ML 5.0 Smoothed Aggregation User's Guide

Michael W. Gee   and   Christopher M. Siefert
Computational Math & Algorithms
Sandia National Laboratories
Mailstop 1320
P.O. Box 5800
Albuquerque, NM 87185-1320

Jonathan J. Hu   and   Ray S. Tuminaro
Computational Math & Algorithms
Sandia National Laboratories
Mailstop 9159
P.O. Box 0969
Livermore, CA 94551-0969

Marzio G. Sala
ETH Zürich Computational Laboratory
CAB F84
ETH Zentrum
8092 Zürich

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 packages [20]. 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

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. A quick start is reported in Section 4. The process of configuring and building **ML** is outlined in Section 5. Section 6 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 6.4. Section 7 reports how to use the Maxwell solvers of **ML**. More advanced uses of **ML** are presented in Section 8. Here, we present how to define and fine-tune smoothers, coarse grid solver, and the multilevel hierarchy. Multigrid options are reported in Section 9. Smoothing options are reported in Section 10, where we also present how to construct a user's defined smoother. Advanced usage of **ML** with EPETRA objects is reported in Section 11. Section 12 reports how to define matrices in **ML** format without depending on EPETRA.

3 Multigrid Background

A brief multigrid description is given (see [4], [10], or [11] 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 ( $k \neq$ **Nlevel** $- 1$)
                $P_k =$ determine_interpolant( $A_k$ );
                $\hat{r} = P_k^T(b - A_k u)$ ;
                $\hat{A}_{k+1} = P_k^T A_k P_k$;    v = 0;
                multilevel($\hat{A}_{k+1}, \hat{r}, v, k + 1$);
                $u = u + P_k$ 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 9. 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 9).

4   Quick Start

This section is intended for the impatient user. It's assumed that you've already have a local copy of TRILINOS[3]. Using the instructions here, your build of TRILINOS will have the following libraries: AZTECOO, EPETRA, EPETRAEXT, IFPACK, LOCA, **ML**, NEW_PACKAGE, NOX, AMESOS and TEUCHOS.

1. `cd` into the TRILINOS directory.

2. Make a build directory, e.g., `mkdir sandbox`.

3. `cd sandbox`.

4. Configure TRILINOS:

    (a) `../configure --enable-teuchos --enable-amesos --enable-aztecoo --enable-epetra`
        if you do not want to use MPI.

    (b) `../configure --enable-teuchos`
        `--enable-amesos --enable-aztecoo --enable-epetra --with-mpi-compilers=/usr/local`
        (or wherever your MPI compilers are located) to use MPI.

---

[2]The P_k's are usually determined as a preprocessing step and not computed within the iteration.
[3]Please refer to the Trilinos web page, trilinos.sandia.gov, to obtain a copy of TRILINOS.

5. Build TRILINOS: `make`.[4]

6. If your build finished without errors, you should see the directory
   `Trilinos/sandbox/packages/`, with subdirectories below that for each individual
   library. **ML**'s subdirectory, `ml`, should contain files `config.log`, `config.status`,
   `Makefile`, and `Makefile.export`, and directories `src` and `examples`. Directory `src`
   contains object files and `libml.a`. Directory `examples` contains executables with exten-
   sion `.exe`, symbolic links to the corresponding source code, and object files. Directory
   `test` is intended primarily for developers and can be ignored.

7. Look in `Trilinos/sandbox/packages/ml/examples` for examples of how to use **ML**.
   File `Trilinos/packages/ml/examples/README` suggests how to use the examples.

5 Configuring and Building ML

ML is configured and built using the GNU autoconf [8] and automake [9] tools. It can
be configured and build as a standalone package without or with AZTEC 2.1 support (as
detailed in Section 5.1 and 5.2), or as a part of the TRILINOS framework [12] (as described
in Section 5.3). 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, Intel, Portland Group
MAC OS X	GNU
IRIX N32, IRIX 64, HPUX, Solaris, DEC	native
ASC Red Storm	native and Portland Group
Windows	Microsoft and Cygwin

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

   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.

5.1 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
```

---

[4]If you are using GNU's make on a machine with more than one processor, then you can speed up the build with `make -j`
`XX`, where `XX` is the number of processors. You can also reduce the size of the link lines with the option `--with-gnumake`.

```
% $ML_HOME/configure --disable-epetra --disable-aztecoo \
    --prefix=$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`.

5.2 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 12.1 and Section 12.2). 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 9. 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.

5.3 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` (here introduced for the sake of simplicity only) identifies the TRILINOS directory, and, for example, that we are compiling under LINUX and MPI.

```
% cd $TRILINOS_HOME
% mkdir LINUX_MPI
% cd LINUX_MPI
% $TRILINOS_HOME/configure \
    --enable-teuchos \
    --enable-amesos \
    --with-mpi-compilers \
    --prefix=$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 5.3.1 and 5.3.2. You can also find a complete list and explanations by typing `./configure --help` in the **ML** home directory.

5.3.1 Enabling Third Party Library Support

ML can be configured with the following third party libraries (TPLs): SUPERLU, SUPERLU_DIST, PARASAILS, ZOLTAN, METIS, and PARMETIS. It can take advantage of the following TRILINOS packages: IFPACK, TEUCHOS, TRIUTILS, AMESOS, EPETRAEXT. Through AMESOS, **ML** can interface with the direct solvers KLU, UMFPACK, SUPERLU, SUPERLU_DIST[5], 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**.

The same configure options that enable certain other Trilinos packages also enable the interfaces to those packages within **ML**.

`--enable-epetra`	Enable support for the EPETRA package. (Recommended)
`--enable-epetraext`	Enable support for the EPETRAEXT package. (Recommended)
`--enable-aztecoo`	Enable support for the AZTECOO package. (Recommended)
`--enable-amesos`	Enables support for the AMESOS package. AMESOS is an interface with several direct solvers. **ML** supports UMFPACK [6], KLU, SUPERLU_DIST (1.0 and 2.0), MUMPS [1]. This package is necessary to use the AMESOS interface to direct solvers. (Recommended)
`--enable-teuchos`	Enables support for the TEUCHOS package. This package is necessary to use the ML_Epetra::MultiLevelPreconditioner class. (Recommended)
`--enable-triutils`	Enables support for the TRIUTILS package. **ML** uses TRIUTILS in some examples and tests to create the linear system matrix.
`--enable-galeri`	Enables support for the GALERI package. **ML** uses GALERI in some examples and tests to create the linear system matrix.
`--enable-ifpack`	Enable support for the IFPACK package [18], to use IFPACK factorizations as smoothers for **ML**.

---

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

To know whether the `--enable-`*package* options below reported are enabled or disabled by default[6], please consult the configure at the Trilinos level by typing

```
$TRILINOS_HOME/configure --help
```

The following configure line options enable interfaces in **ML** to certain TPLs. By default, all the `--with-ml_`*TPL* options are disabled. For the most up-to-date list, please type `configure --help` at the **ML** package level.

`--with-ml_parasails`	Enables interface for PARASAILS [5].
`--with-ml_metis`	Enables interface for METIS [16].
`--with-ml_parmetis2x`	Enables interface for PARMETIS, version 2.*x*.
`--with-ml_parmetis3x`	Enables interface for PARMETIS [15], version 3.*x*.
`--with-ml_superlu`	Enables **ML**'s direct interface for serial SUPERLU [7] 1.0. This interface is deprecated in favor of the AMESOS interface. Note: you cannot configure **ML** with both this option and `--enable-amesos`.
`--with-ml_superlu2`	Enables **ML**'s direct interface for serial SUPERLU [7] 2.0. This interface is deprecated in favor of the AMESOS interface. Note: you cannot configure **ML** with both this option and `--enable-amesos`.
`--with-ml_superlu_dist`	Enables **ML** interface for SUPERLU_DIST [7]. This interface is deprecated in favor of the AMESOS interface. Note: you cannot configure **ML** with both this option and `--enable-amesos`.

If one of the above options is enabled, then 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 , with the option

```
--with-ldflags="<lib-locations>"
--with-libs="<lib-list>"
```

The user might find useful the option

```
--disable-examples
```

---

[6]This refers to configurations from the Trilinos top level (that is, using `$TRILINOS_HOME/configure`). If **ML** if configured from the **ML** level (that is, the user directly calls `$ML_HOME/configure`), then all the `--enable-`*package* are off by default.

which turns off compilation and linking of all Trilinos examples, or

```
--disable-ml-examples
```

which turns off compilation and linking of **ML** examples.

Here is an example configure line, where METIS, PARMETIS, ZOLTAN, and SUPERLU are all used:
```
./configure --with-mpi-compilers="/usr/local/mpich/bin" CXXFLAGS="-O3" CFLAGS="-O3"
FFLAGS="-O3" --cache-file=config.cache --with-gnumake --with-ml_metis --with-ml_zoltan
--with-ml_parmetis3x --with-ldflags="-L/usr/local/superlu-3.0/lib -L/usr/local/parmetis
-L/usr/local/zoltan/lib" --with-libs="-lsuperlu -lparmetis -lmetis -lzoltan"
--with-incdirs="-I/usr/local/superlu-3.0/SRC -I/usr/local/parmetis-3.1 -I/usr/local/par
-I/usr/local/parmetis-3.1/ParMETISLib -I/usr/local/zoltan/include"
```

More details about the installation of TRILINOS can be found at the TRILINOS web site, trilinos.sandia.gov, and [19, Chapter 1].

5.3.2 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.

5.3.3 Linking to ML

The most typical usage of **ML** is as a solver or preconditioner within an application. **ML** provides an easy facility for ensuring that the proper header and library files are available for compilation and linking. The **ML** build directory contains the file `Makefile.export.ml`. (If Trilinos has been installed, this file will be in the installation directory.) `Makefile.export.ml` should be included in application Makefiles, `Makefile.export.ml` defines two macros, `ML_LIBS` and `ML_INCLUDES`, that contain all of **ML**'s library and include dependencies, respectively. Additionally, the macro `HAVE_CONFIG_H` **must** be defined. Here is an example of how to build an sample code that used **ML** has a third-party library:

```
include ${ROOT_OF_TRILINOS_BUILD}/packages/ml/Makefile.export.ml

flowCode.exe: flowCode.o
    g++ -o flowCode.exe flowCode.o ${ML_LIBS}

flowCode.o: flowCode.cpp
    g++ -c -DHAVE_CONFIG_H ${ML_INCLUDES} flowCode.c
```

6 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. This class is derived from the Epetra_RowMatrix class, and is defined in the ML_Epetra namespace.

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[7] and a TEUCHOS parameter list[8].

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_MultiLevelPreconditioner.h"`. Also note that the macro `HAVE_CONFIG_H` must be defined either in the user's code or as a compiler flag.

6.1 Example 1: ml_preconditioner.cpp

We now give a very simple fragment of code that uses the MultiLevelPreconditioner. For the complete code, see `$ML_HOME/examples/BasicExamples/ml_preconditioner.cpp`. 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_MultiLevelPreconditioner.h"
#include "Teuchos_ParameterList.hpp"

...

  Teuchos::ParameterList MLList;

  // 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");

  // create the preconditioner
```

---

[7]Note that not all Epetra matrices can be used with ML. Clearly, the input matrix must be a square matrix. Besides, it is supposed that the `OperatorDomainMap()`, the `OperatorRangeMap()` and the `RowMatrixRowMap()` of the matrix all coincide, and that each row is assigned to exactly one process.

