SANDIA REPORT

SAND2014-16610
Unlimited Release
Printed August 2014

Installing the Anasazi Eigensolver Package with Application to Some Graph Eigenvalue Problems

Erik G. Boman, Karen D. Devine, Richard B. Lehoucq, Nicole L. Slattengren, Heidi Thornquist

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

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

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 ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2014-16610
Unlimited Release
Printed August 2014

Installing the Anasazi Eigensolver Package with Application to Some Graph Eigenvalue Problems

Erik G. Boman
Scalable Algorithms, Org 01426
Sandia National Laboratories
P.O. Box 5800, MS 1318
Albuquerque, NM 87185-1318

Karen D. Devine
Scalable Algorithms, Org 01426
Sandia National Laboratories
P.O. Box 5800, MS 1318
Albuquerque, NM 87185-1318

Richard B. Lehoucq
Computational Mathematics, Org 01442
Sandia National Laboratories
P.O. Box 5800, MS 1320
Albuquerque, NM 87185-1320

Nicole L. Slattengren
Scalable & Secure Sys Research, Org 08961
Sandia National Laboratories
P.O. Box 969, MS 9158
Livermore, CA 94551-0969

Heidi Thornquist
Electrical Models & Simulation, Org 01355
Sandia National Laboratories
P.O. Box 5800, MS 1177
Albuquerque, NM 87185-1177

{egboman, kddevin, rblehou, nlslatt, hkthorn}@sandia.gov

Abstract

The purpose of this report is to document a basic installation of the Anasazi eigensolver package and provide a brief discussion on the numerical solution of some graph eigenvalue problems.

Contents

Figures

Tables

1 Introduction

The purpose of this report is to document a basic installation of the Anasazi [4] eigensolver package and provide a brief discussion on the numerical solution of some graph eigenvalue problems.

Anasazi is a Trilinos [8] package for the numerical solution of the large-scale eigenvalue problem.[1] Anasazi provides a generic interface to a collection of eigensolver algorithms. Matrices and vectors used in computation are treated as opaque objects; only elementary operations on matrices and vectors need to be provided through the interface. After providing the interface implementation, a user may access any of Anasazi's suite of algorithms, including the implementation [9] of the Locally-Optimal Block Preconditioned Conjugate Gradient [10], a Block Krylov-Schur [12], a Block Davidson [3], and an Implicit Riemannian Trust-Region [2] methods, respectively.

In this report, we use Trilinos' Epetra [1] implementation of the Anasazi interface to matrices and vectors. Epetra provides the basic building blocks needed for serial and parallel linear algebra. The *Epetra_Map* class describes the distribution of rows, columns, and vector entries to processes. This class supports both 1D (row-based or column-based) and 2D (nonzero-based) matrix distributions, and plays a key role in enabling the 2D distributions [6] useful for large graphs with skewed degree distributions. The *Epetra_Import* and *Epetra_Export* functions perform communication needed to share data among processes. Sparse matrices can be stored by *Epetra_CrsMatrix*; users may use their own matrix layouts through the virtual *Epetra_RowMatrix* class. Epetra features a multi-vector class *Epetra_MultiVector* that is, in essence, a collection of vectors; this class enables block-based linear and eigensolvers to be efficiently implemented in Trilinos. An extension of Epetra called Epetra64 is also available, which enables Epetra classes to be used for graphs with more than two billion global vertices or edges.

Other packages in Trilinos that have proven useful in our work are the Belos [5] and IFPACK [11] packages. Belos is a block-based linear solver package with a matrix/vector abstraction similar to Anasazi's; we have exploited Belos' block-based solvers to compute commute distances between many pairs of vertices simultaneously. IFPACK is a collection of preconditioners, including Incomplete Cholesky, Symmetric Gauss-Seidel, ILU, and Jacobi preconditioners. A Maximum-weight Spanning Forest (MSF) preconditioner is also implemented in IFPACK; see [7] for details.

[1] See `http://trilinos.sandia.gov/packages/anasazi/` and `http://trilinos.org`, respectively, for online information on Anasazi and Trilinos.

