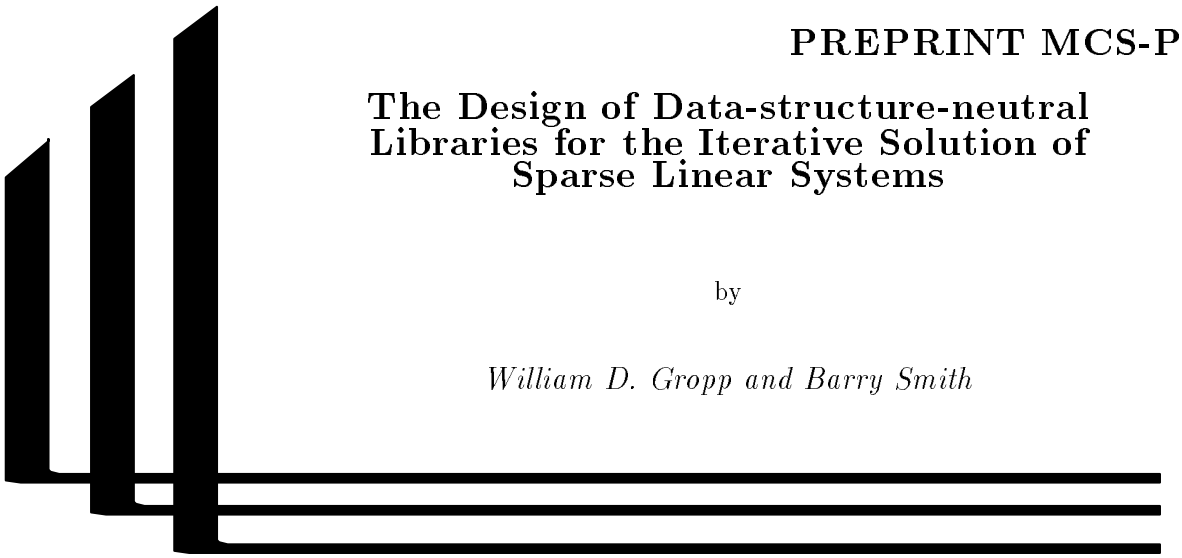

PREPRINT MCS-P356-0393

The Design of Data-structure-neutral Libraries for the Iterative Solution of Sparse Linear Systems

by

William D. Gropp and Barry Smith

March 1993

Mathematics and Computer Science Division
Argonne National Laboratory

THE DESIGN OF DATA-STRUCTURE-NEUTRAL LIBRARIES FOR THE ITERATIVE SOLUTION OF SPARSE LINEAR SYSTEMS

BARRY F. SMITH * WILLIAM D. GROPP†

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 feel 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 since, 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.

Key words. Krylov space methods, software libraries, sparse linear systems

AMS(MOS) subject classifications. 65F10, 65F50, 68N05

1 Introduction

In the 1970s two extremely successful numerical linear algebra software packages, EISPACK and LINPACK, were introduced. They were designed for portability, generality, 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

*Department of Mathematics, University of California, Los Angeles, CA 90025-1555. Electronic mail address: bsmith@math.ucla.edu . This work 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 the Argonne National Laboratory, and by the Office of Naval Research under contract ONR N00014-90-J-1695.

†Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4844. Electronic mail address: gropp@mcs.anl.gov . This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

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 should 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 hiding and object-oriented programming techniques. In object-oriented programming one abstracts out of a data type the actions that one wishes to perform on the data, independent of the underlying representation of the data. So, for instance, in the iterative solution of linear systems one needs to be able to multiply vectors by sparse matrices and their transposes. In addition, one 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 programming in Fortran 77, this may sound like a pipe dream. It is actually relatively easily achieved in some programming languages. In this paper 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 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 (for example, 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.

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 sparse matrices.

To explain this more fully, we give a particular example, the well-known Yale Sparse Matrix Package (YSMP) storage scheme [2]. 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 one desired to write a general-purpose iterative solver routine that used the YSMP storage pattern, it could have a calling sequence like `CG(n, a, ia, ja, ...)`. But if one desired to support both storage formats, one would need something like `CG(n, a, ia, ja, d, flag, ...)`, where the value of flag indicates which of the

two formats is being used. This increases the complexity of the code and makes the addition of a new storage format difficult: it may require not only rewriting the `CG()` code, but also modifying all of the application codes, since the calling sequence of the `CG()` code has been changed.

Other programming languages such as C, C++, and Fortran 90 provide a better and more flexible alternative. The programmer is free to introduce new data types, called structures in C, classes in C++, and derived data types in Fortran 90. 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. In this way the `CG()` routine need not know the storage format of the matrix; only the matrix-multiply routine needs to know it. So, for instance, one may introduce a new data type, `SparseMatrix`, then write conjugate gradient routines like the following that will support any matrix storage format.

```
void MatrixMultiply(SparseMatrix *, Vector *, Vector *);
void CG(void *matrix, ... )
{
  ...
  MatrixMultiply(matrix, x, y);
  ...
}
```

For those not familiar with C's void type, this simply means that the data type is unspecified. If the sparse matrix storage format is changed, only the `MatrixMultiply()` routine must be changed, not the `CG()` routine. In fact, we can do even better than this. Rather than hardwiring into the `CG()` code the matrix-multiply routine, we can pass a pointer to the matrix multiply routine into the `CG()` 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. Some of these are the standard BLAS 1 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 vector context, `VectorContext`. In Fig. 1 we give a part of our C structure that defines the `VectorContext`. All higher-level routines that require access to the vectors act on the vectors only through the vector context, not by directly manipulating the data.

All of the vector routines take, as their first argument, a pointer to a private, implementation-dependent data structure that may contain the vector length, layout, etc. 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 Fig. 2. Associated with each vector operation is a macro to simplify its use; for instance, to use the vector `dot()` operation one may use `VDOT(vp, N, x, y, result)`.

```
typedef struct {
void *(*create_vector)(),   /* Routine returns a single vector */
     (*destroy_vector)(),   /* Free a single vector */
     (*dot)(),              /*  z = x^H * y */
     (*scale)(),            /*  x = alpha * x   */
     (*axpy)(),             /*  y = y + alpha * x */
     .....
} VectorContext;
```

Figure 1: The Vector Context

```
void DVdot( int *N, double *x, double *y, double *result )
{
  int i, n = *N; double sum = 0.0;
  for (i=0; i<n; i++ ) { sum += x[i] * y[i]; } *result = sum;
}
```

Figure 2: A Sample Dot Product

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 feel must be defined. Note that the first argument is defined to be a pointer to `void`. The pointers to `Scalar` or `Vector` are also unspecified; the indication `Scalar` or `Vector` 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, interval arithmetic, etc.

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 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 requiring a rather unnatural style of programming.

Since the various Krylov-based solvers have many optional arguments, we use a context data type, `IterativeContext`, to store this information as well as the location of the right-hand side, solution, etc. The `IterativeContext` has two parts, a

Table 1: Vector Operations

Name	Description	Calling Sequence (first argument is always `void *N`)
Create	a vector	
Destroy	a vector	Vector *v
Obtain	n vectors	int n
Release	n vectors	int n, Vector **v
Dot	$z \leftarrow x^H * y$	Vector *x, Vector *y, Scalar *z
Norm	$z \leftarrow \sqrt{x^H * x}$	Vector *x, Scalar *z
Max	$z \leftarrow \max(\|x\|)$	Vector *x, Scalar *z, int *idx
Scale	$x \leftarrow \alpha x$	Scalar *α, Vector *x
Copy	$y \leftarrow x$	Vector *x, Vector *y
Set	$x_i \leftarrow \alpha, \forall i$	Scalar *α, Vector *x
AXPY	$y \leftarrow \alpha x + y$	Scalar *α, Vector *x, Vector *y
AYPX	$y \leftarrow \alpha y + x$	Scalar *α, Vector *x, Vector *y
Swap	swap x and y	Vector *x, Vector *y
WAXPY	$w \leftarrow \alpha x + y$	Scalar *α, Vector *x, Vector *y, Vector *w

public part which is the same for all Krylov space methods, and a private part which contains particular options, workspace, etc., 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, error, etc., at each iteration; these are stored in the `IterativeContext`, as well. The `IterativeContext` also contains a `VectorContext` for use with operations on the vectors.

In Fig. 3 we show 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 Intel DELTA and BBN TC2000.

In Fig. 4 we give a code fragment that will allow the solution of a linear system using the conjugate gradient method, GMRES, Bi-CG-stab, CGS, or two different versions of transpose-free QMR. The first line currently configures the code for the GMRES method. 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, the routines `amult()` and `binv()` must be provided by the user, and they must also conform to the implementation, though normally a library would provide several default implementations. It has been our experience that the object-oriented design makes this relatively easy to achieve.

In Fig. 5 we give the calling sequence for a conjugate gradient algorithm contained in working notes for a proposed sparse BLAS standard [1]. Within the constraints of

```
for (k=0; k<maxit; k++) {
  VDOT( vp, N, r, z, &beta);                    /* beta <- r'z     */
  c = beta/betaold; betaold = beta;
  VAYPX(vp,N,c,z,p);}                           /* p <- z + c* p   */
  MULT( itp, N, p, z );                         /* z <- A*p        */
  VDOT( vp, N, p, z, &a );
  a = beta/a;                                   /* a <- beta/p'z   */
  VAXPY( vp, N, a, p, u );                      /* u <- u + a*p    */
  VAXPY( vp, N, -a, z, r );                     /* r <- r - a*z    */
  VNORM( vp, N, r, &rnorm );                    /* rnorm <- ||r||  */
  if (CONVERGED( itp, N, rnorm, k)) break;
  PRE( itp, N, r, z );                          /* z <- B*r        */
}
```

Figure 3: Sample Code for PCG Loop: `vp` is the vector context; `itp` is the iterative method context.
Code prior to entering the loop (e.g., setting `betaold`) has been omitted.

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 (as opposed to *used* from Fortran 77) severely limits the flexibility and maintainability of the software.

We also point out that our approach is not intended to duplicate the code in a package such as SPARSKIT [5] 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.

4 Recommendations

Some readers may object that the object-oriented approach merely hides the fact that one 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 Krylov-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.

Another example is in the EAGLE code [4] for external two- and three-dimensional

```
IterativeContext    *itp;            ITMETHOD  itmethod = ITGMRES;
Vector              *x,*b;           int       its, n = 50;
void                amult(), binv();

itp = ITCreate( itmethod );     /* Choose the method */
DVSetDefaultFunctions( itp->vc );  /* set default vector functions */
x = VCREATE( itp->vc, &n ); /* generate space for solution and rhs */
b = VCREATE( itp->vc, &n );
VSET( itp->vc, &n, 1.0, b );    /* set right hand side to ones */
ITSetAmult( itp, amult );       /* set routine for matrix multiply */
ITSetBinv( itp, binv );         /* set routine for preconditioner */
ITSetSolution( itp, x );
ITSetRhs( itp, b );
ITSetUp( itp, &n ); its = ITSolve( itp, &n ); ITDestroy( itp, &n );
```

Figure 4: Sample Code Using Krylov Solvers

```
 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

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 important, 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 solutions (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.

Developing the codes initially takes slightly longer than writing one-use, 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/pdetools`. (We support only "double" as the Scalar, mostly for reasons of limited resources.) These routines are callable from both C and Fortran 77. The linear solvers are part of a larger set of tools, PETSc (Portable, Extensible Tools for Scientific computing) that we have been developing. We have written a set of linear solvers built on top of these libraries called the "Simplified Linear Equation Solvers," SLES. Its user guide [3] is in the file `sles.ps.Z`. SLES is intended as a easy to use, serial interface to the much more powerful underlying routines. We have found this approach to be very easy to use, and we use it both with our research codes and with large computational science applications.

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

[3] W. D. Gropp and B. Smith, *Simplified linear equation solvers users manual*, Tech. Rep. ANL-93/8, Mathematics and Computer Science Division, Argonne National Laboratory, Mar. 1993.

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

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