

ML 3.1 Smoothed Aggregation User's Guide

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

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

Abstract

ML is a multigrid preconditioning package intended to solve linear systems of equations $Ax = b$ where A is a user supplied $n \times n$ sparse matrix, b is a user supplied vector of length n and x is a vector of length n to be computed. **ML** should be used on large sparse linear systems arising from partial differential equation (PDE) discretizations. While technically any linear system can be considered, **ML** should be used on linear systems that correspond to things that work well with multigrid methods (e.g. elliptic PDEs). **ML** can be used as a stand-alone package or to generate preconditioners for a traditional iterative solver package (e.g. Krylov methods). We have supplied support for working with the AZTEC 2.1 and AZTECOO iterative package [16]. However, other solvers can be used by supplying a few functions.

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

(page intentionally left blank)

Contents

| | | |
|----------|--|-----------|
| 1 | Notational Conventions | 7 |
| 2 | Overview | 7 |
| 3 | Multigrid Background | 7 |
| 4 | Configuring and Building ML | 8 |
| 4.1 | Building in Standalone Mode | 9 |
| 4.2 | Building with AZTEC 2.1 Support | 9 |
| 4.3 | Building with TRILINOS Support (RECOMMENDED) | 9 |
| 4.3.1 | Enabling Third Party Library Support | 10 |
| 4.3.2 | Enabling Profiling | 11 |
| 5 | ML and Epetra: Getting Started with the MultiLevelPreconditioner Class | 13 |
| 5.1 | Example 1: ml_example_epetra_preconditioner.cpp | 13 |
| 5.2 | Example 2: ml_example_epetra_preconditioner_2level.cpp | 15 |
| 6 | Parameters for the ML_Epetra::MultiLevelPreconditioner Class | 16 |
| 6.1 | Setting Options on a Specific Level | 17 |
| 6.2 | General Usage of the Parameter List | 17 |
| 6.3 | Default Parameter Settings for Common Problem Types | 18 |
| 6.4 | Commonly Used Parameters | 20 |
| 6.5 | List of All Parameters for MultiLevelPreconditioner Class | 20 |
| 6.5.1 | General Options | 20 |
| 6.5.2 | Aggregation Parameters | 22 |
| 6.5.3 | Smoothing Parameters | 22 |
| 6.5.4 | Coarsest Grid Parameters | 23 |
| 6.6 | Null Space Detection | 24 |
| 6.7 | Aggregation Strategies | 24 |
| 6.8 | Analyzing the ML preconditioner | 25 |
| 6.8.1 | Cheap Analysis of All Level Matrices | 25 |
| 6.8.2 | Eigenvalue Computations | 26 |
| 6.8.3 | Analyze the Effect of Smoothers | 28 |
| 6.8.4 | Analyze the effect of the ML cycle on a random vector | 29 |
| 6.8.5 | Test different smoothers | 29 |
| 6.8.6 | Visualizing the Aggregates | 32 |
| 6.8.7 | Visualizing the effect of the ML Preconditioner and Smoothers | 32 |
| 6.8.8 | Print the Computational Stencil for a 2D Cartesian Grid | 34 |
| 7 | Advanced Usage of ML | 35 |
| 8 | Multigrid & Smoothing Options | 36 |

| | | |
|-----------|--|-----------|
| 9 | Smoothed Aggregation Options | 38 |
| 9.1 | Aggregation Options | 38 |
| 9.2 | Interpolation Options | 39 |
| 10 | Advanced Usage of ML and Epetra | 40 |
| 11 | Using ML without Epetra | 41 |
| 11.1 | Creating a ML matrix: Single Processor | 41 |
| 11.2 | Creating a ML matrix: Multiple Processors | 43 |

1 Notational Conventions

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

```
% a_really_long_command
```

The character % indicates any shell prompt¹. Function names are shown as `ML_Gen_Solver`. Names of packages or libraries as reported in small caps, as `EPETRA`. Mathematical entities are shown in italics.

2 Overview

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

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

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

The **ML** package is used by creating a **ML** object and then associating a matrix, A , and a set of multigrid parameters which describe the specifics of the solver. Once created and initialized, the **ML** object can be used to solve linear systems.

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

3 Multigrid Background

A brief multigrid description is given (see [1], [7], or [8] 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

¹For 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 (  $k \neq \mathbf{Nlevel} - 1$  )
     $P_k = \text{determine\_interpolant}( A_k )$ ;
     $\hat{r} = P_k^T(b - A_k u)$  ;
     $\hat{A}_{k+1} = P_k^T A_k P_k$ ;   $v = 0$ ;
    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 8. For now, it suffices to view them as basic iterative methods (e.g. Gauss-Seidel) which effectively smooth out the error associated with the current approximate solution. The P_k ’s (interpolation operators that transfer solutions from coarse grids to finer grids) are the key ingredient that are determined automatically by the algebraic multigrid method². For the purposes of this guide, it is important to understand that when the multigrid method is used, a hierarchy of grids, grid transfer operators (P_k), and coarse grid discretizations (A_k) are created. To complete the specification of the multigrid method, smoothers must be supplied on each level. There are several smoothers within **ML** or an iterative solver package can be used, or users can write their own smoother (see Section 8).