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

```
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;
```

A MultiLevelPreconditioner may be defined as follows. First, the user defines a TEU-
CHOS parameter list[9]. 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 contain-
ing the parameter name, and Value is the specified parameter value, whose type can be
any C++ object or pointer. A complete list of parameters available for class MultiLevel-
Preconditioner is reported in Section 6.4. Automatic selection of defaults is discussed in
Section 6.5. In particular, the ML_Epetra::SetDefaults command sets multigrid defaults
for specific problem or solution types (e.g. classic smoothed aggregation for positive-definite
systems or a variant more suitable for highly nonsymmetric systems.).

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->DestroyPreconditioner()[10]. 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
...
```

---

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

[10]We suggest to always create the preconditioning object with new and to delete it using delete. Some MPI calls occur in
DestroyPreconditioner(), so the user should not call MPI_Finalize() or delete the communicator used by **ML** before the
preconditioning object is destroyed.

```
// now compute the preconditioner
MLPrec->ComputePreconditioner();

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

// re-compute the elements of A
// now re-compute the preconditioner, using either
MLPrec->ComputePreconditioner();
// or
MLPrec->ReComputePreconditioner();

// re-solve the linear system

// .. finally destroy the object
delete MLPrec;
```

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 [17]). Method `ReComputePreconditioner()` can be used to recompute the preconditioner using already available information about the aggregates. `ReComputePreconditioner()` reuses the already computed tentative prolongator, then recomputes the smoothed prolongators and the other components of the hierarchy, like smoothers and coarse solver[11].

6.2 Example 2: ml_2level_DD.cpp

In the second example, a two level domain decomposition preconditioner is constructed. The coarse space is defined using aggregation. It is worthwhile to compare the parameters selected here to the default parameters stated in Table 6.

File `$ML_HOME/examples/TwoLevelDD/ml_2level_DD.cpp` reports the entire code. In the example, the linear system matrix `A`, is an Epetra_CrsMatrix corresponding to the discretization of a Laplacian on a 2D Cartesian grid. The solution vector and right-hand side are `x` and `b` 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);
```

---

[11]Note that the hierarchy produced by `ReComputePreconditioner()` can differ from that produced by `ComputePreconditioner()` for non-zero threshold values.

```
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:

```
Teuchos::RCP<vector<int>> options = rcp(new vector<int>(AZ_OPTIONS_SIZE));
Teuchos::RCP<vector<double>> params = rcp(new vector<double>(AZ_PARAMS_SIZE));
AZ_defaults(options,params);
AZ_defaults(&(*options)[0],&(*params)[0]);
(*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 TEUCHOS reference-counted pointers, `options` and `params`,
in the parameter list.

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 misspelled, 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 `MLPrec`'s internally stored parameter list, he or she can get a reference to the internally
stored list:

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

and then directly modify `List`. Method `GetList()` should be used carefully, as a change
to `List` may modify the behavior of `MLPrec`.

`Uncoupled`	Attempts to construct aggregates of optimal size (3^d nodes in d dimensions). Each process works independently, and aggregates cannot span processes.
`MIS`	Uses maximal independent set technique [21] to define aggregates. Aggregates can span processes. May provide better quality aggregates than `uncoupled`, but computationally more expensive 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`.
`METIS`	Use graph partitioning algorithm to create aggregates, working process-wise. Number of nodes in each aggregate is specified with option `aggregation: nodes per aggregate`. Requires `--with-ml_metis`.
`ParMETIS`	As `METIS`, but partition global graph. Requires `--with-ml_parmetis2x` or `--with-ml_parmetis3x`. Aggregates can span arbitrary number of processes. Specify global number of aggregates with `aggregation: global number`.
`VBMETIS`	As `METIS`, but for matrices with nonconstant nodal block size. The appearing maximum number of degrees of freedom per nodal block is specified in `PDE equations`. Additional information on layout of variable nodal blocks is required via `aggregation: nblocks`, `aggregation: length blocks`, `aggregation: blocks`, and `aggregation: block_pde`. User-specified null space must be provided using `null space: vectors`. Number of nodes in each aggregate is specified with `aggregation: nodes per aggregate`. Requires `--with-ml_metis`.

Table 3: ML_Epetra::MultiLevelPreconditioner: Available coarsening schemes.

6.3 Tips for Achieving Good Parallel Performance

This section gives a few tips on tuning **ML**'s parallel performance.

6.3.1 Hints for the Impatient User

1. Use the matrix repartitioning option. This is discussed in detail in the next section, §6.3.2.

2. In order to reduce message-passing latency, try limiting the number of multigrid levels to 3 or 4. The syntax is described in §6.4.4.

3. Instead of doing a direct solve on the coarsest level, try a few smoothing sweeps instead. See §6.4.7.

4. For a symmetric positive definite linear system, choose a smoother whose computational kernel is a matvec, such as the Chebyshev polynomial smoother. See §6.4.6.

6.3.2 Dynamic Load-balancing

ML supports dynamic load-balancing of coarse-level matrices in the multigrid preconditioner. This is extremely important for two reasons. First, it is not unusual for the message-passing latency on a coarse level to be roughly equivalent to the latency of the fine level,

`Jacobi`	Point-Jacobi. Specify damping factor with `smoother: damping factor`.
`Gauss-Seidel`	Point Gauss-Seidel. Specify damping factor with `smoother: damping factor`.
`symmetric Gauss-Seidel`	Point symmetric Gauss-Seidel. Specify damping factor with `smoother: damping factor`.
`Chebyshev`	Use Chebyshev smoother. Specify polynomial order by `smoother: sweeps`. Smoother will damp errors between ρ and ρ/α, where ρ is spectral radius of $diag(A)^{-1}A$ and α is specified via `smoother: Chebyshev alpha`.
`IC, ILU, ICT, ILUT`	Local (processor-based) incomplete factorization methods from IFPACK. See Ifpack manual [18] for a complete description of these methods. See §6.4.6 for associated **ML** parameter list options.
`Aztec`	Use AZTECOO's built-in preconditioners as smoothers. Or, if `smoother: Aztec as solver` is `true`, use an AZTECOO Krylov method as smoother. AZTECOO vectors `options` and `params` can be set using `smoother: Aztec options` and `smoother: Aztec params`. See §6.2 for an example.
`user-defined`	User-defined smoothing function with prototype `int (*)(ML_Smoother *, int inLength, double *invec, int outLength, double *outvec)`. See **ML** example `ml_user_smoothing.cpp`.
`petsc`	PETSc smoothing function. A fully-constructed PETSc KSP pointer must be passed as a pointer to `void` via the parameter `smoother: petsc ksp`. Please see the EPETRAEXT homepage for more information.

Table 4: ML_Epetra::MultiLevelPreconditioner: Commonly used smoothers. Unless otherwise noted, number of sweeps is specified with `smoother: sweeps`. In `Gauss-Seidel` and `symmetric Gauss-Seidel`, IFPACK implementation is automatically used (if available) for matrices of type Epetra_CrsMatrix.

even though the number of matrix unknowns on the coarse level is much smaller than on the fine level. Second, as the number of unknowns per processor becomes small, **ML**'s aggregation methods become less effective. The coarsening rate may drop, thus leading to more multigrid levels than one might observe in serial.

We now give two code fragments to demonstrate how to use the repartitioning options. The complete description of options that control repartitioning is given in §6.4.8. The following code demonstrates repartitioning for a matrix with unknowns that are node-based.

```
Teuchos::ParameterList MLList;
MLList.set("repartition: enable",1);
MLList.set("repartition: max min ratio",1.3);
MLList.set("repartition: min per proc",500);
MLList.set("repartition: partitioner","Zoltan");
MLList.set("repartition: Zoltan dimensions",2);

// Zoltan requires coordinates
double *xcoord, *ycoord, *zcoord;
```

Jacobi	Point Jacobi. Specify damping with `coarse: damping factor`.
Gauss-Seidel	Point Gauss-Seidel. Specify damping with `coarse: damping factor`.
symmetric Gauss-Seidel	Symmetric point Gauss-Seidel. Specify damping with `coarse: damping factor`.
Chebyshev	Use degree `coarse: sweeps` Chebyshev polynomial as a solver.
Hiptmair	Two-stage Hiptmair smoother. Take `coarse: node sweeps` (or `coarse: edge sweeps`) iterations with method `coarse: subsmoother type` to solve the nodal (or edge) subproblem.
Amesos-KLU	Use KLU through AMESOS. Coarse grid problem is shipped to proc 0, solved, and solution is broadcast
Amesos-Superlu	Use SUPERLU, version 3.0, through AMESOS.
Amesos-UMFPACK	Use UMFPACK through AMESOS. Coarse grid problem is shipped to proc 0, solved, and solution is broadcasted.
Amesos-Superludist	Use SUPERLU_DIST through AMESOS.
Amesos-MUMPS	Use double precision version of MUMPS through AMESOS.
user-defined	User-defined smoothing function with prototype `int (*)(ML_Smoother *, int inLength, double *invec, int outLength, double *outvec)`. See **ML** example `ml_user_smoothing.cpp`.
SuperLU	Use **ML** interface to SUPERLU(deprecated).