2 Installing Anasazi

The following are directions for building a minimal installation of Anasazi for serial (non-MPI) execution with debug information. These directions were completed on a Dell workstation running Red Hat Enterprise Linux 6. We assume that the Trilinos source code has been downloaded and extracted from either `http://trilinos.sandia.gov/download` or `http://trilinos.org`. These urls provide the most recent version of Trilinos. The following instructions used release tarfile `trilinos-11.8.1-Source.tar.gz`. [2]

We assume that the environment variable `TRILINOS_PATH` contains the directory path for the Trilinos source code. We also need a "build" directory, defined by the environment variable `TRILINOS_BUILD`, where the instance of Trilinos in debug serial mode will be created; this directory should be distinct from `TRILINOS_PATH`.

Step 1. Create a file called `do-configure` with the lines:

```
1  cmake \
2    -D CMAKE_INSTALL_PREFIX:FILEPATH=$TRILINOS_BUILD \
3    -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=OFF \
4    -D Trilinos_ENABLE_Anasazi:BOOL=ON \
5    -D Trilinos_ENABLE_Epetra:BOOL=ON \
6    -D Trilinos_ENABLE_EpetraExt:BOOL=ON \
7    -D Trilinos_ENABLE_Triutils:BOOL=ON \
8    -D Trilinos_ENABLE_Belos:BOOL=ON \
9    -D Trilinos_ENABLE_Ifpack:BOOL=ON \
10   -D Trilinos_ENABLE_TESTS:BOOL=ON \
11   -D TPL_BLAS_LIBRARIES=/usr/lib64/libblas.so.3 \
12   -D TPL_LAPACK_LIBRARIES=/usr/lib64/liblapack.so.3 \
13   -D CMAKE_VERBOSE_MAKEFILE:BOOL=ON \
14   -D Trilinos_ENABLE_DEBUG:BOOL=ON \
15   -D CMAKE_BUILD_TYPE:STRING=DEBUG \
16   -D Trilinos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
17   $TRILINOS_PATH
```

The second line specifies the installation directory for all the header files and libraries; see **Step 4** below. The third line allows us to build and install only the packages we request. The fourth and fifth lines build Anasazi and Epetra, respectively. The sixth and seventh lines build extension libraries for Epetra so that Matrix Market files can be read. Lines 8 and 9 build the Trilinos libraries for block Krylov methods and algebraic preconditioners, respectively. Lines 10 and 11 locate the BLAS and LAPACK libraries. Often, these lines are not needed because CMake can determine the libraries' location. However, if the BLAS and LAPACK libraries are not automatically located, one can specify the appropriate locations as shown in this example.

A minimal build would enable only Anasazi (line 4). Two libraries would be created: `libanasazi.a` and `libteuchos.a`. The latter library provides important software infrastructure needed by Anasazi, including BLAS/LAPACK wrappers, smart pointers, and parameter lists. However, to use Anasazi (and run an example), implementations of the `AnasaziMultiVecTraits` and `AnasaziOperatorTraits` adaptor classes are needed. In this work, we use the implementation, or template specialization, available in files `AnasaziEpetraAdapter.cpp` and `AnasaziEpetraAdapter.hpp`.

Step 2. Execute the command `./do-configure`. This script creates the files needed to build Trilinos.

Step 3. Execute the command `make` to create the executables and libraries.

[2] A `.tar.bz2` download file and a public read-only repository are also available.

Step 4. Execute the command `make install`. This command places the libraries and header files in the two directories `$TRILINOS_BUILD/lib` and `$TRILINOS_BUILD/include`, respectively, as specified in the `do-configure` file (see **Step 1** above).

Step 5. Execute the CMake command `ctest`. This command runs tests of the libraries created. The tail of the output written to the screen is displayed in the box below.

```
.....Output not displayed......
195/198 Test #195: Anasazi_IRTR_complex_lap_test_6 ..........   Passed    0.02 sec
        Start 196: Anasazi_IRTR_complex_lap_test_7
196/198 Test #196: Anasazi_IRTR_complex_lap_test_7 ..........   Passed    0.02 sec
        Start 197: Anasazi_IRTR_complex_test_0
197/198 Test #197: Anasazi_IRTR_complex_test_0 ..............   Passed    0.53 sec
        Start 198: Anasazi_IRTR_complex_test_1
198/198 Test #198: Anasazi_IRTR_complex_test_1 ..............   Passed    0.49 sec

100% tests passed, 0 tests failed out of 198

Label Time Summary:
Anasazi      =  16.76 sec
Belos        =  33.07 sec
Epetra       =  10.33 sec
EpetraExt    =   3.57 sec
Ifpack       =   0.69 sec
Triutils     =   0.10 sec

Total Test time (real) =  64.96 sec
```

This summary indicates that all the libraries were properly created and all the tests completed successfully.