4 Configuring and Building ML

ML is configured and built using the GNU autoconf [5] and automake [6] tools. It can be configured and build as a standalone package without or with AZTEC 2.1 support (as detailed in Section 4.1 and 4.2), or as a part of the TRILINOS framework [9] (as described in Section 4.3). Even though **ML** can be compiled and used as a standalone package, the recommended approach is to build **ML** as part of the TRILINOS framework, as a richer set of features are then available.

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

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

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

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

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

4.1 Building in Standalone Mode

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

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

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

4.2 Building with AZTEC 2.1 Support

To enable the supports for AZTEC 2.1, **ML** must be configured with the options reported in the previous section, plus `--with-ml_aztec2.1` (defaulted to `no`).

All of the AZTEC 2.1 functionality that **ML** accesses is contained in the file `ml_aztec_utils.c`. In principal by creating a similar file, other solver packages could work with **ML** in the same way. For the AZTEC users there are essentially three functions that are important. The first is `AZ_ML_Set_Amat` which converts AZTEC matrices into **ML** matrices by making appropriate **ML** calls (see Section 11.1 and Section 11.2). It is important to note that when creating **ML** matrices from AZTEC matrices information is not copied. Instead, wrapper functions are made so that **ML** can access the same information as AZTEC. The second is `ML_Gen_SmootherAztec` that is used for defining AZTEC iterative methods as smoothers (discussed in Section 8. The third function, `AZ_set_ML_preconditioner`, can be invoked to set the AZTEC preconditioner to use the multilevel 'V' cycle constructed in **ML**. Thus, it is possible to invoke several instances of AZTEC within one solve: smoother on different multigrid levels and/or outer iterative solve.

4.3 Building with TRILINOS Support (RECOMMENDED)

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

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

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

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

4.3.1 Enabling Third Party Library Support

ML can be configured with the following third party libraries (TPLs): SUPERLU, SUPERLU_DIST, PARASAILS, METIS, and PARMETIS. It can take advantage of the following TRILINOS packages: IFPACK, TEUCHOS, TRIUTILS, AMESOS. Through AMESOS, **ML** can interface with the direct solvers KLU, UMFPACK, SUPERLU, SUPERLU_DIST³, MUMPS. It is assumed that you have already built the appropriate libraries (e.g., `libsuperlu.a`) and have the header files. To configure **ML** with one of the above TPLs, you must enable the particular TPL interface in **ML**. All of the options below are disabled by default.

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

| | |
|--------------------------------|--|
| <code>--enable-epetra</code> | Enable support for the EPETRA package. |
| <code>--enable-aztecoo</code> | Enable support for the AZTECOO package. |
| <code>--enable-amesos</code> | Enables support for the AMESOS package. AMESOS is an interface with several direct solvers. ML supports UMFPACK [3], KLU, SUPERLU_DIST (1.0 and 2.0), MUMPS [15]. This package is used only in function <code>ML_Gen_SmoothingAmesos</code> . |
| <code>--enable-teuchos</code> | Enables support for the TEUCHOS package. This package is used only in the definition of class <code>MLEpetra::MultiLevelPreconditioner</code> (see Section 5). and by the Amesos smoother |
| <code>--enable-triutils</code> | Enables support for the TRIUTILS package. ML uses TRIUTILS only in some examples, to create the linear system matrix. |
| <code>--enable-ifpack</code> | Enable support for the IFPACK package [10]. IFPACK is used only to create smoothers via <code>ML_Gen_SmoothingIfpack</code> . |

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

| | |
|-------------------------------|--|
| <code>--enable-anasazi</code> | Enable support for the ANASAZI package. ANASAZI is a high level interface package for various eigenvalue computations. |
|-------------------------------|--|

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

| | |
|-------------------------------------|---|
| <code>--with-ml_parasails</code> | Enables interface for PARASAILS [2]. |
| <code>--with-ml_metis</code> | Enables interface for METIS [13]. |
| <code>--with-ml_parmetis2x</code> | Enables interface for PARMETIS, version 2.x. |
| <code>--with-ml_parmetis3x</code> | Enables interface for PARMETIS [12], version 3.x. |
| <code>--with-ml_superlu</code> | Enables ML interface for serial SUPERLU [4]. The ML interface to SUPERLU is deprecated in favor of the AMESOS interface. |
| <code>--with-ml_superlu_dist</code> | Enables ML interface for SUPERLU_DIST [4]. The ML interface to SUPERLU_DIST is deprecated in favor of the AMESOS interface. |

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

`--with-incdirs=include-locations`

(Header files for TRILINOS libraries are automatically located if **ML** is built through the TRILINOS configure.) In order to link the **ML** examples, the user must indicate the location of all the enabled packages' libraries⁴, with the option

`--with-ldflags=lib-locations`

The user might find useful the option

`--disable-examples`

which turns off compilation and linking of the examples.

More details about the installation of TRILINOS can be found at the TRILINOS web site, <http://software.sandia.gov/Trilinos> and [11, Chapter 1].

4.3.2 Enabling Profiling

All of the options below are disabled by default.

| | |
|---------------------------------|---|
| <code>--enable-ml_timing</code> | This prints out timing of key ML routines. |
|---------------------------------|---|

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

`--enable-ml_flops`

This enables printing of flop counts.

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

5 ML and Epetra: Getting Started with the MultiLevelPreconditioner Class

In this Section we show how to use **ML** as a preconditioner to EPETRA and AZTECOO through the MultiLevelPreconditioner class⁵ in the ML_Epetra namespace.⁶ Although limited to algebraic multilevel preconditioners, this allows the use of **ML** as a black-box preconditioner.