Table 5: ML_Epetra::MultiLevelPreconditioner: Commonly used coarse matrix solvers. Unless otherwise noted, number of sweeps is specified with `coarse: sweeps`. To use Amesos solvers, **ML** must be configured with `with-ml_amesos`, and Amesos must be properly configured. In `Gauss-Seidel` and `symmetric Gauss-Seidel`, IFPACK implementation is automatically used (if available) for matrices of type Epetra_CrsMatrix.

```
// user must load the coordinates, of coarse ...

MLList.set("x-coordinates", xcoord);
MLList.set("y-coordinates", xcoord);
```

The following code demonstrates repartitioning for a system arising from the eddy current approximations to Maxwell's equations.

```
Teuchos::ParameterList MLList;
// these correspond to the main edge matrix
MLList.set("repartition: enable",1);
MLList.set("repartition: max min ratio",1.3);
MLList.set("repartition: min per proc",1000);
MLList.set("repartition: partitioner","Zoltan");

// Zoltan requires coordinates
double *xcoord, *ycoord, *zcoord;

// user must load the coordinates, of coarse ...

MLList.set("x-coordinates", xcoord);
MLList.set("y-coordinates", ycoord);
```

```
    MLList.set("z-coordinates", zcoord);

    // these correspond to the auxiliary node matrix
    MLList.set("repartition: node max min ratio",1.5);
    MLList.set("repartition: node min per proc",300);
    MLList.set("repartition: node partitioner","Zoltan");
    MLList.set("repartition: Zoltan dimensions",3);

    double *node_xcoord, *node_ycoord, *node_zcoord;

    // user must load the coordinates, of coarse ...

    MLList.set("x-coordinates", node_xcoord);
    MLList.set("y-coordinates", node_ycoord);
    MLList.set("z-coordinates", node_zcoord);
```

6.4   List of All Parameters for MultiLevelPreconditioner Class

In this section we give general guidelines for using the MultiLevelPreconditioner class effectively. The complete list of input parameters is also reported. 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 requires `--with-ml_metis --enable-epetra` and `--with-ml_parmetis2x` or `--with-ml_parmetis3x`.

- AMESOS coarse solvers require `--enable-amesos --enable-epetra --enable-teuchos`. Moreover, AMESOS must have been configure to support the requested coarse solver. Please refer to the AMESOS documentation for more details.

- IFPACK smoothers require `--enable-ifpack --enable-epetra --enable-teuchos`.

- PARASAILS smoother requires `--with-ml_parasails`.

- PETSc smoothers require `--with-petsc` and `--enable-epetraext`.

6.4.1   Using Parameters on Individual Levels

Some of the parameters that affect the MultiLevelPreconditioner class 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. There are two ways to change settings for a particular level.

The first method for changing a setting on a particular level, such as level `d`, is to append the string "(level d)" 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 there are 5 levels, **ML** uses `Uncoupled` for levels 0, 2 and 4, `METIS` for level 1 and `MIS` for level 3. Parameters that can be set differently on individual levels are denoted with the symbol $\star$ (this symbol is not part of the parameter name).

The second method for changing a setting on a particular level is by using sublists. Smoother sublists name must be of the form "smoother: list (level XX)", and aggregation sublist names must be of the form "aggregation: list (level XX)", where XX is the level number (0 is the finest). Parameter names in the sublist do not then need a level number. Similarly, options for the coarsest level can be specified via the sublist "coarse: list". Below is a short code fragment to illustrate this usage.

```
ParameterList MLList;
ParameterList &smList = MLList.sublist("smoother: list (level 1)");
smList.set("smoother: type","Jacobi");
smList.set("smoother: sweeps",5);

ParameterList &coarseList = MLList.sublist("coarse: list");
coarseList.set("coarse: type","symmetric Gauss-Seidel");
```

6.4.2 Parameter Validation

By default, the `MultiLevelPreconditioner` class validates the input parameter list. A parameter that is misspelled/unknown or has an incorrect value type will cause an exception to be thrown and execution to halt. **Note that spaces are important within a parameter's name. Do not include non-required leading or trailing spaces. Please separate words by just one space!** The option `ML print initial list` prints the initial list given to the `MultiLevelPreconditioner` constructor. The option `ML print final list` prints the final list actually used by the `MultiLevelPreconditioner` constructor. One may find it useful to print unused parameters by calling `PrintUnused()` *after* the construction of the multilevel hierarchy.

6.4.3 General Options

`ML output`	[`int`] Controls the amount of printed information. Ranges from 0 to 10 (0 is no output, and 10 is incredibly detailed output). Default: 0.
`print unused`	[`int`] If non-negative, will print all unused parameters on the specified processor. If -1, unused parameters will be printed on all processes. If -2, nothing will be printed. Default: -2.

`ML print initial list`	[`int`] If non-negative, will print all parameters in the parameter list supplied to the constructor on the specified processor. If -1, parameters will be printed on all processes. If -2, nothing will be printed. Default: -2.
`ML print final list`	[`int`] If non-negative, will print all parameters in the final list used by the constructor on the specified processor. (The constructor will add entries to the list.) If -1, parameters will be printed on all processes. If -2, nothing will be printed. Default: -2.
`PDE equations`	[`int`] Number of PDE equations at each grid node. Not used for `Epetra_VbrMatrix` objects as this is obtained from the block map used to construct the object. While only block maps with constant element size can be considered, **ML** has some ability to work with a nonconstant number of equations per grid node (see `ml_example_elasticity.c`). Default: 1.
`eigen-analysis: type`	[`string`] Defines the numerical scheme to use when computing spectral radius estimates for $D^{-1}A$, where $D = diag(A)$ and A refers to a discretization matrix on a particular level. Spectral radius estimates are needed to generate the smoothed aggregation prolongator and for some smoothers. Choices are: `cg` (conjugate gradient method), `Anorm` (the A-norm of the matrix), `power-method`. Default: `cg`.
`eigen-analysis: iterations`	[`int`] The number of iterations to perform in estimating the spectral radius. Default: 10.

6.4.4 Multigrid Cycle Options

`cycle applications`	[`int`] Number of multigrid applications within a preconditioner Default: 1.
`max levels`	[`int`] Maximum number of levels. Default: 10.

increasing or decreasing	[`string`] 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. Default: `increasing`.
prec type	[`string`] Multigrid cycle type. Possible values: `MGV`, `MGW`, `full-MGV`, `one-level-postsmoothing`, `two-level-additive`, `two-level-hybrid`, `two-level-hybrid2`, `projected MGV`. Default: `MGV`.
projected modes	[`double**`] Array of double vectors containing modes that should be projected out before and after the AMG V-cycle is applied. For use with the `projected MGV` option. Default: `NULL`.
number of projected modes	[`int`] Number of modes that are to be projected out before and after the AMG V-cycle is applied. For use with the `projected MGV` option. Possible values: 1,2, or 3. Default: 0.

6.4.5 Aggregation and Prolongator Parameters

aggregation: type ⋆	[`string`] Define the aggregation scheme. See Table 3. Default: `Uncoupled`.
aggregation: threshold	[`double`] Dropping threshold in aggregation. Default: 0.0.
aggregation: damping factor	[`double`] Damping factor for smoothed aggregation. Default: 4/3.
aggregation: smoothing sweeps	[`int`] Number of smoothing sweeps to use in the prolongator smoother. This should only be changed when using larger aggregates from either `METIS` or `ParMETIS` aggregation schemes. In general, one wants the number of sweeps to be less than or equal to $log_2(\text{AggregateDiameter} - 1)$. If sweeps equals 1, then `aggregation: damping factor` is used. If sweeps is greater than 1, damping factors are calculated automatically. Default: 1.

24

`aggregation: global aggregates` ⋆	[int] Defines the global (or total) number of aggregates that should be created (only for `METIS` and `ParMETIS` aggregation schemes).
`aggregation: local aggregates` ⋆	[int] 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` ⋆	[int] 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: next-level aggregates per process` ⋆	[int] Defines the maximum number of next-level matrix rows per process (only for `ParMETIS` aggregation scheme). Default: 128.
`aggregation: nblocks`	[int] Defines the number of processor nodal blocks in the matrix (only for `VBMETIS` aggregation scheme). Nodal blocks must not span processors. Default: 0.
`aggregation: length blocks`	[int] Length of vectors specified with `aggregation: blocks` and `aggregation: block_pde` (only for `VBMETIS` aggregation scheme). Usually, this correponds to the processor local number of scalar rows of the problem. Default: 0.
`aggregation: blocks`	[int] Vector of length `aggregation: length blocks`. Contains processor global block numbering starting from `0`. `blocks[i]` is the nodal block the scalar equation `i` of the matrix belongs to, where `i` is a processor local scalar row index of the matrix (only for `VBMETIS` aggregation scheme). The user is responsible for freeing the vector. Default: NULL.

aggregation: block_pde	[int] Vector of length aggregation: length blocks. block_pde[i] is the number of the pde equation inside one nodal block, where numbering starts from 0. Note that a user specified null space matching the supplied pde distribution in this vector should be supplied in null space: vectors (only for VBMETIS aggregation scheme). The user is responsible for freeing the vector. Default: NULL.
aggregation: use tentative restriction	[bool] If true, then the final restriction operator at level ℓ is computed as $R_\ell = P_{\ell,0}^T$, where $P_{\ell,0}$ is the tentative prolongator. Note that the final prolongator is still computed as specified by parameter aggregation: damping factor. Default: false.
aggregation: symmetrize	[bool] If true, then the final prolongator for level ℓ is computed as $P_\ell = (I - \omega/\lambda_{max}(A + A^T))P_{0,\ell}$, where ω is the damping parameter specified using aggregation: damping factor, λ_{max} is the maximum eigenvalue of $D^{-1}A$, and $P_{0,\ell}$ is the tentative prolongator at level ℓ. The restriction is defined as $R_\ell = P_\ell^T$. Default: false.
energy minimization: enable	[bool] If true, then use multigrid variant (based on local damping parameters and restrictor smoothing) suitable for highly nonsymmetric problems. Default: false.
energy minimization: type	[int] Choose the norm used within local minimization of the energy minimizing option. 2 is the more reliable. 1 is the cheapest to set up. 3 is the closest to classic smoothed aggregation. Default: 2.
energy minimization: droptol	[double] Drop entries in intermediate calculations of the energy minimizing prolongator that have absolue value less than this tolerance Default: 0..
energy minimization: cheap	[bool] If true and energy minimization: enable has been set, a version of energy minimization which is less reliable but cheaper to set up is used. Default: false.

