SANDIA REPORT

SAND2004-0154
Unlimited Release
Printed January 2004

Trilinos 3.1 Tutorial

Marzio Sala and Michael Heroux

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-0154
Unlimited Release
Printed January 2004

Trilinos 3.1 Tutorial

Marzio Sala and Michael Heroux
Computational Mathematics and Algorithms Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

This document introduces the use of Trilinos, version 3.1. Trilinos has been written to support, in a rigorous manner, the solver needs of the engineering and scientific applications at Sandia National Laboratories.

Aim of this manuscript is to present the basic features of some of the Trilinos packages. The presented material includes the definition of distributed matrices and vectors with Epetra, the iterative solution of linear system with AztecOO, incomplete factorizations with IFPACK, multilevel methods with ML, direct solution of linear system with Amesos, and iterative solution of nonlinear systems with NOX. With the help of several examples, some of the most important classes and methods are detailed to the unexperienced user. For the most majority, each example is largely commented throughout the text. Other comments can be found in the source of each example.

This document is a companion to the Trilinos User's Guide [10] and Trilinos Development Guides [11, 12]. Also, the documentation included in each of the Trilinos' packages is of fundamental importance.

Acknowledgments

The authors would like to acknowledge the support of the ASCI and LDRD programs that funded development of Trilinos.

Trilinos 3.1 Tutorial

Contents

1 Introduction

The Trilinos Project is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. Goal of the Trilinos Project is develop parallel solver algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific applications. The emphasis is on developing robust, scalable algorithm in a software framework, using abstract interfaces for flexible interoperability of components while providing a full-featured set of concrete classes that implement all abstract interfaces.

1.1 Getting Started with Trilinos

The Trilinos Project uses a two-level software structure designed around collections of packages. A Trilinos package is an integral unit, usually developed to solve a specific task, by a (relatively) small group of expert of the field. Packages exist underneath the Trilinos top level, which provides a common look-and-feel. Each package has its own structure, documentation and set of examples. In principle, Trilinos packages can live independently. However, each package is even more valuable when combined with other Trilinos packages.

Trilinos is a large software project, and currently about twenty packages are included. Fully understanding all the functionalities of the Trilinos packages requires time. The entire set of packages covers a wide range of numerical methods for large scale computing. Some packages are focused on the development of computational schemes, like for instance the solution of linear and nonlinear systems, to the definition of parallel preconditioners for Krylov methods, eigenvalue computation. Other packages are more focused on implementation issues (like definition of matrices and vectors, abstract classes for linear operators). The first Chapters of this tutorial will be focused on implementation issues, while the last Chapters will have a more "mathematical" taste.

Each package offers sophisticated features, that cannot be "unleashed" at a very first usage. For each package, we will outline only the basic features, and we refer to the documentation of each package for a more involved usage. Our goal is to present enough material so that the reader can successfully use the described packages. In fact, for new users, it is neither easy, nor necessary, to manage all the Trilinos functionalities. At the beginning, it is more important for them to understand how to manage the basic classes, such as vector, matrix and linear system classes. However, it is clear that for a fine-tuning, the reader will have to look through each package's documentation and examples.

Although all packages have the same importance in the Trilinos structure, a typical user will probably — at least at the beginning — make use of the following packages:

- **Epetra**. This package defines the basic classes for distributed matrices and vectors, linear operators and linear problems. Epetra classes are the common language spoken by all the Trilinos packages (even if some of them can "speak" other languages). Each Trilinos package is able to accept in input Epetra objects. This allows powerful combinations among the various Trilinos functionalities.
- **AztecOO**. This is a linear solve package based on preconditioned Krylov methods. It supports all the Aztec interfaces and functionality, but also provides significant new functionality.
- **IFPACK**. This is a package to perform various incomplete factorizations, and it is here used in conjunction with AztecOO.
- **ML**. This is an algebraic multilevel preconditioner package, which provided scalable preconditioning capabilities for a variety of problem classes. It is here used in conjunction with AztecOO.
- **Amesos**. This package provides a common interface to various direct solvers (generally available outside the Trilinos framework), both sequential and parallel.
- **NOX**. This is a collection of nonlinear solvers, designed to be easily integrated into an application and used with many different linear solvers.
- **Triutils**. This is a collection of various utilities, that can be extremely useful in some phases of software development.

Table 1 gives a partial overview of what can be accomplished using Trilinos.

This tutorial is divided into 10 chapters:

- Chapter 2 describes the Epetra_Vector class;
- Chapter 3 introduces the Epetra_Matrix class;
- Chapter 4 briefly describes some other Epetra classes;
- Chapter 5 shows how to solve linear systems with AztecOO;
- Chapter 6 presents the basic usage of IFPACK;

Task	Package
Light-weight interface to BLAS and LAPACK:	Epetra, Teuchos*
Definition of serial dense or sparse matrices:	Epetra
Definition of distributed sparse matrices:	Epetra
solve a linear system with preconditioned Krylov accelerators, like CG, GMRES, Bi-CGSTAB, TFQMR:	AztecOO, Belos*
Definition of incomplete factorizations:	AztecOO, IFPACK
Definition of a multilevel preconditioner:	ML
Definition of a one-level Schwarz preconditioner (overlapping domain decomposition):	AztecOO, IFPACK
Definition a two-level Schwarz preconditioner, with coarse grid based on aggregation:	AztecOO+ML
Solution of systems of nonlinear equations:	NOX
interface with various direct solvers, as UMFPACK, MUMPS, SuperLU and others :	Amesos
Computation of eigenvalue of large, sparse matrices:	Anasazi*
Solution of complex linear equations (using equivalent real formulation):	Komplex*
Definition of segregated preconditioners and block preconditioners (for instance, for the incompressible Navier-Stokes equations):	Meros*
Templated interface to BLAS and LAPACK, arbitrary-precision arithmetic, parameter lists:	Teuchos*
Definition of abstract interfaces to vectors, linear operators, and solvers:	TSF*, TSFCore*, TSFExtended*

Table 1. Partial overview of what can be done with Trilinos. $\star$: not covered in this tutorial.

- Chapter 7 introduces multilevel preconditioners based on ML;
- Chapter 8 introduces the Amesos package;
- Chapter 9 outlines the main features of the Trilinos nonlinear solver package, NOX.
- Chapter 10 presents some tools provided with the Triutils package.

Remark 1. *As already pointed out, Epetra objects are meant to be the "common language" spoken by all the Trilinos packages, and therefore the new user must become familiar with those objects. Therefore we suggest to read Chapters 2-4 before considering other Trilinos packages. Also, Chapter 5 should be read before Chapters 6 and 7 (even if both IFPACK and ML can be compiled and run without AztecOO).*

This tutorial assume a basic background in numerical methods for PDEs, and in iterative linear and nonlinear solvers. Although not strictly necessary, the reader is suppose to have a certain familiarity with distributed memory computing and, to a minor extent, with MPI.

Note that this tutorial is not a substitute ofr individual packages documentation. Also, for an overview of all the Trilinos packages, the Trilinos philosophy, and a description of the packages provided by Trilinos, the reader is referred to [7]. Developers should also consider the Trilinos Developers' Guide, which addresses many topics, including the development tools used by Trilinos' developers, and how to include a new package[1].

1.2 Installing Trilinos

To obtain Trilinos, please refers to the instructions reported at the following web site:

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

Trilinos has been compiled on a variety of architectures, including Linux, Sun Solaris, SGI Irix, DEC, and many others. Trilinos has been designed to support parallel applications. However, it can be compiled and run on serial computer. Detailed comments on the installation, and an exhaustive list of FAQs, can be found at the web pages:

[1]Trilinos provides a variety of services to a developer wanting to integrate a package into Trilinos. They include Autoconf [1], Automake [2] and Libtool [3]. Those tools provide a robust, full-featured set of tools for building software across a broad set of platforms. Although these tools are not official standards, they are widely used. All existing Trilinos packages use Autoconf and Automake. Libtool support will be added in future releases.

```
http://software.sandia.gov/Trilinos/installing_manual.html
http://software.sandia.gov/Trilinos/faq.html
```

Before using Trilinos, users might decide to set the environmental variables `TRILINOS_HOME`, indicating the full path of the Trilinos directory, `TRILINOS_LIB`, indicating the location of the compiled Trilinos library, and `TRILINOS_ARCH`, containing the architecture and the communicator currently used. For example, using the BASH shell, command lines of the form

```
export TRILINOS_HOME=/home/msala/Trilinos
export TRILINOS_ARCH=LINUX.MPI
export TRILINOS_LIB=${TRILINOS_HOME}/${TRILINOS_ARCH}
```

can be places in the users' `.bashrc` file.

Here, we briefly report the procedure one should follow in order to compile Trilinos as required by the examples reported in the following chapters 2-10[2]. Suppose we want to compile under LINUX with MPI. The installation procedure can be are reported below. ($ indicates the shell prompt.)

```
$ cd ${TRILINOS_HOME}
$ mkdir ${TRILINOS_ARCH}
$ cd ${TRILINOS_ARCH}
$ ../configure --prefix="${TRILINOS_HOME}/${TRILINOS_ARCH}" \
  --enable-mpi --with-mpi-compilers \
  --enable-triutils --enable-aztecoo \
  --enable-ifpack \
  --enable-ml --enable-nox | tee configure_${TRILINOS_ARCH}.log
$ make | tee make_${TRILINOS_ARCH}.log
$ make install | tee make_install_${TRILINOS_ARCH}.log
```

Remark 2. *All Trilinos packages can be build to run with or without MPI. If MPI is enabled (using* `--enable-mpi`*), the users must know the procedure for beginning MPI jobs on their computer system(s). In some cases, options must be set on the configure line to specify the location of MPI include files and libraries.*

[2]Amesos can be more difficult to compile for the unexperienced user, as it required some information about the packages to interface. Suggestions about the configuration of Amesos are reported in Chapter 8. More details about the installation of Trilinos can be found in [10].

1.3 Compiling and Linking a program using Trilinos

In order to compile and link (part of) the Trilinos library, the use can decide to use a Makefile as reported below. This Makefile refers to one of the examples, reported in the NOX subdirectory of this tutorial.

```
 1: TRILINOS_HOME = /home/msala/Trilinos/
 2: TRILINOS_ARCH - LINUX_MPI
 3: TRILINOS_LIB = $(TRILINOS_HOME)$(TRILINOS_ARCH)
 4:
 5: include $(TRILINOS_HOME)/build/makefile.$(TRILINOS_ARCH)
 6:
 7: MY_COMPILER_FLAGS = -DHAVE_CONFIG_H $(CXXFLAGS) -c -g\
 8:                     -I$(TRILINOS_LIB)/include/
 9:
10: MY_LINKER_FLAGS = $(LDFLAGS) $(TEST_C_OBJ) \
11:          -L$(TRILINOS_LIB)/lib/ \
12:          -lnoxepetra -lnox -lifpack \
13:          -laztecoo -lepetra -llapack -lblas $(ARCH_LIBS)
14:
15: ex1: ex1.cpp
16:          $(CXX)      ex1.cpp $(MY_COMPILER_FLAGS)
17:          $(LINKER)   ex1.o   $(MY_LINKER_FLAGS)     -o ex1.exe
```

Line number have been reported for reader's convenience.

The lines 1-3 can be omitted, see Section 1.2. Line 5 includes basic definitions of Trilinos. (Note that, on some architectures, one may need to use `gmake` instead of `make`.) In line 7, the variable `HAVE_CONFIG_H` is defined. Linker flags of lines 10-13 defines the library to link (location of BLAS and LAPACK can change on different platforms). The variable `ARCH_LIBS` is defined in line 5.

To run the compiled example in a sequential environment, simply type

```
$ ./ex1.exe
```

In a MPI environment, the user might have to use an instruction of type

```
$ mpirun -np 2 ./ex1.exe
```

Please check the local MPI documentation for more details.

1.4 Copyright and Licensing of Trilinos

Trilinos is released under the Lesser GPL GNU Licence.

Trilinos is copyrighted by Sandia Corporation. Under the terms of Contract DE-AC04-94AL85000, there is a non-exclusive license for use of this work by or on behalf of the U.S. Government. Export of this program may require a license from the United States Government.

NOTICE: The United States Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in ths data to reproduce, prepare derivative works, and perform publicly and display publicly. Beginning five (5) years from July 25, 2001, the United States Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in this data to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so.

NEITHER THE UNITED STATES GOVERNMENT, NOR THE UNITED STATES DEPARTMENT OF ENERGY, NOR SANDIA CORPORATION, NOR ANY OF THEIR EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

Some parts of Trilinos are dependent on a third party code. Each third party code comes with its own copyright and/or licensing requirements. It is responsibility of the user to understand these requirements.

1.5 Programming Language Used in this Tutorial

Trilinos is written in C++ (for most packages), and in C. Some interfaces are provided to FORTRAN code (mainly BLAS and LAPACK routines). Even if a limited support is included for C programs (and a more limited for FORTRAN code), to unleashed the full power of Trilinos we suggest to use C++. All the example programs contained in this tutorial will be in C++; some packages contains examples in C.

1.6 Referencing Trilinos

The Trilinos project can be referenced by using the following BiBTeX citation information:

```
@techreport{Trilinos-Overview,
title = "{An Overview of Trilinos}",
author = "Michael Heroux and Roscoe Bartlett and Vicki Howle
Robert Hoekstra and Jonathan Hu and Tamara Kolda and
Richard Lehoucq and Kevin Long and Roger Pawlowski and
Eric Phipps and Andrew Salinger and Heidi Thornquist and
Ray Tuminaro and James Willenbring and Alan Williams ",
institution = "Sandia National Laboratories",
number = "SAND2003-2927",
year = 2003}

@techreport{Trilinos-Dev-Guide,
title = "{Trilinos Developers Guide}",
author = "Michael A. Heroux and James M. Willenbring and Robert Heaphy",
institution = "Sandia National Laboratories",
number = "SAND2003-1898",
year = 2003}

@techreport{Trilinos-Dev-Guide-II,
title = "{Trilinos Developers Guide Part II: ASCI Software Quality
Engineering Practices Version 1.0}",
author = "Michael A. Heroux and James M. Willenbring and Robert Heaphy",
institution = "Sandia National Laboratories",
number = "SAND2003-1899",
year = 2003}

@techreport{Trilinos-Users-Guide,
title = "{Trilinos Users Guide}",
author = "Michael A. Heroux and James M. Willenbring",
institution = "Sandia National Laboratories",
number = "SAND2003-2952",
year = 2003}
```

These BiBTeX information can be downloaded from the web page

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

1.7 A Note on Directory Structure

Each Trilinos package in contained in the subdirectory

```
${TRILINOS_HOME}/packages
```

The structure of all packages is quite similar (although not exactly equal). As a general line, source files are in

```
${TRILINOS_HOME}/packages/<package-name>/src
```

Example files are reported in

```
${TRILINOS_HOME}/packages/<package-name>/examples
```

and test files in

```
${TRILINOS_HOME}/packages/<package-name>/test
```

The documentation is reported

```
${TRILINOS_HOME}/packages/<package-name>/doc
```

Often, Trilinos developers use Doxygen[3]. For instance, to create the documentation for Epetra, we use can type

```
$ cd ${TRILINOS_HOME}/packages/epetra/doc
$ doxygen Doxyfile
```

and then browse it using an HTML reader, or compiling the LaTeXfile using

```
$ cd ${TRILINOS_HOME}/packages/epetra/doc/latex
$ make
```

[3]Copyright ©1997-2003 by Dimitri van Heesch. More information can by found at the web address `http://www.stack.nl/ dimitri/doxygen/`.

