Anasazi software for the numerical solution of large-scale eigenvalue problems

C. G. Baker and U. L. Hetmaniuk and R. B. Lehoucq and H. K. Thornquist
Sandia National Laboratories

Anasazi is a package within the Trilinos software project that provides a framework for the iterative, numerical solution of large-scale eigenvalue problems. Anasazi is written in ANSI C++ and uses modern software paradigms to enable the research and development of eigensolver algorithms. Furthermore, Anasazi provides implementations for some of the most recent eigensolver methods. The purpose of our paper is to describe the design and development of the Anasazi framework. A performance comparison of Anasazi and the popular FORTRAN 77 code ARPACK are given.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematical Software**]: ; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Algorithms, Design, Performance; Reliability, Theory

Additional Key Words and Phrases: Eigenvalue problems, Numerical Algorithms, Generic programming

Anasazi is a package within the Trilinos Project [Heroux et al. 2005] that uses ANSI C++ and modern software paradigms to implement algorithms for the numerical solution of large-scale eigenvalue problems. We define a large-scale eigenvalue problem to be one where a small number (relative to the dimension of the problem) of eigenvalues and the associated eigenspace are computed, and only knowledge of the underlying matrix via application on a vector (or group of vectors) is assumed.

An inspiration for Anasazi is the ARPACK [Lehoucq et al. 1998] FORTRAN 77 software library. ARPACK implements one algorithm, namely an implicitly restarted Arnoldi method [Sorensen 1992]. In contrast, Anasazi provides a software framework, including the necessary infrastructure, to implement a variety of algorithms. We justify our claims by implementing block variants of three popular algorithms: a Davidson [Davidson 1975] method, a Krylov-Schur [Stewart 2001a] method, and an implementation of LOBPCG [Knyazev 2001].

ARPACK has proven to be a popular and successful FORTRAN 77 library for

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. Authors' addresses: C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, Sandia National Laboratories, Computational Mathematics & Algorithms, MS 1320, P.O.Box 5800, Albuquerque, NM 87185-1320; email {cgbaker,ulhetma,rblehou}@sandia.gov. H. K. Thornquist, Sandia National Laboratories, Electrical and Microsystem Modeling, MS 0316, Albuquerque, NM 87185-0316; email hkthorn@sandia.gov.

the numerical solution of large-scale eigenvalue problems. A crucial reason for the popularity of ARPACK is the use of a reverse communication [Lehoucq et al. 1998, p. 3] interface for applying the necessary matrix-vector products. This allows ARPACK to provide a callback for the needed matrix-vector products in a simple fashion within FORTRAN 77. Unfortunately, the reverse communication interface is cumbersome, challenging to maintain, and does not allow data encapsulation. Moreover, because ARPACK uses a procedural programming paradigm where the matrix-vector operations rely upon the physical representation of the data manipulated, ARPACK is susceptible to design changes. Hence, code reuse is limited and software complexity and maintenance are more cumbersome.

The Anasazi framework employs more modern software development paradigms through the use of generic programming via static and dynamic polymorphism [Vandevoorde and Josuttis 2002, Chapter 14]. Static polymorphism, via templating of the linear algebra objects, allows algorithms in Anasazi to be written in a generic manner (i.e., independent of the data types). Dynamic polymorphism, via virtual functions and inheritance, allows eigensolvers to be decoupled from mechanisms such as orthogonalization and stopping conditions; this functionality can then be decided at runtime. The upshot of this decoupling is the facilitation of code reuse and algorithmic flexibility.

The result of these design choices is that Anasazi is an extensible and interoperable software framework. Extensibility is apparent in the infrastructure's support for a significant class of large-scale eigenvalue algorithms. Extensions can be made through the addition of new algorithms or through modification of existing algorithms. This is encouraged by promoting code modularization and multiple levels of access to solvers and their data. Interoperability is enabled via the treatment of both matrices and vectors as opaque objects—only knowledge of the matrix and vectors via elementary operations is necessary. This permits algorithms to be implemented in a generic manner, requiring no knowledge of the underlying linear algebra types or their specific implementations.

We emphasize that our interest is not solely in modern software paradigms. Rather, our paper demonstrates that a rich collection of block eigensolvers is easily implemented using modern programming techniques. Our approach is algorithm-oriented [Musser and Stepanov 1994], in that requirements for efficient implementations of the necessary algorithms is considered first. This is then followed by a formulation of the software abstractions capable of implementing these algorithms and their constituent mechanisms in sufficiently diverse ways. The result is a collection of implementations that are efficient and flexible. We believe that Anasazi is the natural successor to ARPACK, inheriting and extending the quality practices employed by ARPACK.

Related software efforts that implement several algorithms for large-scale eigenvalue (the reader is referred to [Hernández et al. 2005] for a software survey) problems are:

—The Preconditioned Iterative Multi-Method Eigensolver (PRIMME) [Stathopoulos and McCombs 2006] is a C library to find a number of eigenvalues and their corresponding eigenvectors of a real symmetric or complex Hermitian matrix. PRIMME provides a highly parametrized Jacobi-Davidson [Sleijpen and van der

Vorst 1996] iteration, allowing the behavior of multiple eigensolvers to be obtained via the appropriate selection of parameters;

—The Scalable Library for Eigenvalue Problem Computations (SLEPc) [Hernández et al. 2006] library written in C for the solution of large scale sparse eigenvalue problems on parallel computers. SLEPc is an extension of the popular PETSc [Balay et al. 2001] and can be used for either Hermitian or non-Hermitian, standard or generalized, eigenproblems.