6.4.6 Smoothing Parameters

`smoother: type ⋆`	[`string`] Type of the smoother. See Table 4. Default: Chebyshev.
`smoother: sweeps ⋆`	[`int`] Number of smoother sweeps (or polynomial degree if using Chebyshev). Default: 2
`smoother: damping factor ⋆`	[`double`] Smoother damping factor for `Jacobi`, `Gauss-Seidel`, `symmetric Gauss-Seidel`, `block Gauss-Seidel`, or `symmetric block Gauss-Seidel`. Default: 1.0.
`smoother: pre or post ⋆`	[`string`] It can assume one of the following values: `pre, post, both`. Default: `both`.
`smoother: Aztec options ⋆`	[`RCP<vector<int>>`] Teuchos reference-counted pointer to Aztec's options vector (only for `Aztec` smoother) .
`smoother: Aztec params ⋆`	[`RCP<vector<double>>`] Teuchos reference-counted pointer to Aztec's params vector (only for `Aztec` smoother) .
`smoother: Aztec as solver ⋆`	[`bool`] 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). Default: `false`.
`smoother: ifpack level-of-fill ⋆`	[`double`] Specifies fill-level for Ifpack incomplete factorization methods. **ML** will internally interpret this as an integer where appropriate. Default: 0.0.
`smoother: ifpack overlap ⋆`	[`int`] Specifies subdomain overlap if Ifpack smoothers are used. Default: 1.
`smoother: ifpack relative threshold ⋆`	[`double`] Specifies relative threshold for perturbing matrix diagonal. See [18] for details. Default: 1.0.
`smoother: ifpack absolute threshold ⋆`	[`double`] Specifies absolute threshold for perturbing matrix diagonal. See [18] for details. Default: 0.0.

smoother: Chebyshev alpha ⋆	[double] Chebyshev smoother will damps errors between ρ and ρ/alpha where ρ is the spectral radius of $diag(A)^{-1}A$. Default: 20.
smoother: user-defined function	[int (*)(ML_Smoother *, int inLength, double *invec, int outLength, double *outvec)] Pointer to user-defined smoothing function. Default: NULL.
smoother: user-defined name	[string] Label for user-defined smoother. Default: "User-defined".
smoother: Hiptmair efficient symmetric	[bool] Reduce the preconditioner computational work while maintaining symmetry by doing edge-then-node relaxation on the first leg of the V-cycle, and node-then-edge relaxation on the second leg. Default: true.
smoother: petsc ksp ⋆	[void*] A fully-constructed PETScKSP smoother. Can only be used on finest level with a matrix of type Epetra_PETScAIJMatrix, part of the EPETRAEXT package.
subsmoother: type ⋆	[string] Smoother to be used as a subsmoother on the nodal and edge subproblems within Hiptmair smoothing. It can assume one of the following values: Chebyshev, symmetric Gauss-Seidel, ILU, IC, ILUT, ICT. Default: Chebyshev.
subsmoother: Chebyshev alpha ⋆	[double] Hiptmair Chebyshev subsmoother will damps errors between ρ and ρ/alpha where ρ is the spectral radius of $diag(A)^{-1}A$. Default: 20.
subsmoother: SGS damping factor ⋆	[double] Damping factor for Hiptmair subsmoother symmetric Gauss-Seidel. Default: automatically calculated in the range $(0, 2)$.
subsmoother: edge sweeps ⋆	[int] Number of iterations (or polynomial degree) of Hiptmair subsmoother to perform on edge problem. Default: 2.
subsmoother: node sweeps ⋆	[int] Number of iterations (or polynomial degree) of Hiptmair subsmoother to perform on nodal projection problem. Default: 2.

6.4.7 Coarsest Grid Parameters

coarse: max size	[int] Maximum dimension of the coarse grid. **ML** will not coarsen further if current levels̈ size is less than this value. Default: 128.
coarse: type	[string] Coarse solver. See Table 5. Default: AmesosKLU.
coarse: pre or post ⋆	[string] It can assume one of the following values: pre, post, both. Practically, pre and post are the same. Default: post.
coarse: sweeps	[int] (Only for Jacobi, Gauss-Seidel, symmetric Gauss-Seidel, Hiptmair, and Chebyshev.) Number of sweeps (or polynomial degree) in the coarse solver. Default: 2.
coarse: user-defined function	[int (*)(ML_Smoother *, int inLength, double *invec, int outLength, double *outvec)] Pointer to user-defined smoothing function. Default: NULL.
coarse: user-defined name	[string] Label for user-defined smoother. Default: "User-defined".
coarse: damping factor	[double] Smoother damping factor for Jacobi, Gauss-Seidel, symmetric Gauss-Seidel, block Gauss-Seidel, or symmetric block Gauss-Seidel. Default: 1.0.
coarse: subsmoother type	[string] Smoother to be used as a sub-smoother at each step of coarse Hiptmair smoothing. It can assume the following values: Chebyshev, symmetric Gauss-Seidel. Default: Chebyshev.
coarse: node sweeps	[int] Number of sweeps (or polynomial degree) for nodal subproblem within coarse Hiptmair smoothing. Default: 2.
coarse: edge sweeps	[int] Number of sweeps (or polynomial degree) for edge subproblem within coarse Hiptmair smoothing. Default: 2.

`coarse: Chebyshev alpha`	[double] Chebyshev smoother (or sub-smoother) will damps errors between ρ and $\rho/$alpha where ρ is the spectral radius of $diag(A)^{-1}A$. Default: 30.
`coarse: max processes`	[int] Maximum number of processes to be used in the coarse grid solution (only for `Amesos-Superludist, Amesos-MUMPS`). If -1, AMESOS will decide the optimal number of processors to be used. Default: -1.

6.4.8 Load-balancing Options

`repartition: enable`	[int] Enable/disable repartitioning. Default: 0 (off).
`repartition: partitioner`	[string] Partitioning package to use. Can assume the following values: `Zoltan, ParMETIS`. Default: `Zoltan`.
`repartition: max min ratio`	[double] Specifies desired ratio of g_{max}/g_{min}, where g_{max} is the maximum number of rows over all processors, and g_{min} is the minimum number of rows over all processors. If the actual ratio is larger, then repartitioning occurs. Default: 1.3.
`repartition: min per proc`	[int] Specifies desired minimum number of rows per processor. If the actual number is smaller, then repartitioning occurs. Default: 512.
`repartition: node max min ratio`	[double] Maxwell-specific option for auxiliary nodal hierarchy. Specifies desired ratio of g_{max}/g_{min}, where g_{max} is the maximum number of rows over all processors, and g_{min} is the minimum number of rows over all processors. If the actual ratio is larger, then repartitioning occurs. Default: 1.3.
`repartition: node min per proc`	[int] Maxwell-specific option for auxiliary nodal hierarchy. Specifies desired minimum number of rows per processor. If the actual number is smaller, then repartitioning occurs. Default: 170.
`repartition: Zoltan dimensions`	[int] Dimension of the problem. Can assume the following values: $\{2, 3\}$.

6.4.9 Analysis Options

analyze memory   [`bool`] Analyze memory used by the preconditioner. Default: `false`.

viz: enable   [`bool`] Enable/disable visualization. Default: `false`.

viz: output format   [`string`] The format of the visualization files. Can assume the following values: `vtk`, `xyz`, `openx`. Default: `vtk`.

viz: print starting solution   [`bool`] Enable/disable printing of initial starting solution.

viz: equation to plot   [`bool`] In the case of systems of PDEs, the equation number to print. Default: all equations.

6.4.10 Miscellaneous Options

x-coordinates   [`double*`] Pointer to array of x-coordinates of mesh nodes (edges, in the case of Maxwell).

y-coordinates   [`double*`] Pointer to array of y-coordinates of mesh nodes (edges, in the case of Maxwell).

z-coordinates   [`double*`] Pointer to array of z-coordinates of mesh nodes (edges, in the case of Maxwell).

node: x-coordinates   [`double*`] Pointer to array of x-coordinates of mesh nodes. (Maxwell only)

node: y-coordinates   [`double*`] Pointer to array of y-coordinates of mesh nodes. (Maxwell only)

node: z-coordinates   [`double*`] Pointer to array of z-coordinates of mesh nodes. (Maxwell only)

read XML   [`bool`] Read parameter list options from file. Default: `false`

XML input file   [`string`] Name of XML file that contains paraameter list options. Default: `ml_ParameterList.xml`

6.4.11 Smoothed Aggregation and the Null Space

Smoothed aggregation builds grid transfers by improving an initial operator based on a near null space. For scalar PDEs, this null space is often taken to be a constant. In this case, the initial grid transfer is piecewise constant interpolation. For a PDE system with n degrees of freedom per mesh node, the null space if often taken as n constant vectors corresponding to each degree of freedom. While **ML** generates these simple null spaces automatically, it is also possible to pass in a null space (e.g. the set of rigid body modes for elasticity systems) that **ML** uses. A possibility to compute a null space via ANASAZI is currently disabled, but will hopefully get turned back on soon.

null space: type	[string] If default vectors, the default null space is used (no computation required). If pre-computed, a pointer to the already-computed null is obtained from option null space: vectors. If enriched, **ML** will compute, using ANASAZI, an approximation of the null space of the operator. Note: there is a utility ML_Coord2RBM() that converts coordinates to rigid body modes for elasticty problems. Default: default vectors.
null space: vectors	[double*]. Pointer to user-supplied null space vectors.
null space: dimension	[int]. Number of user-supplied null space vectors.
null space: vectors to compute	[int]. If null space: type is set to enriched, this option indicates the number of eigenvectors to compute. Default: 1.
null space: add default vectors	[bool]. If true, the default null space (one constant vector for each unknown) will be added to the computed null space. Default: true.

6.4.12 Aggregation Strategies

In some case, it is preferable to generate the aggregates using a matrix defined as follows:

$$L_{i,j} = (\mathbf{x}_i - \mathbf{x}_j)^{-2}, \quad i \neq j, \qquad L_{i,i} = -\sum_{i \neq j} L_{i,j}, \tag{2}$$

(where $\mathbf{x}_i$ represents the coordinates of node i) in conjunction with a given nonzero dropping threshold. This is often useful in cases where it is difficult to determine directions where smoothing is difficult based only on matrix coefficients. This may happen, for instance, when working with bilinear finite elements on stretched grids. **ML** can be instructed to generate the aggregates using matrix (2), then build the preconditioner using the actual

linear system matrix (i.e. use the original PDE matrix A dropping entries according to (2) in the prolongator smoother), as done in the following code fragment:

```
double* x_coord;
double* y_coord; // set to to 0 for 1D problems
double* z_coord; // set to to 0 for 2D problems
// here we define the nodal coordinates...

MLList.set("x-coordinates", x_coord );
MLList.set("y-coordinates", y_coord );
MLList.set("z-coordinates", z_coord );
```

The double vectors `y_coord` and/or `z_coord` can be set to `0`. `x_coord` (and `y_coord` if 2D, and `z_coord` if 3D) must be allocated to contain the coordinates for all nodes assigned to the calling processor, plus that of the ghost nodes. For systems of PDE equations, the coordinates must refer to the block nodes (that is, x-coordinate of local row `i` will be reported in `x_coord[i/NumPDEEqns]`. The actual PDE matrix A is only modified when developing a prolongator (i.e. for the aggregation and in the prolongator smoother). This modification drops entries *corresponding* to small $L_{i,j}$'s and adds dropped entries to the diagonal. For block PDEs, dropped entries in row i are only added to the diagonal if they correspond to the same degree of freedom within a block as i. The general idea is that contants will still be in the null space of the modified A if they were in the null space of A. The following options can be used:

`aggregation: aux: enable`	[`bool`]. Enable/disable the use of matrix (2). Default: `false`.
`aggregation: aux: threshold`	[`double`]. The matrix entry $A_{i,j}$ is ignored if $L_{i,j} < threshold * max_{j,i \neq j} L_{i,j}$. Note: the diagonal of A is also modified so that constants are still in the null space (assuming that this was true for the unmodified A). Default: `0.0`.

