The Design of Data-Structure-Neutral Libraries for the Iterative Solution of Sparse Linear Systems

BARRY F. SMITH AND WILLIAM D. GROPP

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844; e-mail: bsmith@mcs.anl.gov, gropp@mcs.anl.gov.

ABSTRACT

Over the past few years several proposals have been made for the standardization of sparse matrix storage formats in order to allow for the development of portable matrix libraries for the iterative solution of linear systems. We believe that this is the wrong approach. Rather than define one standard (or a small number of standards) for matrix storage, the community should define an interface (i.e., the calling sequences) for the functions that act on the data. In addition, we cannot ignore the interface to the vector operations because, in many applications, vectors may not be stored as consecutive elements in memory. With the acceptance of shared memory, distributed memory, and cluster memory parallel machines, the flexibility of the distribution of the elements of vectors is also extremely important. This issue is ignored in most proposed standards. In this article we demonstrate how such libraries may be written using data encapsulation techniques.

1 INTRODUCTION

In the 1970s two extremely successful numerical linear algebra software packages, EISPACK and LINPACK, were introduced. They were designed for portability, numerical robustness, and efficiency. They were, however, restricted to dense and banded matrices. The development of serial numerical linear algebra software for dense and banded matrices is greatly simplified by the fact that there are very few natural ways of storing the matrices. Thus, very little effort is needed in designing the data structures used in the codes.

For sparse linear algebra, even on sequential machines, the issues become much more complicated. When one includes various parallel machines, the problems multiply even further. Not only must one make decisions about the storage of the sparse matrices, one must also decide on storage formats for the vectors, since each vector is probably distributed across the parallel processors. We also note that even on sequential machines, the natural storage format for a vector could be dictated by the application. For instance, an adaptive mesh refinement code may represent the solution and other vectors with an octtree data structure.

Software methodologies to overcome these problems do exist; they involve data encapsulation

Received March 1995
Revised February 1996

Scientific Programming, Vol. 5, pp. 329–336 (1996)
CCC 1058-9244/96/040329-08

and object-oriented programming techniques. In object-oriented programming, we abstract out of a data type the actions that we wish to perform on the data, independent of the underlying representation of the data. So, for instance, in the iterative solution of linear systems we need to be able to multiply vectors by sparse matrices and their transposes. In addition, we must be able to perform scalings of vectors, calculate sums of vectors, etc. These operations, not the particular representation of the matrices and vectors, are what define the data. Thus, any storage format, with the corresponding operations defined, should be immediately supported by the software library.

To someone used to prgramming in Fortran 77, this may sound like a pipe dream. It is actually relatively easily achieved in some programming languages. In this article we describe an implementation using C, since many people are familiar with this language and it is portable and available on virtually all machines. Also, it is fairly easy to mix Fortran 77, C, and C++ code in a single application on most platforms.

Note that some people use the term *object oriented* to refer to specifying a data type, operations on that data type, and all of the details of the internal formats (e.g., the sparse matrix format to use). We are using object oriented in a stronger and purer sense: Only the operations are specified. The choice of internal format (and hence, the choice of the actual code to implement the operations) is determined only at run-time rather than at compile-time. This is an important difference; it changes object oriented from being simply a way to organize a code and the argument lists of the routines to a method for flexibly adapting to different situations.

Several publications that discuss these issues in the standard Fortran 77 framework are [1], [2], and [3]. An alternative approach to ours that uses C++ as the implementation language can be found in [4].

2 PROGRAMMER-DEFINED DATA TYPES

In Fortran 77, a limited number of data types are built into the language, essentially scalar integer and floating-point numbers, and dense arrays of integer and floating-point numbers. The language contains no mechanism for the programmer to construct additional data types. Hence, when dealing with higher-level objects such as sparse matrices, the programmer must choose a particular storage format, which, in general, will involve several separate array variables. All of these array variables must be passed to the routines that operate on the spares matrices.

To explain this more fully, we give a particular example, the well-known Yale Sparse Matrix Package (YSMP) storage scheme [5]. In YSMP, the sparse matrix is stored by using four variables: `n`, the size of the matrix; `a`, an array of floating-point numbers that contain the nonzero entries in the matrix; `ia`, an array of integers that contain the locations in `a` of the beginning of each new row; and `ja`, which contains the column number of each entry in `a`. A variation of this storage format is to store the diagonal entries separately in another array, `d`.

If a programmer desired to write a general-purpose iterative solver routine that used the YSMP storage pattern, it could have a calling sequence like `GMRES(n, a, ia, ja, ...)`. But if desired to support both storage formats, something like `GMRES(n, a, ia, ja, d, flag, ...)` would be needed, where the value of a flag indicates which of the two formats is being used. This increases the complexity of the code and makes the addition of a new stoarge format difficult: It may require not only rewriting the `GMRES()` code, but also modifying all of the application codes that use it, since the calling sequence of the `GMRES()` code has been changed.

Other programming languages such as C, C++, and Fortran 90 provide a better and more flexible alternative. The programming is free to introduce new data types, called structures in C and classes in C++. One feature that is useful about these new data types is that pointers to the data may be passed into a routine without the routine needing to know what information they contain and how it is stored. (Fortran 90 has a much more limited construct called derived data types.) In this way the `GMRES()` routine need not know the storage format of the matrix; only the matrix-multiply routine needs to know it. So, for instance, a programmer may introduce a new data type, `Mat`, then write GMRES routines like the following that will support any matrix storage format.

```
int MatrixMultiply(Mat matrix, Vec x, Vec y);
int GMRES(Mat matrix, ... )
{
  ...
   ierr = MatrixMultiply(matrix, x, y);
  ...
}
```

```
typedef struct {
int   (*create)(Vec,Vec*),        /* Routine returns a single vector */
      (*destroy)(Vec),            /* Free a single vector */
      (*dot)(Vec,Vec,Scalar*),    /*  z = x^H * y */
      (*scale)(Scalar *,Vec),     /*  x = alpha * x   */
      (*axpy)(Scalar *,Vec,Vec), /*  y = y + alpha * x */
      .....
} _VectorOps;
```

FIGURE 1 The vector operations structure.

By using a base abstract matrix object, `Mat`, the compiler can still do complete type checking of arguments. If the sparse matrix storage format is changed, only the `MatrixMultiply()` routine must be changed, not the `GMRES()` routine. In fact, we can do even better than this. Rather than hardwiring into the `GMRES()` code the matrix-multiply routine, we can pass a pointer to the matrix-multiply routine into the `GMRES()` routine.

3. OUR APPROACH

Since we would like to support a variety of Krylov-based solvers, we must first determine which vector operations these require. A few of them are the standard Level 1 BLAS operations. Others include routines to generate and free vectors that are needed for temporary or permanent workspace. Since it would be cumbersome to individually pass pointers to all of these routines into the solver routines, we bundle up all of the function pointers and any additional data needed for a particular implementation into a single data type, called a `Vec`. In Figure 1 we give a part of our C structure that defines the vector operations. For those readers who are not fluent in C, this simply defines a data structure whose entries are function pointers. When a function call is needed, the correct function for the particular data storage format is extracted from the data structure and called. Since the function pointers are part of the data structure, the correct function is always called.

All higher-level routines that require access to the vectors act on the vectors only through the pointer, not by directly manipulating the data. The object `Vec` is actually a pointer defined by `typedef struct__Vec* Vec`. The definition of the structure `__Vec` is private to the library and not directly accessible to the application programmer. In this way the library may evolve without requiring any changes to the application codes that rely on it. If the application code had access to the individual data structures in `__Vec`, there would be no data encapsulation.

All of the vector implementations include a pointer to a private, implementation-dependent data structure that may contain the vector length and layout. For a standard serial vector implementation, this can simply be a pointer to an integer containing the length of the vector. For a simple parallel implementation it may be a pointer to two integers, the first containing the length of the part of the vector stored in local memory, the second the length of the entire vector. A sample serial implementation of the `dot()` routine is given in Figure 2.

Currently, our vector structure provides the operations from the Level 1 BLAS, plus the operations $y \leftarrow x + \alpha y$ and $w \leftarrow \alpha x + y$, along with operations to create and free storage for vectors. In Table 1 we list the minimal vector operations we believe must be defined. The pointers to `Scalar` are also unspecified; the indication `Scalar` is there simply to allow type checking of arguments for those languages that support it. These calling sequences will allow the same codes to be used with single precision, double precision, complex, multiple precision, and interval arithmetic. For error handling, all our functions return a nonzero on error and a zero on success.

Remark

Unlike the standard Level 1 BLAS definitions, there is no need to indicate stride information, since the underlying storage format is left up to the particular implementation of the vector operations. Sparse matrix operations may be stored similarly. In addition to the obvious operations such as matrix-vector product and triangular solve, we include such operations as insert and extract row and compute incomplete factorizations. Sparse

```
static int VecDot_Seq(Vec xin, Vec yin, Scalar *z )
{
  Vec_Seq *x = (Vec_Seq *) xin->data, *y = (Vec_Seq *) yin->data;
  int      n = x->n;
  Scalar   sum = 0.0, *xx = x->array, *yy = y->array;
  for ( i=0; i<n; i++ ) {
    sum += xx[i]*yy[i];
  }
  *z = sum;
  return 0;
}
```

FIGURE 2 Sample dot product.

matrices have a similar table, which, to keep this article short, will not be displayed here.

An important feature of the data-hiding approach is that additional operations can be added without disturbing existing code. For example, the operation $w \leftarrow \alpha x + y$ was added when it became apparent that several Krylov methods could make good use of it. The previously coded Krylov space methods did not require any changes. If these routines were passed through argument lists (the only portable mechanism available for Fortran 77 programmers), adding a routine would require modifying each argument list for every routine that used these vector routines.

The only technique available to Fortran programmers that approximates this flexibility is "reverse communication." In this method, for each operation, the library routine sets a flag and returns to the calling program with a request that an operation be performed. However, this method puts the burden on the user, as well as requires a rather unnatural style of programming. In addition, it is difficult to nest routines implemented with reverse communication. For example, if an iterative method, implemented with reverse communication, asks the user to evaluate the preconditioner, which itself makes use of an iterative method (perhaps implementing a block-diagonal preconditioner), implemented with reverse communication, it is the user, not the library, that is responsible for untangling what is happening.

It is extremely important to note that our approach supports both matrix-free as well as out-of-core solvers. In both cases, only the required matrix operations must be provided; no explicit representation of the matrices (or vectors) is needed.

Since the various Krylov-based solvers have

Table 1. Vector Operations

Name	Description	Calling Sequence
VecDuplicate	a vector	Vec in, Vec *out
VecDestroy	a vector	Vec v
VecDuplicateVecs	n vectors	Vec in, int n, Vec **out
VecDestroy Vecs	n vectors	int n, Vec *v
VecDot	$z \leftarrow x^H * y$	Vec x, Vec y, Scalar *z
VecNorm	$z \leftarrow \sqrt{x^H * x}$	Vec x, Scalar *z
VecMax	$z \leftarrow \max(\|x\|)$	Vec x, Scalar *z, int *idx
VecScale	$x \leftarrow \alpha x$	Scalar *α, Vec x
VecCopy	$y \leftarrow x$	Vecx, Vecy
VecSet	$x_i \leftarrow \alpha, \forall i$	Scalar *α, Vec x
VecAXPY	$y \leftarrow \alpha x + y$	Scalar *α, Vec x, Vec y
VecAYPX	$y \leftarrow \alpha y + x$	Scalar *α, Vec x, Vec y
VecSwap	Swap x and y	Vec x, Vec y
VecWAXPY	$w \leftarrow \alpha x + y$	Scalar *α, Vec x, Vec y, Vec w
VecSet Values	$v(idx) = x$	Vec v,int n,int *idx,Scalar *x,int mode

```
for (k=0; k<maxit; k++) {
  VecDot(r, z, &beta);                          /* beta <- r'z     */
  c = beta/betaold; betaold = beta;
  VecAYPX(&c,z,p);                              /* p <- z + c* p   */
  MatMult(ksp->A, p, z );                       /* z <- A*p        */
  VecDot(p, z, &a );
  a = beta/a; ma = -a;                          /* a <- beta/p'z   */
  VecAXPY(&a, p, u );                           /* u <- u + a*p    */
  VecAXPY(&ma, z, r );                          /* r <- r - a*z    */
  VecNorm( r, &rnorm );                         /* rnorm <- ||r||  */
  if (CONVERGED( ksp, rnorm, k)) break;
  PCApply( ksp->B,r, z );                       /* z <- B*r        */
}
```

FIGURE 3 Sample code for preconditioned conjugate gradient loop: Code prior to entering the loop has been omitted.

many optional arguments, we use a context data type, KSP, to store this information as well as the location of the right-hand side and the solution. The KSP has two parts: a public part, which is the same for all Krylov space methods; and a private part, which contains particular options and workspace for each particular Krylov space method. The distinction between the two parts is invisible to the application programmer. The user may also provide optional routines to replace the default convergence tests and optional routines to print out or plot the solution, residual, and error at each iteration; these are also stored in the KSP.

Figure 3 shows an implementation of the inner loop of a preconditioned conjugate gradient. This implementation is portable and works correctly on parallel computers regardless of the distribution of data (all of the difficulty is handled by the specific choices of functions for the vector and matrix operations). In fact, it is taken from the version that we are currently using on both uniprocessors and parallel computers such as the Cray T3D and IBM SP.

Figure 4 gives a code fragment that will allow the solution of a linear system by using the conjugate gradient method, GMRES, Bi-CG-stab, CGS, or two different versions of transpose-free QMR. In the first line, a data structure, `sles`, to contain the control information on the solution process is created. We next set the matrix operator defining the linear system (note that we support matrix-free methods by passing in an abstract matrix object). The next line checks the users command line for solver options and finally the linear system is solved.

```
SLES sles;
Vec  x, b;
Mat  A;
int  its;
/* assemble or define matrix A and vector b */
SLESCreate(&sles);
SLESSetOperators(sles,A,A,0);
SLESSetFromOptions(sles);
SLESSolve(sles,b,x,&its);
```

FIGURE 4 Sample code using Krylov solvers.

The important point is that all of the different methods have the same calling sequences. Optional arguments are passed by calling additional routines, which are ignored if the option is not appropriate. In this way any of the methods in the library may be used without changing the application code at all. In addition, more Krylov space methods may be added to the library without a need for any changes to the application codes. Of course, this flexibility is purchased at a price. Adding a method requires following the object-oriented approach. Further, any matrix vector product or preconditioner provided by the user must conform to the defined calling sequence. But the user may choose any data structure appropriate for his or her application. It has been our experience that the object-oriented design makes this selection relatively easy.

Figure 5 shows the calling sequence for a conjugate gradient algorithm contained in a recent technical report. Within the constraints of Fortran 77 (as a language in which to implement this routine), this is just about the best that can be done. We contend that limiting the design of software to what can be implemented in Fortran 77 severely limits

```
      SUBROUTINE CG(M,DESCRA,AR,IA1,IA2,INFORM,DESCRL,LR,IL1,IL2,DESCRU,
     *              UR,IU1,IU2,DESCRAN,ARN,IAN1,IAN2,DESCRLN,LRN,ILN1,
     *              ILN2,DESCRUN,URN,IUN1,IUN2,VDIAG,B,X,EPS,ITMAX,
     *              ERR,ITER,IERROR,Q,R,S,W,P,PT1,IAUX,LIAUX,AUX,LAUX)
```

FIGURE 5 Calling sequence for a conjugate gradient routine in Fortran 77.

the flexibility and maintainability of the software. However, this limitation does **not** mean that the libraries cannot be implemented in another language and then used from either Fortran 77 or 90. For instance, virtually all aspects of our libraries may be used directly from C, C++, or Fortran.

We also point out that our approach is not intended to duplicate the code in a package such as SPARSKIT [3], but rather to provide an interface that is more flexible and extensible. In fact, we can use carefully crafted implementations of operations involving sparse matrices as the implementation of the operations that we support.

One major concern with object-oriented programming in numerical computing is efficiency. In our approach the "objects" (vectors and matrices) are large grained; this means the OOP overhead is small relative to the time for the numerical computation. Thus, the overall computation time is dictated by the efficiency of the numerical code. In fact, using our package to solve a sparse linear system with direct LU factorization is faster than the Fortran 77 implementation in the YSMP. In Tables 2 and 3 we compare the performance of the direct linear system solver in our package PETSc) to the publicly available YSMP for solving nonsymmetric linear systems using LU factorization and a nested dissection ordering. The first problem is from an industrial oil reservoir simulator and contains 1,501 unknowns and 26,131 nonzeros. The second is from a three-dimensional compressible flow simulation with 15,360 unknowns and 496,000 nonzeros. Runs were made on a Digital Alpha workstation and on an IBM RS6000/370. Times are given in seconds.

The columns *Default* and *Basic* indicate the PETSc default I-node version (a version that takes advantage of rows with identical nonzero structure) and basic version, respectively. Note that the basic version's performance is virtually the same as that from the Fortran 77 YSMP code. Our alpha workstation did not have enough memory to perform the factorization on the larger matrix.

We have chosen the C programming language for our software libraries for a variety of reasons. It is simply not possible to perform true data encapsulation in Fortran 77 or Fortran 90. In addition, the various object-oriented languages such as Smalltalk and Eiffel are too far from the mainstream of scientific computing to be considered. C++ was rejected because it is a moving target. Code that compiles with one compiler will not compile with another; even slightly different generations of the compiler handle very different aspects of the C++ language. We work in an environment where we must maintain robust, high-quality code for a large variety of machines. We can do this in C; and since we can do true data encapsulation and polymorphism in C while supporting users who program in both Fortran and C++, C is clearly the language of choice for our libraries. In many numerical applications and libraries, C++ may be the most appropriate choice.

4 RECOMMENDATIONS

Some readers may object that the object-oriented approach merely hides the fact that users must still write the routines to perform the vector operations and the matrix-vector operations. To some degree this objection is correct. The power of the object-oriented approach is that once the vector and matrix-vector routines are written, they need not be touched, or even understood, to write a new Kry-

Table 2. Oil Reservoir Simulation

	PETSc		
Machine	Default	Basic	YSMP
Alpha	.45	.53	.60
RS6000	.57	.69	.72

Table 3. Compressible Flow Simulation

	PETSc		
Machine	Default	Basic	YSMP
RS6000	112	162	161

lov-based solver that utilizes them. The converse is also true: One need never rewrite the Krylov-based solvers again when a new architecture comes along. As soon as the vector and matrix-vector operations are provided, the Krylov-based solvers will automatically work on that machine—and as efficiently as the underlying operators.

As an example of the flexibility that this approach gives, we mention one of our applications, a magnetostatics code that solves a large, dense linear system in its inner loop. We wished to use iterative methods instead of direct methods to solve this problem. To do this, we simply introduced a new sparse matrix format called "dense." This format uses the same matrix storage that the application is using, and uses Level 2 BLAS for matrix-vector operations (thus providing good efficiency). We were then able to use all of our iterative routines without change. The same approach was used for the parallel version of this application [6].

Another example is in the EAGLE code [7] for external two- and three-dimensional fluid dynamics. In this code, a linear system must be solved within the inner loop. However, the matrix is represented implicitly as coefficients on a grid. The conventional approach to interfacing this code to a solver package is to reformat the matrix into some explicit representation, such as the YSMP format. With our package, we simply added a new sparse matrix type, "Eagle," that is defined by the grid coefficients and a few operations. This simplified the task of using our package in an existing application. Perhaps more importantly, it minimized the amount of additional memory needed, since we did not have to make a separate copy of the matrix elements. Both of these applications codes are written in Fortran 77, demonstrating that the advantages of true object-oriented design can be made available to Fortran users.

We make the following recommendations for the design of truly data-structure-neutral libraries:

1. Do not design the interface based on the limitations of the target language. Just because you cannot implement an interface in Fortran does not mean that you cannot provide that interface to Fortran programmers.
2. Do not assume any particular format in the data structures. Do not assume that vectors are contiguous in computer memory (this is not true even in many serial applications codes).
3. Design the interface so that routines that solve the same problem in different ways are perfectly interchangeable. This approach maximizes the upward compatibility of adding new algorithms.
4. Remember that data-structure-neutral does not mean that the format of the matrix is unspecified: it means specifying vectors and matrices and other objects by the operations that are performed on them in such a way that you can operate on them without knowing their internal structure.
5. Choose the operations carefully so that they can be implemented efficiently. Often this means providing aggregate operations, such as one to set many elements in a matrix, rather than only providing an operation that acts on a single element.
6. Provide implementations of the operations for at least several interesting data structures. For example, our library implementation includes several kinds of sparse matrix formats as well as a dense matrix format.

Developing the codes initially takes slightly longer than writing use-once, data-structure-dependent codes, but the payoff in code reuse more than compensates. Our codes that use these techniques are available via anonymous ftp from the site `info.mcs.anl.gov` in the directory `pub/petsc`. (We will support "double" and "double complex" as the Scalar.) These routines are callable from C, C++, and Fortran 77 (and from Fortran 90 using the Fortran 77 interface). The linear solvers are part of a larger set of tools, PETSc 2.0 (Portable, Extensible Tools for Scientific computing), that we have been developing. The user's manual for PETSc Version 2.0, [8] is also available at the ftp site. In addition, an overview of PETSc may be obtained via the WWW at `http://www.mcs.anl.gov/petsc/petsc.html`.

ACKNOWLEDGMENTS

The work of the first author was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 while the author was at Argonne National Laboratory, and by the Office of Naval Research under contract ONR N00014-90-J-1695 while the author was at the Department of Mathematics, University of California at Los Angeles. The work of the second was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. We thank Rick Dean of Arco for providing the oil reservoir simulation matrix, Lois Curf-

man McInnes for providing the compressible flow matrix, and Satish Balay for providing the I-node code used in the numerical comparison.

REFERENCES

[1] I. S. Duff, M. Marrone, and G. Radicati, "A proposal for user level sparse BLAS," Tech. Rep. TR/PA/92/85, CERFACS, 1992, SPARKER Working note #1.

[2] T. C. Oppe and D. R. Kincaid, "Are there iterative BLAS?" *Int. J. Sci. Comp. Modeling* (in press).

[3] Y. Saad, "SPARSKIT: A basic toolkit for sparse matrix computations," Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Tech. Rep. 1029, Aug. 1990.

[4] A. M. Bruaset and H. P. Langtangen, "Object oriented design of preconditioned iterative methods," Sintef, Oslo, Norway, Tech. Rep. STF33 A94036.

[5] S. C. Eisenstat, H. C. Elman, M. H. Schultz, and A. H. Sherman, The (new) Yale Sparse Matrix Package, Department of Computer Science, Yale University, Tech. Rep. YALE/DCS/RR-265, Apr. 1983.

[6] L. Kettunen, K. Forsman, D. Levine, and W. Gropp, "Computational electromagnetics and parallel dense matrix computations," in *Proc. of the SIAM Parallel Processing for Scientific Computing Conference, 1995.*

[7] J. S. Mounts, D. M. Belk, and D. L. Whitfield, "Program EAGLE user's manual, vol. IV: Multiblock implicit, steady-state Euler code," Air Force Armanent Laboratory (AFATL), Eglin Air Force Base, Florida, Tech. Rep. TR-88-117, Sept. 1988.

[8] S. Balay, W. Gropp, L. Curfman McInnes, and B. Smith, "PETSc 2.0 user's manual, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, Tech. Rep. ANL-95/11.