John L. Volakis
ElectroScience Lab
Electrical Engineering Dept.
The Ohio State University
1320 Kinnear Rd.
Columbus, OH 43212
+1 (614) 292-5846 Tel.
+1 (614) 292-7297 (Fax)
volakis.1@osu.edu (email)

David B. Davidson
Dept. E&E Engineering
University of Stellencosch
Stellenbosch 7600, South Africa
(+27) 21 808 4458
(+27) 21 808 4981 (Fax)
davidson@ing.sun.ac.za (e-mail)

Foreword by the Editors

This month, we have another contribution from Spain, this time from the Technical University of Valencia. This paper continues a theme on practical aspects of Finite-Element programming, started in the December, 2000, column, and continued in the August, 2003, issue (references [1] and [2] in this paper). One issue commented on in the latter was the difficulty of finding a public-domain version of a generalized eigenvalue solver for sparse systems. At the 7th International Workshop on Finite Elements for Microwave Engineering, held in Madrid earlier in 2004, the authors described just such a powerful package, *SLEPc*. We invited them to submit a full-length description for this column, which they have now done. Anyone doing eigenvalue computations using the FEM should find both the paper and the software invaluable. In addition, this paper discusses more general issues of software re-use and publicly available libraries for numerical software, with an extensive list of references and Web sites. We thank the authors for this most useful submission.

High-Quality Computational Tools for Linear-Algebra Problems in FEM Electromagnetic Simulation

V. Hernandez and J. E. Roman

Department of Information Systems and Computation, Technical University of Valencia
E-46022 Valencia, Spain
Tel: (+34) 963 877 356; Fax: (+34) 963 877 359; E-mail: jroman@dsic.upv.es

Abstract

A key ingredient of finite-element analysis programs is the linear-algebra solver, typically either a linear-system solver or an eigensolver. The first part of this paper tries to justify why it is important to have recourse to publicly available software for addressing this part of the computation. A number of libraries are mentioned as successful examples that exhibit a series of desirable qualities. Although some of these libraries force the programmer to somewhat change the programming style and may be difficult to learn, the benefits usually pay off the extra effort. The second part of the paper describes one of these libraries in some detail, namely *SLEPc*, the Scalable Library for Eigenvalue Problem Computations, which is used to illustrate the benefits of modern software paradigms for scientific and engineering computing.

Keywords: Finite element methods; programming; eigenvalues and eigenfunctions; linear algebra; parallel algorithms; matrix decomposition

1. Introduction

The Finite-Element Method (FEM) is becoming a widespread analysis tool among electromagnetics practitioners, especially for high-frequency electromagnetic-field simulations in applications such as the design of antennas and microwave circuits. However, the implementation of an FEM-based program can be very challenging if it is intended to be general enough for analyzing non-trivial problems, in situations with complicated three-dimensional geometries, inhomogeneous media, or lossy materials. Many issues are specific to FEM codes that are not present in other scenarios, such as Finite-Difference Time-Domain analysis: unstructured mesh generation, storage formats for sparse matrices, assembly of elemental matrices, post-processing, etc. Many of these issues have already been nicely addressed in a couple of articles in this column: see [1] and [2]. In this article, we focus on the matrix solution of the linear-algebra problems that arise as a consequence of the FEM discretization of the corresponding differential or integral formulation.

The linear-algebra problems most commonly found in electromagnetic simulations are linear systems of equations and eigensystems. For solving these problems, one may be tempted to implement any of the simple algorithms found in introductory numerical-analysis books. However, these methods soon become ineffective when the size of the problem is considerably increased, for instance, when using a finer mesh for the finite-element analysis. Two of the main difficulties associated with "large" linear-algebra problems are the following. On one hand, there is a dramatic growth of the cost of computing the solution, both in terms of the number of required operations and in terms of the storage requirements. This factor depends on the characteristics of the computer that is being used (processor speed and memory capacity) and, in a way, determines when a problem is considered to be large (e.g., a matrix of order 200 was considered large in the early 1980s, while nowadays a moderate-size FEM analysis may involve tens of thousands of degrees of freedom). On the other hand, numerical difficulties are more likely to arise in large problems than in small problems. Typically, the conditioning of the problem gets worse as the size increases, and numerical stability may become an issue. For all these reasons, it is necessary to make use of algorithms and methods that are efficient as well as numerically robust.

Fortunately, there has been much research in the last few years in this direction. Many methods are available for coping with large, sparse problems, that can be implemented efficiently and appropriately from a numerical point of view. There are many sources of information where surveys of such methods can be found, providing a general view of the topic (see, for instance, [3, Chapter 9] and [4, Chapter 13]). However, these methods and techniques have reached such a level of sophistication that non-experts have great difficulty – or are entirely prevented from – implementing them in a computer program, because there are too many technical details that have be taken into account. For instance, implementing a direct solver for a sparse linear system of equations involves management of sparse-matrix storage; computation of a matrix reordering to minimize fill-in; symbolic factorization to prepare for new nonzero elements; numerical factorization, possibly requiring pivoting; and, finally, two triangular solves. All of these aspects are typically plagued with heuristics, and efficiency and numerical issues that become more and more important as the size of the problem grows. All of these ingredients make it really difficult (or, at least, require a lot of effort) to carry out a good implementation of one of these modern methods, and this difficulty is even higher when targeting a parallel machine in order to solve huge problems with many processors.

The computational science community has the expertise required for providing high-quality implementations of these methods, and there is an increasing demand for transferring this technology to other scientific and engineering communities, in different fields. In this paper, we advocate transferring new solver technology in the form of software libraries satisfying certain desirable properties. Section 2 presents a discussion related to some of these qualities of software, and includes the description of several software packages that provide some of these features. Special emphasis is put in the solution of linear systems of equations. The second part of the paper is more concerned with the solution of large, sparse eigenvalue problems. Section 3 provides a brief overview of the problem, and describes general techniques for computing its solution. In Section 4, a particular software library is presented: *SLEPc*, the Scalable Library for Eigenvalue Problem Computations, is being developed by the authors and other colleagues, and tries to follow the spirit of modern numerical software tools presented in this paper. Some examples of usage are included, in order to illustrate the potential of using this library in real applications.

