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 exploits 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 is 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, Object-oriented programming, Large-scale scientific computing

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. Anasazi has been employed in a number of large-scale scientific codes, for example, performing modal analysis in the Salinas structural dynamics code [Bhardwaj et al. 2002] and stability analysis in LOCA [Salinger et al. 2005].

The purpose of this paper is to document and introduce the Anasazi eigensolver framework to prospective users. These users include practitioners and researchers in need of efficient, large-scale eigensolvers in a modern programming environment. This also includes experts who could exploit the framework provided by Anasazi

---

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.

as a platform for research and development of new methods for solving eigenvalue problems. This paper is intended to outline the benefits of Anasazi's design, as well as the motivation for those decisions.

An inspiration for Anasazi is the ARPACK [Lehoucq et al. 1998] FORTRAN 77 software library. ARPACK implements a single 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 [Morgan and Scott 1986] 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 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. This flexibility has enabled the use of ARPACK in a wide range of applications, and it is this flexibility that Anasazi was designed to emulate.

The design goals of Anasazi were multiple. First, implementation of eigensolvers should be independent from the choice of underlying linear algebra primitives. The benefit is that the resulting eigensolvers are able to exploit existing software, such as the wide variety of linear algebra implementations, solvers, and preconditioners present in Trilinos. This flexibility also eases the incorporation of Anasazi into larger software libraries and application codes.

Another goal of Anasazi is that abstract interfaces should be utilized wherever feasible for algorithmic components, so that the implementation of those components may be separated from the implementation of the eigensolvers. Many benefits result from such a decision. This decoupling facilitates code reuse across and outside of the Anasazi eigensolvers. This underlies Anasazi's existence as a framework for developing novel eigensolver capability. This decoupling also increases algorithmic flexibility. As a result, constituent mechanisms (e.g., orthogonalization routines) can be chosen at runtime. This cements Anasazi's usefulness as a framework for research, not only into eigensolvers, but also the components necessary to implement an eigensolver.

The Anasazi framework accomplishes these design goals by exploiting more recent software development paradigms than available to related eigensolver software. Both generic and object-oriented programming, via static and dynamic polymorphism [Vandevoorde and Josuttis 2002, Chapter 14], are employed to this effect. Static polymorphism, via templating of the linear algebra primitives, 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 constituent mechanisms such as orthogonalization and stopping conditions.

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

The rest of this paper is organized as follows. Section 1 reviews related software for solving large-scale eigenvalue problems. Section 2 briefly discusses algorithms that are implemented in Anasazi, in order to explore the types of operations necessary for an eigensolver framework. Section 3 reviews the Anasazi framework, discusses some of the design decisions, and illustrates the benefits of these decisions. Lastly, Section 4 provides some timings comparing ARPACK and Anasazi to demonstrate that object-oriented overhead has negligible impact on the performance of this modern software framework.

1. RELATED EIGENSOLVER SOFTWARE

There exist a number of related software efforts for solving large-scale eigenvalue problems (the reader is referred to [Hernández et al. 2005] for a more complete survey). We discuss here the ARPACK, IETL, PRIMME and SLEPc software efforts:

—The Arnoldi Package (ARPACK) is a FORTRAN 77 software for the solution of Hermitian or non-Hermitian, standard or generalized, eigenvalue problems. ARPACK implements a single solver, the Implicitly Restarted Arnoldi Method.

—The Iterative Eigensolver Template Library (IETL) is a C++ library which uses C++ templates to provide a collection of generic eigensolvers. It currently provides four solvers for standard Hermitian eigenvalue problems.

—The Preconditioned Iterative Multi-Method Eigensolver (PRIMME) [Stathopoulos and McCombs 2006] is a C library for computing a number of eigenvalues and the 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 is another C library 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.

ARPACK utilizes a reverse-communication interface to access the linear operators defining the eigenvalue problem. As a result, the eigensolver is implemented in a partially generic manner, independent of the underlying linear operator, allowing use of the software for many user-defined eigenproblems. A more recent effort (PARPACK) extends ARPACK to provides a parallel computing capability. These reasons, along with ARPACK's maturity, make it the *de facto* eigensolver in many scientific computing communities. Unfortunately, the reverse communication interface makes maintenance of ARPACK a cumbersome task. Furthermore, while this interface does provide a generic interface for the linear operators, the storage of vector data is fixed. In addition to limiting any flexibility in data representation, this fixed interface results in software which is susceptible to any (albeit unlikely) design changes in ARPACK.

The IETL software library, like ARPACK, strives to ease use of the software in diverse applications through a generic interface to operators and vectors. Implemented in C++, IETL achieves this through the language's template feature. By utilizing generic interfaces for scalar types, vectors and linear operators, IETL solvers can be applied to any data structures that adhere to the prescribed object model. This allows a single implementation of an eigensolver in IETL to be exploited across many different programming environments, e.g., real or complex arithmetic, parallel or serial architecture.