1.8 List of Trilinos Developers

A list of the Trilinos' developers, updated to December 2003, would include the following names (in alphabetical order):

Roscoe A. Bartlett, Jason A. Cross, David M. Day, Robert Heaphy, Michael A. Heroux (project leader), Russell Hooper, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Paul Lin, Kevin R. Long, Roger P. Pawlowski, Michael N. Phenow, Eric T. Phipps, Andrew J. Rothfuss, Marzio Sala, Andrew G. Salinger, Paul M. Sexton, Kendall S. Stanley, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams.

2 Working with Epetra Vectors

Probably, the first mathematical entities defined by a numerical method is a vector. Within the Trilinos framework, vectors are usually constructed starting from Epetra Classes.

Epetra vectors can be used to store double values (like the solution of a PDE problem, the right-hand side of a linear system, or the nodal coordinates), as well as integer data values (such as a set of indexes).

Epetra vectors can be *serial* or *distributed*. Serial vectors are usually small, so that it is not convenient to distribute them across the processes. Possibly, serial vectors are replicated across the processes. On the other hand, distributed vectors tend to be significantly larger, and therefore their elements are distributed across the processors. In this latter case, users must specify the partition they intend to use. In Epetra, this is done by specifying a communicator (introduced in Section 2.1) and an Epetra object called map (introduced in Section 2.2). A map is basically a partitioning of a list of global IDs.

This Chapter will show some of the Trilinos capabilities to work with vectors. Vector classed can be used to perform common vector operations, as dot products, vector scalings and norms, or fill with constant or random values.

During the Chapter, the user be introduced to:

- The Epetra_Comm object (in Section 2.1);
- The Epetra_Map object (in Section 2.2);
- Creating and assembling Epetra vectors (in Sections 2.3 and 2.4);
- Redistributing vectors (in Section 2.5).

2.1 Epetra Communicator Objects

The Epetra_Comm class is an interface that encapsulates the general information and services needed for the other Epetra classes to run on a parallel computer. An Epetra_Comm object is required for building all Epetra_Map objects, which in turn are required for all other Epetra classes.

Epetra_Comm has two basic implementations:

- Epetra_SerialComm (for serial executions);
- Epetra_MpiComm (for MPI distributed memory executions).

For most basic applications, the user can create an Epetra_Comm object using the following code:

```
#include "Epetra_config.h"
#ifdef HAVE_MPI
#include "mpi.h"
#include "Epetra_MpiComm.h"
#else
#include "Epetra_SerialComm.h"
#endif
// .. other include files and others ...
int main( int argv, char *argv[]) {
 // .. some declarations here ...
#ifdef HAVE_MPI
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);
#else
  Epetra_SerialComm Comm;
#endif
// ... other code follows ...
```

Note that the `MPI_Init()` call and the

```
#ifdef HAVE_MPI
  MPI_Finalize();
#endif
```

call, are likely to be the *only* MPI calls users have to explicitly introduce in their code.

Most of Epetra_Comm methods are similar to MPI functions. The class provides methods as `MyPID()`, `NumProc()`, `Barrier()`, `Broadcast()`, `SumAll()`, `GatherAll()`, `MaxAll()`, `MinAll()`, `ScanSum()`. For instance, the number of processes in the communicator, `NumProc`, and the ID of the calling process, `MyPID`, can be obtained as

```
int NumProc = Comm.NumProc();
int MyPID = Comm.MyPID();
```

File `${TRILINOS_HOME}/doc/tutorial/epetra/ex1.cpp` presents the use of some of the above introduced functions. For a description of the syntax, please refer to the Epetra Class Documentation.

2.2 Defining a Map

Very often, various distributed objects such as matrices or vectors, have identical distribution of elements among the processes. This distribution of elements (or points) is here called a *map*, and its actual implementation within the Trilinos project is given by the Epetra_Map class (or, more generally, by an Epetra_BlockMap). Basically, the class handles the definition of:

- global number of elements (called `NumGlobalPoints`);
- the local number of elements (called `NumMyPoints`);
- the global numbering of all local nodes (an integer vector of size `NumMyPoints`, called `MyGlobalElements`).

There are essentially three ways to define an map. The easiest way is to specify the global number of elements:

```
Epetra_Map Map(NumGlobalPoints,0,Comm);
```

In this case, the constructor takes the global dimension of the vector (here indicated as `NumGlobalPoints`), the base index (`0` for C or C++ arrays, `1` for FORTRAN arrays, but it can be any number), and an `Epetra_Comm` object (introduced in Section 2.1). As a result, each process will be assigned a contiguous list of elements.

Another way to build the Epetra_Comm object is to furnish the local number of elements:

```
Epetra_Map Map(-1,NumMyPoints,0,Comm);
```

This will create a vector of size $\sum_{i=0}^{NumProc-1}$ `NumMyPoints`. Each process will get a contiguous set of elements. These two approached are coded in file `${TRILINOS_HOME}/doc/tutorial/epetra/ex2.cpp`.

Another, more involved way, to create an Epetra_Map, is to specify on each process both the number of local elements, and the global numbering of each local element. To better explain this, let us consider the following code, in which a vector, of global dimension 5, is split among 2 processes `p0` and `p1`. `p0` owns nodes 0 an 4, while `p1` nodes 1, 2, and 3.

```
MyPID = Comm.MyPID();
switch( MyPID ) {
case 0:
  MyElements = 2;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 0;
  MyGlobalElements[1] = 4;
  break;
case 1:
  MyElements = 3;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 1;
  MyGlobalElements[1] = 2;
  MyGlobalElements[2] = 3;
  break;
}

Epetra_Map Map(-1,MyElements,MyGlobalElements,0,Comm);
```

The complete code is reported in `${TRILINOS_HOME}/doc/tutorial/epetra/ex3.cpp`.

A Map object can be queried for the global and local number of elements, using

```
int NumGlobalElements = Map.NumGlobalElements();
int NumMyElements = Map.NumMyElements();
```

and for the global ID of local elements, using

```
int * MyGlobalElements = Map.MyGlobalElements();
```

or, equivalently,

```
int MyGlobalElements[NumMyElements];
Map.MyGlobalElements(MyGlobalElements);
```

The class Epetra_Map is derived from Epetra_BlockMap. This class keeps information that describes the distribution of objects that have block elements (for example, one or more contiguous entries of a vector). This situation is common in applications like multiple-unknown PDE problems. A variety of constructors are available for this class. An example of use of block maps is reported in `${TRILINOS_HOME}/doc/tutorial/epetra/ex23.cpp`.

Note that different maps can coexist in the same part of the code. This allows the user to easily define vectors with different distributions (even for vectors of the same size). Two classes are provided to transfer data from one map to an other. Those classes (Epetra_Import and Epetra_Export) are discussed in Section 2.5.

Remark 3. *Most Epetra objects overload the << operator. For example, to visualize information about the* `Map`*, one can simply write*

```
cout << Map;
```

This Section has presented the construction of very basic map objects. However, map objects of very general form can be constructed. First, element numbers are only labels, and they do not have to be consecutive. This means that we can define a map with elements 1, 100 and 10000 on process 0, and elements 2, 200 and 20000 on process 1. This map, composed by 6 elements, is perfectly legal. Second, each element can be assigned to more than one process. Examples `${TRILINOS_HOME}/doc/tutorial/epetra/ex20.cpp` and `${TRILINOS_HOME}/doc/tutorial/epetra/ex21.cpp` can be used to better understand the potentiality of Epetra_Maps.

Remark 4. *The use of "distributed directory" technology facilitates arbitrary global ID support.*

2.3 Creating and Assembling Serial Vectors

Within Epetra, it is possible to define *sequential* vectors, for serial or for parallel runs. A sequential vector is a vector which, in the opinion of the programmer, does not need to be partitioned among the processes. Note that each process defines its own sequential vectors, and that changing an element of this vector on this process will *not* directly affect the vectors stored on other processes (if any have been defined).

To create a sequential vector containing `Length` elements, one can use the following command:

```
Epetra_SerialDenseVector x(Length);
```

Other constructors are available; check the Epetra Class Documentation.

The class Epetra_SerialDenseVector enables the construction and use of real-valued, double-precision dense vectors. The Epetra_SerialDenseVector class is intended to provide convenient vector notation but derives all significant functionality from Epetra_SerialDenseMatrix class. The vector can be filled using the `[]` or `()` operators. Both methods return the specified element of the vector. However, using `()`, bounds, checking is enforced. Using using `[]`, no bounds checking is done unless Epetra is compiled with `EPETRA_ARRAY_BOUNDS_CHECK`.

Remark 5. *To construct replicated Epetra objects on distributed memory machines, the user may consider the class Epetra_LocalMap. This class allows the constructions of those replicated local objects and keeps information that describe the distribution.*

File `${TRILINOS_HOME}/doc/tutorial/epetra/ex4.cpp` shows some basic operations on dense vectors.

2.4 Creating and Assembling a Distributed Vector

To create a distributed vector, the first step is to define a map. (Actually, this is true for all distributed Epetra objects.) After that, an Epetra_Vector object can be constructed with an instruction of type

```
Epetra_Vector x(Map);
```

This constructor allocates space for the vector and set all the elements to zero. A copy constructor can be used as well:

```
Epetra_Vector y(x);
```

Alternatively, the user can pass a pointer to an array of double precision values:

```
Epetra_Vector x(Copy,Map,LocalValues);
```

Note the word `Copy` is input to the constructor. Epetra allows two data access modes:

1. `Copy` mode: Allocates memory and makes a copy of the user-provided data. In this case, the user data is not needed after construction;

2. `View` mode: Creates a "view" of the user's data. In this case, the user data is required to remain untouched for the life of the vector (or modified carefully). It is worth noting that the View mode is very dangerous from a data hiding perspective. Therefore, users are strongly encouraged to develop code using Copy mode first and only use View mode in a secondary optimization phase. To use the View mode, the user has to define the vector entries using a double vector (of appropriate size), than construct an Epetra_Vector with an instruction of type

   ```
   Epetra_Vector z(View,Map,double_vector);
   ```

   where `double_vector` is a pointer to the vector of doubles.

Regardless of how a vector has been created, one can use the `[]` operator to access a vector element:

```
x[i] = 1.0*i;
```

where `i` is in the local index space.

Epetra also defines some functions to set vector elements in local or global index space. `ReplaceMyValues` or `SumIntoMyValues` will replace or sum values into a vector with a given indexed list of values, with indexes in the *local* index space; `ReplaceGlobalValues` or `SumIntoGlobalValues` will replace or sum values into a vector with a given indexed list of values in the *global* index space. It is important to note that a process *cannot* set a vector entries locally owner by another process. In other words, both global and local insert and replace functions refers to the part of a vector assigned to the calling process. Intra-process communications can be performed using Import and Export objects, covered in Section 2.5.

Another way is to put vector values in a user-provided array. For instance, one may have:

```
double *x_values;
x_values = new double[MyLength];
x.ExtractCopy( x_values );
for( int i=0 ; i<MyLength ; ++i ) x_values[i] *= 10;
for( int i=0 ; i<MyLength ; ++i )
  x.ReplaceMyValues( 1, 0, x_values+i, &i );
```

(File ${TRILINOS_HOME}/doc/tutorial/epetra/ex5.cpp reported the complete source.) It is important to note that ExtractCopy does not give access to the vector elements, but only copies them into the user-provided array. The user must commit those changes to the vector object, using, for instance, ReplaceMyValues.

A further, computationally efficient way, is to extract a "view" of the (multi-)vector internal data. To that aim, one has to call

```
double * pointer;
x.ExtractView( &pointer );
```

Now, modifying the values of pointer will affect the internal data of the Epetra_Vector x.
An example of the use of ExtractView is reported in file ${TRILINOS_HOME}/doc/tutorial/epetra

Remark 6. *The class Epetra_Vector is derived from Epetra_MultiVector. Roughly speaking, a multi-vector is a collection of one or more vectors, all having the same length and distribution. The reader may look to the file* ${TRILINOS_HOME}/doc/tutorial/epetra/ex7.cpp *for an example of use of multi-vectors.*

The user can also consider the function ResetView, which allows a (very) light-weight replacement of multi-vector values, created using the Epetra_DataMode View. Note that no checking is performed to see if the values passed in contain valid data. This method can be extremely useful in situation where a vector is needed for use with an Epetra operator or matrix, and the user is not passing in a multi-vector. Use this method with caution as it could be extremely dangerous. A simple example is reported in ${TRILINOS_HOME}/doc/tutorial/epetra/ex8.cpp

It is possible to perform a certain number of operations on vector objects. Some of them are reported in Table 2. Example ${TRILINOS_HOME}/doc/tutorial/epetra/ex18.cpp works with some of the functions reported in the table.

2.5 Epetra_Import and Epetra_Export

Epetra_Import and Epetra_Export are two classes meant for efficient importing of off-processors elements. Epetra_Import and Epetra_Export are used to construct a communication plan that can be called repeatedly by computational classes such the Epetra multi-vectors of the Epetra matrices.

`int NumMyELement()` returns the local vector length on the calling processor
`int NumGlobalElements()` returns the global length
`int Norm1(double *Result) const` returns the 1-norm (defined as $\sum_i^n \|x_i\|$ (see also `Norm2` and `NormInf`)
`Normweigthed(double *Result) const` returns the 2-norm, defined as $\sqrt{\frac{1}{n}\sum_{j=1}^{n}(w_j x_j)^2)}$
`int Dot(const Epetra MultiVector A, double *Result) const` computes the dot product of each corresponding pair of vectors
`int Scale(double ScalarA, const Epetra MultiVector &A` Replace multi-vector values with scaled values of A, `this=ScalarA*A`
`int MinValue(double *Result) const` compute minimum value of each vector in multi-vector (see also `MaxValue` and `MeanValue`
`int PutScalar(double Scalar)` Initialize all values in a multi-vector with constant value
`int Random()` set multi-vector values to random numbers

Table 2. Some methods of the class `Epetra_Vector`

Currently, those classes have one constructor, taking two Epetra_Map or Epetra_BlockMap objects. The first map specifies the global IDs that are owned by the calling processor. The second map specifies the global IDs of elements that we want to import later.

Using an Epetra_Import object means that the calling process knows what it wants to receive, while an Epetra_Export object means that it knows what it wants to send. An Epetra_Import object can be used to do an Export as a reserve operation (and equivalently an Epetra_Export can be used to do an Import). In the particular case of bijective maps, either Epetra_Import or Epetra_Export is appropriate.