2. Numerical Software

Traditionally, scientists and engineers have been forced to introduce simplifications in their analyses so that these could be feasibly solved in a computer. This scenario is changing, thanks to the capacity of today's computers, which is several orders of magnitude larger than a few years ago, allowing the simulation of detailed models even on inexpensive desktop computers. This evolution of hardware is apparent by looking at processor speed and memory capacity of the latest PC available in the market, but it can also be appreciated by the average megaflop rate of the 500 most powerful computers in the world (http://www.top500.org), which has been growing linearly in the last few years.

However, the steady increase of computer capacity is not necessarily leading to a steady increase in the size of the problems being solved by scientists and engineers. The main reason for this is the complexity associated with software implementation of advanced numerical methods, as seen by the examples given in the introduction of this paper. The effort and expertise required for implementing these kinds of methods seem to be affordable only in the context of high-budget projects for grand-challenge applications. In this sense, software development represents a curb that counteracts the positive hardware trend.

Hopefully, this situation is likely to change by the introduction of modern software technologies. The key for a faster and more productive development of numerical software is *reutilization.* As mentioned in the introduction, efficient and stable numerical methods are already available. What the scientific community needs is simply a way to be able to reuse already existing implementations of these methods. The ideal situation is to have a set of off-the-shelf pieces of software, with well-defined interfaces, that can be plugged into our code to solve a particular numerical problem. This is the concept of a *software component*, which emerged in the commercial arena, where reutilization has long been considered of paramount importance to reduce costs and time-to-market. In the case of numerical software, these types of development techniques are being introduced only slowly, due to the complexity

mentioned above, and also due to some specific requirements not present in other scenarios.

2.1. Complexity and Specificities of Numerical Software

There are many sources of complexity in numerical software. One of them comes from the complexity of the problem being addressed. As commented on above, as the available processing power grows, scientists and engineers try to avoid simplifications introduced in the past. They try to take into account more aspects of the problem, such as nonlinearity, for example, or turbulent flow, in the case of fluid dynamics, etc. As another example, multiphysics simulations may require coupling several already existing simulation codes and the use of conforming discretizations.

Another source is the algorithmic complexity, which refers to the method used for different stages of the solution process. A general example of this would be the discretization technique: the finite-element method provides more flexibility with respect to finite differences, but involves a significant increase in complexity. Three-dimensional geometries also contribute to the complexity: managing data structures for structured or unstructured meshes associated with arbitrary three-dimensional domains can be very difficult, especially in problems with moving parts. Some issues related to meshing were covered in [1] and [2]. Complexity grows even further in the case of adaptive mesh-refinement schemes.

The complexity associated with the design of a linear-algebra solver is mainly driven by two requirements that are specific to numerical software: efficiency and numerical stability. Of course, efficiency is desirable in any software project, but in the case of numerical software, it is very important indeed, because some problems become unsolvable unless efficiency is pursued from many different points of view. For instance, efficiency is concerned with maximizing the convergence rate of iterative algorithms, and with keeping memory requirements to a minimum. Efficiency is also directly related to how well the program makes use of the computer resources. For instance, in some cases, it is possible to halve the computing time simply by better exploiting the memory hierarchy characteristics of modern microprocessors, with so-called block-oriented algorithms. Another obvious example is to gather several processors for solving a single problem: parallel computing pushes the level of complexity, and, typically, the effort required for code-development rockets.

With respect to enforcing numerical stability, different techniques have been proposed in different settings, such as pivoting in the LU decomposition. The key point here is that these techniques may not be necessary when addressing small "toy" problems, but become a must with problems coming from real applications. In some cases, numerical difficulties make the algorithms useless (e.g., the loss of orthogonality in the Lanczos method), unless devices that guarantee stability are introduced.

Another requirement specific to numerical software is portability. Being able to port a code from one platform or architecture to a different one is often convenient. Moreover, it is sometimes a necessity, since the life cycle of scientific codes may be several decades, whereas the lifetime of computers is just a few years. The portability problem was largely solved by the generalized adoption of the IEEE-754 standard for floating-point arithmetic, and the MPI standard for message-passing communication in parallel computers. However, there are still many other details to be taken into account in this respect, such as compiler and operating-system issues.

A final comment about specificities of numerical software is that within the scientific community, there are requirements that are distinctive for a particular field. For instance, in electromagnetics, there is often the need to address problems that involve complex arithmetic and, in particular, complex symmetric matrices appear quite frequently. For these cases, specialized numerical methods abound.

2.2. Models of Software Development and Quality Criteria

The simplest model of numerical software development is that of the numerical recipes. Typically in this paradigm, the programmer is also the user of the code and, therefore, the expert in the problem. Whenever requiring a method to solve a mathematical problem, he or she looks it up in a numerical recipes book (see, for example, [5]), and inserts the code into the program. This paradigm has been very popular and successful, especially for simple tasks such as interpolation, integration, root finding, etc., but it is not sufficient for matrix problems, such as those we are concerned with in this article. There are a variety of other paradigms that are more appropriate for such problems, belonging to a wide spectrum, ranging from simple subroutine libraries to more-sophisticated problem-solving environments, as explained next. The objective is to be able to build codes for the solution of complex problems by putting together the functionality of different specialized software packages. This makes life easier to the user who can focus more on research, rather than wasting effort in implementation of programs.

The concept of a library of subroutines was the first to be introduced as an attempt to make effective reutilization of numerical software. A significant example of this idea is represented by the *BLAS*, the Basic Linear Algebra Subprograms [6]. The *BLAS* are a set of subroutines with a well-defined interface that allow the programmer to easily perform basic operations with matrices and vectors. Their introduction in the 1980s represented a giant leap forward for numerical software. Their use reduces the coding effort and significantly improves the readability of programs. More importantly, using the *BLAS* contributes to achieving portability while maintaining efficiency, since machine vendors now include highly optimized implementations as part of the system software. *LINPAK*, the Linear Algebra Package [7], is built on top of the *BLAS*. *LINPAK* provides higher-level routines for solving systems of linear equations, least-squares problems, eigenvalue problems, and singular-value problems.

