SAND REPORT

SAND2004-3796
Unlimited Release
Updated August 2005

AztecOO™ User Guide[a]

Michael A. 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 under Contract DE-AC04-94AL85000.

[a]For **AztecOO**™ Version 3.4 in **Trilinos**™ Release 6.0

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.doe.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 ordering:	http://www.ntis.gov/ordering.htm

SAND2004-3796
Unlimited Release
Updated August 2005

AztecOO™ User Guide†

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

Abstract

The **Trilinos**™ Project is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. **AztecOO**™ is a package within Trilinos that enables the use of the Aztec solver library [19] with **Epetra**™ [9] objects. AztecOO provides access to Aztec preconditioners and solvers by implementing the Aztec "matrix-free" interface using Epetra. While Aztec is written in C and procedure-oriented, AztecOO is written in C++ and is object-oriented.

In addition to providing access to Aztec capabilities, AztecOO also provides some signficant new functionality. In particular it provides an extensible status testing capability that allows expression of sophisticated stopping criteria as is needed in production use of iterative solvers. AztecOO also provides mechanisms for using Ifpack [10], ML [18] and AztecOO itself as preconditioners.

Acknowledgement

The authors would like to acknowledge the support of the ASCI and LDRD programs that funded development of AztecOO and the authors of Aztec 2.1 upon which AztecOO is built: Ray Tuminaro, Mike Heroux, Scott Hutchinson and John Shadid.

†For **AztecOO**™ Version 3.4 in **Trilinos**™ Release 6.0

Contents

Figures

1 Introduction

AztecOO is a collection of C++ classes that support the construction and use of objects for solving linear systems of equations of the form

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

via preconditioned Krylov methods, as provided in Aztec.

This user guide is intended to introduce a new user to the basic features of AztecOO. This document is not intended as a reference manual. Detailed descriptions of AztecOO classes and methods can be found online at the Trilinos Project home page [15].

1.1 Overview of Major AztecOO Classes and Features

AztecOO contains a variety of classes to support the solution of linear systems of equations of the form $Ax = b$ using preconditioned iterative methods. AztecOO also fully contains Aztec, so any application that is using Aztec can use the AztecOO library in place of Aztec.

The primary AztecOO class is of the same name, AztecOO. An AztecOO class instance acts as a manager of Aztec, accepting user data as Epetra objects. If an AztecOO object is instantiated using Epetra objects, all of Aztec's preconditioners and Krylov methods can be applied to the Epetra-defined problem. However, AztecOO provides a variety of mechanisms to override default Aztec capabilities. Users can construct and use preconditioners from Ifpack or ML, or can use another instance of the AztecOO class as a preconditioner. Users can also override the default convergence tests in Aztec and use any combination of available status tests in AztecOO_StatusTest classes, or define their own.

1.2 A Special Note for Aztec Users

AztecOO completely contains the full functionality and interface of Aztec 2.1 [19]. Therefore, any code that calls Aztec 2.1 functions can use AztecOO include files and library in place of Aztec 2.1. Any file that includes Aztec header files must be recompiled and the executable file must be re-linked with the AztecOO library, and the Aztec library must not be linked. In addition, the Epetra [9], LAPACK [1] and BLAS [12, 5, 4] libraries must also be available to the linker. If `--enable-aztecoo-azlu`

is provided to the `configure` command, the Y12M [21] library must also be provided. Finally, it is likely that you will need the linker to be aware of C++ libraries. Often the C++ compiler itself is the best program to use for linking, since it is aware of what system libraries are needed for linking.

Trilinos (and AztecOO as part of Trilinos) use a much different build process than Aztec 2.1. Trilinos uses Autoconf [6] and Automake [7] to support a configure/make process. If you are familar with configure/make procedures, you should find Trilinos fairly easy install. A Trilinos Installation Guide is available from the Trilinos website [15]. If you are familiar with the ad hoc build process of Aztec 2.1 and are unfamiliar with configure/make, you may find building Trilinos and AztecOO challenging. We encourage you to carefully read the Trilinos Installation Guide, since it will help you with understanding configure/make procedures in general, and the use of the processes in Trilinos specifically.

Unlike Aztec, AztecOO does not provide its own copies of BLAS, LAPACK or Y12M libraries. This change is part of a general policy in Trilinos to provide interfaces to third-party libraries but not code. This change is generally considered good practice from a software engineering perspective for a number of reasons, but can be a hindrance to Aztec users making a transition to AztecOO. Optimized BLAS libraries are available for most computer systems. We recommend you obtain one of these libraries from the Internet, if it is not already installed on your computer system. LAPACK is also available in an optimized form for some systems, but this is less critical. LAPACK can also be obtained from the Internet, or may already be installed on your system. Y12M is an old sparse direct solver also available from the Internet. It is used by Aztec's domain decomposition preconditioners as a local subdomain solver. Most Aztec users do not require it. By default, it is not required for building the AztecOO library. You may enable it by passing the argument `--enable-aztecoo-azlu` to the configure command.

1.3 Use of Epetra

AztecOO relies on Epetra for both concrete and abstract classes that describe matrix, vector and linear operator objects. Although concrete classes are needed to construct matrices, AztecOO itself uses these matrices via two Epetra abstract classes. By using abstract interfaces, we can support any of the predefined classes that implement the abstract interfaces and allow users to define new implementations. This allows AztecOO to be easily extended.

Epetra Abstract Classes

The two primary abstract Epetra classes used by AztecOO are:

1. **Epetra_RowMatrix:** Supports the use of any class that is conceptually a linear operator with access to coefficient data. Although row-orientation is assumed, this class works equally well with column oriented data, since transpose operations are supported. This class provides an interface to access matrix data. In addition, it extends the Epetra_Operator interface, so any class that implements Epetra_RowMatrix also implmenents Epetra_Operator, described next. There are four primary classes in Epetra that implement Epetra_RowMatrix, namely the Epetra_CrsMatrix, Epetra_VbrMatrix, Epetra_FECrsMatrix and Epetra_FEVbrMatrix classes. AztecOO also provides an implementation of Epetra_CrsMatrix called Epetra_MsrMatrix. Epetra_MsrMatrix allows users whose application is already forming Aztec DMSR matrix structs to encapsulate the DMSR matrix in a class that implements Epetra_RowMatrix. The encapsulation does not copy the data in the DMSR matrix struct. This feature is important for people making a transition from Aztec to AztecOO.

2. **Epetra_Operator:** Supports the use of any class that is conceptually a linear operator. There are only a handful of methods in this class, the most important of which are the Apply() and ApplyInverse() (ApplyInverse() can be defined as nonexistent). A large number of Epetra classes implement the Epetra_Operator interface, including the Epetra_CrsMatrix, Epetra_VbrMatrix, Epetra_FECrsMatrix and Epetra_FEVbrMatrix since these classes implement Epetra_RowMatrix and Epetra_RowMatrix extends Epetra_Operator. In addition, ML and IFPACK both implement the Epetra_Operator interface, so they can be used as preconditioners for AztecOO. A class called AztecOO_Operator also implements the Epetra_Operator interface using an existing AztecOO class instance. This allows AztecOO to be used as a preconditioner for itself.