PRIMME provides a single, flexible solver capable of emulating a variety of Hermitian eigensolvers. Packaged parameter choices are provided to emulate a number of 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 the implementations provided by the developers of PRIMME. As a result, the diversity of behavior in the solver is limited to those options anticipated by the developers. Furthermore, PRIMME provides implementations only over double precision real and complex fields. Each additional scalar field (such as `float` or extended precision) requires a separate implementation, due to the lack of generic programming ability in the C programming language.

SLEPc extends the PETSc toolkit to provide a library of solvers for standard or generalized, Hermitian or non-Hermitian eigenproblems. SLEPc provides wrappers for several eigensolver packages, most notably ARPACK and PRIMME, as well as native implementations of eigensolvers like Krylov-Schur, Arnoldi, and Lanczos. PETSc uses C language features such as `typedef`s and function pointers to support some generic programming and object-oriented programming techniques, the goal being interoperability with other software packages. Interoperability with PETSc gives SLEPc users access to a large library of linear and nonlinear solvers, preconditioners and matrix formats, though SLEPc's reliance on PETSc requires that users employ PETSc for vector storage. Similar to PRIMME, SLEPc can be compiled with support for double precision real or complex arithmetic. However, only one version of the library can be used at a time. Furthermore, mechanisms such as orthogonalization are hard-coded, allowing only parametrized control over their behavior.

The Anasazi framework was designed to include features from other eigensolver packages that are conducive to algorithm development, while avoiding some of the drawbacks mentioned above. The most important features that have been incorporated into its design are extensibility and interoperability. The extensibility of the Anasazi framework is demonstrated through the infrastructure's support for a significant class of large-scale eigenvalue algorithms. Extensions can be made through the addition of, or modification to, existing algorithms and auxiliary functionality such as orthogonalization, desired eigenvalue selection, and stopping conditions. This is encouraged by promoting code modularization and multiple levels of access to solvers and their data. For example, the question of whether to implement a new solver in PRIMME is a question of whether the PRIMME solver is sufficiently flexible to describe the desired iteration without excessive modification. For libraries such as SLEPc and IETL, this decision is made based largely on the existing functionality in the library that can be exploited in a new implementation. By decoupling eigensolvers from constituent mechanisms, Anasazi allows such functionality to be exploited by new solvers. This is the foundation of extensibility in Anasazi.

Interoperability in the Anasazi framework 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. The Anasazi framework 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, for example, an Anasazi eigensolver using single-precision complex arithmetic can be used alongside another Anasazi eigensolver using an extended precision scalar type; both would be instantiated from the same source code.

As a result of these design features, the Anasazi eigensolver framework is significantly more flexible than previous efforts, easing its inclusion in diverse application environments in addition to providing an arena for research into eigensolvers and their constituent mechanisms.

2. ALGORITHMS IN ANASAZI

The Anasazi framework provides tools that are useful for solving a wide variety of eigenvalue problems. The solvers currently released within Anasazi compute a partial eigen-decomposition for the generalized eigenvalue problem

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

We assume that the matrices $\mathbf{A}$ and $\mathbf{B}$ are large, possibly sparse, and that only their application to a block of vectors is required. For instance, there is no assumption that $\mathbf{A}$ and $\mathbf{B}$ are stored in some sparse matrix format. 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.

We now discuss how the block Davidson eigensolver described in [Arbenz et al. 2005] is implemented within Anasazi to solve

$$\mathbf{A}\mathbf{x} = \mathbf{M}\mathbf{x}\lambda, \qquad \mathbf{A}, \mathbf{M} \in \mathbb{C}^{n\times n} \tag{2}$$

where $\mathbf{A}$ is Hermitian, and $\mathbf{M}$ is Hermitian positive definite. Algorithm 1 lists the salient steps of the block Davidson eigensolver.

The linear operators $\mathbf{A}$ and $\mathbf{M}$ define the eigenproblem to be solved. The linear operator $\mathbf{N}$ is a preconditioner for the problem, and its application to a block of vectors is required. Example preconditioners $\mathbf{N}$ include the inverse of the diagonal of $\mathbf{A}$ (Jacobi preconditioner), an algebraic multigrid preconditioner for $\mathbf{A}$, or

Algorithm 1 Block Davidson Algorithm

Require: Set an initial guess $\mathbf{V}$ and $\mathbf{H} = []$.
1: **for** $iter = 1$ to $iter_{max}$ **do**
2:   **repeat**
3:     Compute $\mathbf{M}$-orthonormal basis $\mathbf{V}$ for $[\mathbf{V},\ \mathbf{H}]$
4:     Project $\mathbf{A}$ onto $\mathbf{V}$: $\hat{\mathbf{A}} = \mathbf{V}^H\mathbf{A}\mathbf{V}$
5:     Compute selected eigenpairs $(\mathbf{Q}, \mathbf{\Gamma})$ of $\hat{\mathbf{A}}$: $\hat{\mathbf{A}}\mathbf{Q} = \mathbf{Q}\mathbf{\Gamma}$
6:     Compute Ritz vectors: $\mathbf{X} = \mathbf{V}\mathbf{Q}$
7:     Compute residuals: $\mathbf{R} = \mathbf{A}\mathbf{X} - \mathbf{M}\mathbf{X}\mathbf{\Gamma}$
8:     Precondition the residuals: $\mathbf{H} = \mathbf{N}\mathbf{R}$
9:     Check convergence and possibly terminate iteration
10:   **until** the matrix $\mathbf{V}$ is not expandable
11:   Restart $\mathbf{V}$
12: **end for**

a preconditioned iteration that approximates the solution of a linear set of equations with $\mathbf{A}$, e.g., the preconditioned conjugate gradient algorithm. We remark that with an appropriate choice for $\mathbf{N}$, Algorithm 1 is easily modified in a Jacobi-Davidson [Sleijpen and van der Vorst 1996] algorithm. We emphasize that the choice and implementation of $\mathbf{N}$ is left to the user.