Both the *BLAS* and *LINPAK* are designed to work with dense matrices: they enforce a fixed and well-defined data structure for storing matrices, namely the *FORTRAN*-style column-major array. This aspect is important in our discussion. The simplicity of this particular data structure allowed the designers of *LINPAK* to concentrate on algorithms. Unfortunately, this simplicity cannot be easily translated into the realm of sparse-matrix computations (or, at least, in a uniquely accepted way), where many formats exist for storing sparse matrices, each of them being more appropriate for certain types of operations. In this context, the traditional subroutine-library paradigm is definitively not sufficient, because it does not allow enough flexibility for the matrix storage. Some software libraries (e.g., *SPARSKIT* [8]) circumvent this difficulty by means

of the so-called reverse-communication scheme. In this scheme, a subroutine that implements a method does not know anything about how the matrix is stored because there is no prescribed format or function prototype. Instead, whenever the solver requires a certain operation related to the matrix, it returns back to the calling program, where such operation is performed. The resulting code is quite obscure, but the goal of independence from the storage format is achieved. This scheme is mostly useful for iterative methods that only use the matrix in matrix-vector products, and is applicable even in the case that the matrix is not explicitly stored. However, this is not a completely satisfactory solution, because the user has to assume the coding effort related to data-structure management and the implementation of matrix operations, and this is usually where most of the complexity resides.

There is an emerging trend that addresses the problem from another angle: the so-called generic programming approach tries to regain focus on algorithms by completely abstracting data structures. This paradigm will not be described further since it is still rather experimental, and requires special programming-language features, such as *C++* templates (more details can be found in [9]).

Problem-solving environments (PSE) are located at the other end of the spectrum with respect to subroutine libraries. These are typically software frameworks that provide a quite comprehensive set of functionalities for addressing a certain type of problem, for instance, for solving partial differential equations. In this type of system, the user has to develop code specifically for running in this framework. An example of this is *Cactus* [10], in which user-defined modules are plugged into the core system in order to build an application. The user code is obliged by a number of rules, and interacts with the framework via a well-defined application-program interface (API). Although this scheme may seem too restrictive, it turns out to provide many benefits that should not be neglected. For instance, collaborative development of applications in the context of large projects developed by multi-disciplinary teams is far easier with this paradigm than with a traditional approach.

Several conclusions can be drawn from the above discussion. One of them is that a good numerical software tool should include support for both algorithms and data structures. Also, no single paradigm is the best, since different paradigms are appropriate for different needs and situations. The paradigm of choice may depend on the scope of our software project, so for small, handmade programs, the simplest paradigm may still be valid. In the case of more advanced paradigms, a higher-level programming style will necessarily have to be employed. In general, most modern numerical software packages lie somewhere between simple subroutine libraries and fully fledged problem-solving environments. Selection of the most appropriate approach should be made on the basis of the particular characteristics of our problem, but also taking into account for the tools a number of the quality criteria that are discussed in the list below:

- Reliability and robustness: Reliability is mainly related to the numerical stability of the algorithms and to the accuracy of the computed results. Robustness refers to the behavior of the software in unexpected situations, such as exceptions, errors, or when the user provides incorrect input data.

- Efficiency and scalability: As discussed above, efficiency is important in terms of algorithms (doing as few operations as possible) and in terms of architecture-specific issues (trying to reach the machine's peak performance). In the case of parallel computers, a related concept is scalability, which refers to maintaining efficiency as the number of processors increases.

- Portability: This is the ability of code to be installed and executed on different computer platforms. Ideally, porting should not imply loss of efficiency.

- Flexibility: Carefully designed libraries should allow the user to address problems formulated in a number of ways, and to easily adapt the solution strategy to the particular characteristics of the problem. This quality is very important, because solving a problem usually implies experimenting with many different methods and algorithms, and tuning the values of a multitude of parameters.

- Generality, extensibility, and interoperability: Generality (e.g., a library that handles both real and complex matrices is more general than another that only supports real arithmetic) is often a desirable feature and fosters reutilization. However, having a very broad scope may be contrary to good quality, and it is preferable to focus on a specific topic. In the latter case, it is sometimes desirable for the user to be able to extend the functionality of the library with new features. Also, focused libraries should be able to interoperate with other libraries, complementing each other in order to effectively serve as building blocks for applications.

There are other quality criteria not included in the list that may be also important. Two of these are stability (the user interface does not change much from one version to the next) and maintenance (bugs and errors are detected and removed). This is difficult to find in public-domain numerical software, since such software is often developed in the context of research projects, and typically suffers from funding discontinuation.

Finally, ease of use is probably one of the most important concerns to the reader. In this sense, high-quality software should provide good documentation. Regardless, in many cases the learning curve may be quite steep, depending on the user's background. For this reason, it is increasingly common to find computer science specialists participating in scientific and engineering computing projects.

2.3. Examples of Successful Numerical Tools

Software design is becoming an increasingly important issue in scientific and engineering computing, not only to enable reusability, but also to pursue desirable features, such as those mentioned above. For these goals, object-oriented design can provide important advantages over traditional design – for instance, hiding details of parallel execution – and it allows the user to work at a higher level of abstraction. The object-oriented approach is being used in many of the most widely accepted numerical tools available today, some of which are described next. More examples can be found in [11], together with a description containing technical details about software design.

One of the most successful numerical tools is *PETSc* [12], the Portable Extensible Toolkit for Scientific Computation. In this statement, successful means "used a lot," and this is assessed on the basis of the number of software downloads, the feedback received from users, and the number of projects in which it is

being employed. *PETSc* is a toolkit for the parallel solution of partial differential equations, written in *C* (although it can also used from *FORTRAN* or *C++*). Its approach is to encapsulate linear-algebra entities and algorithms using object-oriented programming techniques. All the *PETSc* software is freely available, and is used around the world in many application areas.

PETSc focuses on sparse-matrix solvers, providing both data structures and algorithms, which are implemented as objects. The application programmer works directly with these objects, via generic interfaces as explained below, rather than concentrating on the underlying data structures. The three basic data objects are index sets (`IS`), vectors (`Vec`), and matrices (`Mat`). Apart from these, basic support is also included for managing data associated with discretization meshes, both structured (finite differences) and unstructured (finite elements). Built on top of this foundation are various classes of solver objects, including linear, nonlinear, and time-stepping solvers. These solver objects encapsulate virtually all information regarding the solution procedure for a particular class of problems, including the local state and various options, such as convergence tolerances, etc. Options can be specified by means of calls to subroutines in the source code, and also as command-line arguments.