3 Directory structure

Recall that the environment variables `TRILINOS_PATH` and `TRILINOS_BUILD` contain the directory paths for the Trilinos source code and installation directory, respectively. The Anasazi source is located in the directory `$TRILINOS_PATH/packages/anasazi` and contains the subdirectories

cmake doc epetra src **test** testmatrices thyra tpetra

The directory `$TRILINOS_BUILD/packages/anasazi` has a similar directory structure containing the executables and libraries created during **Step 3** and contains the subdirectories

CMakefiles epetra src **test**

In particular, the subdirectory `$TRILINOS_BUILD/packages/anasazi/epetra` contains the subdirectories

CMakefiles example src **test** util

and the subdirectory `$TRILINOS_BUILD/packages/anasazi/epetra/example` contains the subdirectories

BlockDavidson BlockKrylovSchur CMakeFiles GeneralizedDavidson LOBPCG MVOPTester

The subdirectories `BlockDavidson`, `BlockKrylovSchur`, and `LOBPCG` contain the executables associated with the C++ examples located in `$TRILINOS_PATH/packages/anasazi/epetra/example`.

4 Running an Anasazi LOBPCG example

We begin by creating a directory (which is not a subdirectory `$TRILINOS_PATH`) and placing a copy of the file `$TRILINOS_PATH/packages/anasazi/epetra/example/LOBPCG/LOBPCGEpetraExSimple.cpp`. We consider this example in detail and display excerpts from the file below.

```
int main(int argc, char *argv[]) {
  // Get the sorting string from the command line
  std::string which("SM");
  Teuchos::CommandLineProcessor cmdp(false,true);
  cmdp.setOption("sort",&which,"Targetted eigenvalues (SM or LM).");
  // Code to create Epetra matrix A is not shown here; see the Appendix.
  //
  //**********************************
  // Call the LOBPCG solver manager
  //**********************************
  //  Variables used for the LOBPCG Method
  const int    nev        = 10;
  const int    blockSize  = 5;
  const int    maxIters   = 500;
  const double tol        = 1.0e-8;
  typedef Epetra_MultiVector MV;
  typedef Epetra_Operator OP;
  typedef MultiVecTraits<double, Epetra_MultiVector> MVT;
  // Create an Epetra_MultiVector for an initial vector to start the solver.
  // Note:  This needs to have the same number of columns as the blocksize.
  Teuchos::RCP<Epetra_MultiVector> ivec = Teuchos::rcp( new Epetra_MultiVector(Map, blockSize) );
  ivec->Random();
  // Create the eigenproblem.
  Teuchos::RCP<BasicEigenproblem<double, MV, OP> > MyProblem = Teuchos::rcp( new
      BasicEigenproblem<double, MV, OP>(A, ivec) );
  // Inform the eigenproblem that the operator A is symmetric
  MyProblem->setHermitian(true);
  // Set the number of eigenvalues requested
  MyProblem->setNEV( nev );
  // Inform the eigenproblem that you are finishing passing it information
  bool boolret = MyProblem->setProblem();
  if (boolret != true) {
    printer.print(Errors,"Anasazi::BasicEigenproblem::setProblem() returned an error.\n");
    return -1;
  }
  // Create parameter list to pass into the solver manager
  Teuchos::ParameterList MyPL;
  MyPL.set( "Which", which );
  MyPL.set( "Block Size", blockSize );
  MyPL.set( "Maximum Iterations", maxIters );
  MyPL.set( "Convergence Tolerance", tol );
  // Create the solver manager
  SimpleLOBPCGSolMgr<double, MV, OP> MySolverMan(MyProblem, MyPL);
  // Solve the problem
  ReturnType returnCode = MySolverMan.solve();
  // Get the eigenvalues and eigenvectors from the eigenproblem
  Eigensolution<double,MV> sol = MyProblem->getSolution();
  std::vector<Value<double> > evals = sol.Evals;
  Teuchos::RCP<MV> evecs = sol.Evecs;
```

`LOBPCGEpetraExSimple.cpp` is organized into the following four parts:

1. Initialize Anasazi's infrastructure (output manager, command line arguments);
2. Assemble the sparse matrix using Epetra;
3. Set input parameters and call the `LOBPCG` solver; and

4. Output results.

In this example, Line 42 creates an instance `MySolverMan` of the class `SimpleLOBPCGSolMgr`. The constructor is initialized with an instance `MyProblem` of the class `BasicEigenproblem`, that was initialized with the Epetra matrix `A` and starting vector `ivec`. Line 44 calls the `LOBPCG` solver via the command `MySolverMan.solve()`. Line 46 extracts the approximate eigenvalues and eigenvectors via the command `MyProblem->getSolution()`

Algorithm behavior is determined by several input parameters.

- `which`, line 3, specifies which eigenvalues are to be computed and must be a `std::string` equal to ``SM'', ``LM'', ``SR'', ``LR'' representing the eigenvalues with the Smallest Magnitude, Largest Magnitude, Smallest Real value and Largest Real value, respectively.
- `nev`, line 12, represents the number of eigenvalues of interest and is a nonnegative integer no larger than the size of the matrix.
- `blockSize`, line 13, represents the number of vectors in a block and is a nonnegative integer no larger than the size of the matrix.
- `maxIters`, line 14, limits the maximum number of iterations of the algorithm and is a nonnegative integer.
- `tol`, line 15, is the termination criterion that each of the approximate eigenpairs must satisfy and is a floating point number no smaller than machine precision (roughly 10^{-16} in IEEE double precision arithmetic).

The basic organization of `LOBPCGEpetraExSimple.cpp` is used in all of the `LOBPCG` examples. Except for differences in the input parameters, `BlockDavidson` and `BlockKrylovSchur` follow a similar structure.

An example `Makefile` for building applications using Anasazi is shown below.

```
#
# Import a file created by the Trilinos build system
# containing useful information to generate executables.
# First, set the path to the install directory, and second, import the file.
#
include $TRILINOS_BUILD/include/Makefile.export.Trilinos
#
# Copy
# $(TRILINOS_PATH)/packages/anasazi/epetra/example/LOBPCG/LOBPCGEpetraExSimple.cpp
# into the current working directory.
#
LOBPCGEpetraExSimple.exe: LOBPCGEpetraExSimple.cpp
        $(Trilinos_CXX_COMPILER) $(Trilinos_CXX_COMPILER_FLAGS) \
        LOBPCGEpetraExSimple.cpp -o LOBPCGEpetraExSimple.exe \
        $(Trilinos_INCLUDE_DIRS) $(Trilinos_TPL_INCLUDE_DIRS) \
        $(Trilinos_LIBRARY_DIRS) $(Trilinos_LIBRARIES) \
        $(Trilinos_TPL_LIBRARIES) $(Trilinos_EXTRA_LD_FLAGS)
#
clean:
        rm -f LOBPCGEpetraExSimple.exe LOBPCGEpetraExSimple.o
```

The command `make ./LOBPCGEpetraExSimple.exe` creates the executable `LOBPCGEpetraExSimple.exe`. Executing the command `./LOBPCGEpetraExSimple.exe` results in the following output:

Solver manager returned converged.

Eigenvalue	Direct Residual
19.7376	4.04035e−11
49.3345	1.19465e−10
49.3345	3.33375e−11
78.9314	4.09916e−10
98.6308	5.23853e−10

5 Three graph problems of interest

Let A be the adjacency matrix for a simple, undirected, connected graph with n vertices; let D be a diagonal matrix containing the degrees of the vertices for the graph A. These assumptions are only to simplify our presentation. Anasazi can also be used on nonsimple, directed, disconnected graphs; see the section on **Anasazi usage and graph eigenvalue problems**.