The dense rectangular matrices $\mathbf{V}$, $\mathbf{X}$, and $\mathbf{R}$ are stored as a collection of vectors, which we call a *multivector*. The column-vectors for matrix $\mathbf{V}$ form the basis for the Rayleigh-Ritz approximation conducted at Step 5-6. We make the specific choice that these vectors are orthonormal with respect to the inner product induced by the Hermitian positive-definite matrix $\mathbf{M}$, but this is not a requirement.

The block Davidson eigensolver as described above is useful for examining some of the functionality provided by Anasazi. Algorithm 1 highlights three levels of functionality. The first level is given by Steps 1-12 that constitute the eigensolver strategy to solve problem (2). The second level is given by Steps 2-10 that form the eigen-iteration. The third level consists of computational steps that can be implemented in a variety of manners, so encouraging modularization. Step 3 requires a orthonormalization method. The decisions involved in Step 5 require a determination of the eigenvalues and invariant subspace of interest, as well as a definition of accuracy. To check convergence in Step 9, several criteria are possible. For instance, a norm induced by a matrix other than $\mathbf{M}$ may be employed. The restarting of this eigensolver (Step 11) can be performed in a variety of ways and therefore need not be tightly coupled to the eigensolver. Each of these mechanisms provide opportunity for decoupling functionality that need not be implemented in a specific manner.

Anasazi also implements LOBPCG as described in [Hetmaniuk and Lehoucq 2006] to solve (2). For the more general eigenvalue problem given by (1), Anasazi implements a block Krylov-Schur [Stewart 2001a] method. This method allows the use of a matrix operator $\mathbf{OP}$ for implementing a spectral transformation (e.g. shift-invert). We remark that a spectral transformation may be implemented in a number of ways, e.g., via a preconditioned iterative method; this decision resides with the user.

Given these observations, the components that are important to an eigensolver include:

—multivector operations: creation, projection, right multiplication;

—operator-multivector applications: $\mathbf{A\,X}$, $\mathbf{M\,X}$, $\mathbf{OP\,X}$, $\mathbf{N\,R}$;

—solution of typically much smaller dense eigenproblems;

—a sorting method (for the desired portion of the eigenvalues);

—an orthogonalization method;

—a testing capability to terminate the iteration.

A full list of our primitives for operators and multivectors will be presented in Section 3.1. Section 3.2 will discuss our treatment of the other components.

This discussion illustrates that many distinct parts make up a large-scale eigensolver code. 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.

3. ANASAZI SOFTWARE FRAMEWORK

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

We remark that Anasazi is largely independent of other Trilinos packages and third-party libraries. However, Anasazi does rely on the Trilinos Teuchos package [Heroux et al.] to provide tools such as: `RCP`, a reference-counting smart pointer [Detlefs 1992; Bartlett 2004]; `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] C++ wrappers. The only third-party libraries that Anasazi requires are the BLAS and LAPACK libraries, which are essential in performing the dense arithmetic for Rayleigh-Ritz methods.

3.1 The Anasazi Operator/Multivector Interface

Anasazi utilizes traits classes [Meyers 1995; Veldhuizen 1996] to define interfaces for the scalar field, multivectors, and matrix operators. This allows generic programming techniques to be used when developing numerical algorithms in the Anasazi framework. Anasazi's eigensolver framework (Section 3.2) is comprised of abstract numerical interfaces that are all implemented using templates and the functionality of the template arguments is provided through their corresponding trait classes. 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 over the given scalar field, providing a data structure that denotes a collection of vectors; and

—an operator type over the given scalar field, providing linear operators used to define eigenproblems and preconditioners.

We note 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 (a multivector). One advantage of using a multivector data structure is to improve the ratio of floating-point operations to memory references and so better exploit a memory hierarchy.

Templating an eigensolver on operator, multivector, and scalar types makes software reuse easier. Consider in contrast that ARPACK implements the subroutines `SNAUPD`, `DNAUPD`, `CNAUPD`, and `ZNAUPD` for solving non-Hermitian eigenproblems. Separate subroutines are required for each of the 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. By templating abstract numerical interfaces on operator, multivector, and scalar types, it is only necessary to maintain a single code using the Anasazi framework.