The PRIMME software provides a flexible metasolver capable of implementing a variety of eigensolvers. Predefined parameters are provided to emulate a large number of powerful and popular eigensolvers, allowing easy use of the software by novice users. Expert users may manually specify the parameters in order to access the full flexibility available in the solver's behavior. Therefore, PRIMME is valuable both as a convenient eigensolver for practitioners and a platform for experimentation by eigensolver researchers. However, while parameters are provided to control mechanisms such as, e.g., stopping conditions and orthogonalization, the user is limited to those choices provided by the developers of PRIMME. Unsupported behavior requires intrusive modification to the eigensolver. Furthermore, the assumptions made regarding data type and storage may be prohibitive for some users. PRIMME provides implementations only over double precision real and complex fields, but no support exists for `float` or extended precision scalar fields. The reason for this is that the lack of generic programming ability in the C programming language requires a separate implementation for each desired scalar type. The metasolver underlying PRIMME is limited to real symmetric and complex Hermitian standard eigenvalue problems.

SLEPc extends the PETSc toolkit to provide a library of solver for numerous eigenvalue problems. The flexibility of SLEPc is illustrated by its ability to wrap other eigensolvers packages, most notably ARPACK and PRIMME. The use of PETSc gives SLEPc users access to a large library of linear and nonlinear solvers, preconditioners and matrix formats. PETSc uses C language features such as `typedef`s and function pointers to support some generic programming and object-oriented paradigms. However, SLEPc's reliance on PETSc requires that the user employ PETSc for vectors and matrices as well. Furthermore, while PETSc/SLEPc can be compiled with support for complex arithmetic, only one version of the library can be used at a time, and like PRIMME, the support is limited to only double precision real and complex scalar fields. As with PRIMME, mechanisms such as orthogonalization were hard-coded, allowing only parametrized control over their behavior.

Anasazi was designed to include features of these and other eigensolver packages while avoiding some of the pitfalls mentioned above. These features include efficiency, portability, flexibility, ease of use, maintainability and interoperability. Anasazi enjoys interoperability with other Trilinos packages, providing it with a rich set of preconditioners, linear solvers and linear algebra libraries. Like SLEPc, the Anasazi framework allows the description of diverse eigensolvers, including the encapsulation of other eigensolver packages. Unlike SLEPc and PETSc, the eigensolvers use abstract interfaces to access auxiliary functionality such as orthogonalization, selecting desired eigenvalues, stopping conditions, etc. As a result, this

auxiliary functionality can be customized by users without requiring modification to (unrelated) eigensolver code. Specific implementation choices for this functionality can even be decided at runtime.

Another distinction between Anasazi and the previously described packages is that Anasazi makes no specification on the data representation. Just as the authors of SLEPc invested significant effort to ensure that the eigensolvers would be able to exploit a wide variety of matrix operators, the Anasazi code was designed to admit operation with any user choice of scalar field, vector and operator. This is accomplished using the template mechanism in the C++ programming language, an option not available to SLEPc or PRIMME. As a result, an Anasazi eigensolver using single-precision complex arithmetic can be used alongside another Anasazi eigensolver using an extended precision scalar type.

The result is an eigensolver that is significantly more flexible than previous efforts, allowing its inclusion in a diverse applications environments in addition to providing an arena for research into eigensolvers as well as their constituent components.

The rest of this paper is organized as follows. Section 1 briefly reviews one class of algorithms that can be implemented within Anasazi, in order to explore the types of operations necessary for a eigensolver framework. Section 2 reviews the Anasazi framework, arguing for some of the design decisions and illustrating the benefits of these decisions. Lastly, Section 3 provides some timings comparing ARPACK and Anasazi in the hopes of allaying any concerns over the object-oriented overhead of this undertaking.

1. ALGORITHMIC INSPIRATION

The Anasazi software framework provides tools that are useful for solving a wide variety of eigenvalue problems. While development of the package continues to expand its scope to other important eigenvalue problems (e.g., nonlinear eigenproblems, constrained eigenproblems), the solvers currently released with the project are aimed at computing a partial eigen-decomposition for the generalized eigenvalue problem

$$\mathbf{A}\mathbf{x} = \mathbf{B}\mathbf{x}\lambda, \qquad \mathbf{A}, \mathbf{B} \in \mathbb{C}^{n \times n} \ . \tag{1}$$

In this paper, the matrices $\mathbf{A}$ and $\mathbf{B}$ are large, possibly sparse, and we assume that only their application to a block of vectors is required. The reader is referred to [Saad 1992; Sorensen 2002; Stewart 2001b; van der Vorst 2002] for background information and references on the large-scale eigenvalue problem.