6.5 Default Parameter Settings for Common Problem Types

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

1. Classical smoothed aggregation for symmetric positive definite or nearly symmetric positive definite systems.

2. Classical smoothed aggregation-based 2-level domain decomposition

3. 3-level algebraic domain decomposition

4. Eddy current formulation of Maxwell's equations

5. Energy-based minimizing smoothed aggregation suitable for highly convective nonsymmetric fluid flow problems.

Option Name	Type	SA	DD	DD-ML	maxwell	NSSA
max levels	int	10	2	3	10	10
prec type	string	MGV	MGV	MGV	MGV	MGW
increasing or decreasing	string	increasing	increasing	increasing	decreasing	increasing
aggregation: type	string	Uncoupled-MIS	METIS	METIS	Uncoupled-MIS	Uncoupled-MIS
aggregation: local aggregates	int	–	1	–	–	–
aggregation: nodes per aggregate	int	–	–	512	–	–
aggregation: next-level aggregates per process	int	–	–	128	–	–
aggregation: damping factor	double	4/3	4/3	4/3	4/3	–
energy minimization: enable	bool	–	–	–	–	true
eigen-analysis: type	string	cg	power-method	power-method	cg	power-method
eigen-analysis: iterations	int	10	20	20	10	20
smoother: sweeps	int	2	1	1	1	4
smoother: damping factor	double	1.0	–	–	1.0	.67
smoother: pre or post	string	both	both	both	both	post
smoother: type	string	symmetric Gauss-Seidel	Aztec	Aztec	Hiptmair	symmetric Gauss-Seidel
smoother: Aztec as solver	bool	–	false	false	–	–
subsmoother: type	string	–	–	–	Chebyshev	–
subsmoother: Chebyshev alpha	double	–	–	–	27.0	–
subsmoother: node sweeps	int	–	–	–	4	–
subsmoother: edge sweeps	int	–	–	–	4	–
coarse: type	string	AmesosKLU	AmesosKLU	AmesosKLU	AmesosKLU	AmesosKLU
coarse: max size	int	128	128	128	128	256

Table 6: ML_Epetra::MultiLevelPreconditioner Default values for 5 supported problem types SA, DD, DD-ML, Maxwell, NNSA. "_" means not set.

Default values are listed in Table 6. 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. `NSSA` is a nonsymmetric smoothed aggregation variant which may be appropriate for highly nonsymmetric operators.

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);


Teuchos::ParameterList MLList;
ML_Epetra::SetDefaults(ProblemType, MLList, options, params);


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

In the second usage, `options` and `params` are vectors that control the AZTEC smoother. In the third usage, the boolean argument `true` permits `SetDefaults` to override any previously set parameters.

For `DD` and `DD-ML`, the default smoother is `Aztec`, with an incomplete factorization ILUT, and minimal overlap. Memory for the two AZTEC vectors, `options` and `params`, is allocated using TEUCHOS reference-counted pointers, and so will be freed automatically.

6.6 Analyzing the ML preconditioner

A successful multilevel preconditioner require the careful choice of a large variety of parameters, for instance each level's smoother, the aggregation schemes, or the coarse solver. Often, for non-standard problems, there is no theory to support these choices. Also, sometimes it is difficult to understand which component of the multilevel cycle is not properly working. To help to set the multilevel components, **ML** offers a set of tools, to (empirically) analyze several components of the multilevel cycles, and the finest-level matrix.

Two examples are included in the **ML** distribution:

- File `examples/Visualization/ml_viz.cpp` shows how to visualize the effect of the **ML** cycle and each level's smoother on a random vector;

- File `examples/Advanced/ml_analyze.cpp` shows who to get some quantitative information about each level's matrix, and multilevel preconditioner.

In the following subsections, we suppose that a MultiLevelPreconditioner object has already be defined, and the preconditioner computed.

6.6.1 Cheap Analysis of All Level Matrices

Method `AnalyzeMatrixCheap()` will report on standard output general information about each level's matrix. An example of output is as reported below. (Here, we report only the part of output related to the finest level.)

```
                *** Analysis of ML_Operator 'A matrix level 0' ***

        Number of global rows                        = 256
        Number of equations                          = 1
        Number of stored elements                    = 1216
        Number of nonzero elements                   = 1216
        Mininum number of nonzero elements/row       = 3
        Maximum number of nonzero elements/row       = 5
        Average number of nonzero elements/rows      = 4.750000
        Nonzero elements in strict lower part        = 480
        Nonzero elements in strict upper part        = 480
        Max |i-j|, a(i,j) != 0                       = 16
        Number of diagonally dominant rows           = 86 (= 33.59%)
        Number of weakly diagonally dominant rows    = 67 (= 26.17%)
        Number of Dirichlet rows                     = 0 (=  0.00%)
        ||A||_F                                      = 244.066240
        Min_{i,j} ( a(i,j) )                         = -14.950987
        Max_{i,j} ( a(i,j) )                         = 15.208792
        Min_{i,j} ( abs(a(i,j)) )                    = 0.002890
        Max_{i,j} ( abs(a(i,j)) )                    = 15.208792
        Min_i ( abs(a(i,i)) )                        = 2.004640
        Max_i ( abs(a(i,i)) )                        = 15.208792
        Min_i ( \sum_{j!=i} abs(a(i,j)) )            = 2.004640
        Max_i ( \sum_{j!=i} abs(a(i,j)) )            = 15.205902
        max eig(A) (using power method)              = 27.645954
        max eig(D^{-1}A) (using power method)        = 1.878674

        Total time for analysis = 3.147979e-03 (s)
```

This analysis is "cheap" in the sense that it involves only element-by-element comparison, plus the computation of the largest-magnitude eigenvalue (which requires some matrix-vector products). `AnalyzeMatrixCheap()` can be used for both serial and parallel runs.

6.6.2 Analyze the Effect of Smoothers

For each level, method `AnalyzeSmoothers()` computes the eigenvectors of the matrix (say, A). Then, for each eigenvector (say, v) of A, the smoother is applied to the solution of the homogeneous system

$$Ae = 0$$

with starting solution $e_0 = v$. The code reports on file the real and imaginary values of the eigenvalue corresponding to eigenvector v, and the $||e||/||e_0||$.

The syntax is `AnalyzeSmoothers(NumPre, NumPost)`. `NumPre` is the number of pre-smoother applications, and `NumPost` the number of post-smoother applications. This method reports on the standard output the following information:

```
Solving Ae = 0, with a random initial guess
- number of pre-smoother cycle(s) = 5
```

```
- number of post-smoother cycle(s) = 5
- all reported data are scaled with their values
  before the application of the solver
  (0 == perfect solution, 1 == no effect)
- SF is the smoothness factor

Solver                                   Linf      L2        SF
Presmoother  (level 0, eq 0)             0.827193  0.804528  0.313705
Postsmoother (level 0, eq 0)             0.822015  0.810521  0.342827
Presmoother  (level 1, eq 0)             0.972593  0.908874  2.51318
Postsmoother (level 1, eq 0)             0.982529  0.922668  2.53639
```

6.6.3   Analyze the effect of the ML cycle on a random vector

Method `AnalyzeCycle(NumCycles)`, where `NumCycles` is the number of multilevel cycles to apply, applies the already computed **ML** preconditioner to a random vector, and reports on standard output the following information:

```
Solving Ae = 0, with a random initial guess
using 5 ML cycle(s).
- (eq 0) scaled Linf norm after application(s) = 0.0224609
- (eq 0) scaled L2 norm after application(s)   = 0.000249379
- (eq 0) scaled smoothness factor              = 10.6517
```

6.6.4   Test different smoothers

The MultiLevelPreconditioner class offers a very easy way to test the effect of a variety of smoothers on the problem at hand. Once the preconditioning object has been created, a call to `TestSmoothers()` performs the following operations:

1. Creates a new linear system, whose matrix is the one used to construct the MultiLevel-Preconditioner object;

2. Defines a random solution, and the corresponding right-hand side;

3. Defines a zero starting vector for the Krylov solver;

4. Creates a new preconditioning object, with the same options as in the current preconditioner, except for the choice of the smoothers;

5. Solve the linear system with the newly created preconditioner;

6. Reports in a table the iterations to converge and the corresponding CPU time.

The following options, to be set *before* calling `ComputePreconditioner()`, can be used to tune the test session:

`test: max iters`	[int] Maximum number of iterations for the Krylov solver. Default: 500.

test: tolerance	[double] Tolerance for the Krylov solver. Default: 1e-5.
test: Jacobi	[bool] Enable/disable test with Jacobi smoother. Default: true.
test: Gauss-Seidel	[bool] Enable/disable test with Gauss-Seidel smoother. Default: true.
test: symmetric Gauss-Seidel	[bool] Enable/disable test with symmetric Gauss-Seidel smoother. Default: true.
test: block Gauss-Seidel	[bool] Enable/disable test with block Gauss-Seidel smoother. Default: true.
test: Aztec	[bool] Enable/disable test with AztecOO smoother. Default: true.
test: Aztec as solver	[bool] Enable/disable test with AztecOO as a solver smoother. Default: true.
test: ParaSails	[bool] Enable/disable test with ParaSailssmoother. Default: true.
test: IFPACK	[bool] Enable/disable test with Ifpacksmoother. Default: true.

