SIAM J. MATRIX ANAL. APPL.
Vol. 20, No. 4, pp. 953–969




# AN OBJECT-ORIENTED APPROACH TO THE DESIGN OF A USER INTERFACE FOR A SPARSE MATRIX PACKAGE*

ALAN GEORGE† AND JOSEPH LIU‡

**Abstract.** The authors designed and implemented a sparse matrix package called Sparspak in the late 1970s. One of the important features of that package is an interface which shields the user from the complicated calling sequences common to most sparse matrix software. The implementation of the package was challenging because the relatively primitive but widely available Fortran 66 language was used. Modern programming languages such as Fortran-90 and C++ have important features which facilitate the design of flexible and "user-friendly" interfaces for software packages. These features include dynamic storage allocation, function name overloading, user-defined data types, and the ability to hide functions and data from the user. This article describes the redesign of the Sparspak user interface using Fortran-90 and C++, outlining the reasons for its various features and highlighting similarities and differences in the features and capabilities of the two languages. The two new implementations of Sparspak have been named Sparspak-90 and Sparspak++.

**Key words.** sparse matrix software, object-oriented numerical software, user interfaces

**AMS subject classifications.** 65F10, 65F50

**PII.** S0895479897317739

## 1. Introduction.

Sparspak is a sparse matrix package that was designed and implemented by the authors in the late 1970s.[1] One of the important features of the package is a user interface which shields the user from the complicated calling sequences common to most sparse matrix software available at that time. A description of the interface together with motivation for its design can be found in [4].

This paper describes the design and some general implementation issues of the new version of Sparspak. The software is being implemented in C++ and Fortran-90, and the two implementations are referred to as Sparspak++ and Sparspak-90 throughout this paper. An "object-oriented" approach to the design has been adopted, reflecting widely accepted software engineering practice [1, 8].

Software for solving sparse systems of equations involves relatively complicated data structures which are not provided as standard data types in the language in which the software is implemented. Solving a sparse system of equations usually consists of a number of individual steps typically involving different types of data structures; sometimes data used at one step can be discarded at the end of it and the storage released for later use. Sometimes the amount of storage required is not known until at least some of the computation has been completed.

When Sparspak was implemented, Fortran 66 and its successor, Fortran 77, were used almost exclusively for scientific computing. Neither language provides user-defined data types nor dynamic storage allocation. Consequently, the subroutines and functions implementing sparse matrix software tend to have long argument lists,

*Received by the editors February 27, 1997; accepted for publication (in revised form) by E. Ng January 30, 1998; published electronically July 9, 1999. This work was supported by Natural Sciences and Engineering Research Council of Canada grants OGP 000811 and OGP 005509.
http://www.siam.org/journals/simax/20-4/31773.html

†Department of Computer Science, University of Waterloo, Waterloo, ON, N2L 3G1, Canada (jageorge@sparse.uwaterloo.ca).

‡Department of Computer Science, York University, Toronto, ON, M3J 1P3, Canada (joseph@cs.yorku.ca).

[1]Dr. Esmond Ng, currently at the Mathematical Sciences Section, Oak Ridge National Laboratory, collaborated with the authors on a second release of the package in 1984.

most of which have little or no relevance for the user of the software. One of the main roles of the interface is to provide the user with simple subroutines whose argument lists, to the extent possible, include only information specific to the problem; that is, information that the user would inevitably know. The interface provides for the allocation of storage and creation of data structures using the more primitive data types available in the language. In addition, the interface ensures that its subroutines are called in the correct order, provides uniform error message handling, and collects timing and storage statistics.

There are several motivations for revisiting the interface design. First, Sparspak has been in use for more than 15 years, and the users of the package have provided useful feedback concerning both strong points and shortcomings of the interface. In addition, Fortran has evolved substantially, so that the current standard, Fortran-90 [7], contains modern programming language features that make implementation of a user interface far more convenient and effective. Collaterally, Fortran is no longer the exclusive choice of scientific programmers; C [6] and C++ [11] are becoming increasingly popular for implementing scientific software. Both of these languages have features that are desirable in connection with implementing an interface for a sparse matrix package. Finally, experience suggests that the package should cater to a broader audience. When Sparspak was designed, it was assumed that its users would be primarily engineers and scientists whose main interest would be solving sparse problems arising in their applications. That is, it would serve as a utility in various scientific applications and application packages. However, the package also has been used extensively by researchers in sparse matrix computation. Thus, there is a motivation to redesign it so that it can also serve as a research test bed for experts in the field of sparse matrix computation. A corollary of this objective is that the interface should be flexible enough to support techniques for dealing with a wide spectrum of sparse matrix problems. The original package was restricted to positive definite problems, and to least squares problems which could be solved using techniques largely adapted from those used for positive definite problems. This restriction simplified the design of the user interface.

An outline of the paper follows. Section 2 contains a brief review of the basic steps that are performed in connection with solving large sparse systems of equations, along with various contexts in which sparse systems are solved. This establishes the capabilities that one would expect to find in a reasonably comprehensive sparse matrix solver, *regardless of whether it even has an interface.* This in turn provides a framework for identifying the features that we believe an ideal user interface should possess in order that users, sophisticated and otherwise, are able to conveniently exploit the capabilities of the package. Section 3 describes the basic elements of the design of the package along with a description of the major *objects* that are integral parts of it. Section 4 contains some examples of how the package might be used, with particular focus on how users with varying levels of knowledge or sophistication can use the package. This section provides a context in which to demonstrate how the design objectives from section 2 are achieved, with comments about how the design, together with various programming language features, promotes that objective. The last section contains concluding comments about how various programming language features aid in the implementation of the design, drawing attention to the similarities and differences between C++ and Fortran-90, at least with respect to this application.

**2. Design considerations.** Loosely speaking, a user interface is something that allows a user to access the capabilities of a system. In the present context, the interface

is simply a layer of software which shields the user from the complications associated with sparse matrix software, yet allows one to use that software in a natural and convenient way to solve sparse systems of equations. Naturally, the design of the interface is conditioned by the capabilities of the underlying sparse matrix software, so an essential first step is to describe what is assumed to be those capabilities.

For definiteness, the problem to be solved will be denoted by

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{A}$ is an $n \times n$ sparse coefficient matrix, and the method to be used is Gaussian elimination.[2] A triangular factorization of the matrix is computed, followed by the solution of two triangular systems in order to obtain the solution $\boldsymbol{x}$.

There is no general "best method" for solving sparse systems of equations. Even if one restricts the basic algorithm to Gaussian elimination, the way it is best implemented often depends on the characteristics of the given sparse linear system. Therefore, a sparse matrix package should accommodate a variety of methods and allow for convenient inclusion of methods yet to be developed. This should be possible with minimal disruption to the interface.

Sparse systems arise in a variety of contexts. Sometimes many problems having the same structure must be solved, and sometimes many problems differing only in their right-hand sides must be solved. Also, the way in which their structure and numerical values become available is highly variable. The package should be able to handle these situations efficiently and the interface should make it convenient and natural for the user to exploit that capability.

The discussion above suggests the capabilities that one would expect in a reasonably comprehensive sparse matrix package. In particular, there would be software available to handle sparse positive definite systems, sparse symmetric indefinite systems, and general sparse unsymmetric systems. These include software for ordering algorithms, software for creating appropriate data structures, and software for implementing the actual numerical routines. In what follows, we regard the software that does all these tasks, together with the user interface, as the *package*.

In general terms, the interface should support the following objectives:

- *Usability.* The intellectual overhead in learning to use the package should be low.
- *Versatility.* The package should be flexible; that is, it should be convenient to use in a wide variety of situations.
- *Layered accessibility.* The package should serve users having different levels of expertise in sparse matrix computation, ranging from the casual user to the sparse matrix researcher.
- *Extensibility.* The package should be designed so that it can be extended easily to new methods as they are developed and also to other classes of problems, such as sparse least squares problems and sparse nonlinear systems of equations. Ideally, such extensions should cause minimal or no disruption to the basic structure of the interface.
- *User control.* The interface should provide useful feedback, and the amount of such feedback should be under the control of the user.

[2] The authors' ultimate objective is to provide a package which deals with more general problems, such as over- and underdetermined systems. Solving such systems may involve algorithms other than Gaussian elimination, such as orthogonal factorization. However, for purposes of presentation, we restrict our attention to square systems, since their requirements are diverse enough that meeting the needs of these problems also provides the flexibility required to deal with more general classes.

**3. Basic elements of the design.** This section describes some of the essential ingredients of the package, with particular emphasis on its user interface. This provides a context in which to explain how many of the design objectives outlined in the previous section have been achieved.

**3.1. C++ and Fortran-90 classes.** For illustration purposes, some Fortran-90 source codes are included with the description. However, C++ examples could have as easily been included, and the discussion is largely language-independent.[3] The main features used to achieve the design objectives are available in both languages, although the details differ somewhat.

One of the key features used is the ability to encapsulate data structures and functions or subroutines that act upon them together as a single entity. In C++ these are *classes* and each class contains member variables and member functions.[4] Fortran-90 does not really have an analogue of a C++ class. It supports the feature of derived (user-defined) data types. However, unlike member functions in C++ classes, there is no facility to explicitly bind functions or subroutines to a data type.

In Sparspak-90, *modules* are used to support features similar to classes in C++. A Fortran-90 module can contain declarations of derived data types and procedures. A programming style has been adopted in Sparspak-90 such that a module contains a derived type together with routines that act on that data type. In other words, a logical association of a set of routines and a derived type is provided through membership in a module. Furthermore, the binding of these routines with the derived type can be achieved in a natural way by including an object of the derived type as an argument in each of the routines. This organization allows one to treat (and think about) Fortran-90 modules and C++ class definitions in the same way.

In what follows, the term "class" will be used. For C++ programmers, the meaning will be immediate. For Fortran-90 programmers, this should be interpreted as a module that defines a data type, together with a collection of routines that act on instances of that data type.

Instances of a class are called *objects*. A program may have several objects of the same class having different names. From the perspective of a typical user, use of the package involves creating and using two types of objects: **Problem** objects and **Solver** objects. These are described in the sections that follow.

**3.2. The problem class.** Regardless of the level of user sophistication, one task is fundamental and ubiquitous: the user must communicate the sparse matrix problem to the package, and the user interface should make this as convenient as possible. The task is complicated by the variety of ways in which the problem may materialize, as well as by transformations that the user might want to apply to the input. The different ways include the following:

- The structure of the problem and its numerical values may become available simultaneously or at differing times.
- There may be many systems to be solved, differing only in their numerical values.
- There may be numerous problems that differ only in their right-hand sides; these may be available all at once, or each may be the result of computations involving previous right-hand side(s) in a sequence. The latter circumstance

[3] The corresponding C++ examples can be found online at http://www.cs.yorku.ca/~joseph/.

[4] Member variables are sometimes referred to as instance variables, and member functions are sometimes referred to as "methods."

arises naturally if the package is being used in connection with solving a system of nonlinear equations.

- Given the matrix $\boldsymbol{A}$, it may be desirable to generate a right-hand side corresponding to a given solution or, given the structure of $\boldsymbol{A}$, it may be desirable to assign numerical values giving $\boldsymbol{A}$ certain properties (random, symmetric, positive definite, diagonally dominant, etc.). Such capabilities can be useful in developing and testing software.

The list above is far from exhaustive but serves to illustrate the variety of situations which the package should be able to handle. With these considerations in mind, it seems natural and compelling that the package should contain a class that can be viewed as the "problem repository." This class is given the name **Problem**, since objects of this class contain the numerical values and structure of the coefficient matrix, its right-hand side(s), corresponding solution vectors, and related information pertaining to the problem.

The class **Problem** has various member routines which operate on its data. Roughly speaking, these routines fall into four categories:

1. Procedures which provide for input of structural and numerical values, such as **InAij**, **InRow**, **InColumn**, and **InRhs**.
2. Procedures which adjust the input, such as **ReplaceAij**, **MakeGSymmetric**, and **MakeASymmetric**. (The latter two procedures make the problem structurally and numerically symmetric, respectively.)
3. Procedures which retrieve and/or display information, such as **GetRhs**, **GetSolution**, and **PrintSolution**.
4. Procedures which provide information about the problem, such as **IsASymmetric**, **IsGSymmetric**, and **IsAijPresent**. The first two procedures determine whether the matrix is numerically and structurally symmetric, respectively, while the latter two determine whether the $ij$th element of the matrix is present.

An important aspect of the **Problem** class is its general purpose design. It can represent sparse matrix problems with square or rectangular matrices, with symmetric or unsymmetric structures, and with symmetric or unsymmetric numerical values. The general nature of its design allows easy extension of the package in the future to handle more general types of sparse problems, thus promoting one of the design objectives, namely, that the package should be easily extendible.

In order to provide concreteness to the description above, Figure 1 contains a Fortran-90 subroutine which generates an $n \times n$ tridiagonal matrix problem. The example provides an initial opportunity to elaborate on several design features of Sparspak++ and Sparspak-90. Extensive use is made of the programming language feature known as *function name overloading*, which is available in both Fortran-90 and C++. That is, different routines are allowed to have the same names, provided that their parameter lists ("signatures") differ. The compiler detects which routines should be called by matching up the types and number of parameters in the routines. Thus, routines which perform essentially the same role, even though they may employ different internal data structures or operate on different objects, can still have the same name. This name overloading capability is an important way in which the intellectual overhead in learning and using the package is reduced.

Referring to the example in Figure 1, an instance of the problem class is initialized by calling the function **Construct**. *All* objects in the package are initialized by calling

```
subroutine MakeTriDiagProblem( p, n )
    use Sparspak90
    integer :: n, i
    type (Problem) :: p

    call Construct(p, "TriDiagProblem")
    do i = 1, n-1
        call InAij(p, i+1, i, -1D0); call InAij(p, i, i, 4D0)
        call InAij(p, i, i+1, -1D0); call InRhs(p, i, 1D0)
    end do
    call InAij(p, n, n, 4D0); call InRhs(p, n, 1D0)
end subroutine MakeTriDiagProblem
```

FIG. 1. *Tridiagonal test problem generator.*

a function whose name is **Construct**.[5] Of course, the routine that is actually executed will depend on the type of the first member of the argument list (which the compiler can determine). The key point is that the user has to remember only one function name in connection with creating new objects, regardless of their type or class.

Another example demonstrating the convenience of overloading is illustrated by the function **InAij** in the example in Figure 1. As noted in the previous section, the structure and the numerical values of the system of equations can arrive at different times and in different aggregations. If the user does not know the value of $a_{ij}$ but wishes to communicate the fact that the $ij$th element of $\boldsymbol{A}$ is nonzero, the function **InAij** is still used but the last parameter is omitted.[6] Analogously, overloaded input routines **InRow**, **InColumn**, and **InSubMatrix** are available in the event that the nonzeros (or perhaps only their positions in the matrix) become available by rows or columns or as submatrices. Similar remarks apply to **InRhs**. The right-hand side $\boldsymbol{b}$ can be input one element at a time, as shown in the example in Figure 1, or as a subvector with an accompanying list of subscripts, or all at once. In all cases a function with the same name is called. Thus, the user must remember only a small number of function names to use **Problem** objects.

The call to the routine **Construct** in the example in Figure 1 illustrates another design feature. All objects that are created during the use of Sparspak are given names so that if they later generate error messages or other information, they can "identify themselves." Each class has a character string member variable called **objectName**, and the printing of object information always includes its **objectName** for identification. Since there may be numerous objects of the same type present in an executing program, this self-identification is important in helping the user understand output from his or her program. The explicit naming of each object created is optional; if the user omits the parameter, a name for the object is created automatically. Each class has a function **ReName** to make it possible for the user to change the name of an object if that is desirable. The setup allows users to assign unique names to objects for identification.

[5] This statement and the rest of the paragraph applies only to Sparspak-90. In C++, *constructors*, which play the same role as **Construct**, are called automatically whenever an object of a class is instantiated.

[6] Fortran-90 supports the notion of optional arguments, so this feature can be achieved in Fortran-90 by using an optional argument rather than subroutine name overloading.

```
program SimpleExample
    use Sparspak90
    type (Problem) :: p
    type (SparseSpdSolver) :: s

    call MakeTriDiagProblem(p, 5)   ! create test problem
    call Construct(s, p)            ! create solver object
    call Solve (s, p)               ! instruct s to solve p
    call PrintSolution(p)           ! print the solution
    call Destruct(p); call Destruct(s)  ! release storage
end program SimpleExample
```

FIG. 2. *Simple use of the package.*

Another noteworthy feature is that each class has standard member procedures **Save** and **Restore** for saving and restoring the contents of an object to and from an external file.

**3.3. The solver class.** The second major type of object that the typical user of the package will employ is a "solver" object. Loosely speaking, a solver object accepts a problem object as input and produces a solution to the problem. The package contains numerous different "solvers" for sparse systems of equations. That is, in addition to a class of type **Problem** in the package, there are solver classes, each consisting of a particular data structure, ordering algorithm, and numerical factorization and substitution routines; together, these implement a particular overall approach to solving a sparse system. For example, for symmetric positive definite systems, there are several effective algorithms for finding a low fill ordering; there are several efficient methods for storing Cholesky factors; and there are several efficient ways of implementing the factorization using the same data structure (left-looking, right-looking, multifrontal [3, 5, 9]). Various solvers result from selecting different combinations of these options.

The multitude of solvers is necessary because problems vary in several dimensions. They may or may not be square; if square, they may or may not be structurally symmetric. If they are structurally symmetric, they may or may not be numerically symmetric. Regardless of either shape or symmetry, row and/or column interchanges may be necessary to ensure numerical stability. In addition, for any particular combination of problem attributes above, there may be more than one approach that will solve the problem. Of course, a solver that assumes no special features will cope with them all but not as efficiently as one that exploits special features that a problem may possess.

**3.4. Coarse structure of Sparspak.** At a low (coarse) level of resolution, the package can be regarded as consisting of just two fundamental types of classes, namely, **Problem** and **xxxSolver**, where **xxx** denotes one of the numerous possibilities mentioned above. For example, Sparspak contains a solver for symmetric positive definite problems that reorders the problem to reduce fill. This solver class has the name **SparseSpdSolver**, standing for Sparse Symmetric positive-definite Solver. A simple example showing its use is given in Figure 2, where the subroutine in Figure 1 is used to create a small symmetric positive definite tridiagonal problem.

The package also contains a solver for symmetric positive definite problems that orders the problem so that it has a small envelope. This solver class has the name **EnvSpdSolver**, which stands for Envelope-reducing Symmetric positive definite Solver.

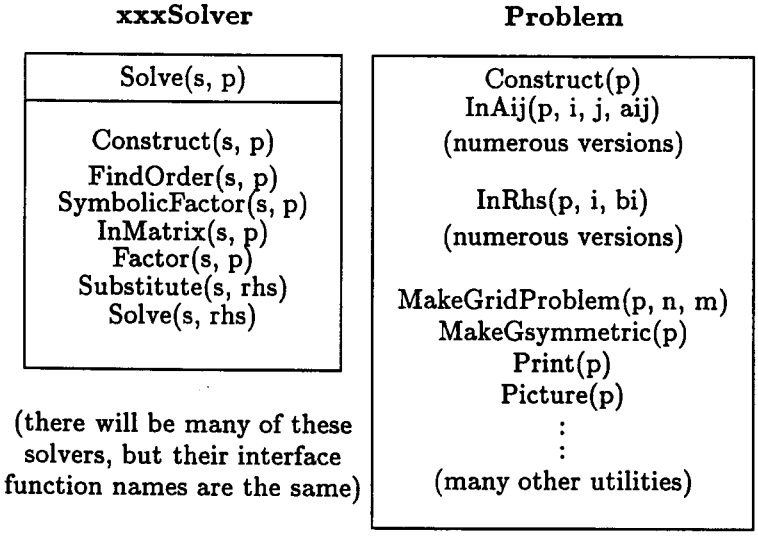


FIG. 3. *Coarse structure of Sparspak.*

The only change necessary in the program in Figure 2 in order to use this solver would be in line 4, where **SparseSpdSolver** would be changed to **EnvSpdSolver**. The use of function name overloading for **Solve** and **Destruct** means that no other changes are required; of course different procedures will be invoked. The compiler detects which procedures should be called by matching up the types and numbers of parameters in the procedures.

"Behind the scenes," **Solve** invokes subroutines which carry out the four standard steps of solving a sparse positive definite system. These subroutine names are also overloaded; they are invoked by the same names, regardless of the actual sparse positive definite system solver class being used:

1. **FindOrder**: find an appropriate ordering;
2. **SymbolicFactor**: symbolic factorization;
3. **Factor**: numerical factorization;
4. **Substitute**: numerical forward and backward substitution.

Before numerical factorization, **Solve** will also invoke **InMatrix** to extract the numerical matrix values from a given matrix **Problem** and place them into the internal data structure. The subroutine name **InMatrix** is also overloaded, so that all **Solve** subroutines use the same name, although the subroutine that it invokes depends on the actual solver, since the data structures that store the Cholesky factor will generally differ across solvers.

Thus, from the perspective of a typical user, Sparspak can be viewed as shown in Figure 3. Two types of objects are involved in its use, namely, problem objects and solver objects.

Figure 3 provides focus for an important point about the solver objects. As the example in Figure 2 illustrates, a solver module can be used to solve a problem using a single call to the subroutine **Solve**. That routine in turn invokes other subroutines mentioned above to perform the various steps in solving a symmetric positive definite system. These are the subroutines listed in the lower subbox in the box labeled

**xxxSolver**. However, these routines are also accessible *directly* by the user; a user who wishes to control when various steps of the computation are performed, or wishes to execute only some of the steps, can invoke these "second-level" procedures directly. The solvers contain private state variables (not accessible by the user) that ensure these subroutines are called in the correct order and that issue warnings or fatal error messages if they are called out of sequence.

It was noted earlier that there is no need for a function **Construct** in C++ because constructors are automatically called whenever a new object is created. Similarly, when objects disappear in C++ as a result of leaving the scope in which the object was declared, *destructors* are automatically invoked. This is not the case in Fortran-90, so Sparspak-90 has **Destruct** routines to explicitly release storage used by objects when they are no longer needed.

**4. Meeting the design objectives.** Section 2 described a number of design objectives, and section 3 provided a general description of the design of Sparspak. The objective of this section is to provide some examples which illustrate how the design of the previous section allows the objectives to be met. Throughout this section, references will be made to the five objectives enumerated in section 2.

One objective of the redesign of Sparspak is that it be able to cater to the needs of a wide variety of users, from the casual user who may know little about sparse matrix technology to the sparse matrix researcher who might use the package as a "toolbox" containing functionality useful in advancing the field. To that end, the examples in this section illustrate how the package might be employed by users with varying levels of sophistication or need.

At a very coarse level, usage of the package can be divided into two categories: *standard usage* and *research usage*. The distinction is based on the objects the user declares and manipulates. Of course this distinction is somewhat arbitrary, but it is useful in understanding the design and usage of the package. Standard users are those who are aware of the coarse structure of the package, as described in the previous section. Of course, within this group of users there will still be substantial variation in the level of sophistication, but the objects that are manipulated by the user will be generally limited to **Problem** objects and **Solver** objects. Some will use only a few of the interface routines available and make no use of options that might be available, while others will make extensive use of them. Research users, on the other hand, will declare and manipulate some or all of the more basic objects within the package, such as graphs, orderings, and so on. These objects will be used as building blocks for the researcher's own software development efforts.

**4.1. Standard usage of the package.** A simple example of basic usage of the package was introduced in Figure 2. The user instantiates an object of the **Problem** class, "loads" it with a problem, creates a **Solver** object, and then uses it to solve the problem. If the user does not know that the problem has any special characteristics (or that it may not have any), then the solver chosen will be a general one that assumes no special properties.

For purposes of this discussion, suppose the problem to be solved is symmetric and positive definite and that an envelope method is to be used to solve the problem. Also, suppose one step of iterative refinement is to be performed on the solution. These computations would be performed as shown in Figure 4. Here a built-in test problem generator **MakeGridProblem** is used. The package provides a number of subroutines to generate standard problems for testing software.

```
program refine
    use Sparspak90
    type (Problem) :: p
    type (EnvSpdSolver) :: s
    real (double), dimension(36) :: x, res

    call MakeGridProblem(p, 6, 6, "9pt") ! create test problem
    call Construct(s, p)                 ! create solver object
    call Solve(s, p)                     ! solve for initial solution
    call ComputeResidual(p, res )        ! compute the residual
    call PrintVector(res, "residual")    ! print the residual
    call Solve(s, res)                   ! solve for correction
    call GetSolution(p, x);              ! retrieve the solution
    x = x + res                          ! compute improved solution
    call PrintVector(x, "refined solution") ! print solution
    call Destruct(s); call Destruct(p)   ! release storage
end program refine
```

FIG. 4. *Sample user program—one step of iterative refinement.*

The function name **Solve** is overloaded within each solver. In the example, the first call to it involves the solver and problem objects **s** and **p**, and the solution is put in the problem object **p**. In the second call to **Solve**, its arguments are the solver and a vector **rhs** containing the right-hand side; its contents are replaced by the solution.

Just as in the example in Figure 2, a change of only one text string in the program (**EnvSpdSolver**) is all that is needed to change the solver being used. All other function names would remain unchanged.

The second example in this section demonstrates how two different methods for solving a problem might be compared in terms of storage requirements and execution times. Such quantities are automatically captured within each solver. Suppose a user wants to compare the performance of the envelope and sparse solvers applied to a $50 \times 50$ grid problem obtained using a nine-point difference operator. One such program to do this is shown in Figure 5. Note again the use of name overloading: **Factor**, **PrintStats**, and **Destruct** invoke different subroutines, depending on the types of their parameters.

It was noted at the end of the previous section that interface routines for solver objects can be invoked at two levels. The user may choose to invoke only the **Solve** function; in that case, the appropriate routines will be called in the required order, and a solution will be produced. However, a user may wish to select various options that might be available within a solver and/or explicitly invoke some of the steps. An example appears in Figure 6. A grid problem is generated, just as in the example in Figure 5. Then the problem is factored using an envelope solver, where the default ordering is overridden by a random ordering. (The function **RandomOrdering** is a built-in function in Sparspak that generates a random ordering.) Then the same problem is factored using a fill-reducing solver, but the default ordering subroutine is overridden by a user-supplied subroutine called **MyOrdering**. To be able to implement such a subroutine, of course, the user must know the interface for such ordering subroutines.

Thus, some users of Sparspak will at times use the solver objects at this next level of resolution, explicitly invoking subroutines which execute the individual steps of the solution process. Some will select various options that may be available to adjust the

```
program compare
    use Sparspak90
    type (Problem) :: p
    type (EnvSpdSolver) :: envSolver
    type (SparseSpdSolver) :: sparseSolver

    call MakeGridProblem(p, 50, 50, "9pt") ! test problem

    call Construct(envSolver, p)    ! create solver objects
    call Construct(sparseSolver, p)
    call Solve(envSolver, p)        ! solve problem using each method
    call Solve(sparseSolver, p)

    call PrintStats(envSolver)      ! print performance statistics
    call PrintStats(sparseSolver)

    call Destruct(p)                ! release storage
    call Destruct(envSolver); call Destruct(sparseSolver)
end program compare
```

FIG. 5. *Sample user program—comparing two solvers.*

```
program myorder
    use Sparspak90
    type (Problem) :: p
    type (EnvSpdSolver) :: envSolver
    type (SparseSpdSolver) :: sparseSolver
    interface
        subroutine MyOrdering( g, order)
            use Sparspak90
            type (Ordering) :: order
            type (Graph) :: g
        end subroutine MyOrdering
    end interface

    call MakeGridProblem(p, 25, 25, "9pt")   ! test problem

    call Construct(envSolver, p)    ! create solver objects
    call Construct(sparseSolver, p)

    call Factor (envSolver, p, RandomOrdering)
    call PrintStats(envSolver, "random ordering: envelope solver")

    call Factor(sparseSolver, p, MyOrdering)
    call PrintStats(sparseSolver, "My ordering: sparse solver")

    call Destruct(p)              ! release storage
    call Destruct(envSolver); call Destruct(sparseSolver)
end program myorder
```

FIG. 6. *Using options in the package.*

```
program SimpleResearchUsage
    use Sparspak90
    type (Problem) :: p
    type (Grid) :: grd
    type (Graph) :: g
    type (Ordering) :: rcmOrder, mmdOrder

    call Construct(grd, 4, 4)    ! create 4 by 4 grid object
    call MakeGridProblem(p, grd, "9pt" )

    call Construct(g, p)    ! create graph object from p
    call Picture(g)         ! draw incidence matrix of the graph
    call Print(g)           ! print the adjacency lists of g

    call RCM(g, rcmOrder)   ! find RCM ordering of g
    call Picture(g, rcmOrder)! draw incidence matrix:RCM-reordered graph

    call MMD(g, mmdOrder)   ! find MMD ordering of g
    call Picture(g, mmdOrder)! draw incidence matrix:MMD-reordered graph

    call Destruct(g); call Destruct(p); call Destruct(grd) ! release storage
    call Destruct(rcmOrder); call Destruct(mmdOrder)
end program SimpleResearchUsage
```

FIG. 7. *Simple research-level usage of the package.*

behavior of the algorithms executed at each step. However, the basic objects that the user manipulates remain the same, and the details of data structures and algorithms are hidden from the user. The user needs only to be concerned with the "essential ingredients" of the problem.

The example in Figure 6 illustrates another design feature of Sparspak. Heavy use is made of optional arguments. Many of the subroutines allow the user to include a text string as a parameter to aid in making output from the package understandable. In the example, such a string is included in the calls to the subroutine **PrintStats**.

**4.2. Research usage of the package.** At this level of usage of the package, the user will be aware of some or perhaps all of the major classes or modules, along with the various routines that operate on objects from these classes. These include such things as graphs, orderings, elimination trees, grids, and so on. A simple example of usage at this level is given in Figure 7. The comments in the program in Figure 7 provide an explanation of what the program does.

The example in Figure 7 illustrates a number of additional features of Sparspak. Essentially all objects have the capability of printing themselves; that is, there is a subroutine called **Print** (heavily overloaded) that will print the contents of an object. For example, if the argument is a graph, the adjacency lists of the graph will be printed. If the object is a problem object, a listing of the nonzeros in the matrix and their positions will be printed, and similarly for other objects. Solver objects print detailed information about their data structures and orderings. These are mainly useful to sparse matrix researchers and for instructional purposes.

Similarly, many objects are able to draw a "picture" of themselves. For example, it is often useful to be able to see the structure of matrices and graphs, and there are **Picture** routines which provide such pictures. Indeed, even the solvers in Sparspak

have **Picture** routines which provide pictures of the data structures used. Again, such routines are useful for sparse matrix researchers as well as helpful in connection with teaching about sparse matrix methods. As usual, the name is overloaded so that the user needs only to remember one name, regardless of what object is being pictured.

In cases where it makes sense to print or provide a picture of a permuted form of the object, such as when the object is a graph or a problem, then an ordering object can be (optionally) provided to the print and picture routines. This option is used in the example in Figure 7, where a picture of the graph under the two different orderings found is printed.

**4.3. Low intellectual overhead.** The examples in the previous subsections make it evident that learning how to use the standard capabilities of the package is relatively simple. Of course, if one wants to utilize the many options available, or use the basic building blocks within the package, more must be learned. However, learning to use the basic power and functionality of the software requires little intellectual investment. This has been achieved through the following:

1. *Simple structure.* The typical user really needs to know about only two kinds of objects, namely, problem objects that contain information about a sparse system of equations, and solver objects which accept a problem as input and produce a solution as output.
2. *Name overloading.* The user must remember only a few function names in order to use the package. Functions which perform essentially the same task, regardless of the underlying data structure and algorithms being used, are invoked by the same name.
3. *Information hiding.* Intricate data structures used in storing sparse factors and detailed implementation of numerical routines are hidden from the user.

**4.4. Flexibility of the package.** The examples presented above illustrate how the package can cope with a wide variety of circumstances. The features that facilitate its flexibility are as follows:

1. Object-oriented design, which allows multiple problem and solver objects to be manipulated in one program.
2. There are a variety of methods available for solving a sparse system. The various scenarios (multiple systems with the same structure, multiple systems differing only in the right-hand side, etc.) can be handled in a natural way, as illustrated by the example in Figure 4.

The example in Figure 6 illustrates how optional parameters allow the user to replace the standard subroutines that find the ordering. Although only a modest number of solution methods exist in the current package, the design allows the simple extension of the package to include additional new methods.

**4.5. Serving different users.** One of the design objectives was to be able to cater to the needs of users with varying levels of expertise. The examples in the previous subsections demonstrate that the design allows this to occur in a more or less seamless fashion.

At a basic level, the user creates one or more problem objects and one or more solver objects. Depending on the context, relatively few member routines of those objects might be invoked. An example of such usage is given in Figure 2.

On the other hand, the context might require slightly more functionality: One or more of the second-level routines in the solver (**FindOrder, SymbolicFactor**, etc.)

might be invoked. An example of this somewhat more sophisticated use of the package is given in Figure 4. Other examples of the usage of these second-level routines in a solver are given in Figures 5 and 6.

Finally, the individual basic objects upon which the package is based can also be employed by the user. This is illustrated by the example in Figure 7.

**4.6. User-package communications.** Messages from the package are grouped into three categories: error messages, warning messages, and information messages. The extent to which these messages are printed is under the control of the user through the value of a **messageLevel** variable. If it is set to zero, all messages are suppressed. Setting it to one allows fatal errors to be printed; setting it to two allows warning messages to be printed as well; and setting it to three permits all messages to be printed. The message level can be reset during execution.

Sparspak allows multiple objects of the same class in the same program. As noted earlier, to provide clear association of messages with objects, each object has a string variable **objectName** for its identification. All messages printed are accompanied by the corresponding **objectName**. This allows the user to relate package messages to the objects that printed them.

To facilitate the use of the package by researchers, a debugging facility has been included in Sparspak, using ideas borrowed in part from a debugging facility provided in the Nachos system [2]. The user can select one of four debugging levels which control the amount of debugging information printed. Setting the level to zero will suppress all debugging information, with levels one, two, and three specifying increasing amounts of information. Additionally, debugging messages can be grouped into *categories* through the use of debug flags. For example, if a researcher is developing a new ordering subroutine, only messages associated with orderings and graphs may be desired. By setting the debug flag to "og," only messages with "o" (ordering) or "g" (graph) in their argument list will be printed. In this way, a software developer can choose to print only messages within specified categories as well as control the amount of information through setting the debug level. These parameters can be reset at any time in the user program so that the user can choose different types and levels of debugging information at selective portions of his or her code.

In Sparspak++, "conditional include" has been used for all the debugging statements. In a production environment, the user can simply choose to exclude all such debugging statements in compiling the code. On the other hand, for research development of new algorithms, the package should be compiled including these statements.

Unfortunately, Fortran-90 has no such conditional include facility, so a much less elegant solution has been used. All statements associated with debugging have a trailing comment of the form **!DEBUG**, so that a simple text stream processor can strip out the debugging statements prior to compilation when a "production version" of the package is desired.

**5. C++ and Fortran-90: Concluding remarks.** The development of the Sparspak++ and Sparspak-90 versions of Sparspak is an ongoing project. This article has described the objectives of the revision to Sparspak, which represent a substantial broadening over those of the original Sparspak package. In particular, supporting a much larger problem class and catering to the needs of a broad spectrum of users are prime objectives. The previous sections have provided the essential features of the design, along with a rationale for it. Also included was a collection of sample programs and commentary about them illustrating the way the design, together with various language features, allows the objectives enunciated in section 2 to be met.

Both C++ and Fortran-90 have a number of features that have been critical in achieving the design objectives set for Sparspak. The ability to overload function names is immensely helpful in keeping low the number of names a user must remember and is also helpful in understanding the structure of the package.

Another important feature of both languages is the ability to create "objects." Although Fortran-90 is not as versatile as C++ in this respect, it is possible with a simple programming protocol to have Fortran-90 modules and C++ classes serve more or less identical roles in the implementations of Sparspak-90 and Sparspak++. This feature is important in meeting one of the design goals, namely, the ability to simultaneously have a number of sparse matrix problems in the process of being solved.

In terms of providing a friendly and versatile interface to the user, one can easily conclude that these two programming language features are among the most important.

An important difference between the two languages that had a significant effect on the *ease of implementation* was support of *inheritance*. Briefly, in C++ new classes can be created from existing *base* classes by "subclassing" them. That is, one can define new classes as extensions of existing classes by declaring additional variables and introducing additional member functions. The new class thus defined *inherits* all the variables and functions from the base class. For example, the underlying structure of Sparspak++ involves the definition of an **SpdSolver** class containing basic objects and functions common to all symmetric positive definite direct solvers. Subclasses of **SpdSolver** will inherit this set of member variables and member functions from **SpdSolver**. Other functions and their interfaces may be declared in **SpdSolver**, but the actual implementations of these member functions will depend on the solution methods and will appear in subclasses of **SpdSolver**. There are currently two subclasses of **SpdSolver**: **EnvSpdSolver** for the envelope solution method and **SparseSpdSolver** for the sparse solution method. This feature has the important and well-known advantage of promoting the reuse of software and making it more manageable. Only one base class has to be maintained, even though it might be used in numerous other classes.

Unfortunately, Fortran-90 does not support the notion of inheritance, so the benefits of the strategy described above are less convenient to realize. There are modules containing the user-defined data types and the subroutines that implement each basic approach; these correspond to the subclasses of **SpdSolver** in the C++ implementation. For example, **EnvSpdBase** is one such Sparspak-90 "class" and another is **SparseSpdBase**. The corresponding solver **EnvSpdSolver** is created in a module with user-defined data type **EnvSpdSolver** having as one of its components **EnvSpdBase**. Thus, **EnvSpdBase** can be viewed as a subclass of **EnvSpdSolver** in the sense of C++. Similarly, a **SparseSpdSolver** is created in a module with user-defined data type **SparseSpdSolver** having as one of its components **SparseSpdBase**. This technique is known as *composition* in the software engineering literature.

At first glance, the advantage of software commonality appears to be lost, since there are two "base classes," namely, **EnvSpdSolver** and **SparseSpdSolver**. However, these solver classes are virtually identical. They differ only in the name of their user-defined data type and in the name of one of its components. Name overloading allows all subroutine calls to be otherwise identical. To change the **EnvSpdSolver** module to the **SparseSpdSolver** solver requires only a simple global text change. Thus, in reality only one solver module needs to be maintained. The use of makefiles

and stream editor scripts allows the various solvers to be generated automatically, and effective reuse of common software is achieved.

The Sparspak++ implementation makes extensive use of a standard C++ library, called the Standard Template Library (STL) [10]. This has been a very substantial aid in implementing many of the fundamental classes in Sparspak++. For example, the STL vector and list classes are used extensively in creating the Sparspak++ Graph class.[7] There does not appear to be anything similar available in Fortran-90; such a library would be very valuable, although it is not obvious that the language supports the generality required to allow such a library to be implemented.

Not surprisingly, Fortran-90 has some powerful features for numerical computation that are absent in C++. Perhaps the most notably useful are various array operations. The so-called colon notation, allowing a subarray to be referenced, is highly useful. Also, array-valued subscripts are extremely useful in gather-scatter operations and in applying permutations to arrays.

There are several features within C++ and absent in Fortran-90 that are potentially useful, although their utility has not been fully explored at the time of writing. One is *dynamic binding*, that is, the ability to establish links to routines at execution time. A need for this feature has not yet been encountered, but it may be in the future. The other notable feature in C++ that is almost certain to be useful is the ability to "throw" and "catch" exceptions, such as floating-point overflow, divide-by-zero, etc. When such exceptions happen many levels down within the procedure hierarchy, it is often necessary to pass such information up the hierarchy to a level where the exception can be addressed. The throw-and-catch facility available in C++ obviates the need to have error flags appear in the argument lists of the routines which either raise the exception, service it, or simply transmit it from one call level to another.

Finally, there are some differences in the way function name overloading is done in the two languages and in the way function parameters are handled. However, the capabilities are comparable and did not lead to a real distinction between the two languages in terms of ease of implementation.

**Acknowledgments.** The authors would like to thank the referees and Dr. John Lewis for many perceptive comments that have greatly improved the layout, notation, and general readability of the paper.

## REFERENCES

[1] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Co., Inc., Redwood City, CA, 1994.

[2] W. A. Christopher, S. J. Procter, and T. E. Anderson, *The nachos instructional operating system*, in Proceedings of USENIX Winter 1993 Technical Conference, The USENIX Association, Berkeley, CA, 1993, pp. 479–488.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, UK, 1987.

[4] A. George and J. W.-H. Liu, *The design of a user interface for a sparse matrix package*, ACM Trans. Math. Software, 5 (1979), pp. 134–162.

[5] A. George and J. W.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981.

[7] One might anticipate that the use of such a library might exact performance penalties. However, C++ allows one to specify that a function be *inline*. This means that short utility functions that are frequently called can instead have their code bodies included directly in the calling program. The function is not "called," and all overhead associated with the function call is eliminated.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice–Hall Software Series, Prentice–Hall, Englewood Cliffs, NJ, 1978.
[7] M. Metcalf and J. Reid, *Fortran* 90 *Explained*, Oxford Science Publications, Oxford, UK, 1990.
[8] B. Meyer, *Object-Oriented Software Construction*, Prentice–Hall, Englewood Cliffs, NJ, 1997.
[9] E. G. Ng and B. W. Peyton, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056.
[10] A. Stepanov and M. Lee, *The Standard Template Library*, Technical Report HPL-94-34, Hewlett-Packard Laboratories, Palo Alto, CA, April 1994.
[11] B. Stroustrup, *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1986.