Algorithm 1.1 is a simple extension of the Rayleigh-Ritz procedure given in [Stewart 2001b, p.281]. This algorithm lists the salient steps found in the majority of large-scale eigensolvers, namely subspace projection methods, and as such is useful for examining some of the functionality that should be provided by a general eigensolver framework. These include:

—matrix-matrix applications: $\mathbf{AU}$;
—block inner products: $\mathbf{V}^H(\mathbf{AU})$;
—solution of typically much smaller eigenproblems (step 3).

Other linear algebra operations include methods for creating vectors and performing vector arithmetic. We list our primitives in Section 2. The functions $\Phi(\cdot)$ and $\Psi(\cdot)$

ALGORITHM 1.1: **Rayleigh-Ritz Algorithm**

(1) *Let the matrices* $\mathbf{M}$, $\mathbf{U}$ *and* $\mathbf{V}$ *be given*
(2) *Form the Rayleigh quotients* $\mathbf{V}^H\mathbf{M}\Phi(\mathbf{A})\mathbf{U}$ *and* $\mathbf{V}^H\mathbf{M}\Psi(\mathbf{B})\mathbf{U}$ *where* $\Phi(\cdot)$ *and* $\Psi(\cdot)$ *are matrix functions*
(3) *Compute an eigen-decomposition for the matrix pencil* $(\mathbf{V}^H\mathbf{M}\Phi(\mathbf{A})\mathbf{U}, \mathbf{V}^H\mathbf{M}\Psi(\mathbf{B})\mathbf{U})$
(4) *Return the eigen-decomposition as an approximation for the pencil* $(\mathbf{A}, \mathbf{B})$

are often used to improve convergence to the eigenvalues and eigenspace of interest. The choice of these operators is beyond the scope of this paper. Algorithm 1.1 needs to be augmented with several steps in order to result in an *eigen-iteration*. Algorithm 1.2 lists these additional steps, so defining an eigen-iteration.

ALGORITHM 1.2: **Eigen-iteration**

(1) *Update the bases* $\mathbf{U}$ *and* $\mathbf{V}$
(2) *Determine whether any portion of the eigen-decomposition is of acceptable accuracy*
(3) *Deflate the accurate portions of the eigen-decomposition*
(4) *Terminate the eigen-iteration*

The decisions involved in Steps 2 and 4 are likely candidates for decoupling from the implementation of a particular eigen-iteration. For a particular eigen-iteration, Step 3 deflation also may be conducted in a manner of ways. Steps 1 and 3, whatever their implementation, are typically implemented using a utility methods for orthogonalization. These methods provide another opportunity for decoupling functionality that need not be explicitly described.

The algorithms that are currently available in Anasazi are:

(1) a block extension of a Krylov-Schur method [Stewart 2001a],
(2) a block Davidson method as described in [Arbenz et al. 2005],
(3) an implementation of LOBPCG as described in [Hetmaniuk and Lehoucq 2006].

We briefly remark that the Anasazi framework was designed to support block methods, defined as those that apply $\mathbf{A}$ or $\mathbf{B}$ to a collection of vectors. One consequence of this design is the use of multivector data structures, defined as a collection of vectors. The use of multivectors can improve the ratio of floating-point operations to memory references and so better exploit a memory hierarchy.

This discussion illustrates that many distinct parts make up a large-scale eigensolver code: orthogonalization, sorting tools, dense linear algebra, convergence testing, multivector arithmetic, etc. Anasazi presents a framework of algorithmic components, decoupling operations where possible in order to simplify component verification, encourage code reuse, and maximize flexibility in implementation.

2. ANASAZI SOFTWARE FRAMEWORK

This section outlines the Anasazi software framework and motivates the design decisions made in the development of Anasazi. Three subsections describe the Anasazi operator/vector interface, the eigensolver framework, and a review of the various classes in Anasazi. The reader is referred to [Baker et al. ; Sala et al. 2004] for software documentation and a tutorial.

2.1 The Anasazi Operator/Vector Interface

Anasazi utilizes abstract interfaces for matrix operators and multivectors. This allows generic programming techniques to be used when developing the numerical algorithms in Anasazi. In C++, generic programming is traditionally implemented using virtual functions or templates. The abstract numerical interfaces used in Anasazi are supported via templates. Most classes in Anasazi accept three template parameters:

—a scalar type, describing the field over which the vectors and operators are defined;

—a multivector type, that depends upon the scalar type, providing a data structure that denotes a collection of vectors; and

—an operator type, that depends upon the multivector and scalar types, providing linear operators used to define eigenproblems and preconditioners.

give small code example of template use in C++

Templating an eigensolver on operator, multivector, and scalar types makes software reuse easier. Consider in contrast that ARPACK implements the four subroutines—`SNAUPD`, `DNAUPD`, `CNAUPD`, and `ZNAUPD`—for solving non-Hermitian eigenproblems. Four separate subroutines are provided for these four FORTRAN 77 floating point types (single and double precision real, and single and double precision complex). Moreover, four additional subroutines are needed for a distributed memory implementation (say, using MPI). In Anasazi, templating on multivector and scalar types requires the maintain of a single code. The multivector templating allows us to separate the eigenvalue algorithm from the linear algebra data structures. The operator type templating is analogous to the reverse communication interface used by ARPACK for providing matrix-vector products.

this is an opportunity to explain why we choose templating over dynamic polymorphism.