Epetra Concrete Classes

Given the above abstract classes, we need some concrete classes in order to construct explicit objects. Specifically, the following concrete Epetra classes are used:

1. **Epetra_Vector:** Supports construction and use of distributed vectors of double-precision numbers. Once constructed, Epetra_Vector objects can be used in multiple ways. Common operations such as norms, dot products and vector updates are supported by methods in this class. Additional functionality is available via several extension techniques discussed later.

2. **Epetra_MultiVector:** An Epetra_MultiVector object is a collection of Epetra_Vector objects (although Epetra_Vector is actually implemented as a specialization of Epetra_MultiVector). Specifically, an Epetra_MultiVector object

is a collection of vectors with the same size and distribution. This kind of object is useful for block algorithms and applications that manage multiple vectors simultaneously. Epetra_Vector and Epetra_MultiVector objects are understood by all Trilinos packages.

3. **Epetra_CrsGraph:** Supports the construction and use of adjacency graphs. These graphs are used to describe the pattern of Epetra sparse matrix classes and provide pattern-based information to load-balancing interfaces. The graphs are also used to implement overlapping subdomain algorithms and a variety of other parallel algorithms.

4. **Epetra_CrsMatrix:** Supports construction and use of distributed sparse matrix objects. Once constructed, an Epetra_CrsMatrix object can be used with any Trilinos solvers or preconditioners. This class also supports common matrix and matrix-vector operations such as matrix scaling, matrix norms and matrix-vector multiplication.

5. **Epetra_FECrsMatrix:** This class inherits from Epetra_CrsMatrix, providing an interface to construct the matrix from element stiffness matrices. Once constructed, this matrix can be used as an Epetra_CrsMatrix.

6. **Epetra_VbrMatrix:** Although less frequently used than the Epetra_CrsMatrix class, this class supports the construction of sparse matrices whose entries are dense matrices. This type of matrix is frequently found in applications where multiple degrees of freedom are tracked at each mesh point. When properly used, the Epetra_VbrMatrix class can offer substantial machine performance and algorithmic robustness improvements.

7. **Epetra_FEVbrMatrix:** This class inherits from Epetra_VbrMatrix, providing an interface to construct the block matrix from element stiffness matrices. Once constructed, this matrix can be used as an Epetra_VbrMatrix.

8. **Epetra_MsrMatrix:** Although not part of Epetra, we list this class here because it is a viable option for existing Aztec users. An Epetra_MsrMatrix object is constructed by passing in an existing AZ_MATRIX struct defining a DMSR matrix as described in the Aztec 2.1 User's Guide [19]. Given the matrix data in this form, the Epetra_MsrMatrix class implements the Epetra_RowMatrix interface using the DMSR matrix with little extra storage required. This class is useful if an existing Aztec user has already constructed a DMSR matrix.

9. **Epetra_LinearProblem:** An Epetra_LinearProblem object is an aggregate object that encapsulates the problem $Ax = b$. It contains a pointer to an Epetra_RowMatrix or Epetra_Operator representing A, and an Epetra_MultiVector

for x and another for b. (NOTE: Although the interface supports multiple right-hand-sides and solution vectors, AztecOO presently supports the solution of only one right-hand-side.) Although use of the Epetra_LinearProblem is not required for passing A, x and b to AztecOO, it is highly recommended. Use of Epetra_LinearProblem guarantees that the matrix, left-hand-side and right-hand-side are compatible. Also, the Epetra_LinearProblem class provides methods for scaling the linear problem using any of a variety of diagonal scaling methods.

2 A First Example

Before proceeding with additional descriptive information, we introduce a simple example in this section in order to explicitly illustrate a straight-forward use of AztecOO. This specific example constructs a tridiagonal matrix and a random RHS vector. Then it constructs an AztecOO object, sets a few parameters for the solver and then solves the problem.

The example code is listed in Figure 1. In the remainder of this section we proceed with a description of the code.

2.1 Explanation of Figure 1

Line 1

Include the AztecOO_config.h file. This file contains macros definitions that were defined during the configuration process. In particular, the macro HAVE_MPI will be defined or undefined in this file, depending on whether AztecOO was built in MPI mode or not. We will use HAVE_MPI below to determine if our example code should be compiled with MPI support or not.

Lines 2–7

Include the appropriate implementation of the Epetra_Comm class. If AztecOO was built in MPI mode, the macro “HAVE_MPI” will be defined and this example will be built with MPI support. If not, then the example will be built in serial mode. Note that these lines of code, lines 15–20 and lines 60–62 are the only difference between a serial and distributed memory version of the example.

```
1  #include "AztecOO_config.h"
2  #ifdef HAVE_MPI
3  #include "mpi.h"
4  #include "Epetra_MpiComm.h"
5  #else
6  #include "Epetra_SerialComm.h"
7  #endif
8  #include "Epetra_Map.h"
9  #include "Epetra_Vector.h"
10 #include "Epetra_CrsMatrix.h"
11 #include "AztecOO.h"
12
13 int main(int argc, char *argv[]) {
14
15 #ifdef HAVE_MPI
16   MPI_Init(&argc,&argv);
17   Epetra_MpiComm Comm( MPI_COMM_WORLD );
18 #else
19   Epetra_SerialComm Comm;
20 #endif
21   cout << Comm <<endl;
22
23   int NumMyElements = 100;
24   // Construct a Map that puts same number of equations on each processor
25   Epetra_Map Map(-1, NumMyElements, 0, Comm);
26   int NumGlobalElements = Map.NumGlobalElements();
27
28   // Create a Epetra_Matrix
29   Epetra_CrsMatrix A(Copy, Map, 3);
30
31   // Add  rows one-at-a-time
32   double negOne = -1.0;
33   double posTwo = 2.0;
34   for (int i=0; i<NumMyElements; i++) {
35     int GlobalRow = A.GRID(i); int RowLess1 = GlobalRow - 1; int RowPlus1 = GlobalRow + 1;
36
37     if (RowLess1!=-1) A.InsertGlobalValues(GlobalRow, 1, &negOne, &RowLess1);
38     if (RowPlus1!=NumGlobalElements) A.InsertGlobalValues(GlobalRow, 1, &negOne, &RowPlus1);
39     A.InsertGlobalValues(GlobalRow, 1, &posTwo, &GlobalRow);
40   }
41
42   // Finish up
43   A.FillComplete();
44
45   // Create x and b vectors
46   Epetra_Vector x(Map);
47   Epetra_Vector b(Map);
48   b.Random(); // Fill b with random values
49
50   // Create Linear Problem
51   Epetra_LinearProblem problem(&A, &x, &b);
52   // Create AztecOO instance
53   AztecOO solver(problem);
54
55   solver.SetAztecOption(AZ_precond, AZ_Jacobi);
56   solver.Iterate(100, 1.0E-8);
57
58   cout << "Solver performed " << solver.NumIters() << " iterations." << endl
59        << "Norm of true residual = " << solver.TrueResidual() << endl;
60 #ifdef HAVE_MPI
61   MPI_Finalize() ;
62 #endif
63   return 0;
64 }
```

Figure 1. Simple AztecOO/Epetra Example

Lines 8–11

Include the other necessary Epetra and AztecOO header files. It is a good practice to explicitly include header files for all classes you explicitly use, and only those header files.