The design of *PETSc* is based mainly on the concept of the interface in object-oriented programming. The key idea is to cleanly separate the interface of an object class from its implementation, in a way so that it is possible to provide more than one implementation for a single interface. By forcing the user code to access the object functionality via the generic interface, common to all implementations, it is easier to later replace an implementation with a different one. This concept is illustrated with a simple example of matrix creation:

```
MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC
_DECIDE,n,n,&A);
for (e=0;e<nelm;e++) {
    ElementMatrix(e,mesh,indices,values);
    MatSetValues(A,6,indices,6,indices,values
        ,ADD_VALUES);
}
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

The generic interface (`Mat`) admits multiple implementations, one per each different storage format (sequential sparse, parallel sparse, sparse symmetric, sparse by blocks, block diagonal, etc.). The operations with the `Mat` prefix in the above example correspond to the generic interface, and are valid for any implementation. In this way, the program is independent of the storage format, which can be selected dynamically by specifying it either in the source code or in the command line at run time. The example builds a finite-element stiffness matrix by assembling elemental matrices of order six.

This philosophy applies to solvers in a similar way. A call to a function of the generic solver interface (e.g., `KSPSolve` for solving linear systems of equations) will invoke one of many different implementations (e.g., different Krylov methods, such as the Conjugate Gradient, GMRES, BiCGStab, etc., as well as different preconditioners). In this way, *PETSc* provides clean and effective code for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables easy comparison and the use of different algorithms (for example, to experiment with different combinations of Krylov subspace methods and preconditioners, or truncated Newton methods for nonlinear problems), since methods can be specified at runtime, along with many different options. Hence, *PETSc* provides a rich environment for modeling scientific applications, as well as for rapid algorithm design and prototyping.

To a large extent, *PETSc* satisfies the quality criteria mentioned in the previous section, flexibility being one of its flagship appeals. With respect to interoperability, *PETSc* interoperates with other libraries by providing a wrapper for each of them, so that its functionality is seamlessly integrated. Some of the libraries integrated in this way are *SuperLU* [13], *MUMPS* [14], *UMFPACK* [15], *DSCPACK* [16], and *SPOOLES* [17] for direct linear solvers; *Hypre* [18] for preconditioning; and *PVODE* [19] for initial-value problems for ordinary-differential-equation (ODE) systems.

Apart from *PETSc*, there are many other high-quality numerical software packages out there. It is worth mentioning two of them. The first one is *Aztec* [20], an older library that offers far less functionality than *PETSc*, but was very successful precisely because of its simplicity. The other one is *Trilinos* [21], a more recent effort that integrates *Aztec* and is rapidly growing to provide even more functionality than *PETSc*. All these software tools are often unknown to the end user, and this is the reason why some initiatives have been established in order to disseminate them in different fields and to promote their use (see [22]).

3. Eigenvalue Problems

This section provides an overview of eigenvalue problems and the techniques available to solve them. These are an important class of linear-algebra problems that arise in many applications in science and engineering, for instance, in stability or bifurcation analyses. In this article, we focus on large sparse eigenproblems, such as those encountered when computing solutions of Maxwell's equations by the Finite-Element Method.

In the standard formulation, the eigenvalue problem consists of the determination of a complex scalar, λ, for which the equation

$$Ax = \lambda x \tag{1}$$

has nontrivial solution, where A is a square matrix of order n. The scalar λ and the vector x are called the eigenvalue and eigenvector, respectively. Quite often, the problem appears in generalized form:

$$Ax = \lambda Bx , \tag{2}$$

where B is also a square matrix of order n. Most applications do not require computing the entire spectrum but only a few selected solutions, e.g., the smallest eigenvalues. Other related linear-algebra problems, such as the quadratic eigenvalue problem or the singular-value decomposition, can be formulated as standard or generalized eigenproblems.

3.1. Numerical Solution Techniques

Many methods have been proposed to compute eigenvalues and eigenvectors. Some of them, such as the QR iteration, are not appropriate for large sparse matrices because they are based on

modifying the matrix by applying similarity transformations, which destroy sparsity. In what follows, we consider only methods that compute a few eigenpairs of large sparse problems (see [23] for a comprehensive survey). These methods usually extract the solution from the information generated by applying the matrix to various vectors. In this way, matrices are only involved in matrix-vector products, which preserves sparsity and also allows the solution of problems in which matrices are not stored explicitly.

The most basic sparse eigensolver is the Power Method, in which an initial vector is repeatedly pre-multiplied by matrix A and conveniently normalized. Under certain conditions, this iteration converges to the eigenvector associated to the largest eigenvalue in magnitude. After this eigenvector has converged, deflation techniques can be applied in order to retrieve the next eigenvector.

The Subspace Iteration is a generalization of the Power Method in which the matrix is applied to a set of m vectors simultaneously, and orthogonality is enforced explicitly in order to avoid the convergence of all of the vectors toward the same eigenvector. This method is often combined with a projection technique to compute approximations to the eigenpairs of matrix A, extracting them from a given low-dimensional subspace on which the problem is projected. In the case of the Subspace Iteration, in the kth iteration this subspace is the one spanned by the set of vectors $\mathrm{A}^k \mathrm{v}_i$.

The projection scheme is common to many other methods. In particular, so-called Krylov methods use a projection onto a Krylov subspace,

$$K_m(\mathrm{A},\mathrm{v}) \equiv \operatorname{span}\left\{\mathrm{v}, \mathrm{Av}, \mathrm{A}^2\mathrm{v}, \ldots, \mathrm{A}^{m-1}\mathrm{v}\right\}. \tag{3}$$

The most basic algorithms of this kind are the Arnoldi and Lanczos. These methods are the basis of a large family of algorithms. The Arnoldi algorithm can be used for non-symmetric problems. It computes approximations of invariant subspaces from Krylov subspaces of increasing size. Since the computational (and storage) costs grow with the size of these subspaces, the algorithm is typically restarted when a maximum is reached. Several restart alternatives have been proposed. The Lanczos algorithm exploits the symmetry of the matrix, and builds the Krylov subspace with a simple three-term recurrence. The main concern in this case is to monitor the loss of orthogonality of the basis vectors, due to round-off errors. A non-symmetric version of Lanczos exists that can be more efficient than Arnoldi, but a robust implementation requires so-called look-ahead techniques to avoid the possibility of serious breakdown of the algorithm.

There are several issues that are common to the methods mentioned above. One of them is the orthogonalization strategy when constructing the basis. Several schemes are available, with different behavior with respect to round-off errors. Another important aspect is the management of convergence. Locking already converged eigenvalues can considerably reduce the cost of an algorithm.

