Why we Couldn't Use Numerical Libraries for PETSc

William D. Gropp†
Mathematics and Computer Science Division,
Argonne National Laboratory,
Argonne, IL 60439, USA

Keywords
Software Libraries, Interoperability, Composition

1 INTRODUCTION

This paper discusses some of the reasons for the crisis in numerical software. Put most bluntly, much numerical software does not address the real needs of users. This situation has resulted not because of numerical deficiencies in the software; measured by traditional criteria such as stability and correctness, the software is quite good. Rather, much numerical software is poorly designed from a software engineering perspective. Problems show up when an attempt is made to use numerical software in complex applications.

One can, of course, point to the success of special function libraries and to linear algebra libraries for dense problems (among others). But these can be misleading models for the current generation of sophisticated applications. Both types of libraries represent special cases because they involve very simple data structures that are natural for the applications. In the case of special functions, the "data structure" is just a scalar or two. For linear algebra, the dense matrix representation is a natural one both for users and for library implementors.* In addition, the amount of work per data value in dense linear algebra often scales as $\mathcal{O}(n)$, hiding the cost of any data rearrangement. For more complex needs, however, the software engineering approaches that worked for these operations become inadequate.

As an illustration of the difficulty, consider an application that is solving large nonlinear partial differential equations (PDEs). The problem is large and complex, and a significant amount of design and coding has gone into the data structures used to model the problem. For example, there may be an irregular mesh, with the problem described by operator

---

†This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

*This ignores the row-major and column-major issues, of course.

coefficients at each mesh point. This is a natural data structure for the application, but not for any numerical algorithm.

Exactly this situation occurred while our group was doing research on domain decomposition methods for solving large PDEs. Our work was directed at both uni- and parallel processors and required combining many different numerical components. For example, in domain decomposition, a physical domain in the problem is divided into a number of subdomains, on each of which a subproblem is solved. The subproblem solutions are combined to form an approximate solution to the full problem; iterations may be required to achieve the desired solution. A typical method may need to solve a linear system on the entire domain. Since the PDE discretization is sparse (in the methods we looked at), this linear system is solved by some Krylov method. The preconditioner for this method is based on the domain decomposition, and requires that a linear system be solved on each subdomain. It may be, of course, that an iterative solution method is the best approach for the sub-domain solve, or alternately a sparse-direct method (the sub-domain is smaller) or other alternative (e.g., FFT or fast Poisson preconditioning for variable coefficient problems). Further, there is no clear single choice of Krylov method, and even the use of an iterative method over a direct method may depend on the data at run time (see Kettunen 1993 for an example). There was no plan to develop anything other than research software. But we rapidly discovered that we needed to write a new numerical library because this application demanded more support for complex data structures and interoperation of components than we could find. The result was the PETSc library (see `http://www.mcs.anl.gov/petsc/petsc.html`).

Specifically, we needed routines to assemble and solve sparse linear systems and to solve sparse nonlinear systems. In many cases, we did not want to specify the particularly algorithm used to solve the individual linear or nonlinear system, but we did need the ability to do so. In the following section we discuss the problems that we encountered.

2 PROBLEMS WITH NUMERICAL SOFTWARE

The major problem with numerical software is that it is designed around individual algorithms (e.g., Bi-CG-Stab or Runge-Kutta 4-5) and designed to be used in isolation, that is, not combined with other libraries or routines.

2.1 Lack of Modularity

Each library module may be used in several ways at the same time (for each of several domains). A classic example is routines for FFTs. A typical FFT implementation contains a routine to initialize some workspace (basically the FFT coefficients) and a second routine to apply the FFT to the user's data. As a "service" to the user, the workspace is often hidden inside the routines, for example, in a common block or static data area. Unfortunately, if the user has two or more different lengths of data to transform, this approach is very inefficient. In one climate modeling example, public-domain FFT code outperforms one vendor's carefully tuned FFT for this very reason.

In parallel codes, another issue arises. In parallel, the operation may not involve all processors. For example, with 16 processors, there may be a decomposition into four separate blocks, each with 4 different processors. The same operation (e.g., solving a linear system) may need to be applied in each of the four blocks. Note that this is still an SPMD (single program, multiple data) model. Some parallel numerical libraries have assumed that all processes are participating in a single operation; this assumption is far more restrictive than the SPMD model. For sophisticated multilevel or hierarchical methods, libraries must be able to handle arbitrary collections of processors.