Line 13

Start of main program.

Lines 15–21

Depending on whether or not AztecOO was built in MPI mode[1], MPI will be initialized and an Epetra_MpiComm object will be constructed, or an Epetra_SerialComm object will be constructed. Please note that, in principle, the serial version of this example would work, even if AztecOO were built in MPI mode. Serial mode is always available. Line 21 prints the Epetra_Comm object to cout.

Line 23

Define the local problem dimension. NumMyElements will be used to define an Epetra_Map that has 100 elements on each processor. In turn the map will be used to construct vectors with 100 entries on each processor and matrices with 100 rows on each processor.

Lines 24–25

Constructs an Epetra_Map object that has NumMyElements elements spread across the parallel (or serial) machine. The first argument indicates that we are not specifying the global number of elements, but allowing the Epetra_Map constructor to compute it as the sum of NumMyElements defined on each calling processor. The second argument is the number of elements assigned to the calling processors. The third argument (a "0") indicates that our global indices are zero-based. Fortran users would typically pass in a "1" here. The fourth argument is the Comm object we just built.

[1] Whether or not Trilinos, and AztecOO as a Trilinos package, is built in MPI mode is determined by how the Trilinos (or AztecOO) `configure` script in invoked. If no MPI-related arguments are passed to the configure script, then packages are built in serial mode only. If one or more MPI options are invoked, then packages are built with MPI support (in addition to serial support).

Line 26

Once an Epetra_Map object is constructed, we can query it for how many total elements are in the map[2]. In this way, the remainder of our code can operate independent of how data is distributed.

Line 29

Instantiates (creates) an Epetra_CrsMatrix. The first argument tells the constructor whether or not data passed in to this object should be copied (user values and indices will be copied to internal storage) or viewed (user values and indices will be pointed to by this object and the user *must* guarantee the integrity of that data). View mode is available across many Epetra classes. In general, this is a very dangerous practice. However, in certain very important situations, it is essential to have this mode. This is especially true when using Epetra with Fortran, or when accepting matrix data from other parts of application where it is too expensive to replicate the data storage.

The second argument is the Epetra_Map object we just constructed. The third argument is an advisory value telling the constructor approximately how many nonzero values will be defined for each row of the matrix[3]. We are constructing a tridiagonal matrix, so the value “3” is appropriate.

At this point the matrix is an empty “bucket” ready to receive matrix values and indices. Also, at this point, most of the methods in the Epetra_CrsMatrix cannot be called successfully for this object.

Lines 31–43

These lines insert values and indices into the matrix we just instantiated. Our matrix is tridiagonal with a value of 2 at each diagonal and -1 on the immediate off-diagonals. We do not go into detail about the methods called here. The reader should look at the Epetra User Guide [11] or the online reference material at the Trilinos home page [15].

[2]For readers who are not familiar with a single-program, multiple data (SPMD) programming model, it may be useful to read a bit about it. Typing “SPMD tutorial” into a web search engine should be a sufficient starting point.

[3]getting this value wrong does not affect the correctness of results, but may affect performance and efficient use of memory

Lines 45–48

Once the matrix is constructed, we create our vectors b and x using the same map that determined the layout of the matrix rows. We also fill b with random values (line 48).

Lines 50–53

Now that A, x and b are formed, we can define a linear problem instance. This object will in turn be used to define an AztecOO instance. Note that it is possible to construct AztecOO objects in other ways, but we strongly recommend use of the constructor shown in line 53. Note that when the AztecOO object is constructed, the parameter and option values listed in Section 3 will be set to their default values. These defaults can be changed by calling the SetAztecOption(), SetAztecParam() and SetParameters() methods (see section 3).

Lines 55–56

Once the solver object is instantiated, we change the value of AZ_precond to AZ_Jacobi. Note that the key/value pairs passed in to this method can be any valid pair as defined in Section 3. Next we call the Iterate() method, passing in the maximum number of iterations that can be performed and a tolerance that should be used to test for convergence. Depending on the values of the Aztec parameters and options, this method will attempt to solve the problem using the prescribed preconditioner (if any) and the specified iterative method. It will also print intermediate results if the user has requested them. Upon exit from this method, the problem will hopefully be solved and the solution will be in x which in turn is part of the linear problem instance. Also upon exit, a number of methods can be called to determine the results of the iterations.

Lines 58–59

Print results from calling the solver.

Lines 60–62

If our code was compile in MPI mode, we need to call MPI_Finalize() for proper clean up.

Line 64

Program exit.

3 Aztec Options and Parameters

Because AztecOO is partly a wrapper around Aztec, much of the control and selection of solver options and parameters is done via three method calls that set Aztec options and parameters, namely SetAztecOption(), SetAztecParam() and SetParameters() (see 3.3). Most of the options and parameters are identical to those found in Aztec 2.1. However, there are a few new options and parameters. Below we list all options and parameters, including the default value and description of each.

3.1 Aztec Options

The following list of key/value pairs can be used with the SetAztecOption() method to change the behavior of the Iterate() method:

Specifications

*options[*AZ_solver*]*	Specifies solution algorithm. DEFAULT: AZ_gmres.
AZ_cg	Conjugate gradient (Applicable to symmetric positive definite matrices, sometimes usable with mildly non-symmetric matrices).
AZ_cg_condnum	Conjugate gradient with condition number estimation. (Similar to AZ_cg. Additionally computes extreme eigenvalue estimates using the generated Lanczos matrix).
AZ_gmres	Restarted generalized minimal residual.
AZ_gmres_condnum	Restarted GMRES with condition number estimation. (Similar to AZ_gmres. Additionally computes extreme eigenvalues using the generated Hessenberg matrix.)
AZ_cgs	Conjugate gradient squared.
AZ_tfqmr	Transpose-free quasi-minimal residual.
AZ_bicgstab	Bi-conjugate gradient with stabilization.

AZ_lu	Sparse direct solver (single processor only). **Note: This option is available only when –enable-aztecoo-azlu is specified on the AztecOO configure script invocation command**
*options[*AZ_precond*]*	Specifies preconditioner. DEFAULT: AZ_none.
AZ_none	No preconditioning.
AZ_Jacobi	k step Jacobi (block Jacobi for DVBR matrices where each block corresponds to a VBR block). The number of Jacobi steps, k, is set via *options*[AZ_poly_ord].
AZ_Neumann	Neumann series polynomial where the polynomial order is set via *options*[AZ_poly_ord].
AZ_ls	Least-squares polynomial where the polynomial order is set via *options*[AZ_poly_ord].
AZ_sym_GS	Non-overlapping domain decomposition (additive Schwarz) k step symmetric Gauss-Siedel. In particular, a symmetric Gauss-Siedel domain decomposition procedure is used where each processor independently performs one step of symmetric Gauss-Siedel on its local matrix, followed by communication to update boundary values before the next local symmetric Gauss-Siedel step. The number of steps, k, is set via *options*[AZ_poly_ord].
AZ_dom_decomp	Domain decomposition preconditioner (additive Schwarz). That is, each processor augments its submatrix according to *options*[AZ_overlap] and approximately "solves" the resulting subsystem using the solver specified by *options*[AZ_subdomain_solve]. Note: *options*[AZ_reorder] determines whether matrix equations are reordered (RCM) before "solving" submatrix problem.
*options[*AZ_subdomain_solve*]*	Specifies the solver to use on each subdomain when *options*[AZ_precond] is set to AZ_dom_decomp DEFAULT: AZ_ilut.