To better illustrate the functionalities of these two classes, we consider the following example. Suppose that vector x of global length 4, is distributed over two processes. Process 0 own nodes 0,1,2, while process 1 owns nodes 1,2,3. This means that nodes 1 and 2 are replicated over the two processes. Suppose that we want to bring all the components of x to process 0, summing up the contributions of node 1 and 2 from the 2 processes. This is done in the following example (the complete code is reported in ${TRILINOS_HOME}/doc/tutorial/epetra/ex9.cpp).

```
int NumGlobalElements = 4; // global dimension of the problem

int NumMyElements; // local nodes
Epetra_IntSerialDenseVector MyGlobalElements;

if( Comm.MyPID() == 0 ) {
  NumMyElements = 3;
  MyGlobalElements.Size(NumMyElements);
  MyGlobalElements[0] = 0;
  MyGlobalElements[1] = 1;
  MyGlobalElements[2] = 2;
} else {
  NumMyElements = 3;
  MyGlobalElements.Size(NumMyElements);
  MyGlobalElements[0] = 1;
  MyGlobalElements[1] = 2;
  MyGlobalElements[2] = 3;
}

// create a map
Epetra_Map Map(-1,MyGlobalElements.Length(),
               MyGlobalElements.Values(),0, Comm);
```

```
// create a vector based on map
Epetra_Vector x(Map);
for( int i=0 ; i<NumMyElements ; ++i )
  x[i] = 10*( Comm.MyPID()+1 );
cout << x;

// create a target map, in which all the elements are on proc 0
int NumMyElements_target;

if( Comm.MyPID() == 0 )
  NumMyElements_target = NumGlobalElements;
else
  NumMyElements_target = 0;

Epetra_Map TargetMap(-1,NumMyElements_target,0,Comm);

Epetra_Export Exporter(Map,TargetMap);

// work on vectors
Epetra_Vector y(TargetMap);

y.Export(x,Exporter,Add);
cout << y;
```

Running this code with 2 processors, the output will be approximatively the following:

```
[msala:epetra]> mpirun -np 2 ./ex31.exe
Epetra::Vector
     MyPID           GID                Value
         0             0                   10
         0             1                   10
         0             2                   10
Epetra::Vector
         1             1                   20
         1             2                   20
         1             3                   20
Epetra::Vector
Epetra::Vector
     MyPID           GID                Value
```

0	0	10
0	1	30
0	2	30
0	3	20

3 Working with Epetra Matrices

Epetra contains several matrix classes. Epetra matrices can be defined to be *serial* or *parallel*:

- Examples of serial matrices are, for instance, the matrix corresponding to a given element in a finite-element discretization, or the Hessemberg matrix in the GMRES method. Those matrices are of small size, and therefore they are not distributed among the processors (but they can be replicated).
- For distributed sparse matrices, the basic class is Epetra_RowMatrix. This class is meant for double-precision matrices with row access (as required in a matrix-vector product), and it is a pure virtual class. Various classes are derived `Epetra_RowMatrix`. Among them, here we recall:
  - `Epetra_CrsMatrix` for point matrices;
  - `Epetra_VbrMatrix` for block matrices (that is, for matrices which have a block structure, for example the ones deriving from the discretization of a PDE problem with multiple unknowns for node);
  - `Epetra_FECrsMatrix` and `Epetra_FEVbrMatrix` for matrices arising from FE discretizations.

This Chapter will show some of the Trilinos capabilities to work with matrices. During the Chapter, the user be introduced to:

- Create (serial) dense matrices (in Section 3.1);
- Create sparse point matrices (in Section 3.2);
- Create sparse block matrices (in Section 3.3);
- Insert non-local elements using finite-element matrices (in Section 3.4).

3.1 Serial Dense Matrices

Epetra provides functionalities for sequential dense matrices with the class Epetra_SerialDenseMatrix. A possible way to create a serial dense matrix `D` of dimension `n` by `m` is

```
Epetra_SerialDenseMatrix D(n,m);
```

One could also create a zero-size object,

```
Epetra_SerialDenseMatrix D();
```

and then shape this object:

```
D.Shape(n,m);
```

(D could be reshaped using ReShape().)

Epetra_SerialDenseMatrix are stored in a column-major order in the usual FORTRAN style. This class is built on the top of the BLAS library, and is derived from Epetra_Blas. Epetra_SerialDenseMatrix is intended to provide a very basic support for dense rectangular matrices.

To access the matrix element at the i-th row and the j-th column, it is possible to use the parenthesis operator (A(i,j)), or the bracket operator (A[j][i], note that i and j are reversed). The bracket approach is in general faster, as the compiler can inline the corresponding function. Instead, some compiler have problems to inline the parenthesis operator.

As an example of the use of this class, in the following code we consider a matrix-matrix product between two rectangular matrices A and B.

```
int NumRowsA = 2, NumColsA = 2;
int NumRowsB = 2, NumColsB = 1;
Epetra_SerialDenseMatrix A, B;
A.Shape( NumRowsA, NumColsA );
B.Shape(NumRowsB, NumColsB);
// ... here set the elements of A and B
Epetra_SerialDenseMatrix AtimesB;
AtimesB.Shape(NumRowsA,NumColsB);
AtimesB.Multiply('N','N',1.0, A, B, 0.0);
cout << AtimesB;
```

The complete code is reported in file ${TRILINOS_HOME}/doc/tutorial/epetra/ex10.cpp.

To solve a linear system with a dense matrix, one has to create an Epetra_SerialDenseSolver. This class uses the most sophisticated techniques available in the LAPACK library. The class is built on the top of BLAS and LAPACK, and thus has excellent performances and numerical capabilities.

Given an Epetra_SerialDenseMatrix and two Epetra_DSerialDenseVectors `x` and `b`, the general approach is as follows:

```
Epetra_SerialDenseSolver Solver();
Solver.SetMatrix(D);
Solver.SetVectors(x,b);
```

Then, it is possible to invert the matrix with `Invert()`, solve the linear system with `Solve()`, apply iterative refinement with `ApplyRefinement()`. Other methods are available; for instance,

```
double rcond=Solve.RCOND();
```

returns the reciprocal of the condition number of matrix `D` (or -1 if not computed).

File `${TRILINOS_HOME}/doc/tutorial/epetra/ex11.cpp` outlines some of the capabilities of the Epetra_SerialDenseSolver class.

The Epetra_LAPACK class provides access to most of the same functionality as Epetra_SerialDenseSolver. The primary difference is that Epetra_LAPACK is a "thin" layer on the top of LAPACK, while Epetra_SerialDenseSolver attempts to provide easy access to the more sophisticated aspects of solving dense linear systems.

As a general rule, we can say that Epetra_LAPACK should be preferred when the user is looking for a convenient wrapper around the FORTRAN LAPACK routines, and the problem at hand is well-conditioned. Instead, when the user wants (or potentially wants to) solve ill-conditioned problems or want to work with a more object-oriented interface, he/she will probably use Epetra_SerialDenseMatrix.

3.2 Distributed Sparse Matrices

Epetra provided an extensive set of methods to create and fill distributed sparse matrices. These classes allow row-by-row or element-by-element constructions. Support is provided

for common matrix operations, as scaling, norm, matrix-vector multiplication and matrix-multivector multiplication[4].

Application do not need to know about the particular storage format, and other implementation details such as data layout, number and location of ghost nodes. Epetra furnishes two basic formats, one suited for point matrices, the other for block matrices. The former is presented in this Section; the latter, generally much more efficient for problems with multiple degree of freedom per node, is introduced in Section 3.3. If required, other matrix formats can be supported via the Epetra_Operator, described in Section 4.3.

Remark 7. *Some numerical algorithms require the application of the linear operator only. For this reason, some applications find convenient to not store a given matrix. Epetra can handle this situation using with the Epetra_Operator class; see Section 4.3.*

The process of creating a sparse matrix is more involved with respect to that of dense matrices. This is because, in order to obtain excellent numerical performances, one has to provide an estimation of the nonzero elements on each row of the sparse matrix. (Recall that dynamic allocation of new memory and copying the old storage into the new one is an expensive operation.)

As a general rule, the process of constructing a (distributed) sparse matrix is as follows:

- allocate an integer array `Nnz`, whose length equals the number of local rows;
- loop over the local rows, and estimate the number of nonzero elements of that row;
- create the sparse matrix using `Nnz`;
- fill the sparse matrix.

As an example, in this Section we will present how to construct a distributed (sparse) matrix, arising from a finite-difference solution of a one-dimensional Laplace problem. This matrix looks like:

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

[4]At the present stage of development, no functions are provided to perform a matrix-matrix product between to distributed objects. However, the interested user can convert the Epetra matrix into an ML matrix (called ML_Operator), perform the matrix-matrix multiplication with ML functions, and convert back the resulting ML_Operator into an Epetra matrix.

The example illustrates how to construct the matrix, and how to perform matrix-vector operations. The code can be found in `${TRILINOS_HOME}/doc/tutorial/epetra/ex12.cpp`.

We start by specifying the global dimension (here is 5, but can be any number):

```
int NumGlobalElements = 5;
```

We create a map, and define the local number of rows and the global numbering for each local row:

```
Epetra_Map Map(NumGlobalElements,0,Comm);
int NumMyElements = Map.NumMyElements();
int * MyGlobalElements = Map.MyGlobalElements( );
```

In particular, we have that `j=MyGlobalElements[i]` is the global numbering for local node `i`. Then, we have to specify the number of nonzeros per row. In general, this can be done in two ways:

- Furnish an integer value, representing the number of nonzero element on each row (the same value for all the rows);
- Furnish an integer vector `NumNz`, of length `NumMyElements()`, containing the nonzero elements of each row.

The second approach can be coded as follows:

```
int * NumNz = new int[NumMyElements];
for( int i=0 ; i<NumMyElements ; i++ )
if( MyGlobalElements[i]==0 ||
    MyGlobalElements[i] == NumGlobalElements-1)
  NumNz[i] = 2;
else
  NumNz[i] = 3;
```

We are building a tridiagonal matrix where each row has (-1 2 -1). So we need 2 off-diagonal terms (except for the first and last equation). Here `NumNz[i]` is the Number of nonzero terms in the i-th global equation on this process.

Now, we create an Epetra_CsrMatrix as

```
Epetra_CrsMatrix A(Copy,Map,NumNz);
```

and we add rows one-at-a-time. A has been created in Copy mode, and relies on the specified map. To fill its values, we need some additional variables: `Indexes` and `Values`. Those will contain the global column number and the values of the nonzeros for each row.

```
double *Values = new double[2];
Values[0] = -1.0; Values[1] = -1.0;
int *Indices = new int[2];
double two = 2.0;
int NumEntries;

for( int i=0 ; i<NumMyElements; ++i ) {
  if (MyGlobalElements[i]==0) {
      Indices[0] = 1;
      NumEntries = 1;
  } else if (MyGlobalElements[i] == NumGlobalElements-1) {
    Indices[0] = NumGlobalElements-2;
    NumEntries = 1;
  } else {
    Indices[0] = MyGlobalElements[i]-1;
    Indices[1] = MyGlobalElements[i]+1;
    NumEntries = 2;
  }
  A.InsertGlobalValues(MyGlobalElements[i], NumEntries, Values, Indices);
  // Put in the diagonal entry
  A.InsertGlobalValues(MyGlobalElements[i], 1, &two, MyGlobalElements+i);
}
```

Note that column indexes have been inserted using global indexes. As a final operation, we can transform the matrix into local indexes. This phase in required in order to perform efficient parallel matrix-vector products and other matrix operations.

```
A.FillComplete();
```

The above presentation refers to a rather common case: In a parallel matrix-vector product

$$AX = B,$$

the map used to define the parallel distribution of the matrix, is the same of the (multi-)vectors X and B. This means that the rows of A are distributed among the processes in the same way of the elements of X and B. However, Epetra allows the user to handle the more general case of a matrix defined using a Map, is different from that of X and that of B. In fact, each Epetra matrix is defined by *four* maps:

- Two maps, called RowMap and ColumnMap, are used to determine the set of rows and the columns of the elements assigned to a given processor. In general, one processor cannot set elements assigned to other processors. (However, some classes, derived from the Epetra_RowMatrix, can perform data exchange; see for instance Epetra_FECrsMatrix or Epetra_FEVbrMatrix.) RowMap and ColumnMap determine the pattern of the matrix, as it is used during the construction. They can be obtained using the methods `RowMap()` and `ColMap()` of the Epetra_RowMatrix class. Usually, the user dos not specify a ColumnMap, which is automatically created by Epetra. RowMap and ColumnMap can differ.
- DomainMap and RangeMap define, instead, the parallel data layout of X and B, respectively. Note that those two maps can completely different from RowMap and ColumnMap, meaning that a matrix can be constructed using a certain data distribution, then used on vectors with another data distribution. DomainMap and RangeMap can differ. Those tow maps can be obtained using the methods `DomainMap()` and `RangeMap()`.

The potentialities of this approach are better explained using an example, reported in the example file `${TRILINOS_HOME}/doc/tutorial/epetra/ex24.cpp`. In this example, to be run using two processors, we build up two maps: `MapA` will be used to construct the matrix, while `MapB` to define the parallel layout of the vectors X and B. For the sake of simplicity, A is diagonal.

```
Epetra_CrsMatrix A(Copy,MapA,MapA,1);
```

As usual in this Tutorial, the integer vector `MyGlobalElementsA` contains the global ID of local nodes. To assemble A, we cycle over all the local rows (defined by `MapA`):

```
for( int i=0 ; i<NumElementsA ; ++i ) {
  double one = 2.0;
  int indices = MyGlobalElementsA[i];
  A.InsertGlobalValues(MyGlobalElementsA[i], 1, &one, &indices );
}
```

Now, as both X and B are defined using MapB, instead of calling `FillComplete()`, we do

```
A.FillComplete(MapB,MapB);
```

Now, we can create X and B as vectors based on MapB, and perform the matrix-vector product:

```
Epetra_Vector VecB(MapB);   Epetra_Vector VecB2(MapB);
A.Multiply(false,VecB,VecB2);
```

Remark 8. *Although presented for Epetra_CrsMatrix objects, the distinction between RowMap, ColMap, DomainMap, and RangeMap is valid for all classed derived from Epetra_RowMatrix.*

Example `${TRILINOS_HOME}/doc/tutorial/epetra/ex14.cpp` shows the use of some of the methods of the Epetra_CrsMatrix class. The code prints out several information about the structure of the matrix, and some of its properties. The output will be approximatively as here reported:

```
[msala:epetra]> mpirun -np 2 ./ex14
*** general Information about the matrix
Number of Global Rows = 5
Number of Global Cols = 5
is the matrix square  = yes
||A||_\infty          = 4
||A||_1               = 4
||A||_F               = 5.2915
Number of nonzero diagonal entries = 5( 100 %)
Nonzero per row : min = 2 average = 2.6 max = 3
Maximum number of nonzero elements/row = 3
min( a_{i,j} )       = -1
max( a_{i,j} )       = 2
min( abs(a_{i,j}) ) = 1
max( abs(a_{i,j}) ) = 2
Number of diagonal dominant rows        = 2 (40 % of total)
Number of weakly diagonal dominant rows = 3 (60 % of total)
*** Information about the Trilinos storage
Base Index                  = 0
```

```
is storage optimized        = no
are indices global          = no
is matrix lower triangular = no
is matrix upper triangular = no
are there diagonal entries = yes
is matrix sorted            = yes
```

Other examples are reported for Epetra_CrsMatrix:

- Example `${TRILINOS_HOME}/doc/tutorial/epetra/ex13.cpp` implements a simple distributed finite-element solver. The code solves a 2D Laplace problem with unstructured triangular grids. In this example, the information about the grid are hardwired. The interested user can easily modify those lines in order to read the grid information from a file.