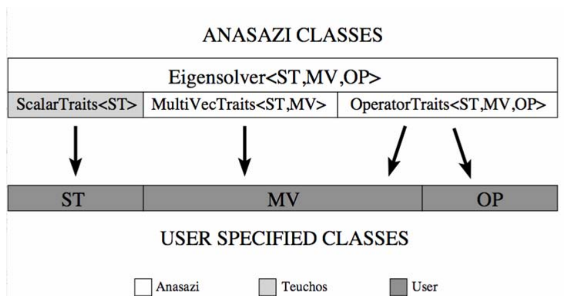

Fig. 1. An eigensolver templated on scalar (ST), multivector (MV), and operator (OP) type.

Another aspect of software reuse that templating alleviates is through the separation of the eigensolver algorithm from the linear algebra data structures. This separation, as shown in Figure 3.1, allows a user of the Anasazi framework to leverage any existing linear algebra software investment. All that is required is the template instantiation of the trait classes, `MultiVecTraits` and `OperatorTraits`, for the user-defined multivector and operator, respectively. The `ScalarTraits` class and respective template instantiations for different scalar types are provided by the Trilinos Teuchos package [Heroux et al.]. Another friendly aspect of employing templates and traits mechanisms is that the Anasazi eigensolver, eigenproblem, and eigensolution are all defined by the specified scalar, multivector, and operator type at compile time. This approach, as opposed to using abstract interfaces and dynamic polymorphism, avoids any dynamic casting of the multivectors and operators in the user's interaction with the Anasazi framework.

The `MultiVecTraits` and `OperatorTraits` classes specify the operations that the multivector and operator type must support in order for them to be used by Anasazi. Through the observations made in Section 2, it is clear that the `OperatorTraits` class only needs to provide one method, described in Table I, that applies an operator to a multivector. This interface defines the only interaction required from an operator, even though the underlying operator may be a matrix, spectral transformation, or preconditioner.

Table I. The method provided by the `OperatorTraits` interface.

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

Table II. The methods provided by the `MultiVecTraits` interface.

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, D,beta,Y)`	Applies a dense matrix D to multivector X and accumulates the result into multivector Y: $Y \leftarrow \alpha XD + \beta Y$
`MvAddMv(alpha,X,beta,Y)`	Performs multivector AXPBY: $Y \leftarrow \alpha X + \beta Y$
`MvTransMv(alpha,X,Y,D)`	Computes the dense matrix $D \leftarrow \alpha X^H Y$
`MvDot(X,Y,d)`	Computes the corresponding dot products: $d[i] \leftarrow \bar{x}_i y_i$
`MvScale(X,d)`	Scales the i-th column of a multivector X by $d[i]$
`MvNorm(X,d)`	Computes the 2-norm of each vector of X: $d[i] \leftarrow \|x_i\|_2$
`SetBlock(X,Y,index)`	Copies the vectors in X to a subset of vectors in Y
`MvInit(X,alpha)`	Replaces each entry in the multivector X with a scalar α
`MvRandom(X)`	Replaces the entries in the multivector X by random scalars
`MvPrint(X)`	Print the multivector X

The methods defined by the `MultiVecTraits` class, listed in Table II, are the creational and arithmetic methods necessitated by the observations in Section 2. The creational methods generate empty or populated multivectors from a previously created multivector. The populated multivectors can be a deep copy, where the object contains the storage for the multivector entries, or a shallow copy, where the object has a view of another multivector's storage. A shallow copy is useful when only a subset of the columns of a multivector is required for computation, which is a situation that commonly occurs during the generation of a subspace. All the creational methods return a reference-counted pointer [Detlefs 1992; Bartlett 2004] to the new multivector (`RCP<MV>`).

The arithmetic methods defined by the `MultiVecTraits` are essential to the computations required by the Rayleigh-Ritz method and the general eigen-iteration. The `MvTimesMatAddMv` and `MvAddMv` methods are necessary for updating the approximate eigenpairs and their residuals in Steps 6–7 of the Algorithm 1. The `MvDot` and `MvTransMv` methods are required by the orthogonalization procedure utilized in Step 3 of the eigen-iteration. The `MvScale` and `MvNorm` methods are necessary, at the very least, for the computation of approximate eigenpairs and for some termination criteria of the eigen-iteration. Deflation and locking of converged eigenvectors necessitates the `SetBlock` method in many cases. Initialization of the bases for the eigen-iteration requires methods such as `MvRandom` and `MvInit`. The ability to perform error checking and debugging in Anasazi is supported by methods that give dimensional attributes (`GetVecLength`, `GetNumberVecs`) and allow the users to print out a multivector (`MvPrint`).

Specialization of the `MultiVecTraits` and `OperatorTraits` classes on given template arguments is compulsory for their usage in the eigensolver framework. Anasazi provides the following specializations of these trait 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. This gives Anasazi the ability to interact with Trilinos packages that support the `Epetra_Operator` interface, e.g., the Amesos direct sparse solver package, the AztecOO and Belos iterative linear solver packages, the Ifpack package of algebraic preconditioners, the ML package for multigrid preconditioners, and NOX/LOCA package of nonlinear solvers.

—`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.