AZ_lu	Approximately solve processor's submatrix via a sparse LU factorization in conjunction with a drop tolerance *params*[AZ_drop]. The current sparse lu factorization is provided by the package Y12M [21]. **Note: This option is available only when –enable-aztecoo-azlu is specified on the AztecOO configure script invocation command**
AZ_ilut	Similar to AZ_lu using Saad's ILUT instead of LU [16]. The drop tolerance is given by *params*[AZ_drop] while the fill-in is given by *params*[AZ_ilut_fill].
AZ_ilu	Similar to AZ_lu using ilu(k) instead of LU with k determined by *options*[AZ_graph_fill]
AZ_rilu	Similar to AZ_ilu using rilu(k,ω) instead of ilu(k) with ω ($0 \geq \omega \geq 1$) given by *params*[AZ_omega] [8].
AZ_bilu	Similar to AZ_ilu using block ilu(k) instead of ilu(k) where each block corresponds to a VBR block.
AZ_icc	Similar to AZ_ilu using icc(k) instead of ilu(k) [14].

*options[*AZ_conv*]* Determines the residual expression used in convergence checks and printing. DEFAULT: AZ_r0. Note that this feature is overridden if the user registers an AztecOO_StatusTest object with the AztecOO solver instance. The iterative solver terminates if the corresponding residual expression is less than *params*[AZ_tol]:

AZ_r0	$\|r\|_2/\|r^{(0)}\|_2$
AZ_rhs	$\|r\|_2/\|b\|_2$
AZ_Anorm	$\|r\|_2/\|A\|_\infty$
AZ_noscaled	$\|r\|_2$
AZ_sol	$\|r\|_\infty/(\|A\|_\infty * \|x\|_1 + \|b\|_\infty)$

AZ_weighted	$\|r\|_{WRMS}$ where $\|\cdot\|_{WRMS} = \sqrt{(1/n)\sum_{i=1}^{n}(r_i/w_i)^2}$, n is the total number of unknowns, w is a weight vector provided by the user via *params*[AZ_weights] and $r^{(0)}$ is the initial residual. **Note: AZ_weighted is not available in AztecOO.**
*options[*AZ_diagnostics*]*	Similar to AZ_output. Specifies diagnostic information that can be used to tune solver performance. In particular, condition estimates for preconditioners, performance ratings and preconditioner memory use. DEFAULT: AZ_all.
AZ_all	Print all possible diagnostics.
AZ_none	Print no diagnostics. Note that if AZ_output is set to AZ_none, diagnostics will not be printed either.
*options[*AZ_output*]*	Specifies information (residual expressions - see *options*[AZ_conv]) to be printed. DEFAULT: 1.
AZ_all	Print out the matrix and indexing vectors for each processor. Print out all intermediate residual expressions.
AZ_none	No intermediate results are printed.
AZ_warnings	Only Aztec warnings are printed.
AZ_last	Print out only the final residual expression.
AZ_summary	Print header containing problem description, iterative method and preconditioner description.
>0	Print residual expression every *options[*AZ_output*]* iterations.
*options[*AZ_pre_calc*]*	Indicates whether to use factorization information from previous calls to Iterate(). DEFAULT: AZ_calc if the user is using the native Aztec preconditioners selected via options[AZ_precond]. If the user has registered an Epetra_Operator object using the SetPrecOperator() method, then this operator will be used as the preconditioner and it is assumed that the preconditioner is already constructed.

AZ_calc — Use no information from previous Iterate() calls.

AZ_recalc — Use preprocessing information from a previous call but recalculate preconditioning factors. This is primarily intended for factorization software which performs a symbolic stage.

AZ_reuse — Use preconditioner from a previous Iterate() call, do not recalculate preconditioning factors. Also, use scaling factors from previous call to scale the right hand side, initial guess and the final solution.

*options[*AZ_graph_fill*]* — The level of graph fill-in (k) for incomplete factorizations: ilu(k), icc(k), bilu(k). DEFAULT: 0

*options[*AZ_max_iter*]* — Maximum number of iterations. DEFAULT: 500, unless an statustest object has been registered using the SetStatusTest() method in which case this option is ignored.

*options[*AZ_poly_ord*]* — The polynomial order when using polynomial preconditioning. Also, the number of steps when using Jacobi or symmetric Gauss-Seidel preconditioning. DEFAULT: 3 for polynomial preconditioners, 1 for Jacobi and Gauss-Seidel preconditioners.

*options[*AZ_overlap*]* — Determines the submatrices factored with the domain decomposition algorithms (see *options*[AZ_precond]). DEFAULT: 0.

AZ_diag — Factor the local submatrix defined on this processor augmented by a diagonal (block diagonal for VBR format) matrix. This diagonal matrix corresponds to the diagonal entries of the matrix rows (found on other processors) associated with external elements. This can be viewed as taking one Jacobi step to update the external elements and then performing domain decomposition with AZ_none on the residual equations.

k — Augment each processor's local submatrix with rows from other processors. The new rows are obtained in k steps (k $\geq$ 0). Specifically at each augmentation step, rows corresponding to external unknowns are obtained. These external unknowns are defined by nonzero columns in the current augmented matrix not containing a corresponding row on this processor. After the k steps, all columns associated with external unknowns are discarded to obtain a square matrix. The resulting procedure is an overlapped additive Schwarz procedure.

*options[*AZ_type_overlap*]* — Determines how overlapping subdomain results are combined when different processors have computed different values for the same unknown. DEFAULT: AZ_standard.

AZ_standard — The resulting value of an unknown is determined by the processor owning that unknown. Information from other processors about that unknown is discarded.

AZ_symmetric — Add together the results obtained from different processors corresponding to the same unknown. This keeps the preconditioner symmetric if a symmetric technique was used on each subdomain.

*options[*AZ_kspace*]* — Krylov subspace size for restarted GMRES. DEFAULT: 30.

*options[*AZ_reorder*]* — Determines whether RCM reordering will be done in conjunction with domain decomposition incomplete factorizations. 1 indicates RCM reordering is used. 0 indicates that equations are not reordered. DEFAULT: 1.

*options[*AZ_keep_info*]* — Determines whether matrix factorization information will be kept after this solve (for example to solve the same system with another right hand side, see *options*[AZ_pre_calc]). 1 indicates factorization information is kept. 0 indicates that factorization information is discarded. DEFAULT: 0.