The MultiLevelPreconditioner class automatically constructs all the components of the preconditioner, using the parameters specified in a TEUCHOS parameter list. The constructor of this class takes as input an Epetra_RowMatrix pointer and a TEUCHOS parameter list⁷.

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.

5.1 Example 1: `ml_example_epetra_preconditioner.cpp`

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

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

...

Teuchos::ParameterList MList;

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

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

⁵The MultiLevelPreconditioner class is derived from the Epetra_RowMatrix class.

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

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

```

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

// create an Aztec00 solver
Aztec00 Solver(Problem)

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

...

delete MLPrec;

```

We now detail the general procedure to define the MultiLevelPreconditioner. First, the user defines a TEUCHOS parameter list⁸. Table 2 briefly reports the most important methods of this class.

| | |
|---------------------------------|--|
| <code>set(Name,Value)</code> | Add entry <code>Name</code> with value and type specified by <code>Value</code> . Any C++ type (like <code>int</code> , <code>double</code> , a pointer, etc.) is valid. |
| <code>get(Name,DefValue)</code> | Get value (whose type is automatically specified by <code>DefValue</code>). If not present, return <code>DefValue</code> . |
| <code>subList(Name)</code> | Get a reference to sublist <code>List</code> . 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 containing 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 MultiLevelPreconditioner is reported in Section 6.

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

```

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

// compute the elements of A
...

```

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

⁹We suggest to create the preconditioning object with `new` and to free memory with `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 (if required) the preconditioner
MLPrec->ComputePreconditioner();

// 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 [14]).

5.2 Example 2: `ml_example_epetra_preconditioner_2level.cpp`

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

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

The AztecOO linear problem is defined as

```

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

```

We create the TEUCHOS parameter list as follows:

```

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

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

```

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

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

The last two commands set the pointer to **options** and **params** in the parameter list¹⁰.

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 the **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**.

6 Parameters for the **ML_Epetra::MultiLevelPreconditioner** Class

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

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

6.1 Setting Options on a Specific Level

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

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

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

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

6.2 General Usage of the Parameter List

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

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

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

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

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

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

6.3 Default Parameter Settings for Common Problem Types

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

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

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

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

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

or as

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

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

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

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

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

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

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

| | |
|---------------|--|
| Uncoupled | Attempts to construct aggregates of optimal size (3^d nodes in d dimensions). Each process works independently, and aggregates cannot span processes. |
| Coupled | As Uncoupled , but aggregates can span processes (deprecated). |
| MIS | Uses a maximal independent set technique to define the aggregates. Aggregates can span processes. May provide better quality aggregates than either Coupled or uncoupled . Computationally more expensive than either because it requires matrix-matrix product. |
| Uncoupled-MIS | Uses Uncoupled for all levels until there is 1 aggregate per processor. Then switches over to MIS. The coarsening scheme on a given level cannot be specified with this option. |
| METIS | Use a graph partitioning algorithm to creates the aggregates, working process-wise. The number of nodes in each aggregate is specified with the option aggregation: nodes per aggregate . Requires ML to be configured with --with-mlmetis . |
| ParMETIS | As METIS , but partition the global graph. Requires --with-mlparmetis2x or --with-mlparmetis3x . Aggregates can span arbitrary number of processes. Global number of aggregates can be specified with the option aggregation: global number . |

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

6.4 Commonly Used Parameters

Table 4 lists parameter for changing aggregation schemes. Table 5 lists common choices for smoothing options. Table 6 lists common choices affecting the coarse grid solve.

Note that, in the parameters name, spaces are important: Do not include non-required leading or trailing spaces, and separate words by just one space! Misspelled parameters will not be detected. One may find useful to print unused parameters by calling `PrintUnused()` *after* the construction of the multilevel hierarchy.

6.5 List of All Parameters for MultiLevelPreconditioner Class

6.5.1 General Options

| | |
|---------------------------------------|---|
| <code>output</code> | Output level, from 0 to 10 (10 being verbose). |
| <code>print unused</code> | If non-negative, will print all the unused parameter on the specified processor. |
| <code>max levels</code> | Maximum number of levels. |
| <code>increasing or decreasing</code> | If set to increasing , level 0 will correspond to the finest level. If set to decreasing , max levels - 1 will correspond to the finest level. |

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

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

| | |
|--------------------|--|
| Jacobi | Use <code>coarse: sweeps</code> steps of Jacobi (with damping parameter <code>coarse: damping parameter</code>) as a solver. |
| Gauss-Seidel | Use <code>coarse: sweeps</code> steps of Gauss-Seidel(with damping parameter <code>coarse: damping parameter</code>) as a solver. |
| Amesos-KLU | Use KLuthrough AMESOS. Coarse grid problem is shipped to proc 0, solved, and solution is broadcast |
| Amesos-UMFPACK | Use UMFPACKthrough AMESOS. Coarse grid problem is shipped to proc 0, solved, and solution is broadcasted. |
| Amesos-Superludist | Use SUPERLU_DISTthrough AMESOS. |
| Amesos-MUMPS | Use double precision version of MUMPS through AMESOS. |
| SuperLU | Use ML interface to SUPERLU. |

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

PDE equations

Number of PDE equations for each grid node. This value is not considered for `Epetra_VbrMatrix` objects, as in this case is obtained from the block map used to construct the object. Note that only block maps with constant element size can be considered.

null space dimension

Dimension of the null space.

null space vectors

Pointer to the null space vectors. If `NULL`, **ML** will use the default null space.