For scalar, multivector and operator types not covered by the provided specializations, alternative specializations of `MultiVecTraits` and `OperatorTraits` must be created. One benefit of the traits mechanism is that it does not require that the data types are C++ classes. Furthermore, the traits mechanism does not require modification to existing data types; it serves only as a translator between the data type's native functionality and the functionality required by Anasazi.

3.2 The Anasazi Eigensolver Framework

In this section we discuss how an eigensolver is implemented in Anasazi's framework. We demonstrate that Anasazi is a framework of algorithmic components, where decoupled operations simplify component verification, encourage code reuse, and maximize flexibility in implementation. This modularized approach utilizes a *solver manager* to define a strategy using these algorithmic components. The high-level class collaboration graph for Anasazi's `SolverManager` class in Figure 3.2 lists all the algorithmic components offered by the Anasazi framework for implementing an eigensolver.

The first component that is essential to the `SolverManager` is the `Eigenproblem` class. `Eigenproblem` is an abstract class that is a container for the components and solution of an eigenvalue problem. By requiring eigenvalue problems to derive from `Eigenproblem`, Anasazi defines a minimum interface that can be expected of all eigenvalue problems by the classes that will work with these problems. The methods provided by this interface, shown in Table III, are generic enough to define an eigenvalue problem that is standard or generalized, Hermitian or non-Hermitian. Furthermore, this interface allows the definition of a preconditioner, for preconditioned eigensolvers, as well as the definition of a spectral transformation, for Arnoldi-based eigensolvers.

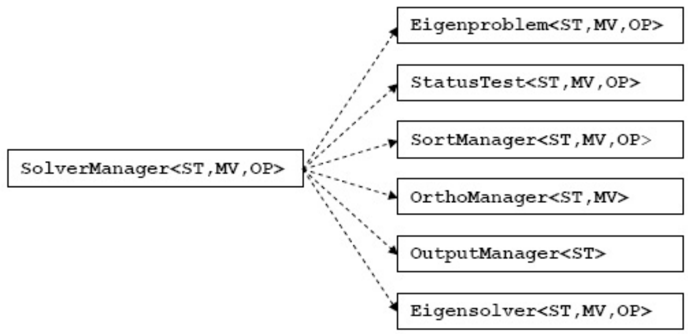

Fig. 2. `SolverManager` class collaboration graph.

Table III. A list of methods provided by any derived `Eigenproblem`.

Eigenproblem<ST,MV,OP>		
Method name		*Description*
`setOperator()`	`getOperator()`	Access the operator for which eigenvalues will be computed
`setA()`	`getA()`	Access the operator $\mathbf{A}$ of the eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{Bx}$
`setB()`	`getB()`	Access the operator $\mathbf{B}$ of the eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{Bx}$
`setPrec()`	`getPrec()`	Access the preconditioner for this eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{Bx}$
`setInitVec()`	`getInitVec()`	Access the initial guess
`setAuxVecs()`	`getAuxVecs()`	Access the auxiliary vectors
`setNEV()`	`getNEV()`	Access the number of eigenvalues (NEV) that are requested
`setHermitian()`	`isHermitian()`	Access the symmetry of the eigenproblem
`setProblem()`	`isProblemSet()`	Access whether the eigenproblem is fully defined
`setSolution()`	`getSolution()`	Access the solution to the eigenproblem

From a user's perspective, the most important part of the interface may be the methods for storing and retrieving the results of the eigenvalue computation:

```
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 scheme enabling compressed storage of eigenvectors for non-Hermitian problems.

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

The `Eigensolution::index` vector has `numVecs` integer entries that take one of three values: $\{0, +1, -1\}$. These values allow the eigenvectors to be retrieved as follows:

—`index[i]==0`: The i-th eigenvector is stored uncompressed in column i of `Evecs`.

—`index[i]==+1`: The i-th eigenvector is stored compressed, with the real component in column i of `Evecs` and the *positive* complex component stored in column $i+1$ of `Evecs`.

—`index[i]==-1`: The i-th eigenvector is stored compressed, with the real component in column $i-1$ of `Evecs` and the *negative* complex component stored in column i of `Evecs`.

The compressed storage scheme is necessary to efficiently store the (potentially) complex eigenvectors of a non-symmetric eigenproblem when the multivectors are defined over a real field. This scheme enables Anasazi to use `numVecs` vectors to store `numVecs` eigenvectors, even when complex conjugate pairs are present. All other eigenproblems (real symmetric, complex Hermitian or non-Hermitian) will return an index vector composed entirely of zeroes, as compression of complex eigenvectors is not an issue.

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. Implementations of the `SolverManager` class are expected to place the results of their computation in the `Eigenproblem` class using an `Eigensolution`.

The second component that is essential to a `SolverManager` is the `Eigensolver` class. The `Eigensolver` abstract base class defines the basic interface that must be met by any eigen-iteration class in Anasazi. A derived class will define two types of methods: status methods and solver-specific methods. A list of these methods is given in Table IV. The status methods are defined by the `Eigensolver` abstract base class and represent the information about the iteration status that can be requested from any eigensolver. Each eigensolver iteration also provides low-level, solver-specific methods for accessing and setting the state of the solver. An eigensolver's state is stored in a solver-specific structure and is expected to fully describe the current state of the solver or the state the solver needs to be initialized to. A simple example of a state structure can be seen in Figure 3.

The eigensolver iterations implemented using the `Eigensolver` class are generic iteration kernels that do not have the intelligence to determine when to stop the

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

Eigensolver<ST,MV,OP>	
Status Methods	
Method name	*Description*
`getNumIters()`	current iteration
`getRitzValues()`	current Ritz values
`getRitzVectors()`	current Ritz vectors
`getRitzIndex()`	Ritz index needed for indexing compressed Ritz vectors
`getResNorms()`	residual norms, with respect to the `OrthoManager` inner product
`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.
`iterate()`	performs eigen-iteration until the status test indicates the need to stop or an error occurs.