*options[*AZ_orthog*]*	GMRES orthogonalization scheme. DEFAULT: AZ_classic.
AZ_classic	2 steps of classical Gram-Schmidt orthogonalization.
AZ_modified	A single step of Modified Gram-Schmidt orthogonalization.
*options[*AZ_aux_vec*]*	Determines $\tilde{r}$ (a required vector within some iterative methods). The convergence behavior varies slightly depending on how this is set. DEFAULT: AZ_resid.
AZ_resid	$\tilde{r}$ is set to the initial residual vector.
AZ_rand	$\tilde{r}$ is set to random numbers between -1 and 1. NOTE: When using this option, the convergence depends on the number of processors (i.e. the iterates obtained with x processors differ from the iterates obtained with y processors if x $\neq$ y).

3.2 **Aztec** parameters

The double precision array *params* is set up by the AztecOO solver instance and is of length AZ_PARAMS_SIZE. Because of this, we do not support *options*[AZ_conv] = AZ_weighted). This type of functionality is still possible by defining an implementation of the AztecOO_StatusTest abstract class.

Below we list the key/value pairs that can be used with the SetAztecParam() method:

Specifications

*params[*AZ_tol*]*	Specifies tolerance value used in conjunction with convergence tests. DEFAULT: 10^{-6}.
*params[*AZ_drop*]*	Specifies drop tolerance used in conjunction with LU or ILUT preconditioners (see description below for ILUT). DEFAULT: 0.0.

*params[*AZ_ilut_fill*]* ILUT uses two criteria for determining the number of nonzeros in the resulting approximate factorizations. For examples, setting *params*[AZ_ilut_fill] $= 1.3$, requires that the ILUT factors contain no more than approximately 1.3 times the number of nonzeros of the original matrix. Additionally, ILUT drops all elements in the resulting factors that are less than *params*[AZ_drop]. Thus, when *params*[AZ_drop] is set to zero, nothing is dropped and the size of the matrix factors is governed only by *params*[AZ_ilut_fill]. However, positive values of *params*[AZ_drop] may result in matrix factors containing significantly fewer nonzeros. [16]
DEFAULT: 1.

*params[*AZ_omega*]* Damping or relaxation parameter used for RILU. When *params*[AZ_omega] is set to zero, RILU corresponds to ILU(k). When it is set to one, RILU corresponds to MILU(k) where k is given by *options*[AZ_graph_fill]. [8]
DEFAULT: 1.

*params[*AZ_weights*]* When *options*[AZ_conv] = AZ_weighted, the *i*'th local component of the weight vector is stored in the location *params*[AZ_weights+i].

*params[*AZ_rthresh*]* Parameter used to modify the relative magnitude of the diagonal entries of the matrix that is used to compute any of the incomplete factorization preconditioners. When *params*[AZ_rthresh] is set to zero, no relative perturbation is performed. See Section 5 and the discussion of *params*[AZ_athresh] for more information.
DEFAULT: 0.0

*params[*AZ_athresh*]* Parameter used to modify the absolute magnitude of the diagonal entries of the matrix that is used to compute any of the incomplete factorization preconditioners. When *params*[AZ_athresh] is set to zero, no absolute perturbation is performed. See Section 5 and the discussion of *params*[AZ_rthresh] for more information.
DEFAULT: 0.0

3.3 ParameterList Interface

The AztecOO method SetParameters() allows options and parameters to be set using values held in a Teuchos::ParameterList container. The SetParameters() method is available only if the option `--enable-aztecoo-teuchos` is provided to the `configure` command.

This method extracts any mixture of options and parameters from a Teuchos::-ParameterList object and uses them to set values in AztecOO's internal options and params arrays. SetParameters() may be called repeatedly, and does not reset default values or previously-set values unless those values are contained in the current Teuchos::ParameterList argument. (Of course, if the method SetAztecDefaults() is called after SetParameters() has been called, any parameters that were set by SetParameters() will be lost.)

A Teuchos::ParameterList is a collection of named Teuchos::ParameterEntry objects. (Please refer to documentation for the Teuchos package for information on using the Teuchos::ParameterList.) AztecOO recognizes names which mirror the macros defined in az_aztec_defs.h and described above in subsections 3.1 and 3.2. In addition, it recognizes case insensitive versions of those names, with or without the prepended 'AZ_'. As an example, the following are equivalent and valid: "AZ_solver", "SOLVER", "Solver". To set an entry in the Aztec options array, the type of the ParameterEntry value may be either a string or an int in some cases. For example, if selecting the solver, the following are equivalent and valid: AZ_gmres (which is an int), "AZ_gmres" (which is a string) or "GMRES" (case-insensitive, 'AZ_' is optional).

To set an entry in the Aztec params array, the type of the ParameterEntry value must be double.

By default, the SetParameters() method will silently ignore parameters which have unrecognized names or invalid types. Users may provide an optional argument specifying that warnings be printed for unused parameters. Alternatively, users may iterate the ParameterList afterwards and check the isUsed attribute on the ParameterEntry objects.

Here is a brief example of setting parameters using the ParameterList interface:

```
Teuchos::ParameterList parameterlist;
parameterlist.set("solver", AZ_gmres); //AZ_gmres stored as int
parameterlist.set("precond", "Jacobi"); //"Jacobi" -> AZ_Jacobi
parameterlist.set("AZ_tol", 1.e-10);
```

```
AztecOO azoo;
azoo.SetParameters(parameterlist);
```

3.4 Return status

The double precision array *status* of length AZ_STATUS_SIZE returned from Iterate()[4]. The contents of *status* are described below.

Specifications

*status[*AZ_its*]*	Number of iterations taken by the iterative method.
*status[*AZ_why*]*	Reason why Iterate() terminated.
AZ_normal	User requested convergence criteria is satisfied.
AZ_param	User requested option is not available.
AZ_breakdown	Numerical breakdown occurred.
AZ_loss	Numerical loss of precision occurred.
AZ_ill_cond	The Hessenberg matrix within GMRES is ill-conditioned. This could be caused by a number of reasons. For example, the preconditioning matrix could be nearly singular due to an unstable factorization (note: pivoting is not implemented in any of the incomplete factorizations). Ill-conditioned Hessenberg matrices could also arise from a singular application matrix. In this case, GMRES tries to compute a least-squares solution.
AZ_maxits	Maximum iterations taken without convergence.
*status[*AZ_r*]*	The true residual norm corresponding to the choice *options*[AZ_conv] (this norm is calculated using the computed solution).
*status[*AZ_scaled_r*]*	The true residual ratio expression as defined by *options*[AZ_conv].

[4] All integer information returned from Iterate() is cast into double precision and stored in *status*.

*status[*AZ_rec_r*]*	Norm corresponding to *options*[AZ_conv] of final residual or estimated final residual (recursively computed by iterative method). Note: When using the 2-norm, **tfqmr** computes an estimate of the residual norm instead of computing the residual.
*status[*AZ_solve_time*]*	Utilization time in Aztec to solve system.
*status[*AZ_Aztec_version*]*	Version number of Aztec.

When AztecOO returns abnormally, the user may elect to restart using the current computed solution as an initial guess.

4 Choosing a Krylov Method

Choosing a Krylov method is determined by many factors, some of which are very problem-specific. However, there are some general guidelines worth mentioning. Selection of a Krylov solver is done by setting the value of options[AZ_solver] as described in Section 3.