2.2 Application Interface

Each application may have its own, natural data structure. For example, the matrix may be stored with the physical (either structured or unstructured) mesh. In fact, many applications have no explicit matrix, even though they are solving linear systems; so-called matrix-free methods are common.

In addition, applications may have optimized the data structures for performance. For example, special blocking may haave been used for efficient vectorization or cache memory usage.

One approach sometimes suggested to handle application data structures is *reverse communication*. In this approach, routines that need to use the application data structures set a flag and return to the user. This can work if there are only a few such operations. But this approach often makes implicit assumptions about the data structures. For example, sparse linear system solvers employing reverse communication often assume that the vectors are contiguous arrays of bytes and that only the matrix is stored in a user-defined way. Such an assumption may not be true; for example, a PDE code may use ghost or other false elements in the arrays to simplify the handling of boundary conditions or, in a parallel context, to simplify performing the matrix-vector products. Note that reverse communication thus can be *extremely* awkward, since *all* vector-scalar and vector-vector operations must be passed back to the user (not just the matrix-vector operations).

2.3 Algorithm Injection

The best algorithm may depend on data or experience. It must be easy to change. This point seems pretty obvious, but most numerical software is oriented by algorithm rather than the problem that it solves. Changing methods thus becomes awkward and error-prone. This is bad for the numerical analysis community not only because it limits the impact of advancements in the field, but because it also complicates the process of performing fair comparisons of different algorithms for the same problem.

An additional issue that is often ignored is that the choice of appropriate algorithm may change not only between runs of an application (as technology advances or the problem changes) but even *during* a single run of an application. For example, in solving a nonlinear system of equations, each step often involves solving a linear system. Initially, this linear system may be very hard to solve with an iterative method, making a direct (sparse) technique preferable. Later in the nonlinear solution process, the linear system may be relatively easy to solve with an iterative method.

All of this implies that argument lists must be independent of algorithms. The functions must be determined at run time (late binding). In addition, procedural analysis by compiler cannot be depended on to give efficient code for fine grain operations.

Further, hierarchical or multilevel methods may generate nested calls; this is not just simple recursion. Hence, implementations of the library routines cannot use static storage or non-recursion-safe languages.

3 OPERATIONAL PROBLEMS

Once a library is found that meets the needs described above, it may still fail to be useful, not because of its design but because of its implementation. Some of these apply to "research" or "experimental" software (particularly where source code is distributed) but they also apply to commercial software. The most common problems are difficult-to-build source distributions, poor documentation, examples, support, and lack of portability and availability on new systems.

4 THE SOLUTION USED IN PETSC

The software engineering aspects of a modern numerical library are just as challenging as the numerical issues. This last section briefly mentions some techniques that are used in PETSc.

Data-structure neutral As much as possible, the operations are referred to by their mathematical properties. Vectors are objects, not contiguous regions in memory.

Component oriented The implementation of algorithms in a data-structure-neutral way is through objects that contain both the data structures and the functions that operate on them. Unfortunately, this approach can not be implemented in standard Fortran.

Contexts The *state* (private data, information that would otherwise be static) that applies to a particular use of a routine or object is its context. By using multiple contexts, a single routine can simultaneously handle many different problems.

Organized by problem Routines are provided by problem (e.g., solve linear system). Choosing a particular algorithm is done not by changing the routine name but by changing an argument (actually, the context for the routine).

Documentation tools Automated tools provide correct, up-to-date documentation in Unix nroff (man), LaTeX, and HTML (Web) forms. The information is extracted from the source files for the implementation, encouraging early production of documentation.

Bug reporting and tracking An e-mail-based bug tracking system is easy for users (no weird forms) and for us.

ANSI C The use of ANSI C provides interoperability with Fortran (an advantage over C++) and ensures a solid development environment.

REFERENCES

Forsman K., Gropp, W., Kettunen, L., Levine, D., and Salonen, J. (1995) Solutions of dense systems of linear equations arising from integral equation formulations. *IEEE Antennas and Propagation Magazine*, 96–100.