```
template <class ST, class MV>
struct SomeEigensolverState {
  /* The current dimension of the subspace.                           *
   * NOTE: This should be equal to SomeEigensolver::getCurSubspaceDim(). */
  int curDim;
  /* The current subspace. */
  RCP<const MV> V;
  /* The current Rayleigh-Ritz projection */
  RCP<const Teuchos::SerialDenseMatrix<int,ST> > H;
};
```

Fig. 3. Example of an `Eigensolver` state structure.

iteration, what the eigenvalues of interest are, which output to send and to where, or how to orthogonalize the basis for a subspace. The intelligence to perform these four tasks is, instead, provided by the `StatusTest`, `SortManager`, `OutputManager`, and `OrthoManager` objects, which are passed into the constructor of an `Eigensolver` (Figure 4). This allows each of these four tasks to be modified without affecting the basic eigensolver iteration. When combined with the status and state-specific `Eigensolver` methods, this provides the user with a large degree of control over eigensolver iterations.

The abstract `StatusTest` class is used to provide the interface for stopping conditions for an eigen-iteration. There are numerous conditions under which an eigen-iteration should be stopped, with the most common conditions being the number

```
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. 4. Basic constructor for an `Eigensolver`

of iterations, convergence criterion, and the deflation strategy for converged eigenpairs. Often the decision to stop an eigensolver iteration is based on a hierarchy of problem-dependent, logically connected stopping conditions. This, possibly complex, reasoning should not be encoded in the `Eigensolver` class, which instead queries the `StatusTest` during its class method `iterate()` to determine whether or not to continue iterating (Figure 5). The `StatusTest` class provides a method,

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

Fig. 5. Example of communication between status test and eigensolver

`checkStatus()`, which queries the methods provided by `Eigensolver` and determines whether the solver meets the criteria defined by the 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 the iteration.

A `StatusTest` is a desirable feature in the Anasazi software framework because it provides the eigensolver user and developer with a flexible interface for interrogating the eigen-iteration. Besides the basic usage, this interface makes it possible to, for example, select stopping conditions at runtime or put application-specific hooks in the eigensolver for debugging and checkpointing. This flexible approach to selecting and developing stopping criteria for an eigensolver is not available in PRIMME or SLEPc. Since an ARPACK user provides the memory for computations and ARPACK is constantly returning control via the reverse communication mechanism, the user has some ability to examine the current state and modify it to force certain behavior. However, this is an advanced use case which requires intimacy with the formatting of the ARPACK control data.

The purpose of the `SortManager` class is to separate the `Eigensolver` from the sorting functionality, giving users the opportunity to choose the eigenvalues of interest 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, shown in Figure 6. The `SortManager` is also expected to provide the permutation vector if the `Eigensolver` passes a non-null pointer for `perm` to the `sort` method. This is necessary, because many eigen-iterations must sort their approximate eigenvectors, as well as their eigenvalues.

```
// Sort n real values stored in evals and return permutation, if required, in perm.
void sort(Eigensolver<ST,MV,OP>* solver,
          const int n,
          std::vector<typename Teuchos::ScalarTraits<ST>::magnitudeType> &evals,
          std::vector<int> *perm)
// Sort n complex values, whose real and imaginary part are stored in r_evals
//   and i_evals, respectively. Return permutation, if required, in perm.
void sort(Eigensolver<ST,MV,OP>* solver,
          const int n,
          std::vector<typename Teuchos::ScalarTraits<ST>::magnitudeType> &r_evals,
          std::vector<typename Teuchos::ScalarTraits<ST>::magnitudeType> &i_evals,
          std::vector<int> *perm)
```

Fig. 6. A list of methods provided by any derived `SortManager`.

Since orthogonalization and orthonormalization are commonly performed computations in iterative eigensolvers and can be implemented in a variety of ways, the `OrthoManager` class separates the `Eigensolver` from this functionality. The `OrthoManager` defines a small number of orthogonalization-related operations, including a choice of an inner product, which are listed in Table V. The `OrthoManager` interface has also been extended, through inheritance, to support orthogonalization

Table V. A list of methods provided by any derived `OrthoManager`.