Convergence problems can arise in the presence of clustered eigenvalues. Selecting a sufficiently large number of basis vectors can usually avoid the problem. However, convergence can still be very slow, and acceleration techniques must be used. Usually, these techniques consist of computing eigenpairs of a transformed operator, and then recovering the solution of the original problem. The most commonly used spectral transformation is called shift-and-invert, and operates with the matrix $(\mathrm{A}-\sigma\mathrm{I})^{-1}$. The value of the shift, σ, is chosen so that the eigenvalues of interest are well separated in the transformed spectrum, thus leading to fast convergence. A linear system of equations must be solved whenever a matrix-vector product is required in the algorithm when using this approach. Typically, this computation must be quite accurate, often requiring use of direct methods. Recently proposed eigensolvers, such as Jacobi-Davidson, have tried to avoid the shift-and-invert spectral transformation by replacing the direct linear solver by approximate solves. These methods are called preconditioned eigensolvers, due to their resemblance to the idea of a preconditioner in the iterative solution of linear systems.

3.2. Available Software

Solving large sparse eigenvalue problems is essentially more difficult than solving linear systems of equations, and effective computational methods have only recently been proposed. For these reasons, the availability of public-domain implementations of such methods is scarcer.

There are several parallel software libraries that approach the problem by some variant of the methods mentioned above. The most complete of these libraries is *ARPACK* [24], which implements the Arnoldi and Lanczos processes with Implicit Restart for standard and generalized problems, in both real and complex arithmetic. Other available parallel libraries are *BLZPACK* [25], *PLANSO* [26], and *TRLAN* [27], which implement different "flavors" of the Lanczos method. A sequential implementation of the subspace iteration is provided by *SRRIT* [28].

All of these libraries require the user to provide a matrix-vector subroutine or to use a reverse-communication scheme. As mentioned above, these two interface strategies are format-independent, but keep part of the complexity on the user's side. All of these libraries are research codes, focused on providing a robust implementation of a single method. They can be used successfully in many situations, but are far from fully satisfying many of the quality criteria discussed above. For instance, flexibility and interoperability are often very limited in these codes. Very desirable features in this case are an easy way of experimenting with different parameter values as well as shifting from one method to another, and straightforward connections to various linear system solvers and preconditioners. The library presented below tries to address all of these issues.

4. Solving Eigenvalue Problems with *SLEPc*

SLEPc [29, 30], the Scalable Library for Eigenvalue Problem Computations, extends the functionality of *PETSc* to solve large, sparse eigenvalue problems on parallel computers. It can be used for the solution of problems formulated in either standard or generalized form, as well as other related problems, such as the singular-value decomposition. It supports either real or complex arithmetic, and includes solvers for both Hermitian and non-Hermitian problems. *SLEPc* is currently being used in numerous applications from different areas, including computational electromagnetics, computational chemistry, nuclear engineering, materials science, statistics and information retrieval, quantum mechanics, etc.

4.1. Design Principles

The functionality provided by *SLEPc* is grouped around two objects, EPS and ST, as described next.

The Eigenvalue Problem Solver (EPS) is the main object provided by *SLEPc*. It is used to specify an eigenvalue problem, either in standard or generalized form, and provides uniform and efficient access to all of the package's eigensolvers. The EPS object provides functions for setting several parameters, such as the number of eigenvalues to compute, the dimension of the subspace, the requested tolerance, and the maximum number of iterations allowed. The user can also specify other things, such as the portion of the spectrum of interest.

The solution of the problem is obtained in several steps. First, the matrices associated with the eigenproblem are specified via EPSSetOperators. Then, a call to EPSSolve is done, which invokes the subroutine for the selected eigensolver. EPSGetConverged can be used afterwards to determine how many of the requested eigenpairs have converged to working precision. Finally, EPSGetSolution is used to retrieve the eigenvalues and eigenvectors. All of these function calls refer to the generic EPS interface, thus making the user code independent of which solver is selected. The solution method can be specified procedurally or via the command line. Currently available solvers are Power Iteration with deflation, Inverse Iteration, Rayleigh Quotient Iteration, Subspace Iteration with non-Hermitian projection and locking, and Arnoldi with explicit restart and deflation. Lanczos and more advanced methods are under development. In addition to these methods, there are also wrappers for integrating the eigenvalue-solver libraries mentioned in the previous section, so that, for instance, one can use the *ARPACK* solver as well as any of the other solvers provided natively by *SLEPc*.

The other main *SLEPc* object is the Spectral Transformation (ST), which encapsulates the functionality required for acceleration techniques based on the transformation of the spectrum. The user does not usually need to create an ST object explicitly. Instead, every EPS object internally sets up an associated ST. One of the design cornerstones of *SLEPc* is to separate spectral transformations from solution methods so that the user can specify any combination of them. To achieve this, all the eigensolvers contained in EPS must be implemented in a way such that they are independent of which transformation the user has selected. That is, the solver algorithm has to work with a generic operator, the actual form of which depends on the transformation used. After convergence, eigenvalues are transformed back appropriately, if necessary. Table 1 lists the operators used in each case, either for standard or generalized eigenproblems.

By default, the shift-of-origin spectral transformation is used, with a zero shift ($\sigma = 0$). Changing the value of the shift can sometimes improve the convergence rate. This benefit is always obtained when using the other two transformation modes, but at the expense of a higher cost per iteration, due to the inverted matrix. In all cases, the value of the shift can be specified at run-time. The expressions shown in Table 1 are not built explicitly. Instead, the appropriate operations are carried out when applying the operator to a certain vector. The inverses imply the solution of a linear system of equations, which is managed internally by setting up an associated KSP object (*PETSc*'s linear-system solver). The user can control the behavior of this object by adjusting the appropriate options, as will be illustrated in the examples in the next section.

Table 1. The operators used in each spectral-transformation mode.

Spectral Transformation	Standard Problem	Generalized Problem
Shift	$A+\sigma I$	$B^{-1}A+\sigma I$
Shift-and-invert	$(A-\sigma I)^{-1}$	$(A-\sigma B)^{-1}B$
Cayley	$(A-\sigma I)^{-1}(A+\tau I)$	$(A-\sigma B)^{-1}(A+\tau B)$

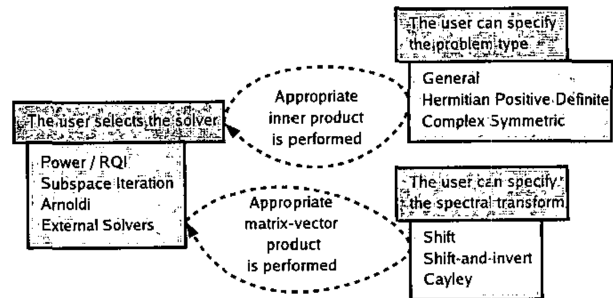

Figure 1. The abstraction used by *SLEPc* solvers.

Apart from separating spectral transforms from solvers, another remarkable design decision is the abstraction of inner products. Different inner products are used depending on the problem type, and this is transparent to the eigensolver, simplifying its implementation. This is done, for instance, for preserving symmetry in positive-definite symmetric generalized problems, in order to be able to use symmetric solvers (e.g., Lanczos). Figure 1 summarizes the abstraction used in *SLEPc*.

4.2. Example of Usage

The following *C* source code illustrates the solution of a simple generalized eigenvalue problem by means of *SLEPc*. The code for creation of matrix A and for error checking is omitted.

