aggregation:`	`type` $\star$	Define the aggregation scheme. Can be: `Uncoupled`, `Coupled`, `MIS`, `METIS`, `ParMETIS`. See Table 4.
`aggregation:`	`global aggregates` $\star$	Defines the global number of aggregates (only for `METIS` and `ParMETIS` aggregation schemes).
`aggregation:`	`local aggregates` $\star$	Defines the number of aggregates of the calling processor (only for `METIS` and `ParMETIS` aggregation schemes). Note: this value overwrites `aggregation: global aggregates`.
`aggregation:`	`nodes per aggregate` $\star$	Defines the number of nodes to be assigned to each aggregate (only for `METIS` and `ParMETIS` aggregation schemes). Note: this value overwrites `aggregation: local aggregates`. If none among `aggregation: global aggregates`, `aggregation: local aggregates` and `aggregation: nodes per aggregate` is specified, the default value is 1 aggregate per process.
`aggregation:`	`damping factor`	Damping factor for smoothed aggregation.
`eigen-analysis:`	`type`	Defines the numerical scheme to be used to compute an estimation of the maximum eigenvalue of $D^{-1}A$, where $D = diag(A)$ (for smoothed aggregation only). It can be: `cg` (use 10 steps of conjugate gradient method), `Anorm` (use A-norm of matrix), `Anasazi` (use the ANASAZI package; the problem is supposed to be non-symmetric), or `power-method`.
`aggregation:`	`threshold`	Threshold in aggregation.
`aggregation:`	`next-level aggregates per process` $\star$	Defines the maximum number of next-level matrix rows per process (only for `ParMETIS` aggregation scheme).

6.5.3 Smoothing Parameters

`smoother:`	`sweeps` $\star$	Number of sweeps of smoother.

`smoother: damping factor` $\star$	Smoother damping factor.
`smoother: pre or post` $\star$	If set to `pre`, only pre-smoothing will be used. If set to `post`, only post-smoothing will be used. If set to `both`, pre- and post-smoothing will be used.
`smoother: type` $\star$	Type of the smoother. It can be: `Jacobi`, `Gauss-Seidel`, `sym Gauss-Seidel`, `Aztec`, `IFPACK`. See Table 5.
`smoother: Aztec options` $\star$	Pointer to AZTEC's options vector (only for `aztec` smoother) .
`smoother: Aztec params` $\star$	Pointer to AZTEC's params vector (only for `aztec` smoother) .
`smoother: Aztec as solver` $\star$	If `true`, `smoother: sweeps` iterations of AZTEC solvers will be used as smoothers. If false, only the AZTEC's preconditioner function will be used as smoother (only for `aztec` smoother) .
`smoother: MLS polynomial order` $\star$	Polynomial order for `MLS` smoothers.
`smoother: MLS alpha` $\star$	Alpha value for `MLS` smoothers.

6.5.4 Coarsest Grid Parameters

`coarse: max size`	Maximum dimension of the coarse grid. **ML** will not coarsen further is the size of the current level is less than this value.
`coarse: type`	Coarse solver. It can be: `Jacobi`, `Gauss-Seidel`, `Amesos_KLU`, `Amesos_UMFPACK`, `Amesos_Superludist`, `Amesos_MUMPS`. See Table 6.
`coarse: sweeps`	(only for `Jacobi` and `Gauss-Seidel`) Number of sweeps in the coarse solver.
`coarse: damping factor`	(only for `Jacobi` and `Gauss-Seidel`) Damping factor in the coarse solver.

coarse: max processes	Maximum number of processes to be used in the coarse grid solution (only for Amesos-Superludist, Amesos-MUMPS, Amesos-ScaLAPACK).

7 Advanced Usage of ML

Sections 5 and 6 have detailed the use of **ML** as a black box preconditioner. In some cases, instead, the user may need to explicitly construct the **ML** hierarchy. This is reported in the following sections.

A brief sample program is given in Figure 2. The function ML_Create creates a mul-

```
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.

tilevel 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 11.1. The multigrid hierarchy is generated via ML_Gen_MGHierarchy_UsingAggregation. Controlling the behavior of this function is discussed in Section 9. 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 [18] [17]. 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 8. 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.

8 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 Section 13.

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 AZTECas 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 [15] 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.

9 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.

9.1 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 two different groups of graph coarsening algorithms in **ML**:

- schemes with fixed ratio of coarsening between levels: uncoupled aggregation, coupled aggregation, and MIS aggregation. A description of those three schemes along with some numerical results are given in [16]. As the default, the `Uncoupled-MIS` scheme is used which does uncoupled aggregation on finer grids and switches to the more expensive MIS aggregation on coarser grids;
- schemes with variable ratio of coarsening between levels: METIS and PARMETISaggregation. Those schemes use the graph decomposition algorithms provided by METIS and PARMETIS, to create the aggregates.

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, ML_Aggregate_Set_CoarsenScheme_MIS, ML_Aggregate_Set_CoarsenScheme_METIS, and ML_Aggregate_Set_CoarsenScheme_ParMETIS.

9.2 Interpolation Options

An interpolation operator is built using coarsening information, seed vectors, and a damping factor. We refer the reader to [17] 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 [17]). 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.

10 Advanced Usage of ML and Epetra

Class ML_Epetra::MultiLevelOperator is defined in a header file, that must be included as