6.5.2 Aggregation Parameters

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

6.5.3 Smoothing Parameters

| | |
|---------------------------------|-------------------------------|
| <code>smoother: sweeps *</code> | Number of sweeps of smoother. |
|---------------------------------|-------------------------------|

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

6.5.4 Coarsest Grid Parameters

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

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

6.6 Null Space Detection

If used in conjunction with ANASAZI, **ML** can compute an approximate null space of the linear operator to precondition. This can be done using the following options:

| | |
|--|---|
| <code>null space: type</code> | [string] If <code>default vectors</code> , the default null space are used (no computation required). If <code>pre-computed</code> , a pointer to the already-computed null is obtained from option <code>null space: vector</code> . If <code>enriched</code> , ML will compute, using ANASAZI, an approximation of the null space of the operator. |
| <code>null space: vectors to compute</code> | [int]. If <code>null space: type</code> is set to <code>enriched</code> , this option indicates the number of eigenvectors to compute. Default: 1. |
| <code>null space: add default vectors</code> | [bool]. If <code>true</code> , the default null space (one constant vector for each unknown) will be added to the computed null space. Default: <code>true</code> . |
| <code>eigen-analysis: tolerance</code> | [double] Tolerance for ANASAZI. Default: 0.1. |
| <code>eigen-analysis: restart</code> | [int] Number of restarts for ANASAZI. Default: 2. |

6.7 Aggregation Strategies

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

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

where \mathbf{x}_i represents the coordinates of node i . This may happen, for instance, when working with stretched elements. **ML** can be instructed to generate the aggregates using matrix (2), then build the preconditioner using the actual linear system matrix, as done in the following code fragment:

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

```

MLList.set("aggregation: x-coordinates", x_coord );
MLList.set("aggregation: y-coordinates", y_coord );
MLList.set("aggregation: z-coordinates", z_coord );
MLList.set("aggregation: use auxiliary matrix", true);

```

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 following options can be used:

`aggregation: use auxiliary matrix [bool]`. Enable/disable the use of matrix (2). Default: `false`.

`aggregation: use symmetric pattern[bool]`. If `true`, matrix (2) will have a symmetric pattern. Default: `false`.

`aggregation: theta [double]`. If θ is different from zero, the code create the aggregates using matrix $\theta A + (1-\theta)L$. Default: 0.0.

6.8 Analyzing the ML preconditioner

A successful multilevel preconditioner require the careful choice of a large variety of parameters, from each level's smoother, to the aggregation scheme, and a lot of others others. Often, for non-standard problems, there is no theory to support this choice. 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 `ml_example_MultiLevelPreconditioner_viz.cpp` shows how to visualize the effect of the **ML** cycle and each level's smoother on a random vector;
- File `ml_example_MultiLevelPreconditioner_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 preconditioned computed.

6.8.1 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
Minimum 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.

6.8.2 Eigenvalue Computations

ML can be used to compute the eigenvalues of the non-preconditioned and the preconditioned finest-level matrix. In the following fragment of code, we compute all the eigenvalues of the finest-level matrix A , and all the eigenvalues of $P^{-1}A$, where P is the already computed **ML** preconditioner:

```

// MLPrec is a ML_Epetra::MultiLevelPreconditioner pointer
// first compute the eigenvalues of the finest-level matrix A
MLPrec->AnalyzeMatrixEigenvaluesDense("A");

// now the eigenvalues of A, with the ML preconditioner
MLPrec->AnalyzeMatrixEigenvaluesDense("P^{-1}A");

```

On standard output, the code will print out information of this kind:


```

-----
*** *****
*** Analysis of the spectral properties using LAPACK ***
*** *****
*** Operator = A

    *** Computing eigenvalues of finest-level matrix
    *** using LAPACK. This may take some time...
    *** results are on file 'eig_A.m'.

    min |lambda_i(A)|      = 0.155462
    max |lambda_i(A)|      = 26.3802
    spectral condition number = 169.689

** Total time = 0.305511(s)
-----

```

```

-----
*** *****
*** Analysis of the spectral properties using LAPACK ***
*** *****
*** Operator = P^{-1}A

    *** Computing eigenvalues of ML^{-1}A
    *** using LAPACK. This may take some time...
    *** results are on file 'eig_PA.m'.

    min |lambda_i(ML^{-1}A)| = 0.776591
    max |lambda_i(ML^{-1}A)| = 1.09315
    spectral condition number = 1.40762

** Total time = 0.730494(s)
-----

```

The eigenvalues of A are saved in file `eig_A.m`, while those of $P^{-1}A$ in file `eig_PA.m`. The file contains, on line i , the real and the imaginary component of the i -th eigenvalue, computed using LAPACK.

Figure 2 shows the eigenvalue distribution of the non-preconditioned and the preconditioned operator.