```
#include "slepceps.h"

EPS          eps;      /* eigensolver context */
Mat          A,B;      /* matrices of Ax=kBx */
Vec          xr,xi;    /* eigenvector, x */
PetscScalar  kr,ki;    /* eigenvalue, k */
int          i,nconv;

EPSCreate(PETSC_COMM_WORLD,&eps);
EPSSetOperators(eps,A,B);
EPSSetProblemType(eps,EPS_GNHEP);
EPSSetFromOptions(eps);
EPSSolve(eps);
EPSGetConverged(eps,&nconv);
for(i=0;i<nconv;i++)
    EPSGetEigenpair(eps,i,&kr,&ki,xr,xi);
EPSDestroy(eps);
```

All of the operations of the program are done over a single EPS object, the eigenvalue problem solver, created in the first call. The next two operations specify the matrices associated with the generalized eigenproblem and set the problem type (generalized

non-Hermitian). At this point, the value of the different options could be set by means of function calls, such as `EPSSetTolerances`. After this, a call to `EPSSetFromOptions` is made, so that options specified at run-time in the command line are appropriately passed to the EPS object. In this way, the user can easily experiment with different combinations of options without having to recompile. The call to `EPSSolve` launches the solution algorithm. The subroutine being actually invoked depends on which solver has been selected by the user. At the end, the data associated with the solution of the eigenproblem is kept internally. The next line queries how many eigenpairs have converged to working precision. The solutions of the eigenproblem are retrieved with the function `EPSGetEigenpair`. Finally, the EPS context is destroyed.

The following are examples of command-line usage.

```
$ program -eps_nev 10 -eps_ncv 24
```

The above line executes the program specifying the number of eigenvalues and the dimension of the subspace.

In this other example, the solution method is given explicitly, and the matrix is shifted with $\sigma = 0.5$.

```
$ program   -eps_type subspace -st_type shift
            -st_shift 0.5 -eps_monitor
```

The last option instructs *SLEPc* to activate the convergence monitor. In cases where the operator contains an inverted matrix (see Table 1), the user can additionally specify options relative to the solution of the linear systems. For example,

```
$ program -st_ksp_type gmres -st_pc_type ilu
```

In the above example, the prefix `st_` is used to indicate an option for the linear system of equations associated with the ST object. In particular, the system is solved with GMRES, preconditioned by incomplete LU factorization. Similarly, for using shift-and-invert:

```
$ program   -st_type sinvert -st_shift 10
            -st_pc_type jacobi
            -st_ksp_type cg -st_ksp_rtol 1e-7
```

In this last example, the value of the shift is $\sigma = 10$, and linear systems are solved via Conjugate Gradients with Jacobi preconditioning, up to a precision of 10^{-7}. A detailed description of these options and others is included in the *SLEPc* user's guide.

4.3. The Resonant Cavity Problem

To finish this section, an example is shown from an application in electromagnetics, namely the resonant cavity problem. The objective is to illustrate how *SLEPc* can be used to address problems that do not exactly match the formulations presented in Equations (1) and (2).

When using the Finite-Element Method to compute the eigenfunctions of Maxwell's equations in a bounded volume, we obtain the following algebraic eigensystem:

$$\left(\mathrm{A} - k_0^2\,\mathrm{B}\right)\mathrm{E} = 0, \tag{4}$$

where A and B are large, sparse matrices, resulting from assembly of elemental matrices; k_0^2 is the eigenvalue of the system; and E is the eigenvector from which the electric field can be reconstructed. Vector elements are used in order to avoid spurious modes (solutions satisfying the algebraic problem but not the original differential equation). More details related to the discretization of the problem can be found in [31].

With the *SLEPc* example code provided above, the program would be ready for computing the extreme eigenvalues or the internal eigenvalues if using shift-and-invert. However, in this particular application, one is usually interested in computing a few of the lowest wavenumbers, k_0^2. The dimension of A's null space is very high (about a sixth of the order of A), and therefore Equation (4) has many zero eigenvalues that have to be avoided during computation. This new problem can be formulated as a constrained eigenproblem

$$\left(\mathrm{A} - k_0^2\,\mathrm{B}\right)\mathrm{E} = 0, \tag{5a}$$

$$\mathrm{C}^{\mathrm{T}}\,\mathrm{B}\,\mathrm{E} = 0, \tag{5b}$$

where the columns of C form a basis of the null space of A. This problem can be addressed in several ways (see [32] for a detailed discussion). One possible way is to solve the following modified problem:

$$\left(\mathrm{M} - k_0^2\,\mathrm{B}\right)\mathrm{E} = 0, \tag{6}$$

where $\mathrm{M} = \mathrm{A} + \mathrm{BCHC}^{\mathrm{T}}\mathrm{B}^{\mathrm{T}}$, and H is a positive definite matrix so that zero eigenvalues are shifted and do not disturb computations. This modified eigenproblem can be easily implemented in *SLEPc* with the aid of shell matrices. These are matrices that do not require explicit storage of the component values. Instead, the user must provide subroutines for all of the necessary matrix operations, typically only the application of the linear operator to a vector. Shell matrices are a simple mechanism of extensibility, in the sense that the package is extended with new, user-defined matrix objects. Once the new matrix has been defined, *SLEPc* can use it as any other matrix. In Equation (6), matrix M should not be computed explicitly because it may be dense, and a shell matrix can instead be used by simply providing a subroutine for computing the product $\mathrm{y} = \mathrm{Mx}$ for any given vector x.

5. Conclusion

The implementation of a finite-element analysis program has to cope with many sources of complexity, in particular in the matrix solution of linear-algebra problems. A large community effort, carried out in the few last years by many computational science specialists, has made possible the fact that today there is a wide panoply of high-quality numerical software tools that can be used in order to avoid much of the complexity. These tools exhibit desirable qualities and provide many benefits. However, they are not simply black-box subroutines, and their use requires adapting our programming habits.

For some classes of problems, such as linear systems of equations, a surprisingly rich variety of implementations of different methods are publicly available, each of them satisfying more or

less the quality criteria mentioned in this article. For different reasons, the offerings related to eigenvalue problems is much more limited. Several software tools exist for this task, but they are far from reaching the same quality level as packages intended for other domains. An exception could be *SLEPc*, a parallel library for the solution of large, sparse eigenvalue problems, described in Section 4. It is based on *PETSc*, and aims at being able to cope with problems arising in real applications by using standard techniques, such as shift-and-invert transformations and state-of-the-art solvers. It offers a growing number of solution methods, as well as interfaces to external eigenvalue packages. With little programming effort, it is possible to easily test different solution strategies for a given eigenproblem. Once the problem has been specified, it is extremely easy to experiment with different solution methods and to carry out studies by varying parameters, such as the value of the shift. It is also possible to easily prepare the program for running with several processors, thus allowing the solution of huge problems without requiring much experience in parallel programming.

SLEPc inherits all the good properties of *PETSc*, including portability, scalability, efficiency, and flexibility. Due to the seamless integration with *PETSc*, the user has at his or her disposal a wide range of linear-equation solvers. The use of shell matrices allows easy formulation of block-structured or implicit eigenproblems, as illustrated with the resonant cavity problem.

6. References

1. D. B. Davidson, "Implementation Issues for Three-Dimensional Vector FEM Programs," *IEEE Antennas and Propagation Magazine*, **42**, December 2000, pp. 100-107.

2. A. Awadhiya, P. Barba, and L. C. Kempel, "Finite-Element Method Programming Made Easy???," *IEEE Antennas and Propagation Magazine*, **45**, August 2003, pp. 73-79.

3. J. L. Volakis, A. Chatterjee, and L. C. Kempel, *Finite Element Method for Electromagnetics: Antennas, Microwave Circuits, and Scattering Applications*, Oxford and New York, Oxford University Press and IEEE Press, 1998.

4. J. Jin, *The Finite Element Method in Electromagnetics, Second Edition*, New York, John-Wiley and Sons, 2002.

5. "Numerical Recipes," http://www.nr.com.

6. "Basic Linear Algebra Subprograms," http://www.netlib.org/blas.

7. *LAPACK* home page, http://www.netlib.org/lapack.

8. Y. Saad, "SPARSKIT: A Basic Tool-Kit for Sparse Matrix Computations," available at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html.

9. The Matrix Template Library, http://www.osl.iu.edu/research/mtl.

10. Cactus Problem Solving Environment, http://www.cactuscode.org.

11. M. E. Henderson, C. R. Anderson, and S. L. Lyons (eds.), "Object Oriented Methods for Interoperable Scientific and Engineering Computing," Proceedings in Applied Mathematics, Philadelphia, SIAM, 1999.

12. S. Balay et al., "PETSc Users Manual," Technical Report ANL-95/11, Rev. 2.2.1, Argonne National Laboratory, 2004; available at http://www.mcs.anl.gov/petsc.

13. X. S. Li and J. W. Demmel, "SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems," *ACM Transactions on Mathematical Software*, **29**, 2, June 2003, pp. 110-140; available at http://crd.lbl.gov/~xiaoye/SuperLU.

14. P. R. Amestoy et al., "MUMPS: A General Purpose Distributed Memory Sparse Solver," *Lecture Notes in Computer Science*, **1947**, 2001, pp. 121-130; available at http://www.enseeiht.fr/lima/apo/MUMPS.

15. T. A. Davis, "UMFPACK V4.3 – An Unsymmetric-Pattern Multifrontal Method," *ACM Transactions on Mathematical Software*, **30**, 2, June 2004, pp. 196-199; available at http://www.cise.ufl.edu/research/sparse/umfpack.