- Example `${TRILINOS_HOME}/doc/tutorial/epetra/ex15.cpp` explains how to export an Epetra_CrsMatrix to file in a MATLAB format. The output of this example will be as follows:

```
[msala:epetra]> mpirun -np 2 ./ex15
A = spalloc(5,5,13);
% On proc 0: 3 rows and 8 nonzeros
A(1,1) = 2;
A(1,2) = -1;
A(2,1) = -1;
A(2,2) = 2;
A(2,3) = -1;
A(3,2) = -1;
A(3,3) = 2;
A(3,4) = -1;
% On proc 1: 2 rows and 5 nonzeros
A(4,4) = 2;
A(4,5) = -1;
A(4,3) = -1;
A(5,4) = -1;
A(5,5) = 2;
```

A companion to this example is `${TRILINOS_HOME}/doc/tutorial/epetra/ex16.cpp`, which exports an Epetra_Vector to MATLAB format.

3.3 Creating VBR Matrices

The following code shows how to work with VBR matrices. This format has been designed for PDE problems with more than one unknown per grid node. The resulting matrix has a sparse block structure, and the size of each dense block equals the number of PDE equations defined on that block. This format is quite general, and can handle matrices with variable block size, as it is done is the following example.

First, we create a map, containing the distribution of the blocks:

```
Epetra_Map Map(NumGlobalElements,0,Comm);
```

Here, a linear decomposition is used for the sake of simplicity, but any map can be used as well. Now, we obtain some information about the map:

```
// local number of elements
int NumMyElements = Map.NumMyElements();
// global numbering of local elements
int * MyGlobalElements = new int [NumMyElements];
Map.MyGlobalElements( MyGlobalElements );
```

A block matrix can have blocks of different size. Here, we suppose that the dimension of diagonal block row i is $i+1$. The integer vector `ElementSizeList` will contain the block size of local element `i`.

```
Epetra_IntSerialDenseVector ElementSizeList(NumMyElements);
 for( int i=0 ; i<NumMyElements ; ++i )
   ElementSizeList[i] = 1+MyGlobalElements[i];
```

Here `ElementSizeList` is declared as Epetra_IntSerialDenseVector, but an int array is fine as well.

Now we can create a map for the block distribution:

```
Epetra_BlockMap BlockMap(NumGlobalElements,NumMyElements,
                         MyGlobalElements,
                         ElementSizeList.Values(),0,Comm);
```

and finally we can create the VBR matrix based on `BlockMap`. In this case, nonzero elements are located in the diagonal and the sub-diagonal above the diagonal.

```
Epetra_VbrMatrix A(Copy, BlockMap, 2);

int Indices[2];
double Values[MaxBlockSize];

for( int i=0 ; i<NumMyElements ; ++i ) {
  int GlobalNode = MyGlobalElements[i];
  Indices[0] = GlobalNode;
  int NumEntries = 1;
  if( GlobalNode != NumGlobalElements-1 ) {
    Indices[1] = GlobalNode+1;
    NumEntries++;
  }
  A.BeginInsertGlobalValues(GlobalNode, NumEntries, Indices);
  // insert diagonal
  int BlockRows = ElementSizeList[i];
  for( int k=0 ; k<BlockRows * BlockRows ; ++k )
    Values[k] = 1.0*i;
  B.SubmitBlockEntry(Values,BlockRows,BlockRows,BlockRows);

  // insert off diagonal if any
  if( GlobalNode != NumGlobalElements-1 ) {
    int BlockCols = ElementSizeList[i+1];
    for( int k=0 ; k<BlockRows * BlockCols ; ++k )
      Values[k] = 1.0*i;
    B.SubmitBlockEntry(Values,BlockRows,BlockRows,BlockCols);
  }
  B.EndSubmitEntries();
}
```

Note that, with VBR matrices, we have to insert one block at time. This required two more instructions, one to start this process (`BeginInsertGlobalValues`), and another one to commit the end of submissions (`EndSubmitEntries`).

Please refer to `${TRILINOS_HOME}/doc/tutorial/epetra/ex17.cpp` for the entire source.

3.4 Insert non-local Elements Using FE Matrices

The most important additional feature provided by the Epetra_FECrsMatrix with respect to Epetra_CrsMatrix, is the capability of setting non-local matrix elements. We will illustrate this using the following example, reported in `${TRILINOS_HOME}/doc/tutorial/epetra/ex23.cpp` In the example, we will set all the entries of a distributed matrix from process 0. For the sake of simplicity, this matrix is diagonal, but more complex cases can be handled as well.

First, we define the Epetra_FECrsMatrix in Copy mode as

```
Epetra_FECrsMatrix A(Copy,Map,1);
```

Now, we will set all the diagonal entries from process 0:

```
if( Comm.MyPID() == 0 ) {
  for( int i=0 ; i<NumGlobalElements ; ++i ) {
    int indices[2];
    indices[0] = i; indices[1] = i;
    double value = 1.0*i;
    A.SumIntoGlobalValues(1,indices,&value);
  }
}
```

The Function `SumIntoGlobalValues` adds the coefficients specified in `indices` (as pair row-column) to the matrix, adding them to any coefficient that may exist at the specified location. In a finite element code, the user will probably insert more than one coefficient at time (typically, all the matrix entries corresponding to an elemental matrix).

At this point, we need to exchange data, to that each matrix element not owned by process 0 could be send to the owner, as specified by `Map`. This is accomplished by calling, on all processes,

```
A.GlobalAssemble();
```

A simple

```
cout << A;
```

can be used to verify the data exchange.

4 Other Epetra Classes

Epetra includes a certain number of classes that can greatly help to develop parallel codes. In this Chapter we will recall the main usage of some of those classes:

- Epetra_Time (in Section 4.1);
- Epetra_Flops (in Section 4.2).
- Epetra_Operator and Epetra_RowMatrix (in Section 4.3);
- Epetra_LinearProblem (in Section 4.4).

4.1 Epetra_Time

To retrieve elapsed and wall-clock time can be problematic because of several platform-dependent and language-dependent issues. To avoid those problems, Epetra furnishes the Epetra_Time class. Epetra_Time is meant to insulate the user from the specifics of timing among a variety of platforms.

Using Epetra_Time, it is possible to measure the elapsed time. This is the time elapsed between two phases of a program.

A Epetra_Time object is defined as

```
Epetra_Time time(Comm);
```

To compute the elapsed time required by a piece of code, then user should put the instruction

```
time.ResetStartTime();
```

before the code to the timed. Then, the methods `ElapsedTime()` and `WallTime()` will return the elapsed time and wall-clock time, respectively. `ElapsedTime()` returns the elapsed time from the creation of *this* object.

4.2 Epetra_Flops

The Epetra_Flops class provides basic support and consistent interfaces for counting and reporting floating point operations performed in the Epetra computational classes. All classes based on the Epetra_CompObject can count flops by the user creating an Epetra_Flops object and calling the SetFlopCounter() method for an Epetra_CompObject.

As an example, suppose you are interested in counting the flops required by a vector-vector product (between, say, x and y). The first step is to create an instance of the class:

```
Epetra_Flops counter();
```

Then, it is necessary to "hook" the counter object to the desired computational object, in the following way:

```
x.SetFlopCounter(counter);
y.SetFlopCounter(counter);
```

Then, we perform the desired computations on Epetra objects (in this case, the vector-vector problem):

```
x.Dot(y,&dotProduct);
```

Finally we can extract the number of performed operations ans stored it in the double variable `total_flops` as

```
total_flops = counter.Flops();
```

which are the toal number of *serial* flops. This will also reset the flop counter.

Epetra_Time objects can be used in conjunction with Epetra_Flops objects to estimate the number of floating point operations per second of a given code (or a part of it). One can proceed as here reported:

```
Epetra_Flops counter;
x.SetFlopCounter(counter);
Epetra_Time timer(Comm);
```

```
x.Dot(y,&dotProduct);
double elapsed_time = timer.ElapsedTime();
double total_flops =counter.Flops();
cout << "Total ops: " << total_flops << endl;
double MFLOPs = total_flops/elapsed_time/1000000.0;
cout << "Total MFLOPs  for mat-vec = " << MFLOPs << endl<< endl;
```

This code is reported in `${TRILINOS_HOME}/doc/tutorial/epetra/ex20.cpp`. The output will be approximatively as follows:

```
[msala:epetra]> mpirun -np 2 ./ex20
Total ops: 734
Total MFLOPs  for mat-vec = 6.92688

Total ops: 734
Total MFLOPs  for mat-vec = 2.48021

Total ops: 246
Total MFLOPs for vec-vec = 0.500985

q dot z = 2
Total ops: 246
Total MFLOPs for vec-vec = 0.592825

q dot z = 2
```

Remark 9. *Operation count are serial count, and therefore keep trace of local operations only.*

Remark 10. *Each computational class has a* `Flops()` *method, that can queried for the flop count of that object.*

4.3 Epetra_Operator and Epetra_RowMatrix Classes

Matrix-free methods can be easily introduced in the Epetra framework using one of the following two classes:

- Epetra_Operator;

- Epetra_RowMatrix.

Technically, both classes are pure virtual classes (that is, they specify interfaces only), that enable the use of real-valued double-precision sparse matrices. Epetra_RowMatrix, derived from Epetra_Operator, is meant for matrices where the matrix entries are intended for row access, and it is currently implemented by Epetra_CrsMatrix, Epetra_VbrMatrix, Epetra_FECrsMatrix, and Epetra_FEVbrMatrix.

In the following, we consider for instance how to apply a matrix to a vector without explicitly constructing the matrix. The matrix is the classical finite-difference discretization of a Laplace on a 1D grid with constant grid-size. For the sake of simplicity, we avoid the issues related to intra-process communication (hence this code can be run with one process only).

The first step is the definition of a class, here called `TriDiagonalOperator`, and derived from the Epetra_Operator class.

```
class TriDiagonalOperator : public Epetra_Operator {
public:
 // .. definitions here, constructors and methods
private:
  Epetra_Map Map_;
  double diag_minus_one_;    // value in the sub-diagonal
  double diag_;              // value in the diagonal
  double diag_plus_one_;     // value in the super-diagonal
}
```

As the class Epetra_Operator implements several virtual methods, we have to specify all those methods in our class. Among them, we are interested in the `Apply` method, which may be coded as follows:

```
int Apply( const Epetra_MultiVector & X, Epetra_MultiVector & Y ) const {
  int Length = X.MyLength();

  // need to handle multi-vectors and not only vectors
  for( int vec=0 ; vec<X.NumVectors() ; ++vec ) {

    // one-dimensional problems here
    if( Length == 1 ) {
```

```
      Y[vec][0] = diag_ * X[vec][0];
      break;
    }

    // more general case (Lenght >= 2)
    // first row
    Y[vec][0] = diag_ * X[vec][0] + diag_plus_one_ * X[vec][1];

    // intermediate rows
    for( int i=1 ; i<Length-1 ; ++i ) {
      Y[vec][i] = diag_ * X[vec][i] + diag_plus_one_ * X[vec][i+1]
        + diag_minus_one_ * X[vec][i-1];
    }
    // final row
    Y[vec][Length-1] = diag_ * X[vec][Length-1]
      + diag_minus_one_ * X[vec][Length-2];
  }
  return true;
}
```

Now, in the `main` function, we can define a TriDiagonalOperatr object using the specified constructor:

```
TriDiagonalOperator TriDiagOp(-1.0,2.0,-1.0,Map);
```

and we can apply this operator to a vector as:

```
DiagOp.Apply(x,y);
```

`${TRILINOS_HOME}/doc/tutorial/epetra/ex21.cpp` reportes the entire source code.

Remark 11. *The clear disadvantage of deriving Epetra_Operator or Epetra_RowMatrix with respect to use Epetra_CrsMatrix or Epetra_VbrMatrix, is that users must specify their communication patterns for intra-process data exchange. For this purpose, Epetra_Import classes can be used. File* `${TRILINOS_HOME}/doc/tutorial/epetra/ex22.cpp` *shows how to extend* `ex21.cpp` *to the multi-process case. This example makes use of the Epetra_Import class to exchange data.*

Another use of Epetra_Operator and Epetra_RowMatrix is to allow support for user's defined matrix format. For instance, suppose that your code generates matrices in MSR format (detailed in the Aztec documentation). You can easily create an Epetra_Operator, that applies the MSR format to Epetra_MultiVectors. For the sake of simplicity, we will limit ourselves to the monoprocess case. Extentions to multi-processes case requires to handle ghost-nodes updates.

As a first step, we create a class, derived from the Epetra_Operator class,

```
class MSRMatrix : public Epetra_Operator
{

public:
  // constructor
  MSRMatrix(Epetra_Map Map, int * bindx, double * val) :
    Map_(Map), bindx_(bindx), val_(val)
  {}

  ~MSRMatrix() // destructor
  {}

  // Apply the RowMatrix to a MultiVector
  int Apply(const Epetra_MultiVector & X, Epetra_MultiVector & Y ) const
  {

    int Nrows = bindx_[0]-1;

    for( int i=0 ; i<Nrows ; i++ ) {
      // diagonal element
      for( int vec=0 ; vec<X.NumVectors() ; ++vec ) {
        Y[vec][i] = val_[i]*X[vec][i];
      }
      // off-diagonal elements
      for( int j=bindx_[i] ; j<bindx_[i+1] ; j++ ) {
        for( int vec=0 ; vec<X.NumVectors() ; ++vec ) {
          Y[vec][bindx_[j]] += val_[j]*X[vec][bindx_[j]];
        }
      }
    }
    return 0;
```

```
  } /* Apply */
  ... other functions ...

private:
  int * bindx_; double * val_;
}
```

As stated by the fragment of code above, the constructor take the two MSR vectors, and an Epetra_Map. The complete code is reported in `${TRILINOS_HOME}/doc/tutorial/epetra/ex`

4.4 Epetra_LinearProblem

A linear problem of type $AX = B$ is defined by an Epetra_LinearProblem class. This class required an Epetra_RowMatrix or an Epetra_Operator object (often an Epetra_CrsMatrix or Epetra_VbrMatrix), and two (multi-)vectors X and B. X must have been defined using a map equivalent to the DomainMap of A, while B using a map equivalent ot the RangeMap of A (see Section 3.2).

Linear problems can be used to solve linear systems with iterative methods (typically, using AztecOO, covered in Chapter 5), or with direct solvers (typically, using Amesos, described in Chapter 8.

Once the linear problem has been defined, the user can:

- scale the problem, using `LeftScale(D)` or `RightScale(D)`, `D` being am Epetra_Vector of compatible size;
- define a preconditioner for the iterative solution;
- change X and B, using `SetRHS(&B)` and `SetLHS(&X)`;
- change A, using `SetOperator(&A)`.

4.5 Concluding Remarks

More details about the Epetra project, and a technical description of classes and methods, can be found in [5, 9].

5 Iterative Solution of Linear Systems with AztecOO

AztecOO is package which extends the Aztec library [20]. Aztec is the legacy iterative solver at the Sandia National Laboratories. It has been extracted from the MPSalsa reacting flow code [17, 15], and it is currently installed in dozens of Sandia's applications. AztecOO extends this package, using C++ classes to enable more sophisticated use.