OrthoManager<ST,MV>	
Method name	*Description*
`innerProd(X,Y,Z)`	Provides the inner product
`norm(X,normvec)`	Provides the norm induced by the inner product
`project(X,Q,C)`	Projects the multivector X onto the subspace orthogonal to the multivectors Q, optionally returning the coefficients of X with respect to the Q.
`normalize(X,B)`	Computes an orthonormal basis for the multivector X, optionally returning the coefficients of X with respect to the computed basis.
`projectAndNormalize(X,Q,C,B)`	Projects the multivector X onto the subspace orthogonal to the multivectors Q and computes an orthonormal basis (orthogonal to the Q) for the resultant, optionally returning the coefficients of X with respect to the Q and the computed basis.

and orthonormalization using matrix-based inner products in the `MatOrthoManager` class. This extended interface allows the eigen-iteration to pass in pre-computed matrix-vector products that can be used in the orthogonalization and orthonormalization process, thus making the computation more efficient.

The `Eigensolver` class combined with the utilities provided by the `StatusTest`, `SortManager`, and `OrthoManager` classes provides a powerful, flexible way to design an eigen-iteration. However, directly interfacing with the `Eigensolver` class can be overwhelming, since it requires the user to construct a number of support classes and 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 iteration 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. The `SolverManager` constructor accepts only two arguments: an `Eigenproblem` specifying the eigenvalue problem to be solved and a `ParameterList` of options specific to this solver manager. The solver manager instantiates an `Eigensolver` implementation, along

```
SolverManager(
   const RCP< Eigenproblem<ST,MV,OP> > &problem,
   ParameterList                       &params
 );
```