4.1 Symmetric (Positive Definite) Problems

AztecOO provides two choices for solving symmetric positive definite (SPD) problems, both of which are conjugate gradient (CG) methods. A third option exists for some rare situations.

1. (AZ_cg) The first is CG (AZ_cg), which is the standard preconditioned CG method. Note that, strictly speaking, CG assume the matrix is symmetric and positive semi-definite. It does sometimes work for mildly non-symmetric problems. Preconditioners for CG must also be symmetric. Thus, it is usually ineffective to use incomplete LU type of preconditioners with CG.

2. (AZ_cg_condnum) The second solver for symmetric problems is also a preconditioned CG solver, but in addition to computing the solution, this solver will construct the tridiagonal Lanczos matrix that provides good estimates of the extreme eigenvalues of the preconditioned[5] matrix. This solver is more expensive than AZ_cg, so it should only be used when this information is needed.

[5] If you want to use AZ_cg_condnum to estimate the extreme eigenvalues of your original matrix, you should not use a preconditioner or scaling method when calling this solver. In general, changing the preconditioner will also change the extreme eigenvalue estimates

3. (AZ_gmres and AZ_gmres_condnum) Although there are specialized Krylov methods for symmetric indefinite problems, AztecOO does not implement them. However, if such a solver is needed, AZ_gmres can be very effective. Note that, if a problem has many eigenvalues on either side of the imaginary access in the complex plane, then the size of the restart value (options[AZ_kspace] should be large. For very difficult problems, it is best to set this value equal to the number of iterations performed.

4.2 Non-symmetric Problems

For SPD problems, CG is both optimal algorithmically and has fixed cost per iteration. For non-symmetric problems there is no such solver. It is well-known that GMRES without restart is an optimal solver. In practice it is very robust, especially with a good preconditioner. However, its cost can be prohibitive, both in memory and computation time. As a result, literally dozens of non-optimal, but lower cost iterative methods have been developed for non-symmetric problems. AztecOO has two implementations of GMRES and several lower cost sub-optimal methods.

Of these options, only two are commonly used, and a third occasionally:

1. (AZ_gmres) Because AztecOO and its predecessor Aztec have often been applied to very difficult non-symmetric systems, and because GMRES is such an effective parallel Krylov method, GMRES is by far the most common non-symmetric solver used in AztecOO. The effectiveness of GMRES can be strongly affected by the choice of restart value and orthogonalization method. We discuss these issues below.

2. (AZ_gmres_condnum) This version of GMRES retains an unfactored copy of the generated Hessenberg matrix and computes its eigenvalues using LAPACK. The extreme eigenvalues of this Hessenberg matrix are often a good approximation to the extreme eigenvalues of the preconditioned matrix.

3. (AZ_bicgstab) Among the many sub-optimal non-symmetric Krylov methods, BiCGSTAB has proven to be the most robust for a reasonable cost. For problems that are well-conditioned, BiCGSTAB can often be a much cheaper alternative to GMRES. However, it is still ineffective for difficult problems. Furthermore, unlike GMRES where the restart value can be increased, there are no parameters that can be used to make it more robust.

4.3 More about GMRES

As mentioned above, GMRES is by far the most robust general-purpose Krylov method available. Part of its robustness comes from the ability to tune two parameters, namely options[AZ_kspace] and options[AZ_orthog]. options[AZ_kspace] determines the number of Krylov vectors that will be used as part of the least-squares problem to generate the next approximate solution. Generally setting this value larger improves the robustness, decreases iteration count, but increases costs. This value is set to 30 by default, but for challenging problems one should set it (much) higher, especially if memory is available on the computer. For very difficult problems, we recommend setting options[AZ_kspace] equal to the maximum number of iterations.

The parameter options[AZ_orthog] can be used to select the type of Gram-Schmidt algorithm to used. We provide two options:

1. Two steps of classical Gram-Schmidt (Double CGS).
2. One step of modified Gram-Schmidt. (Single MGS).

For many years, (single) MGS was the preferred option for GMRES because of its superior numerical accuracy over single CGS. However, in the past several years it has been recognized that double CGS is more effective than single MGS, as effective as double MGS and has superior parallel performance. As a result, AztecOO uses double CGS by default. However, there may be instances where single MGS would be sufficient for robustness and it can have a lower cost in some situations.

5 Diagonal Perturbations and Incomplete Factorizations

One of the new features in AztecOO that was not part of Aztec 2.1 is the ability compute incomplete factorizations of perturbed systems. One attribute of coupled multi-physics problems is that incomplete factorizations can be difficult to compute, even if the original matrix A is well-conditioned. A few sources of difficulty are:

1. Zero diagonal entries. In this case, unless fill-in occurs prior to dividing by the zero diagonal, or we perform some type of pivoting, the factorization will fail or produce unusable factors. In some instances even when fill-in does occur, the diagonal value may be too small to produce a usable factorization.

2. Singular principle sub-matrices. In this case, boundary conditions are missing or insufficient to determine a nonsingular upper left sub-matrix.

3. Singularity due to domain partitioning. When executing in parallel using additive Schwarz methods, we observe situations where an incomplete factorization for the entire domain exists but one or more factorizations for the subdomains do not.

One straightforward technique to address poorly conditioned factors is to introduce diagonal perturbations. In this situation, the incomplete factorization is performed on a matrix that is identical to A except that diagonal entries are perturbed, usually to increase diagonal dominance. This idea was introduced by Manteuffel [13] as a means for computing incomplete Cholesky decompositions for symmetric positive definite systems and extended to nonsymmetric matrices by van der Vorst [20], Saad [17] and Chow [2]. It is used for block entry matrices in a package called BPKIT [3].

The incomplete factorization preconditioners in AztecOO (and in IFPACK [10]) are sensitive to two parameters:

- *params*[AZ_athresh]: Absolute threshold α.
- *params*[AZ_rthresh]: Relative threshold ρ.

Given these two values, the preconditioner is computed for a matrix B such that off-diagonal entries of B are the same as the original matrix A but the diagonals are replaced with:

$$b_{ii} \leftarrow \mathrm{sign}(a_{ii})\alpha + (1+\rho)a_{ii} \tag{2}$$

where sign() is the sign of the value, either $+1$ or -1.

Since Krylov methods such as GMRES are invariant under scaling, and a very large diagonal perturbation essentially makes the off-diagonal elements irrelevant, one way to view diagonal perturbation is as establishing a continuum between an accurate but poorly conditioned incomplete factorization and less accurate but perfectly conditioned Jacobi diagonal scaling. Given this continuum, the strategy is then to choose a minimal perturbation that sufficiently stabilizes the factorization.

5.1 Perturbation Strategies

As mentioned above, we often have difficulty computing usable incomplete factorizations for our problems. The most common source of problems is that the factorization may encounter a small or zero pivot, in which case the factorization can fail, or even if the factorization succeeds, the factors may be so poorly conditioned that use of them in the iterative phase produces meaningless results. Before we can fix this problem, we must be able to detect it. To this end, we use a simple but effective condition number estimate for $(LU)^{-1}$.