An example of output is reported below. Note that some smoothers (ParaSails, Ifpack) will be tested only if **ML** has been properly configured. Note also that `TestSmoothers()` requires **ML** to be configured with option `--enable-aztecoo`.

```
------------------------------------------------------------------------
*** ************************************* ***
*** Analysis of ML parameters (smoothers) ***
*** ************************************* ***

*** maximum iterations = 500
*** tolerance          = 1e-05

*** All options as in the input parameter list, except that
*** all levels have the same smoother

*** M: maximum iterations exceeded without convergence
*** N: normal exit status (convergence achieved)
*** B: breakdown occurred
*** I: matrix is ill-conditioned
*** L: numerical loss of precision occurred

count  smoother type.................its.......||r||/||r_0||..time (s).......
```

```
- Jacobi
#0.....n=5, omega=2.50e-01...........12.......2.97314e-06....0.0839319......N
#1.....n=5, omega=5.00e-01...........12.......6.21844e-06....0.0820519......N
#2.....n=5, omega=7.50e-01...........12.......7.52614e-06....0.082267.......N
#3.....n=5, omega=1.00e+00...........12.......3.80406e-06....0.082956.......N
#4.....n=5, omega=1.25e+00...........12.......2.15858e-06....0.0824361......N

- Gauss-Seidel
#5.....n=5, omega=2.50e-01...........7........2.20736e-06....0.0857691......N
#6.....n=5, omega=5.00e-01...........7........1.91864e-06....0.0789189......N
#7.....n=5, omega=7.50e-01...........6........8.40948e-06....0.076307.......N
#8.....n=5, omega=1.00e+00...........7........1.36415e-06....0.0792729......N
#9.....n=5, omega=1.25e+00...........7........1.4833e-06.....0.0790809......N

- Gauss-Seidel (sym)
#10....n=5, omega=2.50e-01...........4........2.32835e-06....0.149586.......N
#11....n=5, omega=5.00e-01...........4........2.68576e-06....0.092068.......N
#12....n=5, omega=7.50e-01...........4........8.51966e-07....0.0793679......N
#13....n=5, omega=1.00e+00...........4........1.34439e-06....0.0787291......N
#14....n=5, omega=1.25e+00...........4........5.09185e-06....0.0790669......N

- Gauss-Seidel (block)
#15....n=5, omega=2.50e-01...........6........6.56673e-06....0.0920131......N
#16....n=5, omega=5.00e-01...........7........1.77309e-06....0.0881529......N
#17....n=5, omega=7.50e-01...........6........8.53488e-06....0.0846661......N
#18....n=5, omega=1.00e+00...........6........2.66381e-06....0.0839909......N
#19....n=5, omega=1.25e+00...........6........4.87356e-06....0.083786.......N

- Aztec preconditioner
#20....ILU(fill=0)...................7........4.93736e-06....0.0712331......N
#21....ILU(fill=1)...................6........3.54992e-06....0.0647091......N
#22....ILU(fill=2)...................7........1.4724e-06.....0.0678911......N

- Aztec as solver
#23....iterations=1..................7........1.94081e-06....0.140772.......N
#24....iterations=3..................4........8.90029e-08....0.0687031......N
#25....iterations=5..................3........1.00778e-07....0.069193.......N

- ParaSails
#26....default.......................20.......6.60045e-06....0.214094.......N

*** The best iteration count was obtain in test 25
*** The best CPU-time was obtain in test 21
```

```
*** Total time = 2.43798(s)
--------------------------------------------------------------------------
```

6.7   Visualization Capabilities

6.7.1   Visualizing the Aggregates

ML offers the possibility to visualize the aggregates for all levels. Aggregates generated by `Uncoupled` and `METIS` aggregation schemes can be visualized for serial and parallel runs, while aggregates generated using `MIS` and `ParMETIS` can be visualized only for serial runs.

Data can be stored in the "xyz" format, readable by packages such as `XD3D` [**?**]. For each aggregate, the file contains a line of type

```
x-coord y-coord <z-coord> aggregate-number
```

(`z-coord` is present only for 3D computations.) `XD3D` can be used to visualize files in this format, but only for 2D problems. Results are reported in Figure 2.

Data can also be stored in the "vtk legacy" format, readable by packages such as PAR-AVIEW [**?**]. PARAVIEW allows for 2D and 3D point clouds visualization. It is also possible to visualize sections of 3D data via cutting planes. In order to use this format, fine level coordinates must be supplied. Coarse level coordinates are derived from the fine level coordinates via the multigrid restriction operators. Data is stored as follows. The first section begins with the header

```
POINTS XX float
```

where `XX` is the total number of points. For each point, the coordinates are given as

```
x-coord y-coord <z-coord>
```

Next, connectivity information is given in a pair-wise fashion. For example, if the (i, j) entry of matrix A is nonzero, then there is a line in the data file of the form

```
2 i j
```

where the first entry specifies the number of points that follow. (This first entry will always be 2 because the connectivity is pair-wise.) The next data section specifies the aggregate to which each point belongs. It begins with the header

```
POINT_DATA XX
SCALARS aggregate_number int 1
LOOKUP_TABLE default
```

The section then has a line for each point of the form

```
aggregate_number
```

The final section specifies the solution (residual) at each point. The section begins with the header

```
SCALARS solution float 1
LOOKUP_TABLE default
```

and then has a line for each point of the form

```
solution_value
```

With the exception of connectivity, all information is given in the same order. Hence, all information for the jth unknown can be found at the jth line of each section.

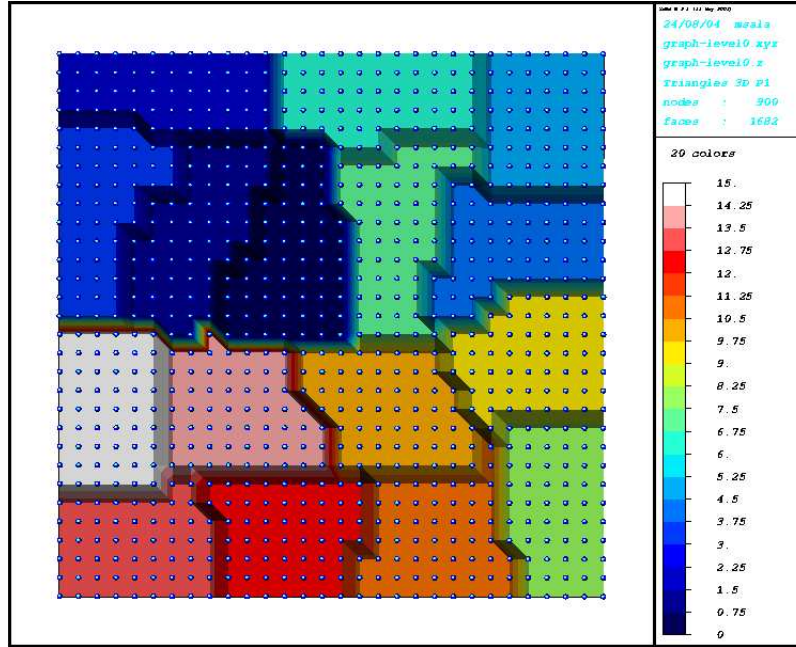Figure 2: Decomposition into aggregates. Uniform colors between blue spheres represent the interior of aggregates. Visualization was done with XD3D.

6.7.2 Visualizing the effect of the ML Preconditioner and Smoothers

In some cases, it may be useful to visualize the effect of the **ML** preconditioner, or of each level's smoother, on a random vector (whose components are contained in the interval

[0.5, 1]), to understand if there are zones or directions that are not affected by the current multilevel preconditioner. This can be easily done with the following code fragment:

```
double* x_coord;
double* y_coord;
// here we define the nodal coordinates...

MLList.set("viz: enable", true);
//you can also specify "xyz" on the line below
MLList.set("viz: output format", "vtk");
MLList.set("viz: x-coordinates", x_coord);
MLList.set("viz: y-coordinates", y_coord);
// by default the starting solution is not printed
MLList.set("viz: print starting solution", true);


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


// visualize the effect of the ML preconditioner on a
// random vector. We ask for 10 multilevel cycles
MLPrec->VisualizeCycle(10);


// visualize the effect of each level's smoother on a
// random vector. We ask for 5 steps of presmoothers,
// and 1 step of postsmoother
MLPrec->VisualizeSmoothers(5,1);
```

(File `ml_viz.cpp` contains the compilable code.) We note that the parameters must be set *before* calling `ComputePreconditioner()`. See also Section 6.4.12 for the requirements on the coordinates vectors. Results will be written in the following files:

- `before-presmoother-eqX-levelY.vtk` contains the random vector before the application of the presmoother, for equation `X` at level `Y`;

- `after-presmoother-eqX-levelY.vtk` contains the random vector after the application of the presmoother, for equation `X` at level `Y`;

- `before-postsmoother-eqX-levelY.vtk` contains the random vector before the application of the postsmoother, for equation `X` at level `Y`;

- `after-postsmoother-eqX-levelY.vtk` contains the random vector after the application of the postsmoother, for equation `X` at level `Y`;

- `before-cycle-eqX-levelY.vtk` contains the random vector before the application of the **ML**cycle, for equation `X` at the finest level;

- `after-cycle-eqX-levelY.vtk` contains the random vector after the application of the ML cycle, for equation `X` at finest level.
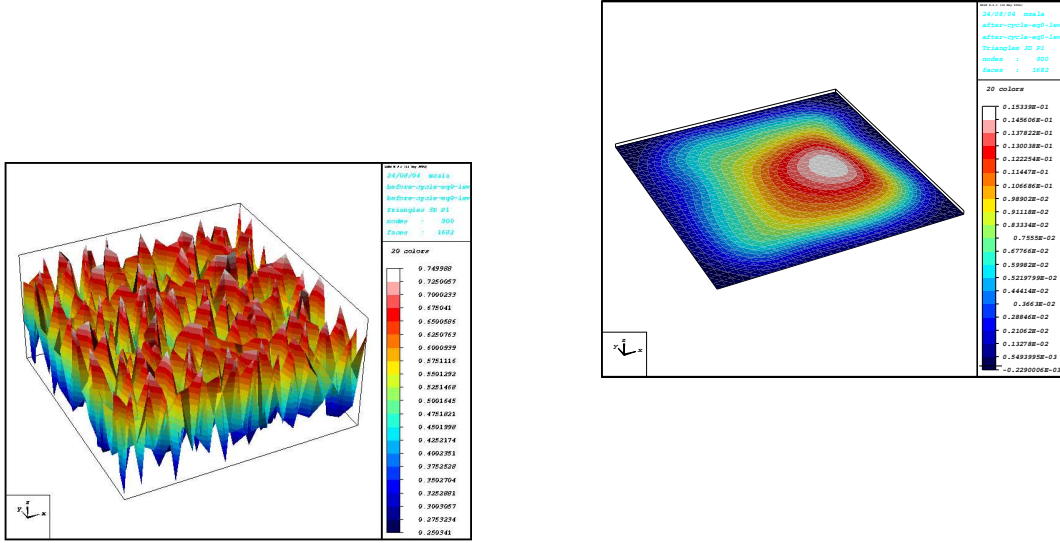
Figure 3: Starting solution (left) and solution after 10 **ML** cycles (right). Visualization was done using XD3D.