AztecOO is intended for the iterative solution of linear systems of the form

$$A\,X = B, \tag{1}$$

when $A \in \mathbb{R}^{n\times n}$ is the linear system matrix, X the solution, and B the right-hand side. Both X and B are Epetra_Vector objects.

In this Chapter, we will:

- Outline the basic issued of the iterative solution of linear systems (in Section 5.1);
- Present the basic usage of AztecOO (in Section 5.2);
- Define one-level domain decomposition preconditioners (in Section 5.3);
- Use of AztecOO problems as preconditioners to other AztecOO problems (in Section 5.4).

5.1 Theoretical Background

Aim of this Section is to briefly present some aspects of the iterative solution of linear systems, to establish a notation. The Section is not supposed to be exhaustive, nor complete on this subject. The reader is referred to the existing literature for a rigorous presentation.

One can distinguish between two different aspects of the iterative solution of a linear system. The first one in the particular acceleration technique for a sequence of iterations vectors, that is a technique used to construct a new approximation for the solution, with information from previous approximations. This leads to specific iteration methods, like conjugate gradient or GMRES. The second aspect is the transformation of the given system to one that can be more efficiently solved by a particular iteration method. This is called *preconditioning*. A good preconditioner improves the convergence of the iterative method, sufficiently to overcome the extra cost of its construction and application. Indeed, without a preconditioner the iterative method may even fail to converge in practice.

The convergence of iterative methods depends on the spectral properties of the linear system matrix. The basic idea is to replace the original system (1) by

$$P^{-1}AX = P^{-1}B$$

(left-preconditioning), or by

$$AP^{-1}PB = B$$

(right-preconditioning), using a linear transformation P^{-1}, called preconditioner, in order to improve the spectral properties of the linear system matrix. In general terms, a preconditioner is any kind of transformation applied to the original system which makes it easier to solve.

In a modern perspective, the general problem of finding an efficient preconditioner is to identify a linear operator P with the following properties:

1. P is a good approximation of A is some sense. Although no general theory is available, we can say that P should act so that $P^{-1}A$ is near to being the identity matrix and its eigenvalues are clustered within a sufficiently small region of the complex plane;

2. P is efficient, in the sense that the iteration method converges much faster, in terms of CPU time, for the preconditioned system. In other words, preconditioners must be selected in such a way that the cost of constructing and using them is offset by the improved convergence properties they permit to achieve;

3. P or P^{-1} can take advantage of the architecture of modern supercomputers, that is, can be constructed and applied in parallel environments.

It should be stressed that computing the inverse of P is not mandatory; actually, the role of P is to "preconditioning" the residual r_m through the solution of the additional system $Pz_m = r_m$. This system $Pz_m = r_m$ should be much easier to solve than the original system.

The choice of P varies from "black-box" algebraic techniques which can be applied to general matrices to "problem dependent" preconditioners which exploit special features of a particular class of problems. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. Between these two extrema, there is a class of preconditioners which are "general-purpose" for a particular – although large – class of problems. These preconditioners are sometimes called "gray-box" preconditioners, since the user has to supply few information about the matrix and the problem to be solved.

AztecOO itself implements a variety of preconditioners, from "classical" methods such as Jacobi and Gauss-Seidel, to polynomial and domain-decomposition based preconditioners. More preconditioners can be given to an AztecOO Krylov accelerator, by using the Trilinos packages IFPACK and ML, covered in Chapter 6 and 7, respectively.

5.2 Basic Usage of AztecOO

To solve a linear system with AztecOO, one must create an `Epetra_LinearProblem` object with the command

```
Epetra_LinearProblem Problem(&A,&x,&b);
```

where `A` is an Epetra matrix, and `x,b` two Epetra vectors[5]. Then, the user must create an AztecOO object,

```
AztecOO Solver(Problem);
```

and specify how to solve the linear system. All AztecOO options are set using two vectors, `options` (integer) and `params` (double), as detailed in the Aztec's User Guide.

To choose among the different AztecOO parameters, the user can create two vectors, usually called `options` and `params`, set them to the default values, and then override with the desired parameters: Default values can be set by

```
int    options[AZ_OPTIONS_SIZE];
double params[AZ_PARAMS_SIZE];
AZ_defaults(options, params);
```

followed by, for instance,

```
Solver.SetAllAztecOptions( options );
Solver.SetAllAztecParams( params );
```

[5]At the current stage of development, AztecOO does not handle Epetra_MultiVectors. It accepts Multi_Vectors, but it will solve the linear system corresponding to the first multivector only.

Those two functions will copy the values of `options` and `params` in internal variables of the AztecOO object.

Alternatively, it is possible to set specific parameters without creating `options` and `params`, using the AztecOO methods `SetAztecOption()` and `SetAztecParams()`. For instance,

```
Solver.SetAztecOption( AZ_precond, AZ_Jacobi );
Solver.SetAztecParams( AZ_tol, 1e-12 );
```

to specify a point Jacobi preconditioner, and a tolerance of 10^{-12}. (We refer to the Aztec documentation for more details about the various Aztec settings.)

To solve the linear system the user may call

```
Solver.Iterate(1000,1E-9);
```

The complete code is in `${TRILINOS_HOME}/doc/tutorial/aztec/ex1.cpp`.

Note that the matrix must be in local coordinates (that is, the command `A.FillComplete()` has been called before attempting to solve the linear system). Note also that the procedure to solve a linear system with AztecOO is identical for sequential and parallel runs. However (for certain choices of the preconditioners), the convergence rate can change as the number of processes used in the computation varies.

When this function returns, one can retrieve the number of iterations performed by the linear solver using `Solver.NumIters()`, while `Solver.TrueResidual()` gives the (nonscaled) norm the residual.

5.3 One-level Domain Decomposition Preconditioners with AztecOO

In this Section, we will consider preconditioners based on one-level overlapping domain decomposition preconditioners, of the form

$$P^{-1} = \sum_{i=1}^{M} R_i^T \tilde{A}_i^{-1} R_i, \tag{2}$$

where P is the preconditioning operator, M the number of subdomains. R_i is a rectangular matrix, composed by 0's and 1's, which restricts a global vector to the subspace defined by

the interior of each subdomain, and $\tilde{A}_i$ is an approximation of

$$A_i = R_i A R_i^T. \tag{3}$$

($\tilde{A}_i$ can be equal to A_i). Typically, $\tilde{A}_i$ differs from A_i when incomplete factorizations are used in (2) to apply $\tilde{A}_i^{-1}$, or when a matrix different from A is used in (3).

In order to use a preconditioner of the form (3), the user has to specify

```
Solver.SetAztecOption( AZ_precond, AZ_dom_decomp );
```

followed by the choice of incomplete factorization (and possibly with that of corresponding parameters, for instance the level-of-fill),

```
Solver.SetAztecOption( AZ_ilu, AZ_subdomain_solve );
Solver.SetAztecOption( AZ_graph_fill, 1 );
```

By default, AztecOO will consider zero-overlap among the rows of A[6]. However, this value of overlap can be changed by, for instance,

```
Solver.SetAztecOption( AZ_overlap, 1 );
```

Remark 12. *By using AztecOO in conjunction with ML, one can easily implement a two-level domain decomposition schemes. The reader is referred to Section 7.3.*

Remark 13. *Another Trilinos package can be used to compute incomplete factorizations, IFPACK. It is covered in Chapter 6.*

5.4 Use of AztecOO Problems as a Preconditioner for AztecOO

One may wish to use an AztecOO solver in the preconditioning phase, as done in `${TRILINOS_HOME}/doc/`. The main steps are here reported.

First, we have to specify the linear problem to be solved (set the linear operator, the solution and the right-hand side), and create an AztecOO object:

[6]For point matrices arising from the FE discretization of the PDE problem with local functions, this is equivalent to one mesh element of overlap.

```
Epetra_LinearProblem A_Problem(&A, &x, &b);
AztecOO A_Solver(A_Problem);
```

Now, we have to define the preconditioner. For the sake of simplicity, we here suppose to use the same Epetra_Matrix A in the preconditioning phase. However, the two matrices can in principle be different (although of the same size).

```
Epetra_CrsMatrix P(A);
```

(This operation is in general expensive as involves the copy constructor.) Then, we create the linear problem which will be used as preconditioner. This requires several steps. (Note that all the P prefix identifies preconditioner' objects.)

1. We create the linear system solve at each prec step, and and we assign the linear operator (in this case, the matrix A itself)

   ```
   Epetra_LinearProblem P_Problem;
   P_Problem.SetOperator(&P);
   ```

2. As we wish to use AztecOO to solve the prec step (in a recursive way), we have to define an AztecOO object:

   ```
   AztecOO P_Solver(P_Problem);
   ```

3. Now, we customize certain parameters:

   ```
   P_Solver.SetAztecOption(AZ_precond, AZ_Jacobi);
   P_Solver.SetAztecOption(AZ_output, AZ_none);
   P_Solver.SetAztecOption(AZ_solver, AZ_cg);
   ```

4. The last step is to create an AztecOO_Operator, so that we can set the Aztec's preconditioner with, and we set the user's defined preconditioners:

   ```
   AztecOO_Operator
   P_Operator(&P_Solver, 10);
   A_Solver.SetPrecOperator(&P_Operator);
   ```

   (Here 10 is the maximum number of iterations of the AztecOO solver in the preconditioning phase.)

5. Finally, we solve the linear system:

```
int Niters=100;
A_Solver.SetAztecOption(AZ_kspace, Niters);
A_Solver.SetAztecOption(AZ_solver, AZ_gmres);
A_Solver.Iterate(Niters, 1.0E-12);
```

5.5 Concluding Remarks

The following methods are often used:

- `NumIters()` returns the total number of iterations performed on *this* problem;
- `TrueResidal()` returns the true unscaled residual;
- `ScaledResidual()` returns the unscaled residual;
- `SetAztecDefaults()` can be used to restore default values in the options and params vectors.

The official documentation of AztecOO can be found in [8].

6 Incomplete Factorizations with IFPACK

IFPACK provides a suite of object-oriented algebraic preconditioners for the solution of preconditioned iterative solvers. IFPACK offers a variety of overlapping (one-level) Schwarz preconditioners, The packages uses Epetra for basic matrix-vector calculations, and accepts user matrices via abstract matrix interface. A concrete implementation for Epetra matrices is provided. The package separates graph construction for factorization, improving performances in a substantial manner with respect to other factorization packages.

In this Chapter we present how to use IFPACK objects as a preconditioner for an AztecOO solver.

In this Chapter, we will

- Set the notation (in Section 6.1);
- Show how to compute incomplete Cholesky factorizations (in Section 6.2);
- Present IFPACK's RILU-type factorizations (in Section 6.3).

6.1 Theoretical Background

Aim of this Section is to briefly present some aspects on incomplete factorization methods, to establish a notation. The Section is not supposed to be exhaustive, nor complete on this subject. The reader is referred to the existing literature for a rigorous presentation.

A broad class of effective preconditioners is based on incomplete factorization of the linear system matrix, and it is usually indicated as ILU. The ILU-type preconditioning techniques lie between direct and iterative methods and provide a balance between reliability and numerical efficiency.

The preconditioner is given in the factored form $P = \tilde{L}\tilde{U}$, with $\tilde{L}$ and $\tilde{U}$ being lower and upper triangular matrices. Solving with P involves two triangular solutions.

The incomplete LU factorization of a matrix A can be described as follows. Let $A_0 = A$. Then, for $k = 2, \ldots, n$, we have

$$A_{k-1} = \begin{pmatrix} B_k & F_k \\ E_k & C_k \end{pmatrix}.$$

Thus, we can write the $k-$step of the Gaussian elimination in a block form as

$$A_{k-1} = \begin{pmatrix} I & 0 \\ E_k B_k^{-1} & I \end{pmatrix} \begin{pmatrix} B_k & F_k \\ 0 & A_k \end{pmatrix},$$

where $A_k = C_k - E_k B_k^{-1} F_k$. If B_k is a scalar, then we have the typical point-wise factorization, otherwise we have a block factorization. Pivoting, if it is necessary, can be accomplished by reordering A_k at every step.

To make the factorization incomplete, entries as dropped in A_k, i.e. the factorization proceeds with

$$\tilde{A}_k = A_k - R_k$$

where R_k is the matrix of dropped entries.

Dropping can be performed by position, for example, dropping those entries in the update matrix $E_k B_k^{-1} F_k$ that are not in the pattern of C_k. This simple ILU factorization is known as ILU(0). Although effective, in some cases the accuracy of the ILU(0) may be insufficient to yield an adequate rate of convergence. More accurate factorizations will differ from ILU(0) by allowing some *fill-in*. The resulting class of methods is called ILU(f), where f is the level-of-fill. A level-of-fill is attributed to each element that is processed by Gaussian elimination, and dropping will be based on the level-of-fill. The level-of-fill should be indicative of the size of the element: the higher the level-of-fill, the smaller the elements.

Other strategies consider dropping by value – for example, dropping entries smaller than a prescribed threshold. Alternative dropping techniques can be based on the numerical size of the element to be discarded. Numerical dropping strategies generally yield more accurate factorizations with the same amount of fill-in than level-of-fill methods. The general strategy is to compute an entire row of the $\tilde{L}$ and $\tilde{U}$ matrices, and then keep only the biggest entries in a certain number. In this way, the amount of fill-in is controlled; however, the structure of the resulting matrices is undefined. These factorizations are usually referred to as ILUT, and a variant which performs pivoting is called ILUTP.

6.2 Incomplete Cholesky Factorizations

Ifpack_CrsIct is a class for constructing and using incomplete Cholecky factorizations of an Epetra_CrsMatrix. The factorization is produced based on several parameters:

- Maximum number of entries per row/column. The factorization will contain at most this number of nonzero elements in each row/column;

- Diagonal perturbation. By default, the factorization will be computed on the input matrix. However, it is possible to modify the diagonal entries of the matrix to be factorized, via functions `SetAbsoluteThreshold()` and `SetRelativeThreshold()`. Refer to the IFPACK's documentation for more details.

It is very easy to compute the incomplete factorization. First, define an Ifpack_CrsIct object,

```
Ifpack_CrsIct * ICT = NULL;
ICT = Ifpack_CrsIct(A,DropTol,LevelFill);
```

where `A` is an Epetra_CrsMatrix (already FillComplete'd), and `DropTop` and `LevelFill` are the drop tolerance and the level-of-fill, respectively. Then, we can set the values and compute the factors,

```
ICT->InitValues(A);
ICT->Factor();
```

IFPACK can compute the estimation of the condition number

$$cond(L_i U_i) \approx \|(LU)^{-1} e\|_\infty,$$

where $e = (1, 1, \ldots, 1)^T$. (More details can be found in the IFPACK's documentation.) This estimation can be computed as follows:

```
double Condest;
ICT->Condest(false,Condest);
```

Please refer to file `${TRILINOS_HOME}/doc/tutorial/ifpack/ex1.cpp` for a complete example of incomplete Cholesky factorization.