Because the underlying data types are unknown to the Anasazi developer, algorithms are developed abstractly. Access to the functionality of the underlying objects is provided via the classes `MultiVecTraits` and `OperatorTraits`. These classes implement the traits mechanism [Meyers 1995] and specify the operations that the multivector and operator classes must support in order to be used within Anasazi. This mechanism hides the low-level details of the underlying data structures. As a result, a single eigensolver implementation in Anasazi can be exploited on a number of diverse computing platforms: serial, distributed memory parallel, multi-core shared memory parallel, graphics processing units, FPGAs, etc.

The methods defined by these traits classes are listed in Table I. Most of the methods listed are self-explanatory. The first three `MultiVecTraits` methods are C++ *virtual* constructors [Meyers 1996, pp. 123–129] that create multivectors from

Table I. Methods provided by the `OperatorTraits` and `MultiVecTraits` interfaces. *fix this table*

OperatorTraits<ST,MV,OP>	
Method name	*Description*
Apply(A, X, Y)	Applies the operator A to the multivector X, placing the result in the multivector Y.
MultiVecTraits<ST,MV>	
Method name	*Description*
Clone$(X, numvecs)$	Creates a new multivector from X with numvecs vectors.
CloneCopy$(X, index)$	Creates a new multivector with a copy of the contents of a subset of the multivector X (deep copy).
CloneView$(X, index)$	Creates a new multivector that shares the selected contents of a subset of the multivector X (shallow copy).
GetVecLength(X)	Returns the vector length of the multivector X.
GetNumberVecs(X)	Returns the number of vectors in the multivector X.
MvTimesMatAddMv$(alpha, X, M, beta, B)$	Applies a serial, dense matrix M to multivector A and accumulates the result into another multivector B: $B \leftarrow \alpha AM + \beta B$.
MvAddMv$(alpha, A, beta, B)$	Performs multivector AXPBY: $B \leftarrow \alpha A + \beta B$.
MvTransMv$(alpha, A, B, C)$	Computes the serial, dense matrix $C \leftarrow \alpha A^H B$.
MvDot(A, B, c)	Computes the vector c where the components are the individual dot-products of the i-th columns of A and B, i.e., $c[i] = A[i]^H B[i]$.
MvScale(A, c)	Scales the columns of a multivector by the entries of c.
MvNorm(A, c)	Computes the 2-norm of each vector of A: $c[i] = \|A[i]\|_2$
SetBlock$(A, B, index)$	Copies the vectors in A to a subset of vectors in B.
MvRandom(A)	Replaces the entries in the multivector A with random numbers.
MvInit(A, α)	Replaces each entry in the multivector A with α.
MvPrint(A)	Prints the Multivector to an output stream.

a multivector provided by the user. Deep and shallow copy denotes whether the object contains the storage for the multivector entries or not. A shallow copy is useful when only a subset of the columns of a multivector is required for computation. This concept provides Anasazi access to the performance benefits previously available only to C/FORTRAN implementations.

The use of `MultiVecTraits` and `OperatorTraits` requires that specializations of these traits classes have been implemented for given template arguments. Anasazi provides the following specializations of these traits classes:

—`Epetra_MultiVector` and `Epetra_Operator` (with scalar type `double`) allow Anasazi to be used with the Epetra [Heroux et al.] linear algebra library provided with Trilinos.

—`Thyra::MultiVectorBase<ST>` and `Thyra::LinearOpBase<ST>` (with arbitrary scalar type `ST`) allow Anasazi to be used with any classes that implement the abstract interfaces provided by the Thyra [Bartlett et al.] package of Trilinos.

—`MultiVec<ST>` and `Operator<ST>` (with arbitrary scalar type `ST`) allow Anasazi to be used with any classes that implement the Anasazi abstract base classes `MultiVec` and `Operator`.

For scalar, multivectors and operators types not covered by these, specializations of `MultiVecTraits` and `OperatorTraits` must be created. The benefit of the traits mechanism is that it does not require that the chosen types are C++ classes. Furthermore, it does not require modification to exiting data types, as the traits class specialization occurs external to the chosen types.

2.2 The Anasazi Framework

We explain how the example Rayleigh-Ritz method of Algorithm 1.1 and the additional steps listed in Algorithm 1.2 may be implemented within the Anasazi framework.

i think the list below is not necessary, at least, the latter two entries. be more clear. In Anasazi, eigensolvers are derived classes of the abstract base class `Eigensolver`. The purpose of a subclass of `Eigensolver` is to encapsulate an eigen-iteration and its associated state/data. An inheritance relationship was chosen for the following reasons:

—the abstract base class defines an interface used for checking the status of a solver by a status test;

—a concrete derived class will perform the iteration associated with a specific eigensolver algorithm; and

—a concrete derived class will act as a container for the state associated with its particular iteration.

explain why we desire status tests: runtime chosen stopping conditions, hook for checking state of solver for numerous other logic (restarting, debugging, etc). contrast against ARPACK.