6.8   Print the Computational Stencil for a 2D Cartesian Grid

Method `PrintStencil2D()` can be used to print out the computational stencil for problems defined on 2D Cartesian grids, if the nodes numbering follows the x-axis. The following fragment of code shows the use of this method:

```
int Nx = 16; // nodes along the x-axis
int Ny = 32; // nodes along the y-axis
int NodeID = -1; // print the stencil for this node
int EquationID = 0; // equation 0, useful for vector problems

// MLPrec is a pointer to an already created
// ML_Epetra::MultiLevelPreconditioner
MLPrec->PrintStencil2D(Nx,Ny,NodeID, EquationID);

// also valid in this case
MLPrec->PrintStencil2D(Nx,Ny);
```

If `NodeID == -1`, the code considers a node in the center of the computational domain. The result can be something as follows:

```
2D computational stencil for equation 0 at node 136 (grid is 16 x 16)


            0                 -1                 0
           -1                  4                -1
            0                 -1                 0
```

7 Using the Maxwell Solvers in ML

This section gives a brief overview of how to ML's two AMG preconditioners for the eddy current formulation of Maxwell's equations. These first solver (Maxwell) is intended only for real-valued equations in the time domain. The second solver (RefMaxwell) is intended for real-valued (not complex) problems in the time or frequency domains.

7.1 Background

The eddy current formulation of Maxwell's equations can be written as

$$\nabla \times \nabla \times \vec{E} + \sigma \vec{E} = \vec{f}, \tag{3}$$

where $\vec{E}$ is the unknown electric field to be computed, σ is the spatially-varying electrical conductivity, and $\vec{f}$ is the known right-hand side. Neumann, Dirichlet, and/or periodic boundary conditions are supported.

Although we assume that (3) is discretized with first-order edge elements, it is possible to use this preconditioner for higher-order discretizations, as long as the user provides the sub-matrix that corresponds to just the first-order discretization. For more theoretical background and algorithm descriptions, please see [14, 2].

7.2 Notational conventions

For the remainder of the discussion, $K^{(e)}$ denotes the matrix corresponding to the first-order element discretization of (3). $K^{(e)}$ can be written as

$$K^{(e)} = S + M,$$

where S is the stiffness matrix corresponding to the $\nabla \times \nabla \times$ term in (3), and M is the mass matrix corresponding to the $\sigma \vec{E}$ term in (3). The null-space of S is given by the discrete gradient matrix, T. T corresponds to the null space of the $\nabla \times \nabla \times$ term in (3). $K^{(n)}$ is an auxiliary nodal matrix that is described in §7.4.2.

7.3 Description of the discrete gradient matrix T

T is a rectangular, $N_e \times N_n$ matrix, where N_e is the number of edges in the mesh (rows in $K^{(e)}$) and N_n is the number of nodes in the mesh (rows in $K^{(n)}$). Each row of T has at most two entries, a $+1$ and/or a -1. There are two ways to construct T. In the first method, it's assumed that the mesh topology is available. T can be viewed as a node-edge incidence matrix of the mesh as a directed graph. As such, each row of T corresponds to an edge, and the $+1/-1$ entries in the row are the head/tail of the edge, respectively. Hence, T can be built one row at a time by visiting each edge and inserting $+1/-1$ in the appropriate columns. The second method assumes that $K^{(n)}$ is already constructed. Each off-diagonal nonzero entry, (i, j), in the upper triangular portion of $K^{(n)}$ corresponds to a row in T containing a 1 and a -1 in columns i and j.

7.4 Solver #1: Maxwell

The Maxwell solver is accessed through the ML_Epetra::MultiLevelPreconditioner class, which is described in more detail in Section 6.

7.4.1 Operators that the user must supply

The user must provide three matrices:

1. the first-order edge element matrix $K^{(e)}$.

2. the auxiliary nodal matrix $K^{(n)}$.

3. the discrete gradient matrix T.

7.4.2 Description of the auxiliary nodal matrix $K^{(n)}$

$K^{(n)}$ is a square, $N_n \times N_n$ matrix, where N_n is the number of nodes in the mesh. There are two ways to construct $K^{(n)}$. The first method is to discretize the PDE

$$\int_\Omega \sigma u \cdot v + \int_\Omega \nabla u \cdot \nabla v, \tag{4}$$

using nodal linear finite elements on the same mesh as (3). The second method assumes that you already have T available. $K^{(n)}$ can be formed via the triple-matrix product

$$T^\mathrm{T} K^{(e)} T = T^\mathrm{T} M T,$$

where we have used the fact that $ST = 0$.

7.4.3 Smoother options

The default smoother for the Maxwell solver is a two-stage smoother that we call Hiptmair smoothing [13]. This smoother consists of three steps: one step of a smoother $\mathcal{S}$ on the entire system $Kx = f$; one step of a smoother $\mathcal{S}$ on the projected system $T^\mathrm{T} M T e = T^\mathrm{T} r$, where $r = f - Kx$; and finally one step of a smoother $\mathcal{S}$ on the entire system with initial guess $x + e$.

The default sub-smoother $\mathcal{S}$ is a degree 3 Chebyshev polynomial. The coefficients of this polynomial are calculated automatically based on the coarsening rate of the multigrid hierarchy. The Chebyshev degree for the nodal (or edge) problem can be changed with the options `subsmoother: node sweeps` (or `subsmoother: edge sweeps`) and `coarse: node sweeps` (or `coarse: edge sweeps`) depending on whether the Hiptmair smoothing is done on a fine or the coarsest level.

7.4.4 Sample Usage

```
#include "ml_MultiLevelPreconditioner.h"

// Allocate Matrices
Epetra_CrsMatrix CurlCurlMatrix(...);
```

```
Epetra_CrsMatrix TMatrix(...);
Epetra_CrsMatrix KnMatrix(...);

// Fill Matrices Here

// Build Maxwell Preconditioner
ML_Epetra::MultiLevelPreconditioner(CurlCurlMatrix, TMatrix, KnMatrix);
```

7.5 Solver #2: RefMaxwell

The RefMaxwell solver is accessed through the `ML_Epetra::RefMaxwellPreconditioner` class, which can be found in the directory `ml/src/RefMaxwell`. This preconditioner handles the range and null space of the curl operator with a dedicated `MultiLevelPreconditioner` object for each and has demonstrated excellent parallel scalability in practice. The solver algorithm is detailed in [3]

7.5.1 Operators that the user must supply

The user must provide four matrices:

1. the first-order edge element matrix $K^{(e)}$.

2. the discrete gradient matrix T.

3. the inverse of the (lumped) nodal mass matrix $\mathcal{M}_0$.

4. the edge mass matrix $\mathcal{M}_1$.

The matrices $K^{(e)}$ and T are identical to the matrices used in Maxwell (described in Section 7.4). The matrix $\mathcal{M}_0$ is used for the algebraic gauging of the edge space and should include appropriate factors for the permeability (μ). The matrix $\mathcal{M}_1$ is used only for the edge space, meaning that it does not include the conductivity (σ).

7.5.2 Coarse nullspaces

RefMaxwell will automatically calculate the nullspace for the coarse edge problem from the discrete gradient matrix T and nodal coordinates. For periodic problems, however, this calculation will not be correct (since the nodes on the periodic boundary are really in two places at the same time). In this case, the user can specify nullspace options (using the standard syntax) as part of `refmaxwell: 11list`.

7.5.3 Smoother options

Unlike the Maxwell solver, it is not particularly advantageous to use the so-called Hiptmair hybrid smoother with RefMaxwell. A more traditional smoother, such as Chebyshev (in parallel) or Gauss-Seidel (in serial) should be used.

7.5.4 Setting parameter options

RefMaxwell uses a nested parameter list in order to logically separate the options for each individual solver. Each sublist would include the appropriate options for that particular solver. A full list of options supported by the `MultiLevelPreconditioner` can be found in Section 6.4.

In addition to the options for `MultiLevelPreconditioner` (almost all of which are valid), the RefMaxwell solver has the following unique options:

`refmaxwell: 11solver`	[`string`] Sets the type of solver to use on the (1,1) block (edges). Default: `edge matrix free`.
`refmaxwell: 11list`	[`ParameterList`] Contains all of the options for the (1,1) block preconditioner (edges).
`refmaxwell: 22solver`	[`string`] Sets the type of solver to use on the (2,2) block (nodes). Default: `multilevel`.
`refmaxwell: 22list`	[`ParameterList`] Contains all of the options for the (2,2) block preconditioner (nodes).
`refmaxwell: mode`	[`string`] Sets the specific type of preconditioner used. Default: `additive`.

The `EdgeMatrixFreePreconditoner` for the (1,1) block (currently the only supported option) also has one unique option:

`edge matrix free: coarse`	[`ParameterList`] Contains all of the options for the coarse (1,1) block preconditioner (edges).

7.5.5 Sample Usage

```
#include "ml_RefMaxwell.h"

// Allocate Matrices
Epetra_CrsMatrix CurlCurlMatrix(...);
Epetra_CrsMatrix TMatrix(...);
Epetra_CrsMatrix M0invMatrix(...);
Epetra_CrsMatrix M1Matrix(...);

// Fill Matrices Here

// Build RefMaxwell Preconditioner
ML_Epetra::RefMaxwellPreconditioner(CurlCurlMatrix, TMatrix,
                                    M0invMatrix, M1Matrix);
```

8 Advanced Usage of ML

Sections 6 and 6.4 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 4. The function ML_Create creates a multilevel

```
ML_Create          (&ml_object, N_grids);

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

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

Figure 4: High level multigrid sample code.

solver object that is used to define the preconditioner. It requires the maximum number of multigrid levels be specified. In almost all cases, N_grids= 20 is more than adequate. The three 'Amatrix' statements are used to define the discretization matrix, A, that is solved. This is discussed in greater detail in Section 12.1. The multigrid hierarchy is generated via ML_Gen_MGHierarchy_UsingAggregation. Controlling the behavior of this function is discussed in Section 10. 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 [23] [22]. 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 9. 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.

9   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.

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 [20] 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(ML_Smoother *smoother, int x_length, double x[],
                   int rhs_length, double rhs[])
```

where `smoother` is an internal **ML** object, `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`. The function ML_Get_MySmootherData(smoother) can be used to get a pointer back to the user's data (i.e. the data pointer given with the ML_Set_Smoother invocation). A simple (and suboptimal) damped Jacobi smoother for the finest grid of our example is given below:

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

   Poisson_matvec(ML_Get_MySmootherData(smoother), 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.

10 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.

10.1 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 $0 \le \epsilon \le 10^{-6}$) , 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

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

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 [21]. 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.