Estimating Preconditioner Condition Numbers

The condition of a matrix B, called $cond_p(B)$, is defined as $cond_p(B) = \|B\|_p \|B^{-1}\|_p$ in some appropriate norm p. $cond_p(B)$ gives some indication of how many accurate floating point digits can be expected from operations involving the matrix and its inverse. A condition number approaching the accuracy of a given floating point number system, about 15 decimal digits in IEEE double precision, means that any results involving B or B^{-1} may be meaningless.

The ∞-norm of a vector y is defined as the maximum of the absolute values of the vector entries, and the ∞-norm of a matrix C is defined as $\|C\|_\infty = \max_{\|y\|_\infty=1} \|Cy\|_\infty$. A crude lower bound for the $cond_\infty(C)$ is $\|C^{-1}e\|_\infty$ where $e = (1, 1, \ldots, 1)^T$. It is a lower bound because $cond_\infty(C) = \|C\|_\infty \|C^{-1}\|_\infty \geq \|C^{-1}\|_\infty \geq |C^{-1}e\|_\infty$.

For our purposes, we want to estimate $cond_\infty(LU)$, where L and U are our incomplete factors. Chow [2] demonstrates that $\|(LU)^{-1}e\|_\infty$ provides an effective estimate for $cond_\infty(LU)$. Furthermore, since finding z such that $LUz = y$ is a basic kernel for applying the preconditioner, computing this estimate of $cond_\infty(LU)$ is performed by setting $y = e$, calling the solve kernel to compute z and then computing $\|z\|_\infty$.

AztecOO provides the ability to query the condition number of the preconditioner, if one is available. This value is obtained by calling the Condest() method on an AztecOO object instance.

A priori Diagonal Perturbations

Given the above method to estimate the conditioning of the incomplete factors, if we detect that our factorization is too ill-conditioned we can improve the conditioning by perturbing the matrix diagonal and restarting the factorization using this more diagonally dominant matrix. In order to apply perturbation, prior to starting the factorization, we compute a diagonal perturbation of our matrix A in Eq. 1 and

perform the factorization on this perturbed matrix. The overhead cost of perturbing the diagonal is minimal since the first step in computing the incomplete factors is to copy the matrix A into the memory space for the incomplete factors. We simply compute the perturbed diagonal at this point. The actual perturbation values we use are discussed below.

5.2 Strategies for Managing Condition Numbers

Without any prior knowledge of a problem, the first step to take when computing a preconditioner is to compute the original factors without any diagonal perturbation. This usually gives the most accurate factorization and, if the condition estimate of the factors is not too big, will lead to the best convergence. If the condition estimate of the original factors is larger than machine precision, say greater than 1.0e15, then it is possible that the factorization will destroy convergence of the iterative solver. This will be evident if the iterative solver starts to diverge, stagnates, or aborts because it detects ill-conditioning. In these cases, diagonal perturbations may be effective. If the condition estimate of the preconditioner is well below machine precision (less than 1.0e13) and one is not achieving convergence, then diagonal perturbation will probably not be useful. Instead, one should try to construct a more accurate factorization by increasing fill.

Strategies for *a priori* Diagonal Perturbations

The goal when applying *a priori* perturbations is to find a close to minimal perturbation that reduces the condition estimate below machine precision (roughly 1.0e16). In some practical settings, we use the strategy outlined in Figure 2. Essentially, we replace the diagonal values $(d_1, d_2, \ldots, d_n)$ with $d_i = sign(d_i)\alpha + d_i(1+\rho)$, $i = 1, 2, \ldots, n$, where n is the matrix dimension and $sign(d_i)$ returns the sign of the diagonal entry. This has the effect of forcing the diagonal values to have minimal magnitude of α and to increase each by an amount proportional to ρ, and still keep the sign of the original diagonal entry.

6 Optimal reuse of AztecOO for Repeated Solves

In many practical situations, a linear solver is being called repeatedly to solve problems that have similar numeric properties and similar or identical nonzero structure. In these cases, it is often beneficial to reuse in later solves some or all of the work

1. Set the absolute threshold $\alpha = 0.0$ and the relative threshold $\rho = 0.0$ (equivalent to no perturbation).
2. Define perturbed diagonal entries as $d_i = sign(d_i)\alpha + d_i(1+\rho)$ and compute the incomplete factors L and U.
3. Compute $condest = \|(LU)^{-1}e\|_\infty$ where $e = (1, 1, \ldots, 1)^T$.
4. If failure ($condest > 10^{15}$ or convergence is poor), set $\alpha = 10^{-5}$, $\rho = 0.0$. Repeat Steps 2 and 3.
5. If failure, set $\alpha = 10^{-5}$, $\rho = 0.01$. Repeat Steps 2 and 3.
6. If failure, set $\alpha = 10^{-2}$, $\rho = 0.0$. Repeat Steps 2 and 3.
7. If failure, set $\alpha = 10^{-2}$, $\rho = 0.01$. Repeat Steps 2 and 3.
8. If still failing, continue alternate increases in the two threshold values.

Figure 2. Simple *a priori* Threshold Strategy

performed in formulating the preconditioner for an earlier solve. The Aztec option options[AZ_pre_calc] allows the user to specify that the preconditioner from the previous solve should be retained for subsequent solves.

6.1 Reuse and the SetPrecOperator() Method

A user can overrides the Aztec preconditioner options by forming their own preconditioner that conforms to the Epetra_Operator interface. In particular, users may construct ML and Ifpack preconditioners for use with AztecOO. Once an Epetra_Operator-compliant preconditioner is constructed, it can be registered with an AztecOO object as the preconditioner. In this situation, the preconditioner will never be reset by AztecOO. Instead, the user has this responsibility.

7 Accessing AztecOO from Thyra and Python

In addition to accessing AztecOO via its C++ interfaces, bindings exist that allow use via the Trilinos abstract interfaces in Thyra. Thyra is a package of abstract C++ interfaces for accessing one or more concrete solver capabilities via a single solver interface.

AztecOO also is accessible via the PyTrilinos package using Python. Python provides an excellent interactive prototyping environment, as well as a flexible application development environment. AztecOO, Epetra, ML, IFPACK and several other Trilinos packages are available to Python users via the PyTrilinos package.

Detailed discussions of Thyra and PyTrilinos can be found by going to the Trilinos home page [15].

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Pub., third edition, 1999.

[2] E. Chow. *Robust Preconditioning for Sparse Linear Systems*. PhD thesis, University of Minnesota, Minneapolis, MN, 1997.

[3] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, June 1998.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[5] J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14, 1988.

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

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

[8] I. Gustafsson. A class of first-order factorization methods. *Bit*, 18:142–156, 1978.

[9] M. A. Heroux. Epetra home page. http://software.sandia.gov/Trilinos/packages/epetra.

[10] M. A. Heroux. Ifpack home page. http://software.sandia.gov/Trilinos/packages/ifpack.

[11] M. A. Heroux. *Epetra User Guide*, 2.0 edition, 2002.

[12] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5, 1979.

[13] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[14] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for systems of which the coefficient matrix is a symmetric m-matrix. *Math. Comp.*, 31(137):148–162, 1977.

[15] M. N. Phenow. Trilinos home page. http://software.sandia.gov/Trilinos.