1. Find the `nev` smallest magnitude (`which="SM"`) approximate eigenvalues λ and eigenvectors x of the *normalized* graph Laplacian matrix eigenvalue problem:

$$Lx := \big(I - D^{-1/2} A D^{-1/2}\big)x = x\lambda, \qquad A = A^T, \quad 0 \leq \lambda \leq 2.$$

2. Find the `nev` smallest magnitude (`which="SM"`) approximate eigenvalues ν and eigenvectors y of the *combinatorial* graph Laplacian matrix eigenvalue problem:

$$L_c y := \big(D - A\big)y = y\nu, \qquad A = A^T, \quad 0 \leq \nu \leq 2n.$$

3. Find the `nev` largest magnitude (`which="LM"`) approximate eigenvalues η and eigenvectors z of the matrix eigenvalue problem

$$Az = z\eta, \qquad A = A^T, \quad -n \leq \eta \leq n$$

Table 1. Primary parameters

	BlockDavidson	BlockKrylovSchur	LOBPCG	Description
which	x	x	x	Portion of the eigenvalues
tol	x	x	x	Residual tolerance
nev	x	x	x	Eigenpair approximations requested
blockSize	x	x	x	Number of vectors in a block
numBlocks	x	x		Number of blocks
maxRestarts	x	x		Max number of restarts
maxIters			x	Max number of iterations

6 Input parameters

We now provide some guidance on the influence of the primary parameters on the performance of the Anasazi eigensolvers `BlockDavidson`, `BlockKrylovSchur`, and `LOBPCG`. Table 1 lists the primary input parameters for `BlockDavidson`, `BlockKrylovSchur` and `LOBPCG`.

Let N be a preconditioner for T, i.e., $N^{-1}T \approx I$. Here T represents the normalized graph Laplacian $I - D^{-1/2}AD^{-1/2}$, combinatorial graph Laplacian $D - A$, or the adjacency matrix A.

1. An approximate eigenvalue θ and approximate eigenvector u for the matrix T is accepted when

$$\|u\theta - Tu\|_2 \leq \theta\tau, \quad \|u\|_2 = 1, \quad \theta \neq 0,$$

   for the residual tolerance τ and $\|\cdot\|_2$ is the Euclidean norm. This is an approximation to a relative residual error and is not defined when $\theta = 0$. The above test is invariant under a (nonzero) scaling of T, i.e. αT, but is not invariant under a shift of T because

$$\|u(\theta+\alpha) - (T+\alpha I)u\|_2 \leq (\theta+\alpha)\tau. \tag{6.1}$$

   Selecting an appropriate τ requires some care and invariably depends upon the underlying problem.

2. `BlockKrylovSchur` iteratively constructs `blockSize` vectors at a time, an orthogonal basis for the block Krylov subspace

$$\text{span}\{\underbrace{v, Tv, \cdots, T^{\texttt{numBlocks}-1}v}_{\texttt{blockSize*numBlocks}}\},$$

   where v is an initial vector consisting of `blockSize` vectors.

3. `BlockDavidson` iteratively constructs `blockSize` vectors at a time, an orthogonal basis for the subspace

$$\text{span}\{\underbrace{r_1, N^{-1}r_2, \cdots, N^{-\texttt{numBlocks}+1}r_{\texttt{numBlocks}}}_{\texttt{blockSize*numBlocks}}\},$$

   where $r_i = u_i\theta_i - Tu_i$ consisting of `blockSize` vectors, is determined at iteration i.

4. `BlockKrylovSchur` and `BlockDavidson` are restarted a maximum of `maxRestarts` number of times.

5. A "restart" is a scheme for extracting a subspace of smaller dimension from the subspace of dimension `blockSize*numBlocks`. The reduced size subspace is then iteratively extended `blockSize` vectors at a time until the subspace is of dimension `blockSize*numBlocks` again.

6. `LOBPCG` constructs an orthogonal basis for the subspace

$$\text{span}\{\underbrace{u_{i-1}, u_i, N^{-1}\left(\theta_i u_i - T u_i\right)}_{\texttt{3*blockSize}}\}$$

   where u_i and u_{i-1} are the `blockSize` approximate eigenvectors corresponding to the `which="SM"` approximate eigenvalues at iterations i and $i-1$, respectively. The variable `maxIters` denotes the maximum number of iterations.

7. The `BlockDavidson` and `LOBPCG` eigensolvers can directly use a preconditioner. `BlockKrylovSchur` cannot directly use a preconditioner and instead requires a preconditioned iteration.

8. `BlockKrylovSchur` can also be used in *shift-invert* mode; e.g., $T = (L + \sigma I)^{-1}$ where $0 < \sigma < \lambda_1$ and `which="LM"`. The restriction on the shift σ enables a preconditioned conjugate gradient iteration with the preconditioner N.