16. P. Raghavan, DSCPACK: Domain-Separator Codes For Solving Sparse Linear Systems, http://www.cse.psu.edu/~raghavan/Dscpack.

17. SPOOLES, Sparse Object Oriented Linear Equations Solver, http://www.netlib.org/linalg/spooles.

18. R. D. Falgout and U. Meier Yang, "Hypre: A Library of High Performance Preconditioners," *Lecture Notes in Computer Science*, **2331**, 2002, p. 632-641; available at http://www.llnl.gov/CASC/hypre.

19. G. D. Byrne and A. C. Hindmarsh, "PVODE, an ODE Solver for Parallel Computers," *The International Journal of High Performance Computing Applications*, **13**, 4, Winter 1999, pp. 354-365; available at http://www.llnl.gov/CASC/sundials.

20. Aztec: A Massively Parallel Iterative Solver Library for Solving Sparse Linear Systems, http://www.cs.sandia.gov/CRF/aztec1.html.

21. The Trilinos Project, http://software.sandia.gov/trilinos.

22. ACTS: The Advanced Computational Software Collection, http://acts.nersc.gov.

23. Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst (eds.), *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, Philadelphia, SIAM, 2000.

24. R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*, Philadelphia, SIAM, 1998; available at http://www.caam.rice.edu/software/ARPACK.

25. O. A. Marques, "BLZPACK: Description and User's Guide," Technical Report TR/PA/95/30, CERFACS, Toulouse, France, 1995; available at http://www.nersc.gov/~osni/#Software.

26. K. Wu and H. D. Simon, "A Parallel Lanczos Method for Symmetric Generalized Eigenvalue Problems," Technical Report LBNL-41284, Lawrence Berkeley National Lab., 1997; available at http://www.nersc.gov/~kewu/planso.html.

27. K. Wu and H. D. Simon, "Thick-Restart Lanczos Method for Symmetric Eigenvalue Problems," *Lecture Notes in Computer Science*, **1457**, 1998, pp. 43-55; available at http://www.nersc.gov/~kewu/trlan.html.

28. G.W. Stewart and Z. Bai, "SRRIT – A Fortran Subroutine to Calculate the Dominant Invariant Subspace of a Nonsymmetric Matrix," *ACM Transactions on Mathematical Software*, **23**, 4, 1997, pp. 494-513; available at http://www.netlib.org/toms/776.

29. V. Hernandez, J. E. Roman, and V. Vidal, "SLEPc: Scalable Library for Eigenvalue Problem Computations," *Lecture Notes in Computer Science*, **2565**, 2003, pp. 377-391; available at http://www.grycap.upv.es/slepc.

30. V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal, "SLEPc Users Manual," Technical Report DSIC-II/24/02, Rev. 2.2.1, Dept. Information Systems and Computation, Technical University of Valencia, Spain, 2004.

31. J. V. Balbastre, L. Nuño, A. Díaz, A. Muelas, I. Crespo, "Finite-Element Analysis of General Lossy Anisotropic Resonant Cavities Using Edge Elements," *Microwave and Optical Technology Letters*, **19**, 4, 1998, pp. 304-307.

32. V. Simoncini, "Algebraic Formulations for the Solution of the Nullspace-Free Eigenvalue Problem Using the Inexact Shift-and-Invert Lanczos Method," *Numer. Linear Algebra Appl.*, **10**, 2003, pp. 357-375.

Editor's Comments *Continued from page 53*

The Italian government is funding a major study on the impact of electromagnetic-field-emitting systems – including cell phones – on humans and the environment. Similar studies are also being undertaken in other European countries. In his Telecommunications Health and Safety column, Jim Lin describes a recent conference held in connection with the Italian study, and discusses the importance and need for such work.

Interruptions: Etiquette, Safe Computing, and Time Management

Randy Haupt's Ethically Speaking column is devoted to a discussion of the interruptions we experience from e-mail, cell phones, call-waiting, pagers, and the other electronic communication mechanisms upon which we have come to depend. There are certainly ethical and etiquette issues here: you need to read the column.

In reading it, I was reminded of the reason why I have rarely had a problem with e-mail interruptions (and why I've never inflicted e-mail interruptions on those with whom I'm working!): I don't "stay connected." When my Internet connection was a dial-up connection, I would only check e-mail once or twice per day, because I had to dial in to it: it was inherently not an always-on connection. My primary Internet connection is now a broadband (cable modem) connection, which is inherently an always-on connection. However, when I first got the broadband connection, I installed a switch between my computer and the Internet for security reasons. I thus again have to take action in order to connect to the Internet to receive my e-mail. A side effect of this is that as a result, I typically only check e-mail once per day.

I think there are at least two benefits from this situation. First, by turning off the Internet connection when not in use, I make myself *much* less vulnerable to the various attacks and malware that have become a dominant factor in Internet usage. I have a hub between my cable modem and my computer and internal network, and I *simply turn it off* when I'm not accessing the Internet. I'll comment more on this below. Second, I thus end up managing my time, rather than allowing e-mail to manage my time for me. I decide when to process my e-mail, and how much time I devote to doing so. If you have your e-mail set to notify you whenever a new message comes in – and worse, if you then interrupt whatever you're doing to process that e-mail – you are probably far less productive because of the interruptions, and you have turned control of your time over to whoever is sending you e-mail.

There are some disadvantages to not "staying connected" all the time. The primary disadvantage is that I might not immediately receive and respond to an urgent e-mail. However, e-mails that are so urgent that they need to be responded to immediately are really very rare, in my experience. If someone needs to communicate something to me that urgently, they can (and probably should) telephone me. The cost and effort of a telephone call (compared to an e-mail) serves as an excellent filter for determining what is really urgent! Furthermore, that kind of urgent communication is usually better done by telephone, since it often involves the need for rapid two-way communication.

I've adopted the same approach with my cell phone. When I do carry one, I leave it turned off unless I'm either expecting a call or I need to make a call. I do check voice mail periodically, but I can do that when it is efficient and convenient (and safe!) to do so. Among other benefits, this means that unless I'm in a situation where I'm expecting an emergency call, I'm not using a cell phone when I'm driving, which is much safer. It also means that I'm not receiving cell-phone calls when I'm where I would prefer not to receive them – such as in public places – so I'm not disturbing others, nor am I holding private conversations in public.

Who controls your time when it comes to communications?

From the Screen of Stone Lite: Thoughts on Connecting a New Computer

Connecting to the Internet has become almost an inherent part of setting up a new computer. It is also a very hazardous step. The CERT® Coordination Center (http://www.cert.org), located at Carnegie Mellon University, is a worldwide leader in computer security. CERT estimates that the mean time between connecting a computer to the Internet and the first attack on that computer is measured in minutes. I have experienced attacks less than 10 seconds after connecting a computer via a broadband connection. These attacks usually do not have direct human involvement. They are typically the result of software implanted on large numbers of systems around the world (and typically, without the knowledge of the owners of the systems).

This situation creates a paradox when you're trying to set up a new computer. On the one hand, you typically need to connect the new computer to the Internet to download operating-system

Continued on page 134