```
#include "ml_epetra_operator.h"
```

Users may also need to include `ml_config.h`, `Epetra_Operator.h`, `Epetra_MultiVector.h`, `Epetra_LinearProblem.h`, `AztecOO.h`. Check the EPETRA and AztecOO documentation for more details.

Let `A` be an Epetra_RowMatrix for which we aim to construct a preconditioner, and let `ml_handle` be the structure **ML** requires to store internal data (see Section 7), created with the instruction

```
ML_Create(&ml_handle,N_levels);
```

where `N_levels` is the specified (maximum) number of levels. As already pointed out, **ML** can accept in input very general matrices. Basically, the user has to specify the number of local rows, and provide a function to update the ghost nodes (that is, nodes requires in the matrix-vector product, but assigned to another process). For Epetra matrices, this is done by the following function

```
EpetraMatrix2MLMatrix(ml_handle, 0, &A);
```

and it is important to note that `A` is *not* converted to ML format. Instead, EpetraMatrix2MLMatrix defines a suitable getrow function (and other minor data structures) that allows **ML** to work with `A`.

Let `agg_object` a ML_Aggregate pointer, created using

```
ML_Aggregate_Create(&agg_object);
```

At this point, users have to create the multilevel hierarchy, define the aggregation schemes, the smoothers, the coarse solver, and create the solver. Then, we can finally create the ML_Epetra::MultiLevelOperator object

```
ML_Epetra::MultiLevelOperator MLop(ml_handle,comm,map,map);
```

(`map` being the Epetra_Map used to create the matrix) and set the preconditioning operator of our AztecOO solver,

```
Epetra_LinearProblem Problem(A,&x,&b);
AztecOO Solver(Problem);
solver.SetPrecOperator(&MLop);
```

where `x` and `b` are `Epetra_MultiVector`'s defining solution and right-hand side. The linear problem can now be solved as, for instance,

```
Solver.SetAztecOption( AZ_solver, AZ_gmres );
solver.Iterate(Niters, 1e-12);
```

11 Using ML without Epetra

11.1 Creating a ML 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 Epetra and Aztec users do not need to read this (or the next) section as there are special functions to convert Epetra objects and Aztec matrices to **ML** matrices (see Section 4.2). Further, functions for some common matrix storage formats (CSR & MSR) already exist within ML and do not need to be rewritten[11].

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.

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

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 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:

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

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.

11.2 Creating a ML 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 (2) 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[12]. A specific communication function must 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;
```

[12] 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.

```
   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);
```

12 Visualization Capabilities

ML supports limited capabilities for the visualization of the aggregates, with an interface to OpenDX. Currently, only `Uncoupled`, `METIS` and `ParMETIS` aggregation routines can dump files in OpenDX format.

The procedure to create the OpenDX input files is as follows:

1. Add the following line after the creation of the ML_Aggregate object

   ```
   ML_Aggregate_Viz_Stats_Setup( ag, MaxMgLevels );
   ```

   where `MaxMgLevels` is the maximum number of levels (this is the same value used to create the **ML** object).

2. Create the multilevel hierarchy;

3. Write OpenDX file using the instruction

   ```
   ML_Aggregate_Visualize( ml, ag, MaxMgLevels, x, y, z, option, filename);
   ```

   where `ml` is the **ML** object, `ag` the ML_Aggregation object, and `x,y,z` are double vectors, whose size equals the number of local nodes in the fine grid, containing the coordinates of fine grids nodes. `option` is an integer value defined so that:

- option = 1 : solution of 1D problem (`y` and `z` can be `NULL`);
- option = 2 : solution of 2D problems (`z` can be `NULL`);
- option = 3 : solution of 3D problems.

Processor X will write its own file, `filename_levelY_procX`, where `Y` is the level. `filename` can be set to `NULL` (default value of `.graph` will be used in this case).

Note that, as in smoothed aggregation there is no grid for coarser levels, ML_Aggregate_Visualize needs to assign to each aggregate a set of coordinates. This is done by computing the center of gravity of each aggregates (starting from the fine grid, up to the coarsest level).