6.3 RILU Factorizations

IFPACK implements various incomplete factorization for non-symmetric matrices. In this Section, we will consider the Epetra_CrsRiluk class, that can be used to produce RILU factorization of a Epetra_CrsMatrix. The class required an Ifpack_OverlapGraph in the construction phase. This means that the factorization is split into two parts:

1. Definition of the level filled graph;
2. Computation of the factors.

This approach can significantly improve the performances of code, when an ILU preconditioner has to be computed for several matrices, with different entries but with the same sparsity pattern. An Ifpack_IlukGraph object of an Epetra matrix `A` can be constructed as

```
Ifpack_IlukGraph Graph =
  Ifpack_IlukGraph(A.Graph(),LevelFill,LevelOverlap);
```

Here, `LevelOverlap` is the required overlap among the subdomains.

A call to `ConstructFilledGraph()` completes the process.

Remark 14. *An Ifpack_IlukGraph object has two Epetra_CrsGraph objects, containing the L_i and U_i graphs. Thus, it is possible to manually insert and delete graph entries in L_i and U_i via the Epetra_CrsGraphInsertIndices and RemoveIndices functions. However, in this cas FillComplete must be called before the graph is used for subsequent operations.*

At this point, we can create an Ifpack_CrsRiluk object,

```
ILUT = Ifpack_CrsRiluk(Graph);
```

This phase defined the graph for the incomplete factorization, without computing the actual values of the L_i and U_i factors. Instead, this operation is accomplished with

```
int initerr = ILUT->InitValues(A);
```

The ILUT object can be used with AztecOO simply setting

```
solver.SetPrecOperator(ILUK);
```

where `solver` is an AztecOO pbject. Example `${TRILINOS_HOME}/doc/tutorial/ifpack/ex2.c` shows the use of Ifpack_CrsRiluk class.

The application of the incomplete factors to a global vector, $z = (L_i U_i^{-1})r$, results in redundant approximation for any element of z that correspond to rows that are part of

more than one local ILU factor. The OverlapMode defines how those redundant values are managed. OverlapMode is an Epetra_CombinedMode enum, that can assume the following values: `Add`, `Zero`, `Insert`, `Average`, `AbxMax`. The default is to zero out all the values of z for rows that were not part of the original matrix row distribution.

6.4 Concluding Remarks

More documentation on the IFPACK package can be found in [6, 4].

7 Multilevel Methods with ML

The ML package defines a class of preconditioners based on multilevel methods [18]. While technically any linear system can be considered, ML should be used on linear systems on linear systems, like elliptic PDEs, that are known to work well with multilevel methods.

ML is a large package, that can be used to a variety of purposes. ML provides multilevel solvers, as well as multilevel preconditioners, and it can handle geometric as well as algebraic methods.

In this Chapter we present:

- Outline the basic issues of multilevel schemes (in Section 7.1);
- Present the use of ML objects as a preconditioner for an AztecOO solver objects (in Section 7.2);
- Outline the steps required to implement two-level domain decomposition methods, with a coarse grid defined using aggregation procedures (in Section 7.3).

As other Trilinos packages, ML can be compiled and run independently from Epetra, that is, it can accept input matrix in formats different from the Epetra_RowMatrix or Epetra_Operator. Should the reader be interested in running ML without Epetra, or using a C code (and not a C++ code), then we refer to the ML guide, contained in the `${TRILINOS_HOME}/packages/ml/doc/`.

7.1 Theoretical Background

Aim of this Section is to briefly present some aspects on multilevel methods. The Section is not supposed to be exhaustive, nor complete on this subject. The reader is referred to the existing literature for a rigorous presentation.

Multilevel methods require the operator to be defined on a sequence of coarser spaces, an iterative method that evolves the solution (called a smoother) and interpolation operators that transfer information between the spaces. The principle behind the algorithm is that the high-frequency errors can be efficiently solved on the fine space, while the low-frequency are treated on the coarser one, where there frequencies manifest themselves as

high-frequencies. A very popular multilevel methods are multigrid methods. Geometric multigrid (GMG) methods cannot be applied without the existence of an underlying grid (this is their major limitation). This led to the development of algebraic multigrid method (AMG), initiated by Ruge and Stüben. In AMG, both the matrix hierarchy and the prolongation operators are constructed just from the stiffness matrix. Since the automatic generation of a grid-hierarchy for GMG and especially the proper assembly of all components would be a very difficult task for unstructured problems, the automatic algebraic construction of a virtual grid is a big advantage.

A function to solve (1) using a multilevel method can be defined as follows:

```
MGM( X, B, k)
{
  if( k == 0 ) X = A_k \ B;
  else {
    X = S_k^1 (X, B);
    D = R_{k-1,k} ( B - A_k X );
    V = 0;
    MGM( V, D, k-1 );
    X = X + P_{k,k-1} V;
    X = S_k^2( U, B );
  }
}
```

In the above method, S_k^1 and S_k^2 are two smoothers, $R_{k-1,k}$ is a restriction operator from level k to $k-1$, and $P_{k,k-1}$ is a prolongator from $k-1$ to k.

In a variational setting, the matrices A_k can be constructed as

$$A_k = R_{k-1,k} A_k P_{k,k-1}.$$

Alternatively, when a grid is available at level $k-1$, one can discretize the PDE operator on grid $k-1$.

Remark 15. *In this tutorial, we will consider multilevel methods based on aggregation schemes only.*

7.2 ML as a Preconditioner for AztecOO

In order to use ML as a preconditioner, we need to define an AztecOO Solver, as outlined in Chapter 5.

ML requires the user to define a structure, to store internal data. This structure is usually called `ml_handle`:

```
ML *ml_handle;
```

We intend to use ML as a "black-box" (or gray-box) multilevel preconditioner, using aggregation procedures to define the multilevel hierarchy. The variable

```
int N_levels = 10;
```

defines the maximum number of levels, while

```
ML_Set_PrintLevel(3);
```

toggle the output level (from 0 to 10, 10 being verbose mode and 0 silent mode).

The ML handle is created using

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

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);
```

Note that `A` is *not* converted to ML format. Instead, proper wrappers are defined. (Here, `A` is the Epetra matrix for which we aim to construct a multilevel preconditioner.)

ML requires another structure, called ML_Aggregate, to store the information about the aggregates at various levels:

```
ML_Aggregate *agg_object;
ML_Aggregate_Create(&agg_object);
```

The multilevel hierarchy is constructed with the instruction

```
N_levels = ML_Gen_MGHierarchy_UsingAggregation(ml_handle, 0,
                                               ML_INCREASING,
                                               agg_object);
```

Here, `0` is the index of the finest level, and the index of coarser levels will be obtained by incrementing this value. (We refer to the ML manual for more details about the input parameters.)

We still need to define the smoother, for instance a symmetric Gauss-Seidel:

```
ML_Gen_Smoother_SymGaussSeidel(ml_handle, ML_ALL_LEVELS,
                               ML_BOTH, 1, ML_DEFAULT);
```

and to generate the solver as

```
ML_Gen_Solver     (ml_handle, ML_MGV, 0, N_levels-1);
```

Finally, we can create an Epetra_Operator, based on the previously defined ML hierarchy

```
Epetra_ML_Operator  MLop(ml_handle,comm,map,map);
```

and set the preconditioning operator of our AztecOO solver,

```
solver.SetPrecOperator(&MLop);
```

At this point, we can call `Iterate()` as usual,

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

The entire code is reported in `${TRILINOS_HOME}/doc/tutorial/ml/ex1.cpp`. The output will be approximatively as reported below.

```
[msala:ml]> mpirun -np 2 ./ex1.exe
*****************************************************************
* ML Aggregation information                                    *
```

```
============================================================
ML_Aggregate : ordering            = natural.
ML_Aggregate : min nodes/aggr      = 2
ML_Aggregate : max neigh selected = 0
ML_Aggregate : attach scheme       = MAXLINK
ML_Aggregate : coarsen scheme      = UNCOUPLED
ML_Aggregate : strong threshold    = 0.000000e+00
ML_Aggregate : P damping factor    = 1.333333e+00
ML_Aggregate : number of PDEs      = 1
ML_Aggregate : number of null vec = 1
ML_Aggregate : smoother drop tol   = 0.000000e+00
ML_Aggregate : max coarse size     = 1
ML_Aggregate : max no. of levels   = 10
************************************************************
ML_Gen_MGHierarchy : applying coarsening
ML_Aggregate_Coarsen begins
ML_Aggregate_CoarsenUncoupled : current level = 0
ML_Aggregate_CoarsenUncoupled : current eps = 0.000000e+00
Aggregation(UVB) : Total nonzeros = 128 (Nrows=30)
Aggregation(UC) : Phase 0 - no. of bdry pts  = 0
Aggregation(UC) : Phase 1 - nodes aggregated = 28 (30)
Aggregation(UC) : Phase 1 - total aggregates = 8
Aggregation(UC_Phase2_3) : Phase 1 - nodes aggregated = 28
Aggregation(UC_Phase2_3) : Phase 1 - total aggregates = 8
Aggregation(UC_Phase2_3) : Phase 2a- additional aggregates = 0
Aggregation(UC_Phase2_3) : Phase 2 - total aggregates = 8
Aggregation(UC_Phase2_3) : Phase 2 - boundary nodes   = 0
Aggregation(UC_Phase2_3) : Phase 3 - leftovers = 0 and singletons = 0
 Aggregation time       = 1.854551e-03
Gen_Prolongator : max eigen = 1.883496e+00
ML_Gen_MGHierarchy : applying coarsening
ML_Gen_MGHierarchy : Gen_RAP
RAP time for level  0 = 5.319577e-04
ML_Gen_MGHierarchy : Gen_RAP done
ML_Gen_MGHierarchy : applying coarsening
ML_Aggregate_Coarsen begins
ML_Aggregate_CoarsenUncoupled : current level = 1
ML_Aggregate_CoarsenUncoupled : current eps = 0.000000e+00
Aggregation(UVB) : Total nonzeros = 46 (Nrows=8)
Aggregation(UC) : Phase 0 - no. of bdry pts  = 0
```

```
Aggregation(UC) : Phase 1 - nodes aggregated = 6 (8)
Aggregation(UC) : Phase 1 - total aggregates = 2
Aggregation(UC_Phase2_3) : Phase 1 - nodes aggregated = 6
Aggregation(UC_Phase2_3) : Phase 1 - total aggregates = 2
Aggregation(UC_Phase2_3) : Phase 2a- additional aggregates = 0
Aggregation(UC_Phase2_3) : Phase 2 - total aggregates = 2
Aggregation(UC_Phase2_3) : Phase 2 - boundary nodes   = 0
Aggregation(UC_Phase2_3) : Phase 3 - leftovers = 0 and singletons = 0
 Aggregation time       = 1.679042e-03
Gen_Prolongator : max eigen = 1.246751e+00
ML_Gen_MGHierarchy : applying coarsening
ML_Gen_MGHierarchy : Gen_RAP
RAP time for level  1 = 4.489557e-04
ML_Gen_MGHierarchy : Gen_RAP done
ML_Gen_MGHierarchy : applying coarsening
ML_Aggregate_Coarsen begins
Aggregation total setup time = 8.903003e-02 seconds
Smoothed Aggregation : operator complexity = 1.390625e+00.

          ***********************************************************
          ***** Preconditioned CG solution
          ***** Epetra ML_Operator
          ***** No scaling
          ***********************************************************

          iter:    0           residual = 1.000000e+00
          iter:    1           residual = 1.289136e-01
          iter:    2           residual = 4.710371e-03
          iter:    3           residual = 7.119470e-05
          iter:    4           residual = 1.386302e-06
          iter:    5           residual = 2.477133e-08
          iter:    6           residual = 6.141025e-10
          iter:    7           residual = 6.222216e-12
          iter:    8           residual = 1.277534e-13


          Solution time: 0.005845 (sec.)
          total iterations: 8
Residual    = 6.99704e-13
```

7.3 Two-level Domain Decomposition Preconditioners with ML

In order to use the example reported in this Section, one should compile ML with the configure flag `--with-ml_metis`. In this way, ML will use the graph decomposition library METIS to create the coarse-level matrix[7].

Two-level domain decomposition methods have been proved to be very effective for the iterative solution of many different kind of linear systems. For some classes of problems, a very convenient way to define the coarse grid operator is to use aggregation procedure. This is very close to what presented in Section 7.2. The main difference is that only two level methods are considered, and that the coarse grid remains of (relatively) small size. The idea is to define a small number of aggregates on each process, using a graph decomposition algorithm (as implemented in the library METIS, for instance)[8]. This can be done as follows.

First, we need to define an AztecOO problem, an ML structure, and an ML_Aggregate structure. Then, we limit ourself to 2-level scheme.

```
int N_levels = 2;
```

Then, we specify the aggregation scheme as

```
ML_Aggregate_Set_CoarsenScheme_METIS(agg_object);
```

and define the number of aggregates (here, 4) to be defined on each process as

```
ML_Aggregate_Set_LocalNumber( ml_handle, agg_object, 0, 4 );
```

As smoother, we can adopt a subdomain-based Gauss-Seidel smoother.

The creation of the multilevel hierarchy and the solution of the linear system will be as reported in Section 7.2.

The entire code is reported in `${TRILINOS_HOME}/doc/tutorial/ml/ex2.cpp`.

[7]Note that ML has to be aware of the location of the METIS include files and the METIS library. The user can use the configure flags `--with-incdirs` and `--with-ldflags`. Please type `configure --help` for more information. If you don't have METIS, or you don't want to re-configure ML, you will be able to run the example of this Section. However, you will be limited to use only one aggregate per process.

[8]Aggregation schemes based on ParMETIS as also available. Please refer to the help of the ML `configure` for more details.

7.4 Concluding Remarks

More documentation about ML can be found in [21, 19, 19].

8 Interfacing Direct Solvers with Amesos

The Amesos package provides an object-oriented interface to several direct sparse solvers. Amesos will solve (using a direct factorization method) the linear systems of equations

$$AX = B \tag{4}$$

where A is stored as an Epetra_RowMatrix object, and X and B are Epetra_MultiVector objects.

The Amesos package has been designed to face some of the challenges of direct solution of linear systems. In fact, many solvers have been proposed in the last years, and often each of them requires different input formats for the linear system matrix. Moreover, it is not uncommon that the interface changes between revisions. Amesos aims to solve those problems, furnishing a clean, consistent interface to many direct solvers.

Using Amesos, users can interface their codes with a (large) variety of direct linear solvers, sequential or parallel, simply by a code instruction of type

```
AmesosProblem.Solver();
```

Amesos will take care of redistributing data among the processors, if necessary.

This Chapter starts with few notes on the installation of the third-part packages required by Amesos. Then, the Chapter will present the use of Amesos objects, to interface with the following packages:

- UMFPACK, version 4.1 (in Section 8.2);
- SuperLUdist, version 2.0 (in Section 8.3);
- A generic interface to various direct solvers is presented (in Section 8.4).

8.1 Installation of Trilinos third-part Packages

Amesos is an interface to other packages, mainly developed outside the Trilinos framework[9]. In order to use those packages, the user should carefully check copyright and licensing of those third party codes. Please refer to the web page or the documentation of each particular package for details.

[9]Currently, SuperLU is included in the Trilinos framework.

Amesos supports a variety of direct solvers for linear systems of equations, and you are likely to use Amesos with only few of them. We suggest to define the shell variable `TRILINOS_3PL` to define the directory used to stored third-part packages. For instance, under BASH, you may have a line of type

```
export TRILINOS_3PL=/home/msala/Trilinos3PL
```

in your `.bashrc` file. Then, you may decide to create a directory to hold include files and libraries. For instance, to compile under LINUX with MPI:

```
$ mkdir ${TRILINOS_3PL}/LINUX_MPI
$ mkdir ${TRILINOS_3PL}/LINUX_MPI/include
$ mkdir ${TRILINOS_3PL}/LINUX_MPI/lib
```

(Note that this will reflect the directory structure used by Trilinos, see Section 1.2.) While installing a package, you can now copy all include files and libraries in these directories.

Using this setting, you can configure Amesos with a command of type

```
$ cd ${TRILINOS_HOME}/packages/amesos
$ ./configure --prefix=${TRILINOS_HOME}/LINUX_MPI \
  --enable-mpi --with-mpi-compilers \
  --enable-amesos-umfpack \
  --enable-amesos-superludist \
  --with-amesos-superludistlib=\
  "${TRILINOS_3PL}/SuperLU_DIST_2.0/libsuperlu_LINUX.a"
```

(This command is followed by `make` and `make install`, as usual.) This will enable UMFPACK and SuperLUdist, which are the two packages covered in this Chapter.

For more details about the configuration options of Amesos, please refer to Amesos documentation.

8.2 UMFPACK

File `${TRILINOS_HOME}/doc/tutorial/amesos/ex1.cpp` shows how to use Amesos to solve a linear system with UMFPACK[10].

[10]UMFPACK is a set of routines solving sparse linear systems via LU factorization. It is copyrighted by Timothy A. Davis. More information can be obtained at the web page

Suppose that A, x and b are an Epetra_RowMatrix and two Epetra_MultiVector, respectively, or compatible dimensions. Amesos objects for the solution of linear systems requires an Epetra_LinearProblem object, plus another object, AMESOS::Parameter::List, used to specify the parameters.