Fig. 7. Basic constructor for a `SolverManager`

with the status tests and other support classes needed by the eigensolver iteration, as specified by the parameter list. To solve the eigenvalue problem, the user simply calls the `solve()` method of the `SolverManager`, which returns either `Converged` or `Unconverged`, and retrieves the computed `Eigensolution` from the `Eigenproblem` (Figure 8).

```
// create an eigenproblem
RCP< Anasazi::Eigenproblem<ST,MV,OP> > problem = ...;
// create a parameter list
ParameterList params;
params.set(...);
// create a solver manager
Anasazi::SolverManager<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. 8. Sample code for solving an eigenvalue problem using a `SolverManager`

The simplicity of the `SolverManager` interface often conceals a complex eigensolver strategy. The purpose of many solver managers is to manage and initiate the repeated calls to the underlying `Eigensolver::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.

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

—Use the existing solver managers, which we will discuss in the next section. 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 iteration. The user can extend the functionality provided by the eigen-iteration, specifying custom configurations for status tests, orthogonalization, restarting, locking, etc.

—Implement a new eigensolver iteration, thus taking advantage of Anasazi's extensibility. The user can write an eigensolver iteration that is not provided by Anasazi. The user still has the benefit of the available support classes and the knowledge that this effort can be easily employed by anyone already familiar with Anasazi.

In the next section we will discuss the current implementations of eigensolver iterations and managers, as well as utility classes, provided by the Anasazi eigensolver framework.

3.3 Anasazi Class Implementations

Anasazi is an eigensolver software framework 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 class implementations provided by Anasazi.

3.3.1 *`Anasazi::Eigenproblem`.* The `Eigenproblem` class describes an interface for encapsulating the information necessary to define an eigenvalue problem. Anasazi provides users with a concrete implementation of `Eigenproblem`, called `BasicEigenproblem`. This basic implementation contains the matrices and functionality necessary to describe generalized and standard, Hermitian and non-Hermitian linear eigenvalue problems. A user may specify the $\mathbf{A}$ and $\mathbf{B}$ operators defining the eigenvalue problem, a preconditioner and a spectral transformation. The user also may specify the symmetry of the problem, as well as the number of eigenvalue that should be computed.

3.3.2 *`Anasazi::Eigensolver`.* The `Eigensolver` class is intended to capture only the essential steps of an eigensolver iteration, capturing the second level of functionality discussed in Section 2. For example, the `BlockDavidson` class describes only Steps 2–11 of Algorithm 1. Anasazi provides concrete implementations for the iterations associated with the following three methods:

(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].

These implementations can be found in the `BlockKrylovSchur`, `BlockDavidson`, and `LOBPCG` classes, respectively. Only the block Krylov-Schur method can be used for non-Hermitian generalized eigenvalue problems. In contrast, all three algorithms can be used for symmetric positive semi-definite generalized eigenvalue problems.

3.3.3 *Anasazi::SolverManager.* Anasazi provides solver managers to implement a strategy for solving an eigenvalue problem. At the heart of each strategy is the iteration encapsulated in an `Eigensolver` object. Solver managers therefore fulfill the highest level of functionality involved in the solution of an eigenvalue problem. As described in Section 3.2, such a strategy is implemented in the instantiation of constituent objects that are to be passed to the underlying `Eigensolver` object. The strategy is also defined by any additional functionality provided by the solver manager (e.g., subspace restarting). The current solver managers are:

—`BlockKrylovSchurSolMgr` - a solver manager that provides a restarting mechanism for the block Krylov-Schur iteration. When the block size is one, this Krylov-subspace method is mathematically equivalent to the implicitly-restarted Arnoldi method in ARPACK.

—`BlockDavidsonSolMgr` - a solver manager that provides restarting and locking/deflating mechanisms of converged eigenvectors (Step 11 of Algorithm 1) for the block Davidson iteration.

—`LOBPCGSolMgr` - a solver manager that provides a lock/deflating mechanism of converged eigenvectors for the LOBPCG iteration.

3.3.4 *Anasazi::StatusTest.* 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. 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.

With respect to these reasons, 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;

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

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

3.3.5 *Anasazi::SortManager.* The purpose of the `SortManager` is to give users the opportunity to choose the eigenvalues of interest in whatever manner is deemed to be most appropriate. Typically, the eigenvalues of interest are those with:

—the smallest or largest magnitude;

—the smallest or largest real part;

—the smallest or largest imaginary part.

Anasazi provides the ability to perform these six sorts in the `BasicSortManager` class. Other implementations of `SortManager` may, for example, be tailored to account for the effects of chosen spectral transformations.

3.3.6 *`Anasazi::OrthoManager`.* The eigen-iteration implementations provided by Anasazi are all orthogonal Rayleigh-Ritz methods where an orthonormal basis representation is computed. Motivated by the plethora of available methods for performing these computations, Anasazi has left as much leeway to the users as possible. Anasazi provides two concrete orthogonalization managers:

—`BasicOrthoManager` - performs orthogonalization using classical Gram-Schmidt with a possible DGKS correction step [Daniel et al. 1976];

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

4. BENCHMARKING

The benefits of an object-oriented eigensolver framework such as Anasazi are many: modularization provides improved code reuse, static polymorphism via templating allows easier code maintenance and a larger audience through software interoperability, and dynamic polymorphism via inheritance allows easy extension of capability and 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. In this section we address this important issue by comparing Anasazi and ARPACK on a model problem. Our interest is in addressing concerns about the overhead of Anasazi and ARPACK, C++ and FORTRAN 77 software.

We compared Anasazi's `BlockKrylovSchurSolMgr` (with a block size of one) and ARPACK's `dnaupd`, which each compute approximations to the eigenspace of a nonsymmetric 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 classical Gram-Schmidt with the DGKS [Daniel et al. 1976] correction 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. The results of this study can be found in Table VI.

Note that the operator application in Anasazi records approximately twice as much time as the ARPACK implementation. This is because the Anasazi code used an Epetra sparse matrix representation for the linear operator, while the ARPACK implementation applies the block tridiagonal matrix via a stencil (which would have been possible via a different choice of operator). Note however that the operator application comprised only a small portion of the clock time in these tests. The majority of the clock time was consumed via the orthogonalization routines which support the Arnoldi iteration. It should be noted that while these routines are hard-coded in the ARPACK FORTRAN 77 library, they are accessed through a dynamic polymorphic interface in Anasazi. Regardless, the performance of the Anasazi library in computing the Arnoldi vectors is similar to that of ARPACK.

Table VI. Illustrating the overhead of Anasazi as compared to ARPACK; "—" denotes a measurement below the clock resolution. Each timing is the average over three runs.

	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

This illustrates that a well-designed library in C++ can be as efficient as a FORTRAN 77 library.

5. CONCLUSION

Our paper described the design and development of a large-scale eigensolver framework using modern software paradigms. Anasazi achieves this goal by exploiting the generic and object-oriented language features of C++. As a case study, three algorithms were implemented, demonstrating the flexibility and utility of the Anasazi framework. A benchmark demonstrates that an efficient eigensolver implementation is possible using these programming techniques.

6. ACKNOWLEDGMENTS

We thank Roscoe Bartlett, Mike Heroux, Roger Pawlowski, Eric Phipps, and Andy Salinger for many helpful discussions. We appreciate the suggestions and corrections of the anonymous reviewers.

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 documentation. 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/.

BARTLETT, R. A. 2004. Teuchos::RCP Beginner's Guide. Tech. Rep. SAND2004-3268, Sandia National Laboratories.

BHARDWAJ, M., PIERSON, K., REESE, G., WALSH, T., DAY, D., ALVIN, K., PEERY, J., FARHAT, C., AND LESOINNE, M. 16-22 Nov. 2002. Salinas: A scalable software for high-performance structural and solid mechanics simulations. *Supercomputing, ACM/IEEE 2002 Conference*, 35–35.

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.

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.

MORGAN, R. B. AND SCOTT, D. S. 1986. Generalizations of davidson's method for computing eigenvalues of sparse symmetric matrices. *SIAM J. Scientific Computing 7*, 817–825.

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.* John Wiley and Sons, New York.

SALA, M., HEROUX, M. A., AND DAY, D. M. 2004. Trilinos Tutorial. Tech. Rep. SAND2004-2189, Sandia National Laboratories.

SALINGER, A. G., BURROUGHS, E. A., PAWLOWSKI, R. P., PHIPPS, E. T., AND ROMERO, L. A. 2005. Bifurcation tracking algorithms and software for large scale applications. *Int. J. Bifurcation Chaos 15*, 1015–1032.

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.

VELDHUIZEN, T. 1996. Using C++ trait classes for scientific computing. See http://oonumerics.org/blitz/traits.html.