Method `AnalyzeMatrixEigenvaluesDense()` can be used only for serial runs. Alternatively, methods `AnalyzeMatrixEigenvaluesSparse()` can be used to compute the lowest-magnitude and the largest-magnitude of the spectrum. This method requires **ML** to be configured with support for ANASAZI (see Section 4.3.1). The following options, that must be set *before* the construction of the preconditioner, can be used to tune the sparse eigen-solver:

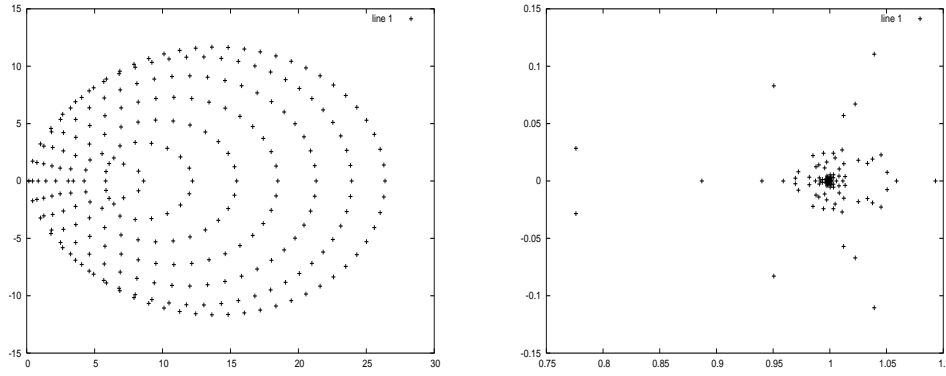


Figure 2: Eigenvalue distribution on the complex plane of the non-preconditioned (left) and the preconditioned operator (right), obtained by calling `AnalyzeMatrixEigenvaluesDense()`, and post-processing with `octave`.

| | |
|---|---|
| <code>analysis: eigenvalues</code> | [int] Number of eigenvalues to compute. Default = 128. |
| <code>analysis: tolerance</code> | [double] Tolerance for eigensolver. Default = 1e-2. |
| <code>analysis: anasazi restart</code> | [int] Maximum number of restarts. Default = 2. |
| <code>analysis: anasazi output</code> | [int] Output level, from 0 to 10 (10 being verbose). Default = 0. |
| <code>analysis: print anasazi status</code> | [bool] If true, prints out the status of the eigensolver after convergence. Default: false. |

This fragment of code computes the eigenvalues of A and $P^{-1}A$ using ANASAZI:

```
// first compute the eigenvalues of the finest-level matrix A
MLPrec->AnalyzeMatrixEigenvaluesSparse("A");

// now the eigenvalues of A, with the ML preconditioner
MLPrec->AnalyzeMatrixEigenvaluesSparse("P^{-1}A");
```

As for the dense case, the (computed) eigenvalues of A are saved in file `eig_A.m`, while those of $P^{-1}A$ in file `eig_PA.m`.

6.8.3 Analyze the Effect of Smoothers

For each level, method `AnalyzeSmoothersDense()` 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 `AnalyzeSmoothersDense(NumPre, NumPost)`. `NumPre` is the number of pre-smoother applications, and `NumPost` the number of post-smoother applications.

Method `AnalyzeSmoothersDense()` can be used only for serial runs. Alternatively, method `AnalyzeSmoothersSparse(NumPre, NumPost)` can be used. This function 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 |

6.8.4 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
```

6.8.5 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 `MultiLevelPreconditioner` 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:

| | |
|---|---|
| <code>test: max iters</code> | [int] Maximum number of iterations for the Krylov solver. Default: 500. |
| <code>test: tolerance</code> | [double] Tolerance for the Krylov solver. Default: 1e-5. |
| <code>test: Jacobi</code> | [bool] Enable/disable test with Jacobi smoother. Default: <code>true</code> . |
| <code>test: Gauss-Seidel</code> | [bool] Enable/disable test with Gauss-Seidel smoother. Default: <code>true</code> . |
| <code>test: symmetric Gauss-Seidel</code> | [bool] Enable/disable test with symmetric Gauss-Seidel smoother. Default: <code>true</code> . |
| <code>test: block Gauss-Seidel</code> | [bool] Enable/disable test with block Gauss-Seidel smoother. Default: <code>true</code> . |
| <code>test: Aztec</code> | [bool] Enable/disable test with AZTECOO smoother. Default: <code>true</code> . |
| <code>test: Aztec as solver</code> | [bool] Enable/disable test with AZTECOO as a solver smoother. Default: <code>true</code> . |
| <code>test: ParaSails</code> | [bool] Enable/disable test with PARASAILSSmoothing. Default: <code>true</code> . |
| <code>test: IFPACK</code> | [bool] Enable/disable test with IFPACKsmoothing. Default: <code>true</code> . |

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)
-----
```

6.8.6 Visualizing the Aggregates

ML offers the possibility to visualize the aggregates for all levels.

Data are stored on a file in a simple format. 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.) A visualization software, called XD3D¹¹ can be used to visualize files in this format, but only for 2D problems. Results can be as reported in figure 3.