10.2 Interpolation Options

An interpolation operator is built using coarsening information, seed vectors, and a damping factor. We refer the reader to [22] 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 (which can be computed with the help of ML_Coord2RBM()) 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 [22]). 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 a few ways to determine the Jacobi damping parameter and each require some estimate of the spectral radius of the discretization operator. The current default is to use a few iterations of the power method method (subspace size of two) to estimate this value (see ML_Set_SpectralNormScheme_PowerMethod) However, if the matrix is symmetric conjugate-gradient method should be used via ML_Set_SpectralNormScheme_Calc. It is also possible to change the default number of iterations used in the eigensolver from ten via ML_Set_SpectralNorm_Iterations. 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.

11   Advanced Usage of ML and Epetra

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

```
#include "ml_MultiLevelOperator.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 8), 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);
```

12   Using ML without Epetra

12.1   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 5.2). Further, functions for some common matrix storage formats (CSR & MSR) already exist within ML and do not need to be rewritten[12].

Size information is indicated via ML_Init_Amatrix. The third parameter in the Figure 4 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.

---

[12]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 4 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(ML_Operator *Amat, int in_length, double p[], int out_length,
                double ap[])
```

where `A_mat` is an internal **ML** object, `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`. The function **ML_Get_MyMatvecData(Amat)** can be used to get a pointer back to the user's data (i.e. the data pointer given with the **ML_Init_Amatrix** invocation).

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

where `Amat` is an internal **ML** object, `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. The function **ML_Get_MyGetrowData(Amat)** can be used to get a pointer back to the user's data (i.e. the data pointer given with the **ML_Init_Amatrix** invocation). 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:

$$
\begin{pmatrix}
2 & -1 & & & \\
-1 & 2 & -1 & & \\
& -1 & 2 & -1 & \\
& & -1 & 2 & -1 \\
& & & -1 & 2
\end{pmatrix}.
\tag{5}
$$

To implement this matrix, the following functions are defined:

```
int Poisson_getrow(ML_Operator *Amat, 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(ML_Operator *Amat, 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 `Amat` was not used. In the next section we give a parallel example which makes use of `Amat`. The complete sample program can be found in the file `mlguide.c` within the **ML** code distribution.

12.2   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 (5) 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 kth 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[13]. 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(ML_Operator *Amat, int N_requested_rows, int requested_rows[],
   int allocated_space, int cols[], double values[], int row_lengths[])
```

---

[13]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.

```
{
    int m = 0, i, row, proc, *itemp, start;

    itemp = (int *) ML_Get_MyGetrowData(Amat);
    proc  = *itemp;

    for (i = 0; i < N_requested_rows; i++) {
        row = requested_rows[i];
        if (allocated_space < m+3) return(0);
        values[m] = 2; values[m+1] = -1; values[m+2] = -1;
        start = m;
        if (proc == 0) {
            if (row == 0) {cols[m++] = 0; cols[m++] = 2;                 }
            if (row == 1) {cols[m++] = 1; cols[m++] = 3;}
        }
        if (proc == 1) {
            if (row == 0) {cols[m++] = 0; cols[m++] = 1; cols[m++] = 4;}
            if (row == 1) {cols[m++] = 1; cols[m++] = 0; cols[m++] = 2;}
            if (row == 2) {cols[m++] = 2; cols[m++] = 1; cols[m++] = 3;}
        }
        row_lengths[i] = m - start;
    }
    return(1);
}
int Poisson_matvec(ML_Operator *Amat, int in_length, double p[], int out_length,
                   double ap[])
{
    int i, proc, *itemp;
    double new_p[5];

    itemp = (int *) ML_Get_MyMatvecData(Amat);
    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);
```

13 MLMEX: The MATLAB Interface for ML

MLMEX is ML's interface to the MATLAB environment. It allows access to a limited set of routines either using the ML_Epetra or the MLAPI interfaces. It is designed to provide access to ML's aggregation and solver routines from MATLAB and does little else. MLMEX allows users to setup and solve arbitrarily many problems, so long as memory suffices. More than one problem can be setup simultaneously.

13.1 Configure and Make

In order to use MLMEX, ML must be configured with (at least) the following options:

```
../configure -enable-epetra --enable-teuchos --enable-ifpack \
  --enable-aztecoo --enable-galeri --enable-amesos --enable-epetraext \
  --enable-ml-matlab \
  --with-matlab-exec=<directory with matlab binaries> \
  --with-matlab-root=<root directory of matlab install>
```

Most additional options (such as `--enable-zoltan`) can be specified as well. It is important to note that MLMEX does not work properly with MPI, hence MPI must be disabled in order to compile MLMEX .

On 64-bit Intel/AMD architectures, ML and all required Trilinos libraries must be compiled with the `-fPIC` option. This necessitates adding `CFLAGS=-fPIC CXXFLAGS=-fPIC FFLAGS=-fPIC` to the Trilinos configure line. Currently, MLMEX will not compile under Solaris.

The build system uses the `mexext` script from Mathworks to detect the appropriate file extension for the architecture. At present, this means that cross-compilation is not possible. It also means that MLMEX will only support MATLAB 7.2 (R2006a) and up.

13.2   Using MLMEX

MLMEX is designed to be interfaced with via the MATLAB script `ml.m`. There are five modes in which MLMEX can be run:

1. Setup Mode — Performs the problem setup for ML. Depending on whether or not the `mlmex:  interface` option is used, MLMEX creates either a ML_Epetra or MLAPI object. This call returns a problem handle used to reference the problem in the future.

2. Solve Mode — Given a problem handle and a right-hand side, ML solves the problem specified. Setup mode must be called before solve mode.

3. Cleanup Mode — Frees the memory allocated to internal ML objects. This can be called with a particular problem handle, in which case it frees that problem, or without one, in which case all MLMEX memory is freed.

4. Status Mode — Prints out status information on problems which have been set up. Like cleanup, it can be called with or without a particular problem handle.

5. Aggregate Mode — Uses MLMEX as an interface to ML's aggregation routines.

All of these modes, with the exception of status and cleanup take option lists which will be directly converted into `Teuchos::ParameterList` objects by MLMEX. This means that the options described in Section 6 for ML_Epetra and those in the MLAPI guide (for MLAPI) will work correctly for MLMEX.

13.2.1   Setup Mode

Setup mode is called as follows:

```
>> [h,oc]=ml('setup',A,['parameter',value,...])
```

The parameter `A` represents the sparse matrix to perform aggregation on and the parameter/value pairs represent standard ML options.

The routine returns a problem handle, `h`, and the operator complexity `oc` for the operator. In addition to the standard options, setup mode has one unique option of its own:

`mlmex:  interface`	[string] Whether to use ML_Epetra ('epetra') or MLAPI ('mlapi'). Default: 'epetra'.

13.2.2 Solve Mode

Solve mode is called as follows:

```
>> x=ml(h,A,b,['parameter',value,...])
```

The parameter `h` is a problem handle returned by the setup mode call, `A` is the sparse matrix with which to solve and `b` is the right-hand side. Parameter/value pairs are as above. However, a few additional MLAPI options, normally not supported by ML_Epetra, are supported as well:

krylov:	tolerance	[`double`] Tolerance for the solver. Default: 1e-9.
krylov:	max iterations	[`int`] Maximum number of krylov iterations. Default 1550.
krylov:	type	[`string`] Solver to use, e.g. 'cg', 'gmres', 'fixed point'. Default 'gmres'.
krylov:	output level	[`int`] ML output level. Default 10.
krylov:	conv	[`string`] ML convergence criterion. Default 'r0'.

All of these options are taken directly from MLAPI, so consult its manual for more information.

13.2.3 Cleanup Mode

Cleanup mode is called as follows:

```
>>  ml('cleanup',[h])
```

The parameter `h` is a problem handle returned by the setup mode call and is optional. If `h` is provided, that problem is cleaned up. If the option is not provided all currently set up problems are cleaned up.

13.2.4 Status Mode

Status mode is called as follows:

```
>>  ml('status',[h])
```

The parameter `h` is a problem handle returned by the setup mode call and is optional. If `h` is provided, status information for that problem is printed. If the option is not provided all currently set up problems have status information printed.

13.2.5 Aggregate Mode

Status mode is called as follows:

```
>>  agg=ml('aggregate',A,['parameter',value,...])
```

The parameter `A` represents the sparse matrix to perform aggregation upon. The aggregates are returned and all memory is immediately freed. The purpose of this option is to allow access to ML's aggregation routines in MATLAB.

13.3 Tips and Tricks

Internally, MATLAB represents all data as doubles unless you go through efforts to do otherwise. MLMEX detects integer parameters by a relative error test, seeing if the relative difference between the value from MATLAB and the value of the `int`-typecast value are less than 1e-15. Unfortunately, this means that MLMEX will choose the incorrect type for parameters which are doubles that happen to have an integer value (a good example of where this might happen would be the parameter 'smoother Chebyshev: alpha', which defaults to 30.0). Since MLMEX does no internal typechecking of parameters (it uses ML's internal checks), it has no way of detecting this conflict. From the user's perspective, avoiding this is as simple as adding a small perturbation (greater than a relative 1e-15) to the parameter that makes it non-integer valued.

References

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

[2] P. B. Bochev, C. J. Garasi, J. J. Hu, A. C. Robinson, and R. S. Tuminaro, *An improved algebraic multigrid method for solving Maxwell's equations*, SIAM J. Sci. Comput., 25 (2003), pp. 623–642.

[3] P. B. Bochev, J. J. Hu, C. M. Siefert, and R. S. Tuminaro, *An algebraic multigrid approach based on a compatible gauge reformulation of Maxwell's equations*, Tech. Rep. SAND2007-1633J, Sandia National Laboratories, 2007. Submitted to SIAM J. Sci. Comput.

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

[5] E. Chow, *Parasails user's guide*, Tech. Rep. UCRL-MA-137863, Lawrence Livermore National Laboratory, 2000.

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

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

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

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

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

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

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

[13] R. Hiptmair, *Multigrid method for Maxwell's equations*, SIAM J. Numer. Anal., 36 (1998), pp. 204–225.

[14] J. J. Hu, R. S. Tuminaro, P. B. Bochev, C. J. Garasi, and A. C. Robinson, *Toward an h-independent algebraic multigrid method for Maxwell's equations*, SIAM J. Sci. Comput., (2005). Accepted for publication.

[15] 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.

[16] G. Karypis and V. Kumar, *METIS: Unstructured graph partitining and sparse matrix ordering sy stem*, tech. rep., University of Minnesota, Department of Computer Science, 1998.

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

[18] M. Sala and M. Heroux, *Robust algebraic preconditioners with IFPACK 3.0*, Tech. Rep. SAND-0662, Sandia National Laboratories, 2005.

[19] M. Sala and M. A. Heroux, *Trilinos Tutorial*, 3.1 ed., 2004.

[20] 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.

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

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

[23] 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.