The class `StatusTest` is used to specify stopping conditions for an eigen-iteration. `Eigensolver` queries the `StatusTest` during its class method `iterate()` to determine whether or not to continue iterating. Concrete subclasses of `StatusTest` provide particular stopping criteria. A typical interaction between these two classes is illustrated in Figure 1.

```
SomeEigensolver::iterate() {
  while ( somestatustest.checkStatus(this) != Passed ) {
    //
    // perform eigensolver iterations
    //
  }
  return;  // return back to caller
}
```

Fig. 1. Example of communication between status test and eigensolver

Each `StatusTest` provides a virtual method, `checkStatus()`, which queries the methods provided by `Eigensolver` and determines whether the solver meets the criteria defined by a particular status test. After a solver returns from `iterate()`, the caller has the ability to access the solver's state and the option to re-initialize the solver with a new state and continue iterating.

perhaps utility classes should be introduced before discussion of solver managers. emphasize: solver managers driven by parameter lists.

While this approach to interfacing with the solver is powerful, it can be overwhelming. It requires the user to construct a number of support classes and to manage calls to `Eigensolver::iterate()`. The `SolverManager` class was developed to encapsulate an instantiation of `Eigensolver`, providing additional functionality and handling low-level interaction with the eigensolver that a user may not want to specify. Solver managers are intended to be easy to use, while still providing the features and flexibility needed to solve large-scale eigenvalue problems.

For example, the constructor of `BlockDavidsonSolMgr` accepts only two arguments: an `Eigenproblem` specifying the eigenvalue problem to be solved and a `ParameterList` of options specific to this solver manager. This solver manager instantiates a `BlockDavidson` subclass of `Eigensolver`, along with the status tests and other support classes needed by the eigensolver, as specified by the parameter list. To solve the eigenvalue problem, the user simply calls the `solve()` method of `BlockDavidsonSolMgr`. The solver manager calls `iterate()`, performs restarts and locking, and places the final solution into the `Eigenproblem`.

Under this framework, users have a number of options for performing eigenvalue computations with Anasazi:

—Use an existing solver manager. In this case, the user is limited to the functionality provided by the existing solver managers.

—Develop a new solver manager for an existing eigensolver. The user can extend the functionality provided by the eigensolver, specifying custom configurations for status tests, orthogonalization, restarting, locking, etc.

—Implement a new eigensolver (and so extend Anasazi). The user can write an eigensolver for an iteration that is not represented in Anasazi. The user still has the benefit of the support classes provided by Anasazi, and the knowledge that this effort can be easily employed by anyone already familiar with Anasazi.

2.3 Anasazi Classes

Anasazi is designed with extensibility in mind, so that users can augment the package with any special functionality that may be needed. However, the released version of Anasazi provides all functionality necessary for solving a wide variety of problems. This section lists and briefly describes the classes used in Anasazi.

We remark that Anasazi is largely independent of Trilinos. Anasazi only relies on the Trilinos Teuchos package [Heroux et al.] that provides a common suite of tools, such as: `RCP`, a reference-counting smart pointer [Detlefs 1992]; `ParameterList`, a list for algorithmic parameters of varying data types; and the BLAS [Lawson et al. 1979; Blackford et al. 2002] and LAPACK [Anderson et al. 1999].

The abstract base class `Eigenproblem` is a container for the components and solution of an eigenvalue problem. By requiring eigenproblems to derive from `Eigenproblem`, Anasazi defines a minimum interface that can be expected of all eigenvalue problems by the classes that will work with the problems (e.g., eigensolvers and status testers). Anasazi provides users with a concrete implementation of `Eigenproblem`, called `BasicEigenproblem`. This basic implementation provides all the functionality necessary to describe both generalized and standard, Hermitian

and non-Hermitian linear eigenvalue problems.

The methods for storing and retrieving the results of the eigenvalue computation in an `Eigenproblem` are:

```
const Eigensolution & Eigenproblem::getSolution();
void Eigenproblem::setSolution(const Eigensolution & sol);
```

The `Eigensolution` class was developed in order to facilitate setting and retrieving the solution data from an eigenproblem. Furthermore, the `Eigensolution` class was designed for storing solution data from both Hermitian and non-Hermitian eigenproblems. This structure contains the following information:

—`RCP< MV > Evecs`
The computed eigenvectors.

—`RCP< MV > Espace`
An orthonormal basis for the computed eigenspace.

—`std::vector< Value< ST > > Evals`
The computed eigenvalue approximations.

—`std::vector< int > index`
An index into `Evecs` to enable compressed storage of eigenvectors for non-Hermitian problems.

—`int numVecs`
The number of computed eigenpair approximations.

The `Value` structure is a simple container, templated on scalar type, that has two members: the real and imaginary part of an eigenvalue. The real and imaginary parts are stored as the magnitude type of the scalar type. The `Value` structure along with the `index` vector enable the `Eigensolution` structure to store the solutions from either real or complex, Hermitian or non-Hermitian eigenvalue problems.