4. Deallocate memory using

```
ML_Aggregate_Viz_Stats_Clean( ag, MaxMgLevels )'.
```

At this point, one should copy file `viz_aggre.net` and `viz_aggre.cfg` (located in `$ML_HOME/util/`) in the directory where the output files are located, and run OpendDX with the instruction

```
% dx -edit viz_aggre.net
```

Other instructions are reported in file `$ML_HOME/util/viz_aggre.README`. An example of code can be found in file `$ML_HOME/examples/ml_aztec_simple_METIS.c`.

13 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 AZTECmatrix 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 AZTECdata structure representing a matrix. See the AZTECUser's Guide.
proc_config	On input, an AZTECdata structure representing processor information. See the AZTECUser'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 AZTECpreconditioner. Thus, when Precond and options are passed into the AZTECiterative solver, it will invoke the V cycle multigrid algorithm described by ml_object.

Parameters

Precond	On input, an AZTECdata structure representing a preconditioner. On output, the multigrid V cycle method described by ml_object will be associated with this preconditioner. See the AZTECUser's Guide.
Amat	On input, an AZTECdata structure representing a matrix. See the AZTECUser's Guide.
ml_object	On input, **ML** object pointer (see ML_Create) representing a V cycle multigrid method.
options	On input, an AZTECdata 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 9).

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 9).

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 9).

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_CoarsenScheme_METIS(ML_Aggregate *agg_object)
```

Description

Set the aggregate coarsening scheme to be used as 'METIS (see Section 9).

Parameters

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

Prototype

```
int ML_Aggregate_Set_CoarsenScheme_ParMETIS(ML_Aggregate *agg_object)
```

Description

Set the aggregate coarsening scheme to be used as 'ParMETIS (see Section 9).

Parameters

agg_object On input, aggregate object pointer (see ML_Aggregate_Create). On output, the 'ParMETIS' 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 8).

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 $|A(i,j)| \leq tol\_d * \sqrt{|A(i,i)A(j,j)|}$. See Section 9 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

```
int ML_Gen_SmootherAmesos(ML *ml_object, int k, int AmesosSolver,
                          int MaxProcs)
```

Description

Use Amesos interface to direct solvers 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 Amesos.
k	On input, indicates level within ml_object hierarchy (must be the coarsest level in the multigrid hierarchy).
AmesosSolver	On input, indicates the direct solver library to use in the coarse solution. It can be: ML_AMESOS_UMFPACK, ML_AMESOS_KLU, ML_AMESOS_SUPERLUDIST, ML_AMESOS_MUMPS, ML_AMESOS_SCALAPACK.
MaxProcs	On input, indicates maximum number of processors to use in the coarse solution (only for ML_AMESOS_SUPERLUDIST).

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 AZTECscheme is given by the AZTECarrays: options, params, proc_config, and status and AZTECpreconditioning 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, AZTECarrays that determine the AZTECscheme and are used for AZTECto return information. See the AZTECUser's Guide.
N_iterations	On input, maximum AZTECiterations 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, AZTECpreconditioning function indicating what preconditioner will be used within AZTEC. Normally, this is set to AZ_precondition. See the AZTECUser'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 11.1.
comm	On input, a function pointer to the user-defined communication function. See Section 11.2.

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 11.1

.

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 8.
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] T. Davis, *UMFPACK home page.* http://www.cise.ufl.edu/research/sparse/umfpack, 2003.

[3] J. W. Demmel, J. R. Gilbert, and X. S. Li, *SuperLU Users' Guide*, 2003.

[4] Free Software Foundation, *Autoconf Home Page.* http://www.gnu.org/software/autoconf.

[5] ——, *Automake Home Page.* http://www.gnu.org/software/automake.

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

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

[8] M. A. Heroux, *Trilinos home page.* http://software.sandia.gov/trilinos.

[9] M. A. Heroux, *IFPACK Reference Manual*, 2.0 ed., 2003. http://software.sandia.gov/trilinos/packages/ifpack/doxygen/latex/IfpackReferenceManual.pdf.

[10] M. M. A. Heroux, *Trilinos Tutorial*, 3.1 ed., 2004.

[11] G. Karypis and V. Kumar, *ParMETIS: Parallel graph partitioning and sparse matrix ordering li brary*, Tech. Rep. 97-060, Department of Computer Science, University of Minnesota, 1997.

[12] ——, *METIS: Unstructured graph partitining and sparse matrix ordering sy stem*, tech. rep., University of Minnesota, Department of Computer Science, 1998.

[13] T. G. Kolda and R. P. Pawlowski, *Nox home page.* http://software.sandia.gov/nox.

[14] I. S. D. P. R. Amestoy and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods in Appl. Mech. Eng., (2000), pp. 501–520.

[15] 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.

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

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

[18] 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.