6.8.7 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);
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
```

¹¹, More information about XD3D can be found at the web site <http://www.cmap.polytechnique.fr/~jouve/xd3d/index.php>.

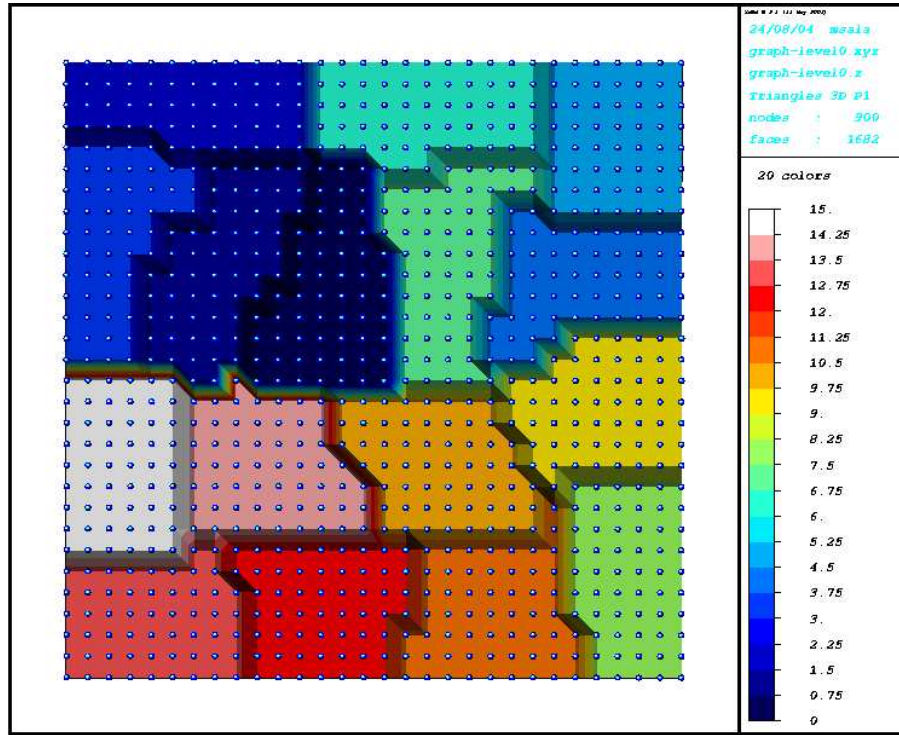


Figure 3: Decomposition into aggregates. Uniform colors between blue spheres represent the interior of aggregates.

```
// 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_example_MultiLevelPreconditioner_viz.cpp` contains the compilable code.) We note that the parameters must be set *before* calling `ComputePreconditioner()`. See also Section 6.7 for the requirements on the coordinates vectors. Results will be written in the following files:

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

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

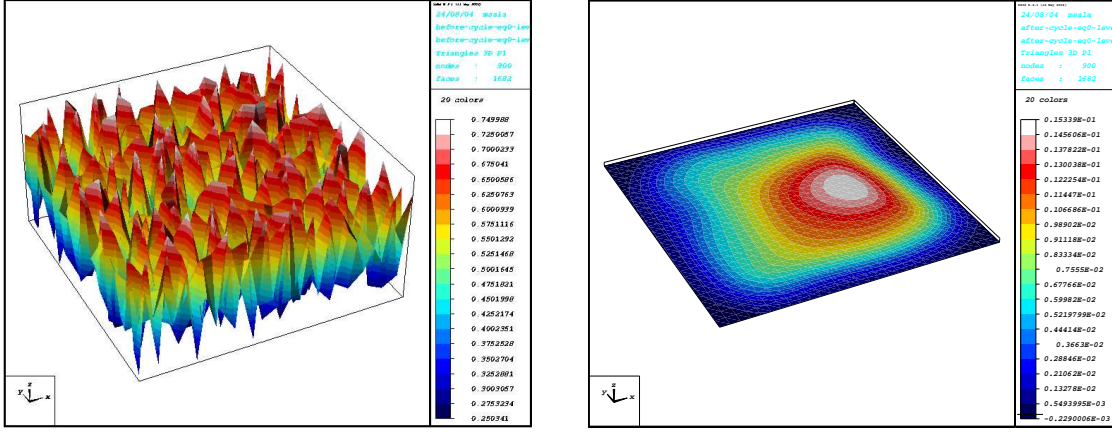


Figure 4: Starting solution (left) and solution after 10 **ML** cycles (right).

6.8.8 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; // 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 Advanced Usage of ML