Anasazi solver managers are expected to place the results of their computation in the `Eigenproblem` class using an `Eigensolution`. However, a user working directly with an eigensolver (i.e., not with a solver manager) will need to recover the solution directly from the eigensolver state.

The `Eigensolver` abstract base class defines the basic interface that must be met by any eigensolver class in Anasazi. Specific eigensolver iterations are implemented as derived classes of `Eigensolver`. This class defines two types of methods: status methods and solver-specific methods. A list of these methods is given in Table II. The status methods are defined by the `Eigensolver` abstract base class and represent the information that any status test can request from any eigensolver. Each eigensolver iteration also provides low-level, solver-specific methods for accessing and setting the state of the solver. The combination of these two types of methods, along with the flexibility provided by status tests, provides the user with a large degree of control over eigensolver iterations.

`SolverManager` defines only two methods: a constructor accepting an `Eigenproblem` and a parameter list of options specific to the solver manager; and a `solve()` method, taking no arguments and returning either `Converged` or `Unconverged` (Figure 2).

The goal of the solver manager is to instantiate a subclass of `Eigensolver`, along with the necessary support objects. Another purpose of many solver managers is

Table II. A list of methods provided by any derived `Eigensolver`.

Status Methods	
Method name	*Description*
`getNumIters`	current number of iterations.
`getRitzValues`	most recent Ritz values.
`getRitzVectors`	most recent Ritz vectors.
`getRitzIndex`	Ritz index needed for indexing compressed Ritz vectors.
`getResNorms`	residual norms, with respect to the `OrthoManager`.
`getRes2Norms`	residual Euclidean norms.
`getRitzRes2Norms`	Ritz residual Euclidean norms.
`getCurSubspaceDim`	current subspace dimension.
`getMaxSubspaceDim`	maximum subspace dimension.
`getBlockSize`	block size.
Solver-specific Methods	
Method name	*Description*
`getState`	returns a specific structure with read-only pointers to the current state of the solver.
`initialize`	accepts a solver-specific structure enabling the user to initialize the solver with a particular state.

```
// create an eigenproblem
RCP< Anasazi::Eigenproblem<ST,MV,OP> > problem = ...;
// create a parameter list
ParameterList params;
params.set(...);
// create a solver manager
Anasazi::BlockDavidsonSolMgr<ST,MV,OP> solman(problem,params);
// solve the eigenvalue problem
Anasazi::ReturnType ret = solman.solve();
// get the solution from the problem
Anasazi::Eigensolution<ST,MV> sol = problem->getSolution();
```

Fig. 2. Sample code for solving an eigenvalue problem using a `SolverManager`

to manage and initiate the repeated calls to the underlying solver's `iterate()` method. For solvers that increase the dimension of trial and test subspaces (e.g., Davidson and Krylov subspace methods), the solver manager may also assume the task of restarting (so that storage costs may be fixed). This decoupling of restarting from the eigensolver is beneficial due to the numerous restarting techniques in use.

Performing an eigen-iteration requires a number of support classes. These are passed through the objects constructor, defined by `Eigensolver` to take the form listed in Figure 3.

These support classes are employed for the following purposes:

—`problem` - the eigenproblem to be solved; problem operators are defined.

—`sorter` - the sort manager selects the eigenvalues of interest.

—`printer` - the output manager dictates the verbosity level in addition to process-

```
Eigensolver(
   const RCP< Eigenproblem<ST,MV,OP> > &problem,
   const RCP< SortManager<ST,MV,OP>  > &sorter,
   const RCP< OutputManager<ST>      > &printer,
   const RCP< StatusTest<ST,MV,OP>   > &tester,
   const RCP< OrthoManager<ST,OP>    > &ortho,
   ParameterList                                &params
 );
```

Fig. 3. Constructor for eigensolver

ing output streams.

—`tester` - the status tester dictates the termination of the iteration `iterate()`.

—`ortho` - the orthogonalization manager defines the inner product in addition to performing orthogonalization for the solver.

—`params` - the parameter list specifies eigensolver-specific options.

The purpose of the `StatusTest` is to give the user or solver manager flexibility in terminating the eigensolver iterations in order to interact directly with the solver. For instance, typical reasons for terminating the iteration are:

—some convergence criterion has been satisfied;

—some portion of the subspace has reached sufficient accuracy to be deflated from the iterate or locked;

—the solver has performed a sufficient number of iterations.

The variation that exists for monitoring these and other conditions requires an abstract mechanism controlling the iteration.

The following is a list of Anasazi-provided status tests:

—`StatusTestMaxIters` - monitors the number of iterations performed by the solver; it can be used to halt the solver at some maximum number of iterations or even to require some minimum number of iterations.

—`StatusTestResNorm` - monitors the residual norms of the current iterate.

—`StatusTestOrderedResNorm` - monitors the residual norms of the current iterate, but only considers the residuals associated with the most significant eigenvalues.

—`StatusTestCombo` - a boolean combination of other status tests, creating near unlimited potential for complex status tests.

—`StatusTestOutput` - a wrapper around another status test, allowing for printing of status information on a call to `checkStatus()`.