```
Epetra_LinearProblem Problem(&A,&x,&b);
AMESOS::Parameter::List params;
```

Then, only few lines are required: We can define an Amesos object and solve the problem,

```
Amesos_Umfpack UmfpackProblem(Problem,params);
UmfpackProblem.Solve();
```

or, alternatively, it is possible to specify when symbolical factorization, numerical factorization and solution occur,

```
Amesos_Umfpack UmfpackProblem(Problem,params);
UmfpackProblem.SymbolicFactorization();
UmfpackProblem.NumericFactorization();
UmfpackProblem.Solve();
```

Note that *exactly* the same code can be run with more than one processor. In this case, being UMFPACK a serial solver, Amesos will take care to gather all required data on a processor, solve sequentially the linear system, and then broadcast the solution.

8.3 SuperLUdist

Solving using SuperLUdist[11] is not much different from what presented in Section 8.2. Instead of declaring an Amesos_Umfpack object, one can proceed as follows:

```
Amesos_Superludist * SuperludistProblem =
 new Amesos_Superludist(Problem,params);
```

http://www.cise.ufl.edu/research/sparse/umfpack.

[11] SuperLU_DIST is a parallel extension to the serial SuperLU library. It is targeted for the distributed memory parallel machines. Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory. Please refer to the web site http://www.nersc.gov/ xiaoye/SuperLU for more information.

followed by a call to Solve(), possibly preceded by SymbolicFactorization() and NumericFactorization().

Remark 16. *We have declared a pointer to and Amesos_Superludist object because the destructor of this object contains some MPI calls. As in example* `${TRILINOS_HOME}/doc/tutorial/amesos/ex2.cpp` *the destructor is called at the end of the main function (after a call to* `MPI_Finalize()`*, we have to delete this object using the C++ statement*

```
delete SuperludisProblem;
```

before the call to `MPI_Finalize()`.

8.4 A Generic Interface to Various Direct Solvers

All Amesos objects are derived from the Amesos_BaseClass object. Using the capabilities of C++, one may decide to code a generic interface to a direct solver as follows:

```
// parameter vector for Amesos
AMESOS::Parameter::List ParamList;

// prepare the linear solver
Amesos_BaseSolver * AmesosProblem;

switch( choice ) {
case ML_SOLVE_WITH_AMESOS_UMFPACK:
    AmesosProblem =
      new Amesos_Umfpack( *LinearProblem, ParamList );
    break;
  case ML_SOLVE_WITH_AMESOS_SUPERLUDIST:
    AmesosProblem =
      new Amesos_Superludist( *LinearProblem, ParamList );
    break;
  default:
    cerr << ''Error'' << endl;
}
```

Now, factorization and solution are the same for all the packages:

```
AmesosProblem->SymbolicFactorization();
AmesosProblem->NumericFactorization();
AmesosProblem = (void *) AmesosProblem ;
```

9 Solving Nonlinear Systems with NOX

NOX is a suite of solution methods for the solution of nonlinear systems of type

$$F(x) = 0, \tag{5}$$

with

$$F(x) = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix}$$

is a nonlinear vector function. The Jacobian matrix of F, J, is defined by

$$J_{i,j} = \frac{\partial F_i}{\partial x_j}(x).$$

NOX aims to solver (5) using Newton-type methods. NOX uses an abstract vector and "group" interface. Current implementation are provided for Epetra/AztecOO objects, but also for LAPACK and PETSc. It provides various strategies for the solution of nonlinear systems, and it has been designed to be easily integrated into existing applications.

In this Chapter, we will

- Outline the basic issued of the solution of nonlinear systems (in Section 9.1);
- Introduce the NOX package (in Section 9.2);
- Describe how to introduce a NOX solver in an existing code (in Section 9.3);
- Present Jacobian-free methods (in Section 9.6).

9.1 Theoretical Background

Aim of this Section is to briefly present some aspects of the solution of nonlinear systems, to establish a notation. The Section is not supposed to be exhaustive, nor complete on this subject. The reader is referred to the existing literature for a rigorous presentation.

To solve system of nonlinear equations, NOX makes use of Newton-like methods. The Newton method defines a suite $\{x_k\}$ that, under some conditions, converges to x, solution

of (5). The algorithm is as follows: given x_0, for $k = 1, \ldots$ until convergence, solve

$$J_k(x_{k-1})\,(x_k - x_{k-1}) = -F(x_{k-1}), \quad J_k(x_{k-1}) = \left[\frac{\partial F}{\partial x}(x_{k-1})\right]. \tag{6}$$

Newton method introduces a local full linearizion of the equation. Solving a system of linear equations at each Newton step can be very expensive if the number of unknowns is large, and may not be justified if the current iterate is far from the solution. Therefore, a departure from the Newton framework consists of considering *inexact* Newton methods, which solve system (6) only approximatively.

In fact, in practical implementation of the Newton method, one or more of the following approximations are used:

1. The Fréchet derivative J_k for the Newton step is not recomputed at every Newton step;
2. The equation for the Newton step (6) is solved only inexactly;
3. Defect-correction methods are employed, that is, J_k is numerically computed using low-order (in space) schemes, while the right-hand side is built up using high-order methods.

For a given initial guess, "close enough" to the solution of (5), the Newton method with exact linear solves converges quadratically. In practice, the radius of convergence is often extended via various methods. NOX provides, among others, line search techniques and trust region strategies.

9.2 Creating NOX Vectors and Group

NOX is not based on any particular linear algebra package. Users are required to supply methods that derive from the abstract classes `NOX::Abstract::Vector` (which provides support for basic vector operations as dot products), and `NOX::Abstract::Group` (which supports the linear algebra functionalities, evaluation of the function G and, optionally, of the Jacobian J).

In order to link their code with NOX, users have to write their own instantiation of those two abstract classes. In this tutorial, we will consider the concrete implementations provided for Epetra matrices and vectors. As this implementation is separate from the

NOX algorithms, the configure option `--enable-nox-epetra` has to be specified (see Section 1.2)[12].

9.3 Introducing NOX in an Existing Code

Two basic steps are required to implement a `NOX::Epetra` interface. First, a concrete class derived from `NOX::Epetra::Interface` has to be written. This class must define the following methods:

1. A method to compute $y = F(X)$ for a given x. The syntax is

   ```
   computeF(const Epetra_Vector & x, Epetra_Vector & y,
            FillType flag)
   ```

   with `x` and `y` two Epetra_Vectors, and `flag` an enumerated type that tells why this method was called. In fact, NOX has the ability to generate Jacobians based on numerical differencing. In this case, users may want to compute an inexact (and hopefully cheaper) F, since it is only used in the Jacobian (or preconditioner).

2. A function to compute the Jacobian, whose syntax is

   ```
   computeJacobian(const Epetra_Vector & x,
                   Epetra_Operator * Jac)
   ```

   This method is optional optional method. It should be implemented when users wish to supply their own evaluation of the Jacobian. If the user does not wish to supply their own Jacobian, they should implement this method so that it throws an error if it is called. This method should update the Jac operator so that subsequent Epetra_Operator::Apply() calls on that operator correspond to the Jacobian at the current solution vector x.

3. A method which fills a preconditioner matrix, whose syntax is

   ```
   computePrecMatrix(const Epetra_Vector & x,
                     Epetra_RowMatrix & M)
   ```

[12]Other two concrete implementation are provided, for LAPACK and PETSc. The user may wish to configure NOX with `--enable-nox-lapack` or `--enable-nox-petsc`. Examples can be compiled with the options `--enable-nox-lapack-examples`, `--enable-nox-petsc-examples`, and `-enable-nox-epetra-exemples`.

It should only contain an estimate of the Jacobian. If users do not wish to supply their own Preconditioning matrix, they should implement this method such that if called, it will throw an error.

4. A method to apply the user's defined preconditioner. The syntax is

   ```
   computePreconditioner(const Epetra_Vector & x, Epetra_Operator & M)
   ```

   The method should compute a preconditioner based upon the solution vector x and store it in the Epetra_Operator M. Subsequent calls to the Epetra_Operator::Apply method will apply this user supplied preconditioner to epetra vectors.

Then, the user can construct a `NOX::Epetra::Group`, which contains information about the solution technique. All constructors require:

- A parameter list for printing output and for input options, defined as `NOX::Parameter::List`.
- An initial guess for the solution (stored in an Epetra_Vector object);
- an operator for the Jacobian and (optionally) and operator for the preconditioning phase. Users can write their own operators. In particular, the Jacobian can be defined by the user as an Epetra_Operator,

  ```
  Epetra_Operator & J = UserProblem.getJacobian(),
  ```

  created as a NOX matrix-free operator,

  ```
  NOX::Epetra::MatrixFree & J = MatrixFree(userDefinedInterface,
                                           solutionVec),
  ```

  or created by NOX using a finite differencing,

  ```
  NOX::Epetra::FiniteDifference & J = FIXME...
  ```

At this point, users have to create an instantiation of the `NOX::Epetra::Interface` derived object,

```
UserInterface interface(...),
```

and finally construct the group,

```
NOX::Epetra::Group gourp(printParams, lsParams, interface, FIXME).
```

9.4 A Simple Nonlinear Problem

As an example. define $F : \mathbb{R}^2 \to \mathbb{R}^2$ by

$$F(x) = \begin{pmatrix} x_1^2 + x_2^2 - 1 \\ x_2 - x_1^2 \end{pmatrix}.$$

With this choice of F, the exact solutions of (5) are the intersections of the unity circle and the parabola $x_2 - x_1^2$. Simple algebra shows that one solution lies in the first quadrant, and has coordinates

$$\alpha = \left(\sqrt{\frac{\sqrt{5}-1}{2}}, \frac{\sqrt{5}-1}{2} \right),$$

the other being the reflection of α among the x_2 axis.

Code `${TRILINOS_HOME}/doc/tutorial/nox/ex1.cpp` applies the Newton method to this problem, with $x_0 = (0.5, 0.5)$ as a starting solution. The output is approximatively as follows:

```
[msala:nox]> mpirun -np 1 ./ex1.exe
************************************************************
-- Nonlinear Solver Step 0 --
f = 5.590e-01  step = 0.000e+00  dx = 0.000e+00
************************************************************

************************************************************
-- Nonlinear Solver Step 1 --
f = 2.102e-01  step = 1.000e+00  dx = 3.953e-01
************************************************************

************************************************************
-- Nonlinear Solver Step 2 --
f = 1.009e-02  step = 1.000e+00  dx = 8.461e-02
************************************************************

************************************************************
-- Nonlinear Solver Step 3 --
f = 2.877e-05  step = 1.000e+00  dx = 4.510e-03 (Converged!)
************************************************************
```

```
**************************************************
-- Final Status Test Results --
Converged....OR Combination ->
  Converged....F-Norm = 2.034e-05 < 2.530e-04
                (Length-Scaled Two-Norm, Relative Tolerance)
  ??..........Number of Iterations = -1 < 20
**************************************************

-- Parameter List From Solver --
Direction ->
  Method = "Newton"    [default]
  Newton ->
    Linear Solver ->
      Max Iterations = 400    [default]
      Output ->
        Achieved Tolerance = 8.6e-17    [unused]
        Number of Linear Iterations = 2    [unused]
        Total Number of Linear Iterations = 6    [unused]
      Tolerance = 1e-10    [default]
    Rescue Bad Newton Solve = true    [default]
Line Search ->
  Method = "More'-Thuente"
  More'-Thuente ->
    Curvature Condition = 1    [default]
    Default Step = 1    [default]
    Interval Width = 1e-15    [default]
    Max Iters = 20    [default]
    Maximum Step = 1e+06    [default]
    Minimum Step = 1e-12    [default]
    Optimize Slope Calculation = false    [default]
    Recovery Step = 1    [default]
    Recovery Step Type = "Constant"    [default]
    Sufficient Decrease = 0.0001    [default]
    Sufficient Decrease Condition = "Armijo-Goldstein"    [default]
  Output ->
    Total Number of Failed Line Searches = 0    [unused]
    Total Number of Line Search Calls = 3    [unused]
    Total Number of Line Search Inner Iterations = 0    [unused]
    Total Number of Non-trivial Line Searches = 0    [unused]
Nonlinear Solver = "Line Search Based"
```

```
Output ->
  2-Norm of Residual = 2.88e-05    [unused]
  Nonlinear Iterations = 3    [unused]
Printing ->
  MyPID = 0    [default]
  Output Information = 2
  Output Precision = 3    [default]
  Output Processor = 0    [default]
Computed solution :
Epetra::Vector
     MyPID           GID                 Value
         0             0                 0.786
         0             1                 0.618
Exact solution :
Epetra::Vector
     MyPID           GID                 Value
         0             0                 0.786
         0             1                 0.618
```

9.5 A 2D Nonlinear PDE Problem

In this Section, we consider the solution of the following nonlinear PDE problem:

$$\begin{cases} -\Delta u + \lambda e^u &= 0 \quad \text{in } \Omega = (0,1) \times (0,1) \\ u &= 0 \quad \text{on } \partial\Omega. \end{cases} \tag{7}$$

For the sake of simplicity, we use a finite difference scheme ona Cartesian gri, with constant mesh sizes h_x and h_y. Using standard procedures, the discrete equation at node (i, j) reads

$$\frac{-u_{i-1,j} + 2u_{i,j} - u_{i+1,j}}{h_x^2} + \frac{-u_{i,j-1} + 2u_{i,j} - u_{i,j+1}}{h_y^2} - \lambda e^{u_{i,j}} = 0.$$

In example `${TRILINOS_HOME}/doc/tutorial/nox/ex2.cpp`, we build the Jacobian matrix as an Epetra_CrsMatrix, and we use NOX to solve problem (7) for a given value of λ. The example shows how to use NOX for more complex cases. The code defines a class, here called PDEProblem, which contains two main methods: One to compute $F(x)$ for a given x, and the other to update the entries of the Jacobian matrix. The class contains all the problem definitions (here, the number of nodes along the x-axis and the y-axis and the value of λ). In more complex cases, a similar class may have enough information to

compute, for instance, the entries of J using a finite-element approximation of the PDE problem.

The interface to NOX, here called SimpleProblemInterface, accepts a PDEProblem as a constructor,