9. For the normalized and combinatorial graph Laplacian matrices, `which="SM"` is the same as `which="SA"` (smallest algebraic) and `which="SR"` (smallest real part).

10. Anasazi eigensolvers `BlockDavidson`, `BlockKrylovSchur`, and `LOBPCG` can be used for all three problems since these problems are symmetric eigenvalue problems when the graph is undirected.

7 Anasazi usage and graph eigenvalue problems

Suppose the graph is connected, simple and undirected. The eigenvectors of the normalized $L = D^{-1/2} L_c D^{-1/2}$ and combinatorial L_c graph Laplacian matrices coincide if and only if the matrices commute; e.g.,

$$\underbrace{D^{-1/2}(D-A)D^{-1/2}}(D-A) = (D-A)\underbrace{D^{-1/2}(D-A)D^{-1/2}}.$$

This occurs only when the graph is regular and, then, our experience is that `BlockKrylovSchur` is an excellent choice. When the graph is not regular, and has a skew degree distribution, our experience is as follows:

- `LOBPCG` performs well for computing the smallest eigenvalues of the combinatorial graph Laplacian $D-A$;
- `BlockKrylovSchur` performs well for computing the smallest eigenvalues of the normalized graph Laplacian $I - D^{-1/2} A D^{-1/2}$ and the largest eigenvalues of both graph Laplacians;
- `BlockDavidson` was not competitive;
- Preliminary findings demonstrated that Implicit Riemannian Trust-Region `IRTR` available within Anasazi was competitive with `LOBPCG` and `BlockKrylovSchur`.

The Anasazi eigensolvers can be used on disconnected and directed graphs. The complication with disconnected graphs is that the number of connected components determines the dimension of the nullspace for the graph Laplacian. In many applications, the graph consists of a number of connected components, leading to a potentially large number of zero eigenvalues and corresponding eigenvectors. A good approach is to invoke the eigensolver on each connected component. A directed graph leads to a nonsymmetric graph Laplacian since the adjacency matrix A is no longer symmetric. Therefore, only `BlockKrylovSchur` may be used.

Assume that the graph is connected. Then the normalized and combinatorial graph Laplacian matrices both have a zero eigenvalue $\lambda_0 = \nu_0 = 0$ with eigenvector $D^{1/2}c$ and c, respectively, where $c \in \text{span}\{e\}$ for the vector e of all ones. There are (at least) three ways to handle the nullspace vector. First, let the eigensolver compute the approximation (`which="SM"` or `"SA"` or `"SR"`). Second, after an application of the graph Laplacian matrix vector product to a vector, orthogonalize against the nullspace vector. Third, add a multiple α of the identity matrix to the graph Laplacian. This last approach modifies the zero eigenvalue to be the multiple and the eigenvector remains the same. However, the bound (6.1) explains that a positive α modifies the termination criterion, so accuracy of the computed eigenvector approximation may be affected.

Our experience is that a Jacobi preconditioner works quite well on the combinatorial Laplacian corresponding to a non-regular graph, and a symmetric Gauss-Seidel preconditioner works well for the normalized graph Laplacian; see the preliminary findings of Deweese and Boman [7].

A good metric by which to compare different algorithms (and implementations) is the number of matrix-vector products with L (or L_C) and applications of the preconditioner N needed to achieve a prescribed level of accuracy (as measured by the residual tolerance). In the final analysis, though, wall-clock time is the most important consideration. There are two important sources of additional cost:

1. The cost of maintaining orthogonality of the basis vectors to machine precision. The number of floating point operations is `blockSize*numBlocks*n`^{2} where `n` is the number of vertices of the graph.

2. The cost of solving a dense eigenvalue problem of order `blockSize*numBlocks`. The number of floating point operations is `(blockSize*numBlocks)`^{3}.

As the problem size and/or the number of eigenvalues of interest increases, the relative cost of orthogonalization and/or the dense eigenvalue problem increases. Constructing an orthogonal basis efficiently and stably is critical to the success of an eigensolver; this is automatically accomplished by the Anasazi software.

Anasazi usage errors are of two types:

1. Simple errors that occur from improper input-parameter values.

2. Far more challenging errors involve the matrix T and preconditioner N, such as incorrect construction of the matrix or preconditioner. Such errors may compile and execute, but may not represent the system intended by the user. Unit testing of the matrix and preconditioner outside of Anasazi is suggested.