[16] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[17] Y. Saad. Preconditioned krylov subspace methods for CFD applications. In W. G. Habashi, editor, *Proceedings of the International Workshop on Solution Techniques for Large-scale CFD problems*, pages 179–195, 1994.

[18] R. S. Tuminaro. Ml home page. http://software.sandia.gov/Trilinos/packages/ml.

[19] 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.

[20] H. van der Vorst. Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from PDE-problems. 44:1–19, 1981.

[21] Z. Zlatev, V.A. Barker, and P.G. Thomsen. SSLEST - a FORTRAN IV subroutine for solving sparse systems of linear equations (user's guide). Technical report, Institute for Numerical Analysis, Technical University of Denmark, Lyngby,Denmark, 1978.

AztecOO `configure` Options

Most often AztecOO's `configure` command will be invoked automatically as part of the Trilinos-level `configure` command. Regardless of how AztecOO's `configure` command is invoked, the following options can be used to customize the configure process. These options can also be listed by executing

Command: `./configure --help`

in the main AztecOO directory. In fact, obtaining the options this way is preferred, since option may have changed since the publiction of this document. However, for convenience, we list the configuration options presently available:

```
‘configure’ configures aztecoo 3.3d to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print ‘checking...’ messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for ‘--cache-file=config.cache’
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or ‘..’]

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
  [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
  [PREFIX]

By default, ‘make install’ will install all the files in
```

```
'/usr/local/bin', '/usr/local/lib' etc.  You can specify
an installation prefix other than '/usr/local' using '--prefix',
for instance '--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
  --sbindir=DIR          system admin executables [EPREFIX/sbin]
  --libexecdir=DIR       program executables [EPREFIX/libexec]
  --datadir=DIR          read-only architecture-independent data [PREFIX/share]
  --sysconfdir=DIR       read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR   modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR    modifiable single-machine data [PREFIX/var]
  --libdir=DIR           object code libraries [EPREFIX/lib]
  --includedir=DIR       C header files [PREFIX/include]
  --oldincludedir=DIR    C header files for non-gcc [/usr/include]
  --infodir=DIR          info documentation [PREFIX/info]
  --mandir=DIR           man documentation [PREFIX/man]

Program names:
  --program-prefix=PREFIX            prepend PREFIX to installed program names
  --program-suffix=SUFFIX            append SUFFIX to installed program names
  --program-transform-name=PROGRAM   run sed PROGRAM on installed program names

System types:
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on HOST [BUILD]
  --target=TARGET   configure for building compilers for TARGET [HOST]

Optional Features:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-maintainer-mode  enable make rules and dependencies not useful
  (and sometimes confusing) to the casual installer
  --enable-mpi            MPI support
  --disable-dependency-tracking  speeds up one-time build
  --enable-dependency-tracking   do not reject slow dependency extractors
  --enable-export-makefiles
                          Creates export makefiles in the install (prefix)
                          directory. This option requires perl to be set in
                          your path or defined with --with-perl=<perl
```

```
                          executable>. Note that the export makefiles are
                          always created and used in the build directory, but
                          will not be installable without this option to
                          change the paths. (default is yes)
  --enable-aztecoo-azlu   Enable az-lu preconditioner. Default is no. Requires
                          y12m.
  --enable-teuchos        Build Teuchos ParameterList support in AztecOO. Can
                          be overridden with --disable-aztecoo-teuchos.
  --enable-aztecoo-teuchos
                          Build Teuchos ParameterList support in AztecOO.
  --enable-tests          Build tests for all Trilinos packages (not all
                          packages are sensitive to this option) (default is
                          yes)

  --enable-aztecoo-tests  Build AztecOO tests (default is yes if
                          --disable-tests is not specified)
  --enable-examples       Build examples for all Trilinos packages (not all
                          packages are sensitive to this option) (default is
                          yes)
  --enable-aztecoo-examples
                          Build AztecOO examples (default is yes if
                          --disable-examples is not specified)
  --enable-libcheck       Check for some third-party libraries including BLAS
                          and LAPACK. (Cannot be disabled unless tests and
                          examples are also disabled.) (default is yes)
  --enable-python[=PYTHON]
                          absolute path name of Python executable

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-install=INSTALL_PROGRAM
                          Use the installation program INSTALL_PROGRAM rather
                          the default that is provided. For example
                          --with-install="/path/install -p"
  --with-mpi-compilers=PATH
                          use MPI compilers mpicc, mpif77, and mpicxx, mpic++
                          or mpiCC in the specified path or in the default
                          path if no path is specified. Enables MPI
  --with-mpi=MPIROOT      use MPI root directory (enables MPI)
  --with-mpi-libs="LIBS"  MPI libraries ["-lmpi"]
  --with-mpi-incdir=DIR   MPI include directory [MPIROOT/include] Do not
```

```
                          use -I
--with-mpi-libdir=DIR     MPI library directory [MPIROOT/lib] Do not use
                          -L
--with-ccflags            additional CCFLAGS flags to be added: will prepend
                          to CCFLAGS
--with-cxxflags           additional CXXFLAGS flags to be added: will
                          prepend to CXXFLAGS
--with-cflags             additional CFLAGS flags to be added: will prepend
                          to CFLAGS
--with-fflags             additional FFLAGS flags to be added: will prepend
                          to FFLAGS
--with-libs               List additional libraries here. For example,
                          --with-libs=-lsuperlu or
                          --with-libs=/path/libsuperlu.a
--with-ldflags            additional LDFLAGS flags to be added: will prepend
                          to LDFLAGS
--with-ar                 override archiver command (default is "ar cru")
--with-perl               supply a perl executable. For example
                          --with-perl=/usr/bin/perl.
--with-gnumake            Gnu's make has special functions we can use to
                          eliminate redundant paths in the build and link
                          lines. Enable this if you use gnu-make to build
                          Trilinos. This requires that perl is in your path or
                          that you have specified the perl executable with
                          --with-perl=<perl executable>. Configure will check
                          for the existence of the perl executable and quit
                          with an error if it is not found. (default is no)
--with-python[=PYTHON]
                          absolute path name of Python executable
--with-swig[=SWIG]
                          enable swig and set swig binary
--with-libdirs            OBSOLETE use --with-ldflags instead. (ex.
                          --with-ldflags="-L<DIR> -L<DIR2>")
--with-incdirs            additional directories containing include files:
                          will prepend to search here for includes, use -Idir
                          format
--with-lapacklib          name of library containing LAPACK: will search lib
                          directories for -lname
--with-blaslib            name of library containing BLAS: will search lib
                          directories for -lname
--with-blas=<lib>         use BLAS library <lib>
--with-lapack=<lib>       use LAPACK library <lib>
```

```
Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you have
              headers in a nonstandard directory <include dir>
  CXX         C++ compiler command
  CXXFLAGS    C++ compiler flags
  F77         Fortran 77 compiler command
  FFLAGS      Fortran 77 compiler flags
  CXXCPP      C++ preprocessor
  PYTHON      Python Executable Path

Use these variables to override the choices made by ‘configure’ or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <maherou@sandia.gov>.
```