The purpose of a sort manager is to separate the eigensolver classes from the sorting functionality required by those classes. This satisfies the flexibility principle sought by Anasazi, by giving users the opportunity to perform the sorting in whatever manner is deemed to be most appropriate. Anasazi defines an abstract class `SortManager` with two methods, one for sorting real values and one for sorting complex values. Anasazi provides a concrete implementation called `BasicSort`. This class provides basic functionality for selecting significant eigenvalues: by largest or

smallest real part, by largest or smallest imaginary part, or by largest or smallest magnitude.

Orthogonalization and orthonormalization are commonly performed computations in iterative eigensolvers. As explained in Section 1, all our current implementations are orthogonal Rayleigh-Ritz methods where an orthonormal basis representation is computed. The abstract base class `OrthoManager` defines a small number of orthogonalization-related operations, including choice of an inner product (e.g., Euclidean, induced by a symmetric positive semi-definite $\mathbf{B}$). Combined with the plethora of available methods for performing these computations, Anasazi has left as much leeway to the users as possible. To this end, Anasazi provides two concrete orthogonalization managers:

—`BasicOrthoManager` - performs orthogonalization using multiple steps of classical Gram-Schmidt [Daniel et al. 1976].

—`SVQBOrthoManager` - performs orthogonalization using the SVQB orthogonalization technique described by Stathapoulos and Wu [Stathopoulos and Wu 2002].

In order to perform the Rayleigh-Ritz analysis used by the algorithms illustrating this section, Anasazi utilizes the classes `Teuchos::BLAS` and `Teuchos::LAPACK`. The purpose of these classes is to provide templated interfaces to the dense linear algebra routines provided by the BLAS and LAPACK libraries. Therefore, even such operations as dense matrix-matrix multiplication are made independent of the scalar field defining the eigenvalue problem. Users are therefore currently limited to algorithms provided by LAPACK.

3. BENCHMARKING

The benefits of an object-oriented eigensolver framework such as Anasazi are manifold: modularization provides improved code reuse, static polymorphism via templating allows easier code maintenance and a larger audience, and dynamic polymorphism via inheritance allows flexible runtime behavior. However, none of these benefits should come at the expense of code performance. Concern over overhead has long been an inhibiting factor in the adoption of object-oriented programming paradigms in scientific computing scenarios.

We now discuss the important issue of comparing Anasazi and ARPACK on a model problem. Our interest is in assessing any overhead of Anasazi and ARPACK, C++ and FORTRAN 77 software.

We benchmarked Anasazi's `BlockKrylovSchurSolMgr` (with a block size of one) and ARPACK's `dnaupd` that compute approximations to the eigenspace of a non-symmetric matrix. Our goal was to benchmark the cost of computing $50, 100, 150$ Arnoldi vectors for a finite difference approximation to a two dimensional convection diffusion problem. Both codes use the DGKS [Daniel et al. 1976] method for maintaining the numerical orthogonality of the Arnoldi basis vectors. The Intel 9.1 C++ and FORTRAN compilers were used with compiler switches "-O2 -xP" on an Intel Pentium D, 3GHz, 1MB L2 cache, 2GB main, Linux/FC5 PC.

rewrite this operator

The operator application in Anasazi records approximately twice as much time as the ARPACK implementation. This is because the Anasazi code used an Epetra

Table III. Comparing the overhead of Anasazi with ARPACK; "—" denotes a measurement below the clock resolution.

	Computing 50 Arnoldi vectors			
	Matrix-vector time [s]		Total runtime [s]	
Matrix size	ARPACK	Anasazi	ARPACK	Anasazi
10000	—	0.01	0.14	0.15
62500	0.04	0.09	1.20	1.17
250000	0.15	0.32	4.98	4.79
1000000	0.66	1.23	19.2	18.8
	Computing 100 Arnoldi vectors			
	Matrix-vector time [s]		Total runtime [s]	
Matrix size	ARPACK	Anasazi	ARPACK	Anasazi
10000	0.03	0.02	0.53	0.55
62500	0.03	0.17	4.37	4.29
250000	0.34	0.64	17.8	17.5
1000000	1.27	2.40	68.4	67.1
	Computing 150 Arnoldi vectors			
	Matrix-vector time [s]		Total runtime [s]	
Matrix size	ARPACK	Anasazi	ARPACK	Anasazi
10000	0.03	0.04	1.15	1.22
62500	0.14	0.26	9.53	9.39
250000	0.50	0.96	38.1	38.0
1000000	1.97	3.56	149	146

sparse matrix representation, while the ARPACK implementation applies the block tridiagonal matrix via a stencil. Note that the operator application comprised only a small portion of the clock time in these tests. The performance of the Anasazi library in computing the Arnoldi vectors is similar to that of ARPACK. Our conclusion is that a well-designed library in C++ is as efficient as a FORTRAN 77 library.

4. CONCLUSION

reinforce their clarity

issues yet to be handled: anasazi provides only three eigensolvers, it also provides a framework capable of implementing multiple eigensolvers. for example, eigen-iterations requiring an iteration have been implemented using anasazi, such as RTR, TRACEMIN and Jacobi-Davidson.