```
SimpleProblemInterface Interface(&Problem);
```

Once a NOX::Epetra:Interface object has been defined, the procedure is almost identical to that of the previous Section.

9.6 Jacobian-free Methods

In Section 9.5, the entries of the Jacobian matrix have been explicitly coded. FOr more complex discretization schemes, it is not always possible nor convenient to compute the exact entries of J. For those cases, NOX can automatically build Jacobian matrices based on finite difference approximation, that is,

$$J_{i,j} = \frac{F_i(u + h_j e_j) - F_i(x)}{h_j},$$

where e_j is the j-unity vector. Sophisticated schemes are provided by NOX, to reduce the number of function evaluations.

9.7 Concluding Remarks

The documentation of NOX can be found in [13].

A library of continuation classes, called LOCA [14, 16], is included in the NOX distribution. LOCA is a generic continuation and bifurcation analysis package, designed for large-scalr applications.The algorithms are designed with minimal interface requirements over that needed for a Newton method to read an equilibrium solution. LOCA is built upon the NOX package. LOCA provided functionalities for single parameter continuation and multiple continuation. Also, LOCA provides a stepper class that repeatedly class the NOX nonlinear solver to compute points along a continuation curve. We will not cover LOCAL in this tutorial. The interested reader is referred to the LOCA documentation.

10 TriUtils

Triutils is a collection of various utilities, that can help development and testing. Mainly, triutils contains functions or classes to generate matrices in various formats (MSR, VBR, Epetra), to read matrices (in HB or COO format), to convert matrices from one format to another, and to process the command line. Programs using triutils should include the file `Trilinos_Util.h`.

In this Chapter, we will present:

- How to read a matrix (and possibly right-hand side and solution vectors) from an Harwell/Boeing file format (in Section 10.1);
- How to retrive a parameter specified on the command line (in Section 10.2).

10.1 Reading a HB problem

It is possible to read matrix, solution and right-hand side, from a file written in the Harwell/Boeing format. This is done in `${TRILINOS_HOME}/doc/tutorial/triutils/ex1.cpp`. The key instructions are the following.

First, we define pointers to Epetra_Vector and Epetra_Matrix objects:

```
// Pointers because of Trilinos_Util_ReadHb2Epetra
Epetra_Map * readMap;
Epetra_CrsMatrix * readA;
Epetra_Vector * readx;
Epetra_Vector * readb;
Epetra_Vector * readxexact;
```

The HB problem is read with the instruction

```
Trilinos_Util_ReadHb2Epetra(FileName, Comm, readMap, readA, readx,
                            readb, readxexact);
```

Here, `Comm` is an Epetra_SerialComm or Epetra_MpiComm object, and `FileName` an array of character containing the name of the HB file.

This creates an Epetra_Matrix and two Epetra_Vectors, with all the elements assigned to processor zero. This is because the HB file does not contain any information about the distribution of the elements to the processors. Should the user need to solve the linear problem in parallel, thus he has to redistributed `readA`. In this case, the first step is to specify a map. For instance, we can use a linear map:

```
int NumGlobalElements = readMap->NumGlobalElements();
Epetra_Map map(NumGlobalElements,0,Comm);
```

and create and exporter to distribute read-in matrix and vectors:

```
Epetra_Export exporter(*readMap, map);
Epetra_CrsMatrix A(Copy, map, 0);
Epetra_Vector x(map);
Epetra_Vector b(map);
Epetra_Vector xexact(map);
// this is the data distribution phase
x.Export(*readx, exporter, Add);
b.Export(*readb, exporter, Add);
xexact.Export(*readxexact, exporter, Add);
A.Export(*readA, exporter, Add);
```

Finally, we can destroy the objects used to store the non-distributed HB problem:

```
delete readA;
delete readx;
delete readb;
delete readxexact;
delete readMap;
```

and solve the distributed linear system with the method of choice.

10.2 ShellOptions

ShellOptions is a class to manage the input arguments and shell variables. With this class, it is easy to handle input line arguments and shell variables. For instance, the user can write

```
$ ./ex2.exe -nx 10 -tol 1e-6 -solver=cg
```

and then easily retrieve the value of nx, `tol`, and `solver`.

A simple code using this class is as follows:

```
int main(int argc, char *argv[])
{

  ShellOptions Args(argc,argv);
  int nx = Args.GetIntOption("-nx", 123);
  int ny = Args.GetIntOption("-ny", 145);
  double tol = Args.GetDoubleOption("-tol", 1e-12);
  string solver = Args.GetIntOption("-solver");

  cout << "nx = " << nx << endl;
  cout << "ny = " << ny << " (default value)" << endl;
  cout << "tol = " << tol << endl;
  cout << "solver = " << solver << endl;

  return 0;
}
```

Each line option can have a value or not. For options with a value, the user can specify this values as follows. Let `-tolerance` be the name of the option and `1e-12` its value. Both choices are valid:

- `-tolerance 1e-12` (with one or more spaces)
- `-tolerance=1e-12` (with = sign and no spaces)

Option names must begin with one or more dashes ('-'). Each option cannot have more than one value.

To use this class, the user has to build the database using the `argc,argv` input arguments. Then, to retrieve the option value, the user has to use one of the following functions: `GetIntOption`, `GetDoubleOption`, and `GetStringOption`.

If option name is not found in the database, a value of 0, 0.0 or an empty string is returned. If needed, the user can also specify a default value to return when the option name

is not found in the database. Method `HaveOption` can be used to query the database for an option.

File `${TRILINOS_HOME}/doc/tutorial/triutils/ex2.cpp` gives an example of the usage of this class.

References

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

[2] Free Software Foundation. Automake Home Page. http://www.gnu.org/software/automake.

[3] Free Software Foundation. Libtool Home Page. http://www.gnu.org/software/libtool.

[4] M. A. Heroux. *IFPACK User Guide*, 1.0 edition, 2001.

[5] M. A. Heroux. *Epetra Reference Manual*, 2.0 edition, 2002. http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/EpetraReferenceManual.pdf.

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

[7] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[8] Michael A. Heroux. AztecOO Users Guide. Technical Report SAND2003-XXXX, Sandia National Laboratories, 2003.

[9] Michael A. Heroux, Robert J. Hoekstra, and Alan Williams. Epetra Users Guide. Technical Report SAND2003-XXX, Sandia National Laboratories, 2003.

[10] Michael A. Heroux and James M. Willenbring. Trilinos Users Guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[11] Michael A. Heroux, James M. Willenbring, and Robert Heaphy. Trilinos Developers Guide. Technical Report SAND2003-1898, Sandia National Laboratories, 2003.

[12] Michael A. Heroux, James M. Willenbring, and Robert Heaphy. Trilinos Developers Guide Part II: ASCI Software Quality Engineering Practices Version 1.0. Technical Report SAND2003-1899, Sandia National Laboratories, 2003.

[13] Tamara G. Kolda and Roger P. Pawlowski. Nox home page. http://software.sandia.gov/nox.

[14] A. G. Salinger, N. M. Bou-Rabee, R. P. Pawlowski, E. D. Wilkes, E. A. Burroughs, R. B. Lehoucq, and L. A. Romero. LOCA: A library of continuation algorithms - Theroy and implementation manual. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2001. SAND 2002-0396.

[15] A. G Salinger, K. D. Devine, G. L. Hennigan, H. K. Moffat, S. A Hutchinson, and J. N. Shadid. MPSalsa: A finite element computer program for reacting flow problems part 2 - user's guide. Technical Report SAND96–2331, Sandia National Laboratories, 1996.

[16] A. G. Salinger, R. B. Lehoucq, R. P. Pawlowski, and J. N. Shadid. Computational bifurcation and stability studies of the 8:1 thermal cavity problem. *Internat. J. Numer. Meth. Fluids*, 40(8):1059–1073, 2002.

[17] John N. Shadid, Harry K. Moffat, Scott A. Hutchinson, Gary L. Hennigan, Karen D. Devine, and Andrew G. Salinger. MPSalsa: A finite element computer program for reacting flow problems part 1 - theoretical development. Technical Report SAND95–2752, Sandia National Laboratories, 1995.

[18] C. Tong and R. Tuminaro. ML2.0 Smoothed Aggregation User's Guide. Technical Report SAND2001-8028, Sandia National Laboratories, Albq, NM, 2000.

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

[20] Ray S. Tuminaro, Michael A. Heroux, Scott. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide, Version 2.1*. Sandia National Laboratories, Albuquerque, NM 87185, 1999.

[21] Ray S. Tuminaro and Jonathan Hu. Ml home page. http://www.cs.sandia.gov/ tuminaro/ML_Description.html.

Distribution list:

Internal Distribution:

1	MS 0316	G. L Hennigan, 9233
1	MS 0316	R. Hooper, 9233
1	MS 0316	R. J. Hoekstra, 9233
1	MS 0316	R. P. Pawlowski, 9233
1	MS 0316	S. A. Hutchinson, 9233
1	MS 0316	S. J. Plimpton, 9212
1	MS 0316	W. F. Spotz, 9233
1	MS 0819	A. C. Robinson, 9231
1	MS 0819	M. A. Christon, 9231
1	MS 0826	A. B. Williams, 8961
1	MS 0826	J. R. Stewart, 9143
1	MS 0827	H. C. Edwards, 9143
1	MS 0827	P. A. Sackinger, 9113
1	MS 0828	C. C. Ober, 9233
1	MS 0834	H. K. Moffat, 9114
1	MS 0834	M. M. Hopkins, 9114
1	MS 0834	R. P. Schunk, 9114
1	MS 0834	R. R. Rao, 9114
1	MS 0835	A. A. Lorber, 9141
1	MS 0835	K. H. Pierson, 9142
1	MS 0835	S. R. Subia, 9141
1	MS 0835	S. W. Bova, 9141
1	MS 0847	B. G. van Bloemen Waanders, 9211
1	MS 0847	C. R. Dohrmann, 9124
1	MS 0847	G. M. Reese, 9142
1	MS 0847	M. S. Eldred, 9211
1	MS 1110	D. E. Womble, 9214
1	MS 1110	D. M. Day, 9214
1	MS 1110	H. K. Thornquist, 9214
1	MS 1110	J. M. Willenbring, 9214
10	MS 1110	M. A. Heroux , 9214
10	MS 1110	M. Sala , 9214
1	MS 1110	R. A. Bartlett, 9214
1	MS 1110	R. B. Lehoucq, 9214
1	MS 1110	R. Heaphy, 9215
1	MS 1111	A. G. Salinger, 9233
1	MS 1111	C. A. Phillips, 9233
1	MS 1111	E. R. Keiter, 9233
1	MS 1111	E. T. Phipps, 9233
1	MS 1111	J. N. Shadid, 9233

1 MS 1111 K. D. Devine, 9215
1 MS 1152 J. D. Kotulski, 1642
1 MS 1166 C. R. Drumm, 15345
1 MS 9217 J. J. Hu, 9214
1 MS 9217 K. R. Long, 8962
1 MS 9217 P. T. Boggs, 8962
1 MS 9217 R. S. Tuminaro, 9214
1 MS 9217 T. Kolda, 8962
1 MS 9217 V. E. Howle, 8962
1 MS 9217 P. D. Hough, 8962
1 MS 9915 A. J. Rothfuss, 8961
1 MS 9915 N. M. Nachtigal, 8961

1 MS 9018 Central Technical Files, 8945-1
2 MS 0899 Technical Library, 9616

External distribution:

Ken Stanley
322 W. College St.
Oberlin OH 44074

Matthias Heinkenschloss
Department of Computational and Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

Dan Sorenson
Department of Computational and Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

Yousef Saad
Department of Computer Science and Engineering
University of Minnesota,
4-192 EE/CSci Building, 200 Union Street S.E.
Minneapolis, MN 55455

Kris Kampshoff
Department of Computer Science and Engineering
University of Minnesota,
EE/CSci Building, 200 Union Street S.E.
Minneapolis, MN 55455

Eric de Sturler
2312 Digital Computer Laboratory, MC-258
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, IL 61801-2987

Jason Cross
Box 429
St. John's University
Collegeville, MN 56321

Paul Sexton
Box 1560
St. John's University
Collegeville, MN 56321

Mike Phenow
PO Box 1392
St. John's University
Collegeville, MN 56321

Tim Davis, Assoc. Prof.
Room E338 CSE Building
P.O. Box 116120
University of Florida-6120
Gainesville, FL 32611-6120

Padma Raghavan
Department of Computer Science and Engineering
308 Pond Laboratory
The Pennsylvania State University
University Park, PA 16802-6106

Xiaoye Li
Lawrence Berkeley Lab
50F-1650
1 Cyclotron Rd
Berkeley, CA 94720

Richard Barrett
Los Alamos National Laboratory
Mail Stop B272
Los Alamos, NM 87545

Victor Eijkhout
Department of Computer Science,
203 Claxton Complex, 1122 Volunteer Boulevard,
University of Tennessee at Knoxville,
Knoxville TN 37996, USA

Jack Dongarra
Computer Science Department
1122 Volunteer Blvd
Knoxville, TN 37996-3450

David Keyes
Appl Phys & Appl Math
Columbia University
200 S. W. Mudd Building
500 W. 120th Street
New York, NY, 10027

Lois Curfman McInnes
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Paul Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Jeffrey J. Derby
CEMS Department, U. of MN
421 Washington Ave SE
Minneaplolis, MN 55455-0132

Carol Woodward
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Box 808, L-561
Livermore, CA 94551

Craig Douglas
325 McVey Hall - CCS
Lexington, KY 40506-0045

Juan Meza
Department Head, High Performance Computing Research
Lawrence Berkeley National Laboratory
Mail Stop 50B-2239
Berkeley, CA 9472

C.T. Kelley
Department of Mathematics, Box 8205
Center for Research in Scientific Computation
North Carolina State University
Raleigh, NC 27695-8205

Chuck Romine
Program Manager, Applied Mathematics
U.S. Department of Energy
1000 Independence Ave., SW
Washington, DC 20585-1290

Prof. Luca Formaggia
Mathematics Department
"F. Brioschi" Politecnico di Milano
Piazza L. da Vinci 32, 20133 MILANO, Italy

Prof. Alfio Quarteroni
IACS-CMCS
EPFL
CH-1015 Lausanne (VD) Switzerland