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

A brief sample program is given in Figure 5. The function **ML_Create** creates a mul-

```
ML_Create          (&ml_object, N_grids);

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

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

Figure 5: High level multigrid sample code.

tilevel solver object that is used to define the preconditioner. It requires the maximum number of multigrid levels be specified. In almost all cases, **N_grids**= 20 is more than adequate. The three ‘Amatrix’ statements are used to define the discretization matrix, *A*, that is solved. This is discussed in greater detail in Section 11.1. The multigrid hierarchy is generated via **ML_Gen_MGHierarchy_UsingAggregation**. Controlling the behavior of this function is discussed in Section 9. For now, it is important to understand that this function takes the matrix *A* and sets up relevant multigrid operators corresponding to the smoothed aggregation multigrid method [19] [18]. In particular, it generates a graph associated with *A*, coarsens this graph, builds functions to transfer vector data between the original graph and the coarsened graph, and then builds an approximation to *A* on the coarser graph. Once this second multigrid level is completed, the same operations are repeated to the second level approximation to *A* generating a third level. This process continues until the current graph is sufficiently coarse. The function **ML_Gen_Smoother_Jacobi** indicates that a Jacobi smoother should be used on all levels. Smoothers are discussed further in Section 8. Finally, **ML_Gen_Solver** is invoked when the multigrid preconditioner is fully specified. This function performs any needed initialization and checks for inconsistent options. After **ML_Gen_Solver** completes **ML_Iterate** can be used to solve the problem with an initial guess of *sol* (which will be overwritten with the solution) and a right hand side of *rhs*. At the present time, the external interface to vectors are just arrays. That is, *rhs* and *sol* are simple one-dimensional arrays of the same length as the number of rows in *A*. In addition to **ML_Iterate**, the function **ML_Solve_MGV** can be used to perform one multigrid ‘V’ cycle as a preconditioner.

8 Multigrid & Smoothing Options

Several options can be set to tune the multigrid behavior. In this section, smoothing and high level multigrid choices are discussed. In the next section, the more specialized topic of the grid transfer operator is considered.

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

| | |
|--|--|
| <code>ML_Gen_Smoothing_Jacobi</code> | 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. |
| <code>ML_Gen_Smoothing_GaussSeidel</code> | Probably the most popular smoother. Typically, faster than Jacobi and damping is often not necessary nor advantageous. |
| <code>ML_Gen_Smoothing_SymGaussSeidel</code> | 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. |
| <code>ML_Gen_Smoothing_BlockGaussSeidel</code> | 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. |
| <code>ML_Gen_Smoothing_VBlockJacobi</code> | Variable block Jacobi smoother. This allows users to specify unknowns to be grouped into different blocks when doing block Jacobi. |
| <code>ML_Gen_Smoothing_VBlockSymGaussSeidel</code> | 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 [16] can invoke `ML_Gen_SmoothingAztec` 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_Smoothen`. The signature of the user defined smoother function is

```
int user_smoothing(ML_Smoothen *smoothen, int x_length, double x[],
                  int rhs_length, double rhs[])
```

where `smoothen` 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_MySmoothenData(smoothen)` can be used to get a pointer back to the user's data (i.e. the data pointer given with the `ML_Set_Smoothen` invocation). A simple (and suboptimal) damped Jacobi smoother for the finest grid of our example is given below:

```
int user_smoothing(ML_Smoothen *smoothen, 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_MySmoothenData(smoothen), 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_Smoothen`.

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

9 Smoothed Aggregation Options

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

9.1 Aggregation Options

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

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

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

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

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

There are two different groups of graph coarsening algorithms in **ML**:

- schemes with fixed ratio of coarsening between levels: uncoupled aggregation, coupled aggregation, and MIS aggregation. A description of those three schemes along with some numerical results are given in [17]. As the default, the **Uncoupled-MIS** scheme is used which does uncoupled aggregation on finer grids and switches to the more expensive MIS aggregation on coarser grids;

- schemes with variable ratio of coarsening between levels: METIS and PARMETISaggregation. Those schemes use the graph decomposition algorithms provided by METIS and PARMETIS, to create the aggregates.

Poorly done aggregation can adversely affect the multigrid convergence and the time per iteration. In particular, if the scheme coarsens too rapidly multigrid convergence may suffer. However, if coarsening is too slow, the number of multigrid levels increases and the number of nonzeros per row in the coarse grid discretization matrix may grow rapidly. We refer the reader to the above paper and indicate that users might try experimenting with the different schemes via `ML_Aggregate_Set_CoarsenScheme_Uncoupled`, `ML_Aggregate_Set_CoarsenScheme_Coupled`, `ML_Aggregate_Set_CoarsenScheme_MIS`, `ML_Aggregate_Set_CoarsenScheme_METIS`, and `ML_Aggregate_Set_CoarsenScheme_ParMETIS`.

9.2 Interpolation Options

An interpolation operator is built using coarsening information, seed vectors, and a damping factor. We refer the reader to [18] for details on the algorithm and the theory. In this section, we explain a few essential features to help users direct the interpolation process.

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

the graph. In some cases, however, information is known as to what components will be difficult for the smoother or what null space is associated with an operator. In elasticity, for example, it is well known that a floating structure has six rigid body modes (three translational vectors and three rotation vectors) that correspond to the null space of the operator. In this case, the logical choice is to take these six vectors as the seed vectors in smoothed aggregation. When this type of information is known, it should be given to **ML** via the command `ML_Aggregate_Set_NullSpace`.

Once the tentative prolongator is created, it is smoothed via a damped Jacobi iteration. The reasons for this smoothing are related to the theory where the interpolation basis functions must have a certain degree of smoothness (see [18]). However, the smoothing stage can be omitted by setting the damping to zero using the function `ML_Aggregate_Set_DampingFactor`. Though theoretically poorer, unsmoothed aggregation can have considerably less set up time and less cost per iteration than smoothed aggregation. When smoothing, **ML** has two ways to determine the Jacobi damping parameter and each require some estimate of the largest eigenvalue of the discretization operator. The current default is to use a few iterations of a conjugate-gradient method to estimate this value. However, if the matrix is nonsymmetric, the infinity norm of the matrix should be used instead via `ML_Aggregate_Set_SpectralNormScheme_Anorm`. There are several other internal parameters that have not been discussed in this document. In the future, it is anticipated that some of these will be made available to users.