5. ACKNOWLEDGMENTS

We thank Roscoe Bartlett, Mike Heroux, Roger Pawlowski, Eric Phipps, and Andy Salinger for many helpful discussions.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1999. *LAPACK Users' Guide*, third ed. SIAM, Philadelphia.

Arbenz, P., Hetmaniuk, U., Lehoucq, R., and Tuminaro, R. 2005. A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods. *Int. J. Numer. Meth. Engng. 64*, 204–236.

Baker, C. G., Hetmaniuk, U., Lehoucq, R. B., and Thornquist, H. K. Anasazi: Block eigensolver package. See http://software.sandia.gov/trilinos/packages/anasazi/index.html.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2001. PETSc Web page. http://www.mcs.anl.gov/petsc.

Bartlett, R., Boggs, P., Coffey, T., Heroux, M., Hoekstra, R., Howle, V., Long, K., Pawlowski, R., Phipps, E., Spotz, B., Thornquist, H., and Williams, A. Thyra: Interfaces for abstract numerical algorithms. See http://software.sandia.gov/trilinos/packages/thyra/.

Blackford, L. S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., and Whaley, R. C. 2002. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software 28,* 2 (June), 135–151.

Daniel, J., Gragg, W. B., Kaufman, L., and Stewart, G. W. 1976. Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. *Mathematics of Computation 30*, 772–795.

Davidson, E. R. 1975. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics 17*, 87–94.

Detlefs, D. 1992. Garbage collection and run-time typing as a C++ library. In *Proceedings: USENIX C++ Technical Conference, August 10–13, 1992, Portland, OR*, USENIX, Ed. USENIX, pub-USENIX:adr, 37–56.

Hernández, V., Román, J., Tomás, A., and Vidal, V. 2005. A survey of software for sparse eigenvalue problems. Tech. Rep. SLEPc Technical Report STR-6, Universidad Politecnica de Valencia. See http://www.grycap.upv.es/slepc.

Hernández, V., Román, J., Tomás, A., and Vidal, V. 2006. SLEPc users manual: Scalable library for eigenvalue problem computations. Tech. Rep. DISC-II/24/02, Universidad Politecnica de Valencia. See http://www.grycap.upv.es/slepc.

Heroux, M., Hoekstra, R., Sexton, P., Spotz, B., Willenbring, J., and Williams, A. Epetra: Linear algebra services package. See http://software.sandia.gov/trilinos/packages/epetra/.

Heroux, M. A., Baker, C. G., Bartlett, R. A., Kampschoff, K., Long, K. R., Sexton, P. M., and Thornquist, H. K. Teuchos: The trilinos tools library. See http://software.sandia.gov/trilinos/packages/teuchos/.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. 2005. An overview of the Trilinos project. *ACM Trans. Mathematical Software 31,* 3 (Sept.), 397–423.

Hetmaniuk, U. and Lehoucq, R. 2006. Basis selection in LOBPCG. *J. Comput. Phys. 218*, 324–332.

Knyazev, A. V. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Scientific Computing 23*, 517–541.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5,* 3 (Sept.), 308–323.

Lehoucq, R. B., Sorensen, D. C., and Yang, C. 1998. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia, PA.

Meyers, N. C. 1995. Traits: A new and useful template technique. *C++ Report 7*, 32–35.

Meyers, S. 1996. *More Effective C++.* Addison-Wesley Professional Computing Series. Addison-Wesley.

Musser, D. R. and Stepanov, A. A. 1994. Algorithm-oriented generic libraries. *Software Practice and Experience 24,* 7 (July), 623–642.

Saad, Y. 1992. *Numerical Methods for Large Eigenvalue Problems.* Halsted Press.

Sala, M., Heroux, M. A., and Day, D. M. 2004. Trilinos Tutorial. Tech. Rep. SAND2004-2189, Sandia National Laboratories.

Sleijpen, G. L. G. and van der Vorst, H. A. 1996. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Analysis and Applications 17,* 2, 401–425.

Sorensen, D. 2002. *Numerical Methods for large eigenvalue problems.* Acta Numerica, vol. 11. Cambridge University Press, 519–584.

Sorensen, D. C. 1992. Implicit application of polynomial filters in a k-step Arnoldi method. *SIAM J. Matrix Analysis and Applications 13*, 357–385.

Stathopoulos, A. and McCombs, J. R. 2006. PRIMME home page. See http://www.cs.wm.edu/~andreas/software/.

Stathopoulos, A. and Wu, K. 2002. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput. 23*, 2165–2182.

Stewart, G. W. 2001a. A Krylov-Schur algorithm for large eigenproblems. *SIAM J. Matrix Analysis and Applications 23*, 601–614.

Stewart, G. W. 2001b. *Matrix Systems: Eigensystems.* Vol. II. SIAM.

van der Vorst, H. A. 2002. Computational methods for large eigenvalue problems. P. Ciarlet and J. Lions, Eds. Handbook of Numerical Analysis, vol. VIII. North-Holland (Elsevier), Amsterdam, 3—179.

Vandevoorde, D. and Josuttis, N. M. 2002. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc.