UCRL-JC-132025
PREPRINT

# Design of the *hypre* Preconditioner Library

E. Chow
A.J. Cleary
R.D. Falgout

This paper was prepared for submittal to the
*Society for Industrial and Applied Mathematics*
*Workshop on Object-Oriented Methods for Intero-Operable*
*Scientific and Engineering Computing*
*Yorktown Heights, NY*
*October 21-23, 1998*

September 22, 1998

# Design of the *hypre* Preconditioner Library [*]

Edmond Chow[†] Andrew J. Cleary[*] Robert D. Falgout[*]

**Abstract**

we discuss the design of *hypre*, an object-oriented library for the solution of extremely large sparse linear systems on parallel computers. The mathematical emphasis of *hypre* is on modern powerful and scalable preconditioners. The design of *hypre*allows it to be used as both a solver package and a framework for algorithm development. The object model used for *hypre*is more general and flexible than the current generation of solver libraries.

## 1 Introduction

In this paper, we discuss the design of *hypre*, a project to create an object-oriented library for the solution of extremely large sparse linear systems on parallel computers. The need for *hypre* is motivated by the demands of computationally challenging programs such as the Accelerated Strategic Computing Initiative (ASCI) in the Department of Energy. The linear systems that arise in these applications are extremely large and difficult, and require a new generation of solvers and massively parallel computers for their solution. Current solver technologies are insufficient by themselves for solving these problems, though they can provide components out of which more advanced solvers can be built.

This paper represents work-in-progress. As of this writing, *hypre* reflects its origin as a collection of subroutines more than a designed library. This paper documents our plans to evolve *hypre* into an integrated, flexible, and coherent object oriented library.

In the remainder of this paper, we first review the state of the art of parallel linear solver libraries. In the next chapter, we discuss in more detail the design goals for *hypre*. We then describe the two key elements of *hypre*'s design that provide the framework for meeting *hypre*'s design goals, and finish with a summary.

## 2 Review of state of the art

Not long ago, many application writers coded linear solvers directly into their application codes. This was possible because the solvers at that time were relatively simple, as were the computer architectures. This has become an increasingly unworkable solution with the emergence of parallel computing and complicated data-structure specific preconditioners. By now, it is much more common for applications to turn to linear solver libraries. The older linear solver libraries are generally organized as sets of monolothic, stand alone subroutines, and are procedural in nature. To use such a library, an application writer has to decide

[*]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. The work was supported by the DOE2000 program and the ASCI PSE program.

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551.

ahead of time on a particular algorithm from a particular package, and then code the application in terms of the solver's data structure. This process is particularly difficult for parallel sparse data structures, making the required investment of effort so large that application writers rarely try any more than one solver. However, it is very difficult *apriori* to know which specific solver will work best for a particular application, meaning that the ability to experiment with different solvers is very important. Also, the adoption of new algorithms is often slow due to the investment necessary to switch solvers.

Increasingly, more modern solver libraries have adopted object oriented techniques that make them easier and more extensible to use. The object oriented solver libraries in widest use within the DOE are the PETSc library from Argonne ([?]), and the ISIS++ library from Sandia ([?]). Although there are several other object oriented libraries in the wider community, we will concentrate on these two libraries for illustrative purposes.

The core object models for both libraries are similar. The model consists of base classes capturing the mathematics, including Matrix, Vector, Solver, LinearSystem, KrylovMethod, and Preconditioner, as well as an auxiliary class that encapsulates information about the parallel computer and the distribution of objects to distributed memories, generally called a Map class. Note that the names we use here do not necessarily match exactly the names in these libraries but have obvious counterparts.

- The Vector class is fairly straightforward. It provides functions for inserting elements into the vector, as well as the standard set of linear algebraic functions such as dot products. In both libraries, its implementation is straightforward so that it can be used efficiently within critical operations such as matrix-vector products (mat-vec).

- The Matrix class has several uses. It is used in the LinearSystem class to define problems to be solved. Its matrix-vector multiply member function is used by the KrylovMethod class to implement Krylov-based algorithms. In PETSc, building preconditioners is the responsibility of the matrix class. In ISIS++, preconditioners can be implemented in this way, but another paradigm is supported as well. Here, subclasses of Matrix introduce *access functions* that provide abstractions for accessing the underlying data structure, and preconditioners can be written in terms of these access functions. In both libraries, interfaces are also defined for inserting coefficients into matrices.

- The LinearSystem class is basically a composition of a Matrix and two vectors, representing the equation $Ax = b$.

- In both libraries, the Solver class is essentially a composition of a KrylovMethod object and one (PETSc) or two (ISIS++) Preconditioner objects, along with parameters like convergence tolerances.

- The KrylovMethod class generalizes algorithms such as Generalized Minimum Residual (GMRES) and Conjugate Gradient (CG), of which there are many varieties. This class is used by the Solver class, and provides the functionality of providing the next iteration of the Krylov method based on the history of previous iterates and the action of the preconditioner. Note that in some libraries, what they call KrylovMethod is what we call Solver, that is, the Krylov class knows about and interacts with the Preconditioner class itself. The distinction is a matter of style rather than substance.

- The Preconditioner class includes all of the algorithms that are not Krylov algorithms, such as simple iteration schemes like Jacobi's method and Gauss-Seidel iterations, along with more sophisticated "preconditioners" such as incomplete factorizations and sparse approximate inverses. Objects in this class use Matrix objects, as discussed under the Matrix class.

## 2.1 Language

We note that while ISIS++ is written in C++, PETSc is written entirely in C language, and yet it is entirely object oriented, with encapsulation, inheritance, and polymorphism. Arguably, implementing object oriented code in C is more difficult and less natural than in C++ or other object oriented languages, but it does illustrate the point that the object oriented design is far more important than the choice of language in which that design is implemented. For this reason, we do not stress any particular language in this paper. Indeed, the choice of language of implementation for *hypre* is not entirely settled, and in the end, *hypre* will probably be implemented in several languages. In [?], a technique for achieving language interoperability for object oriented libraries is presented, and it is our intention to leverage this project to allow flexibility in both the calling language and the implementation language for *hypre*.

## 3 Design goals for *hypre*

We now compare and contrast the design goals for *hypre* with the designs of the current generation of object oriented solver libraries.

One major difference is that the mathematical emphasis of *hypre* is on modern powerful and scalable preconditioners, whereas the traditional emphasis has been on Krylov-type solvers, splitting-based iterative methods like Jacobi's method and Gauss-Seidel, and other high level preconditioners like polynomial preconditioning. Libraries like PETSc and ISIS++ are notably weak in parallel, scalable preconditioners, though this is more a function of the paucity of such preconditioners rather than a design choice.

To fill this void, we are designing *hypre* so that it can be used in two different modes: like existing libraries, it can be used as a stand-alone linear solver library, but in addition, it can be used as an *add-on package* to existing solver libraries. The first mode allows us to leverage existing libraries by allowing *hypre* to use Krylov solvers, basic iterative methods, and concrete matrix classes from other libraries, but all from a *hypre* interface, while the second mode allows parallel scalable preconditioners from *hypre* to be used from other libraries. This presents many new interoperability challenges. For instance, the scalable preconditioners in *hypre* require access to the coefficients of matrices to build data structures that implement the preconditioning step, whereas the majority of algorithms in current libraries do not need significant access to coefficients. Meeting this dual-mode goal requires relaxation of the standard library model in which a library assumes it "owns the world" to a "peer model" where each library is equal.

Another area that highly impacts our design is the emergence of "physics-based interfaces", e.g., the Finite Element Interface defined at Sandia ([?]). For the most part, solver libraries have a distinctly linear-algebraic interface that may not be particularly well matched to a physicist's view of their problem. In our opinion, linear-algebraic interfaces are more appropriate for algorithm developers than for users. The concept behind the physics-based layers is to provide a layer that allows application developers to describe their linear system in the language of their problem domain. For a problem discretized by finite elements, for example, it is much more natural for the application to describe the problem in terms of element stiffness matrices and element connectivities than to describe

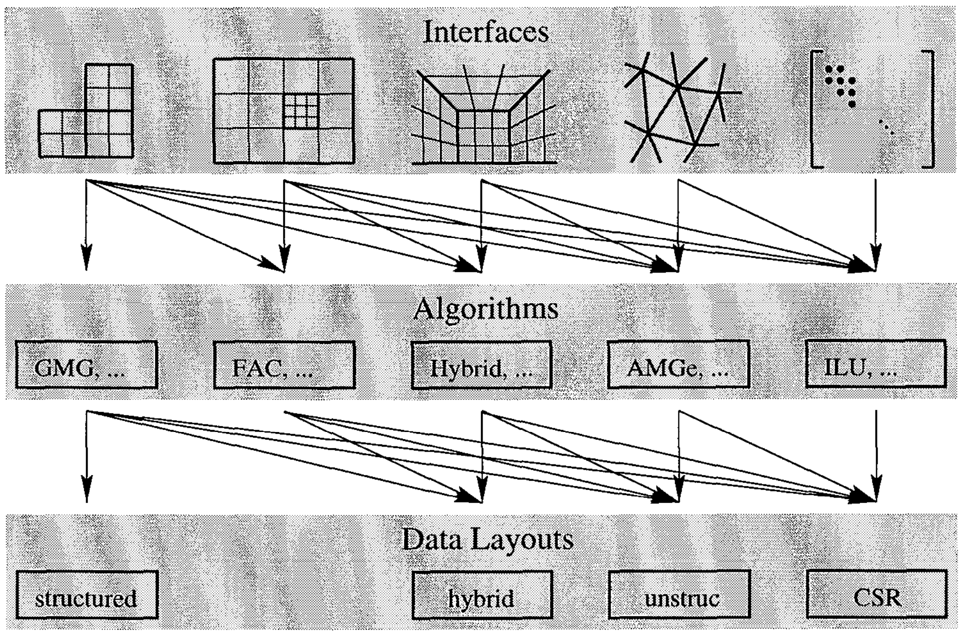

FIG. 1. *Interfaces.*

the problem in terms of matrix columns and rows. Under the covers, an implementation of the finite element interface performs this mapping from the finite element interface to a more solver-appropriate interface. At the worst, to the extent that these interfaces can be standardized, they can provide a single user-level interface to multiple solver libraries and allow applications to experiment with different libraries.

The physics-based interfaces are more than just conveniences for applications writers, however. As problems grow in size and difficulty, it becomes increasingly necessary for solvers to have more information about the problem than what is encapsulated in the traditional matrix. Multigrid solvers, for instance, often need to know information about the grids that defined the problem, and finite element-specific solvers like AMGe [?] must know the element stiffness matrices in unassembled form. Some solvers are not even defined for general matrices, for instance, alternating direction methods only make sense on structured grids for which "direction" is meaningful. Figure 1 illustrates this concept. The level of generality moves from left-to-right in this figure. On the left are specific interfaces with algorithms and data structures that take advantage of this specificity. On the right are the more general interfaces, algorithms, and structures. The arrows represent the fact that many of the more specific interfaces can be mapped to more general algorithms and interfaces, and it is this fact that allows seamless experimentation with different algorithms. However, there is a subsequent loss of performance, both in the algorithms and the data structures, that comes from discarding information. Note that the linear-algebraic interface is the most general and interoperable, but also the least efficient.

Recent trends have introduced another evolution in requirements for linear solver libraries that we are incorporating into the *hypre* design. Many of the newer solvers do not conform to the current solver model of a Krylov method plus a preconditioner, as we now illustrate. Some libraries (e.g. ISIS++) provide *composed preconditioners* that are defined as several different preconditioners applied in sequence. Multigrid solvers can be viewed as compositions of operators, including splitting methods (smoothers in multigrid terminology), matrix-vector multiplies (restriction and interpolation), and direct methods for the coarsest grid solve. Parallelism, through domain partitioning, is encouraging hierarchical solvers that are really solvers within solvers, e.g., a Krylov solver on the outside, block Jacobi as a preconditioner, a Krylov method to solve the individual blocks, and an ILU method to precondition the block solvers. Several research groups have developed "inner-outer" iterations that consists of at least two nested iterative methods, one for producing a Schur complement, and one for solving the Schur complement. These examples illustrate the fact that there is a trend towards algorithms that encompass an increasingly general set of possible combinations of solvers, preconditioners, matrices, and iterative methods, and thus a new and far more flexible model for combining methods is needed.

This viewpoint is important because *hypre* is intended for use by two mostly separate groups: applications writers will use *hypre* for solving linear systems, while *developers* will use *hypre* as a platform for developing new algorithms. It should be noted that PETSc and ISIS++, as well as other libraries, are also intended to be used in both of these modes and that they succeed in this endeavor to a certain extent. However, we feel that their object models are not quite general enough for the most recent algorithmic trends, and we are attempting to address this in *hypre*.

Though it is our intent that *hypre* supports and promotes object oriented technology, the fact is that most new algorithms are developed by mathematicians who are not trained in this paradigm. Therefore, it is a requirement that we provide a relatively lightweight mechanism by which such a code can be included in *hypre*, while still allowing for the inclusion of more interoperable and flexible solvers and matrices. To achieve these goals, *hypre* has been designed to support an arbitrary level of interoperability of each component, so that the level is decided by the component developer rather than dictated by *hypre*.

## 4 OO Model

In this section, we present the basic object model that forms the foundation of the design for *hypre*. Central to the design is the use of multiple inheritance of interfaces for allowing algorithm developers a flexible system for mixing and matching exactly the interfaces that they want to support for their objects. Note that we adopt the Java model of multiple inheritance in which at most one of the inherited base classes can be concrete. The core object model, as described in the design goals listed above, needs to be far more flexible than existing models that hinge on Krylov solver objects and preconditioner objects only. Note, however, that because many solvers will still be formed in this way, it is important that the traditional notions of Krylov objects and preconditioner objects are supportable by inheriting and extending interfaces from the core *hypre* model.

Our core model consists of two layers. The top layer consists of three base abstract classes representing the mathematics of linear solver libraries, and a Map class analagous to that in current libraries. This top layer's main function is to provide type safety and polymorphism. The second level consists of a myriad of separate *interfaces*, where we define interface to mean a collection of functions that can be invoked on an object. In C++ terms, these interfaces are pure virtual classes. The function of the second layer is to provide a "menu" of interfaces out of which other classes can build their more complete interfaces through multiple inheritance. This mix-and-match paradigm is vital for supporting a wide variety of concrete classes developed by varied development teams.

The classes in the top layer are **Map**, **Vector**, **Operator**, and **CoefficientAccess**. The Map and Vector classes are essentially unchanged from existing libraries, and thus we will not focus on them any further. The other two classes are less obvious, and in particular,

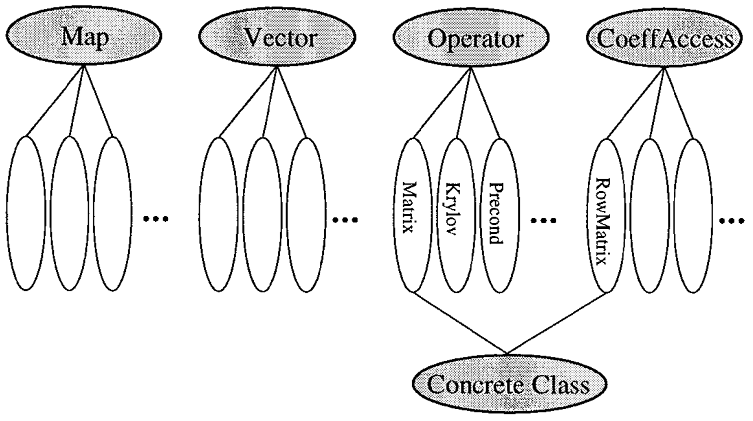

FIG. 2. *Object model.*

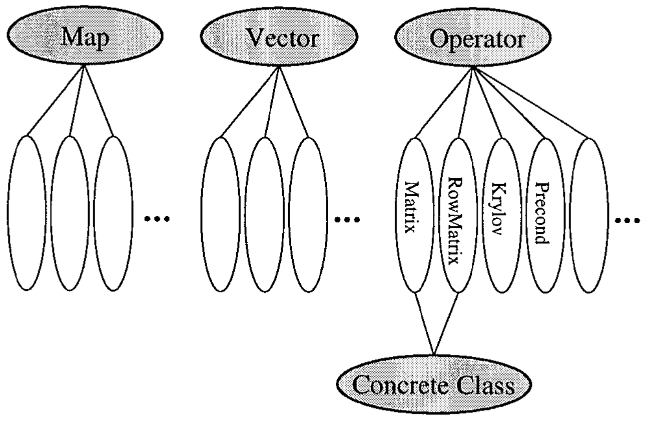

FIG. 3. *Simple object model.*

there is a notable absence of standard objects such as matrices and solvers from our core model. Our design builds these more standard objects from the last two classes, and thus it is important to justify their existence as peer classes. We use the matrix to illustrate the intent of these classes.

As described in section 2, the traditional matrix class supports both the linear-algebraic mat-vec operation, as well as being a container for the matrix coefficients that allows input or output of coefficients through member functions. Fundamental to the *hypre* design is the notion that these are orthogonal functionalities, and that it should be possible for objects to support one of these functionalities without supporting the other. The Operator class in *hypre* contains (a more general form of) matrix-vector product functionality, while the CoefficientAccess class represents container functionality. Specific classes are built by multiply inheriting from subclasses of the Operator and CoefficientAccess classes.

The classic example of an object that performs mat-vec but does not offer access to coefficients is the so-called *matrix-free matrix*, which by definition can perform mat-vec through some procedure (differencing formulas in non-linear methods, for example) without actually storing a matrix data structure. The matrix-free matrix is already supported in current libraries: ISIS++ has a root Matrix class that includes the mat-vec operation, for example, and PETSc makes provision for user-defined matrix operations. However, the classes that supply coefficient access functionality inherit from the Matrix class in current libraries, and this precludes having objects that provide coefficient access without providing mat-vec. We prefer to break this inheritance tree into a flatter inheritance relationship.

A coefficient-based preconditioner (i.e. one that must access coefficients and cannot be implemented with just mat-vecs, e.g. incomplete factorization), needs a container object but does not care whether that object can perform a mat-vec operation. Under the standard model, the matrix used to form the preconditioner is also used to define an outer Krylov method, in which case the matrix has to provide mat-vec. We believe that there are occasions where it is appropriate to use iterative methods without Krylov wrappers, in which case the mat-vec functionality is unnecessary.

Direct methods are the most straightforward example, as they clearly do not need to be wrapped in a Krylov solver. A direct method like Gaussian elimination is not usually considered an iterative method, but in the context of *iterative refinement*, it can be considered as an iterative method that almost always converges in a single step. There is precedent for this: PETSc treats direct methods as special cases of iterative methods. Even if direct methods are not particularly practical for solving huge sparse problems, they still have many uses as components for building algorithms, e.g. direct methods are often used to solve the coarsest grid problem in multigrid. When the coarse grid operator is formed, it will not have to be able to perform a mat-vec, so forcing the object that encapsulates the coarse-grid operator to implement a mat-vec is an unnecessary burden on the developer.

This design decision allows innovations such as implementing an algorithm that does not explicitly form a "matrix" for the coarse grid,but rather implements the container access functions necessary for the direct method in terms of the data structures and operations from the grid above it. Several Schur-complement preconditioners are based on this kind of a representation for the Schur-complement operator. (**??**).

In general, it is our claim that there is a spectrum of objects that support these two functionalities in fairly arbitrary combinations, that is, some classes support (flavors of) one or the other or both. Requiring the two interfaces to go hand in hand is too restrictive.

## 4.1 CoefficientAccess Class

The CoefficientAccess class is essentially an abstraction of matrix data structures. As a simple example, consider that many of the parallel solver libraries in use today have a matrix data structure that consists of a contiguous block of rows of the parallel matrix stored in each processor's memory in a compressed sparse row (CSR) format. These data structures are very similar and codes that implement a particular algorithm for these different data structures are logically the same. Nonetheless, because of the minor differences, separate code for each data structure is necessary. A solution that allows more reuse of algorithmic code is to write algorithms on top of abstracted data structures, and then wrap each compatible data structure in terms of the abstract data structure. Not all data structures are compatible with all abstracted data structures, but using an abstraction layer allows interoperability of those data structures that are functionally equivalent in terms of t! he access patterns that they support.

There are two basic dimensions of coefficient access, input and output (or "put" and "get"). Again, we claim that these are orthogonal interfaces in that derived classes may mix and match input and output coefficient access interfaces in arbitrary combinations through multiple inheritance.

Input coefficient access is already fairly common. The finite element interface referenced earlier has an input interface which is defined in a language of finite elements. To the user of this interface, the object looks like a matrix data structure that supports input of coefficients in blocks corresponding to element stiffness matrices. This allows the user to build his matrix through the abstractions of the FEI rather than building a particular data structure. Likewise, PETSc's base matrix class has a member function called MatSetValues. Users build their matrix data structures through this member function, and PETSc does the work under the covers to place coefficients into the requested data structure.

In some senses, the input coefficient access is less controversial. For one, building a matrix is done only once and is a much lower-ordered cost than solving a linear system. This means that these functions do not have to be particularly efficient, which in turn means that input patterns can have a looser coupling to the underlying data structures.

Output coefficient access is more controversial. For the CSR-based data structures described above, the key access pattern is random access to locally stored rows. This can easily be abstracted by providing an abstract class with a GetRow function. Given a row number, this function returns the given row in a sparse format. The exact semantics are arguable, that is, whether the function should return a pointer, whether it should use copy semantics, which format the row should be in, etc. The concept remains the same, though: an algorithm written for one data structure in terms of GetRow will work with other matrices that support the same function.

In the past, there has been considerable resistance towards implementing preconditioners on top of abstraction layers rather than for specific data structures. The argument is performance-based: performance is crucial and therefore a preconditioner needs to know exactly on which data structure it is operating. We have two responses to this argument. First, we leave the decision about the level of interoperability of solvers up to the developer of the solver. There is obviously some tradeoff between interoperability and performance, and the developer should be allowed to make the design decision of where to draw this line. Secondly, we argue that expressing algorithms in terms of access patterns is more natural than it seems at first: there is *no* sparse matrix data structure that supports efficient random access to its coefficients, and therefore there is *no* preconditioner that accesses the coefficients randomly. By definition, anything that is not random is patterned.

Another way to look at input coefficient functionality is as a generalized Iterator. In this context, an Iterator is a construct that allows the elements of complex data structures to be accessed in some sequence. This concept has proven very powerful in allowing so-called *generic algorithms* for certain problem domains. Unfortunately, while some preconditioners and iterative methods can be written with only serial access to the matrix, many others cannot. The idea behind CoefficientAccess is to establish other access patterns besides pure serialization in which preconditioning algorithms can be expressed. While we have worked through several examples as part of *hypre*'s design, we acknowledge that it is still an open question as to what the ratio of algorithms-to-access patterns is across the spectrum of preconditioners.

The obvious benefit of the coefficient access classes is code reuse, but there are other benefits. As discussed, one of our design goals is to use our core object model as a framework for developing complicated algorithms out of components developed across development teams. Assume that such an algorithm wants to use Operator A, B, and C (the use of the Operator class to build algorithms is explained more fully below). This can only work if the matrix data structure that is input is compatible with all of the operators. In an environment with many diverse data structures, this will not be possible very often. However, with the coefficient access concept, it is only necessary that the input data structure supports access patterns that are compatible with all of the operators. Since it is anticipated that there will be far fewer access patterns than data structures, this becomes much more likely.

The input coefficient access concept may also be useful in capturing data locality, much the same as the BLAS capture data locality for the LAPACK dense matrix library. As illustrated by the GetRow example, the return value of an access function does not have to be floating point numbers representing coefficients. It is perfectly natural for some access functions to return other coefficient access objects. Generally, these will encapsulate some smaller portion of the matrix. At the right level of granularity, these intermediate portions can be used to build efficient kernels that can be used as building blocks for codes that are both efficient and interoperable. The potential benefits along these lines are still an area of research.

## 4.2 Operator Class

We discuss the Operator class, which is the unifying concept on which many of the more commonly used classes are based. The unification comes from the observation that the fundamental action of most mathematical objects in linear solver libraries is operating on vectors to produce an output vector. This includes the classical notions of matrices, solvers, preconditioners, and iterative methods.

- Matrices are the most obvious subclass of Operator. The fundamental defining characteristic of a matrix is that it is a linear operator that maps vectors to vectors.

- Solvers are actually quite similar to matrices. Solving a (nonsingular and square) linear system is the same as applying the *inverse* of a matrix, which is itself a matrix. As a class, a solver is really just an operator that has a different set up than a matrix.

- Preconditioners, which incude basic iterative methods like Jacobi and Gauss-Seidel as well as more complicated methods such as multigrid and incomplete factorizations, are mathematically "approximate solvers". They are defined by a matrix and a right-hand side and return an approximate solution. Their implementations are somewhat different, though, particularly in their setup functions.

- An iterative methods is different than a solver or approximate solver in that it is defined by a matrix and a right hand side, and operates on a guess at a solution to the system and returns a new and hopefully better guess as output. These **use** approximate solvers to solve residual equations.

- Krylov methods can be Operators in two different ways. They can be considered (approximate) solvers, in that they are defined by a matrix and operate on right hand sides to give approximate solutions. Or, with the right-hand side fixed, the Krylov method takes an initial guess as input and then produces a sequence of iterates, so that it is a type of iterative method.

We believe that generalizing all of these classes into subclasses of an Operator base class not only encapsulates common behavior, but more importantly, it allows the flexibility for building complicated solvers from nontrivial combinations of Operators. We illustrate with an example.

The Solver class in PETSc is composed of a KrylovMethod and a Preconditioner, and optionally, a separate PreconditionerMatrix. It is not our intention to review the mathematics of Krylov solvers in depth, but we point out that mathematically, a preconditioner to a Krylov method is a matrix $M$ such that $M^{-1}A$ is better conditioned than $A$. Mathematically, then, any linear operator can be used as a preconditioner, since a matrix and a linear operator are equivalent. Since our goal is to provide a flexible framework for algorithm development, we prefer to let the preconditioner be *any linear operator* rather than dictating that it be a "preconditioner". This obviously encompasses the standard preconditioner model: the user inputs the Matrix to a Preconditioner object, and then composes the Preconditioner object with the Solver. In addition, this generalized definition can be used to implement the PreconditionerMatrix concept in almost the exact same way.

The concept of a preconditioning matrix is useful for applications that have a simpler representation (a lower order discretization, for example, or the symmetric part of a mildly nonsymmetric matrix) of the problem to be solved. In PETSc, there is special code and special semantics surrounding the PreconditionerMatrix, namely, if one is supplied, the "preconditioner" is the result of an iterative method applied to the preconditioning matrix. In our model, a user or developer who wants to use this method can first input the preconditioning matrix to an appropriate Operator like an iterative method or an incomplete factorization method, and then compose this Operator with the Solver. No special code or semantics are necessary: the core model encompasses this combination of objects.

Should this user have a matrix that somehow approximates the inverse of $A$ directly, this matrix could be composed with the Solver as a preconditioner directly. The key concept is that if the mathematical requirement of a preconditioner is that it should be a linear operator, than we would prefer to let *any* linear operator be allowed to fill this role. We are not smart enough to anticipate all possible future algorithmic developments, so we allow full generality wherever it makes sense. The Operator class is our mechanism for providing this generality.

## 5 Summary

We have presented the preliminary design of *hypre* as a framework for developing and providing advanced linear solver algorithms for extremely large systems. We have reviewed the standard object oriented models for linear solver libraries and argued that they are not particularly flexible for building new solvers. We presented a core object model that replaces the common classes of matrix, preconditioner, solver, etc., with only two classes, the CoefficientAccess class and the Operator class. We showed how the former class supports physics-based interfaces, allows generic programming of algorithms, and eases interoperability between modules developed independently, while the latter class supports innovative algorithm development as well as interoperability. The fundamental technique used to provide this framework is the use of multiple inheritance and a flat inheritance tree to allow developers the freedom to implement objects with exactly the interfaces that they think are appropriate for their objects.

The design of *hypre* has been heavily influenced by two forums within the DOE and academia. The Linear Equation Solver Interface forum ([]) is attempting to define a standard object model and standard interfaces for linear solvers, and the *hypre* design is proceeding in parallel, influencing and being influenced by this effort. Much of the object model presented in this work mirrors the object model being developed in this forum. The other group is the Common Component Architecture forum, which is attempting to standardize a component architecture for high performance scientific computing. The *hypre* model of multiply inheriting desired interfaces is influenced in large part by the fact that it maps one-for-one onto a *query-interface* (or *introspection*) model, which is at the core of most component models.