10 Advanced Usage of ML and Epetra

Class `MLEpetra::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 7), created with the instruction

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

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

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

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

Let `agg_object` a `ML_Aggregate` pointer, created using

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

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

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

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

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

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

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

11 Using ML without Epetra

11.1 Creating a ML matrix: Single Processor

Matrices are created by defining some size information, a matrix-vector product and a `getrow` function (which is used to extract matrix information). We note that `EPETRA` and `AZTEC` users do not need to read this (or the next) section as there are special functions to convert `EPETRA` objects and `AZTEC` matrices to **ML** matrices (see Section 4.2). Further, functions for some common matrix storage formats (`CSR` & `MSR`) already exist within **ML** and do not need to be rewritten¹².

Size information is indicated via `ML_Init_Amatrix`. The third parameter in the Figure 5 invocation indicates that a matrix with `nlocal` rows is being defined. The fourth parameter gives the vector length of vectors that can be multiplied with this matrix. Additionally, a data pointer, `A_data`, is associated with the matrix. This pointer is passed back into the matrix-vector product and `getrow` functions that are supplied by the user. Finally, the number '0' indicates at what level within the multigrid hierarchy the matrix is to be stored. For discussions within this document, this is always '0'. It should be noted that there appears to be some redundant information. In particular, the number of rows and the vector length in `ML_Init_Amatrix` should be the same number as the discretization matrices are square. Cases where these 'apparently' redundant parameters might be set differently are not discussed in this document.

The function `ML_Set_Amatrix_Matvec` associates a matrix-vector product with the discretization matrix. The invocation in Figure 5 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

¹²The functions `CSR_matvec`, `CSR_getrows`, `MSR_matvec` and `MSR_getrows` can be used.

```
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 5 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 `i`th 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:

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}. \quad (3)$$

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.

11.2 Creating a ML matrix: Multiple Processors

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

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

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

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

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

ML has no notion of global indices and uses only the local indices. In most cases, another package or application already mapped the global indices to local indices and so **ML** works with the existing local indices. Specifically, the parallel version of `user_getrow` and `user_matvec` should correspond to each processor's local matrix. This means that when giving the column information with `ML_Set_Amatrix_Getrow`, the total number of columns in the local matrix should be given and that when row k is requested, `user_getrow` should return the k^{th} local row using local column indices. Likewise, the matrix-vector product takes a local input vector and multiplies it by the local matrix. It is important to note that this local input vector does not contain ghost node data (i.e. the input vector is of length `nlocal` where `nlocal` is the number of matrix rows). Thus, `user_matvec` must perform the necessary communication to update ghost variables. When invoking `ML_Init_Amatrix`, the local number of rows should be given for the number of rows and the vector length¹³. 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[])
{
    int m = 0, i, row, proc, *itemp, start;

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

¹³In contrast to `ML_Set_Amatrix_Getrow` in which the number of local columns are given (including those that correspond to ghost variables), `ML_Init_Amatrix` does not include ghost variables and so both size parameters should be the number of local rows.

```

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

int Poisson_matvec(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);

```

References

- [1] A. BRANDT, *Multi-level Adaptive Solutions to Boundary-Value Problems*, Math. Comp., 31 (1977), pp. 333–390.
- [2] E. CHOW, *ParaSails User’s Guide*, 2000.
- [3] T. DAVIS, *UMFPACK home page*. <http://www.cise.ufl.edu/research/sparse/umfpack>, 2003.
- [4] J. W. DEMMEL, J. R. GILBERT, AND X. S. LI, *SuperLU Users’ Guide*, 2003.
- [5] FREE SOFTWARE FOUNDATION, *Autoconf Home Page*. <http://www.gnu.org/software/autoconf>.
- [6] ———, *Automake Home Page*. <http://www.gnu.org/software/automake>.
- [7] W. HACKBUSCH, *Multi-grid Methods and Applications*, Springer-Verlag, Berlin, 1985.
- [8] ———, *Iterative Solution of Large Sparse Linear Systems of Equations*, Springer-Verlag, Berlin, 1994.
- [9] M. A. HEROUX, *Trilinos home page*. <http://software.sandia.gov/trilinos>.
- [10] M. A. HEROUX, *IFPACK Reference Manual*, 2.0 ed., 2003. <http://software.sandia.gov/trilinos/packages/ifpack/doxygen/latex/IfpackReferenceManual.pdf>.

- [11] M. M. A. HEROUX, *Trilinos Tutorial*, 3.1 ed., 2004.
- [12] G. KARYPIS AND V. KUMAR, *ParMETIS: Parallel graph partitioning and sparse matrix ordering library*, Tech. Rep. 97-060, Department of Computer Science, University of Minnesota, 1997.
- [13] —, *METIS: Unstructured graph partitioning and sparse matrix ordering system*, tech. rep., University of Minnesota, Department of Computer Science, 1998.
- [14] T. G. KOLDA AND R. P. PAWLOWSKI, *Nox home page*. <http://software.sandia.gov/nox>.
- [15] I. S. D. P. R. AMESTOY AND J.-Y. L'EXCELLENT, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods in Appl. Mech. Eng., (2000), pp. 501–520.
- [16] 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.
- [17] R. TUMINARO AND C. TONG, *Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines*, in SuperComputing 2000 Proceedings, J. Donnelley, ed., 2000.
- [18] P. VANEK, M. BREZINA, AND J. MANDEL, *Convergence of Algebraic Multigrid Based on Smoothed Aggregation*, Tech. Rep. report 126, UCD/CCM, Denver, CO, 1998.
- [19] 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.