Our recommendation is to first experiment with the various example programs provided with Anasazi. This includes modifying the default tolerances and parameters (including specifying them incorrectly). The example programs also check the eigenvalue and eigenvector approximations computed via a residual check; this is a recommended practise. A further useful check is whether the computed eigenvectors are orthogonal to machine precision. We recommend that the user then the (slowly) modify the example program to use another matrix.

Algorithms that compute the eigenvalues and eigenvectors of a matrix are given by a nonlinear iteration. As a result, rerunning the same code will not provide the exact same performance unless the floating-point arithmetic can be guaranteed to be performed in exactly the same manner. In parallel environments, for example, the order of operations and the resulting floating-point arithmetic will differ with the number of processes. A good practice is to run the eigensolver several times (including varying the starting vector) followed by averaging the resulting runtime statistics.

That the algorithms implement a nonlinear iteration also typically manifest in the disappointing realization that "small" changes to the graph, e.g., insertion or deletion of edges, cannot exploit computed eigenvector approximations. Our experience is that the "small" changes do not result in savings when reusing the eigenvectors previously computed.

References

[1] *Epetra home page*. http://trilinos.sandia.gov/packages/epetra.

[2] P.-A. ABSIL, C. BAKER, AND K. GALLIVAN, *A truncated-CG style method for symmetric generalized eigenvalue problems*, Journal of Computational and Applied Mathematics, 189 (2006), pp. 274–285. See http://dx.doi.org/10.1016/j.cam.2005.10.006.

[3] P. ARBENZ, U. HETMANIUK, R. LEHOUCQ, AND R. TUMINARO, *A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods*, Intl. J. for Numer. Methods Engrg., 64 (2005), pp. 201–236. See http://dx.doi.org/10.1002/nme.1365.

[4] C. G. BAKER, U. L. HETMANIUK, R. B. LEHOUCQ, AND H. K. THORNQUIST, *Anasazi software for the numerical solution of large-scale eigenvalue problems*, ACM Trans. Math. Softw., 36 (2009), pp. 13:1–13:23. See http://dx.doi.org/10.1145/1527286.1527287.

[5] E. BAVIER, M. HOEMMEN, S. RAJAMANICKAM, AND H. THORNQUIST, *Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems*, Scientific Programming, 20 (2012), pp. 241–255. See http://dx.doi.org/10.3233/SPR-2012-0352.

[6] E. G. BOMAN, K. D. DEVINE, AND S. RAJAMANICKAM, *Scalable matrix computations on large scale-free graphs using 2d graph partitioning*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, 2013, pp. 50:1–50:12. http://doi.acm.org/10.1145/2503210.2503293.

[7] K. DEWEESE AND E. G. BOMAN, *A comparison of preconditioners for solving linear systems arising from graph Laplacians*, report SAND-2013-9772P, Sandia National Laboratories, 2013. See http://www.sandia.gov/~egboman/papers/DeweeseBoman2013.pdf.

[8] M. A. HEROUX, R. A. BARTLETT, V. E. HOWLE, R. J. HOEKSTRA, J. J. HU, T. G. KOLDA, R. B. LEHOUCQ, K. R. LONG, R. P. PAWLOWSKI, E. T. PHIPPS, ET AL., *An overview of the trilinos project*, ACM Transactions on Mathematical Software (TOMS), 31 (2005), pp. 397–423. See http://dx.doi.org/10.1145/1089014.1089021.

[9] U. HETMANIUK AND R. LEHOUCQ, *Basis selection in LOBPCG*, Journal of Computational Physics, 218 (2006), pp. 324–332. See http://dx.doi.org/10.1016/j.jcp.2006.02.007.

[10] A. V. KNYAZEV, *Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method*, SIAM. J. Sci. Comput., 23(2) (2001), pp. 517–541. See http://dx.doi.org/10.1137/S1064827500366124.

[11] M. SALA AND M. HEROUX, *Robust algebraic preconditioners with IFPACK 3.0*, Tech. Report SAND-0662, Sandia National Laboratories, 2005.

[12] G. W. STEWART, *A Krylov-Schur algorithm for large eigenproblems*, SIAM J. Matrix Anal. Appl., 23 (2000), pp. 601–614. See http://dx.doi.org/10.1137/S0895479800371529.