Object-Oriented Design of Preconditioned Iterative Methods in Diffpack

ARE MAGNUS BRUASET
SINTEF Applied Mathematics, P.O. Box 124 Blindern, N-0314 Oslo, Norway. Phone: +47 2206 7610. Email: Are.Magnus.Bruaset@si.sintef.no.
and
HANS PETTER LANGTANGEN
Dept. of Mathematics, University of Oslo, P.O. Box 124 Blindern, N-0314 Oslo, Norway. Phone: +47 2285 5833. Email: hpl@math.uio.no.

As modern programming methodologies migrate from computer science to scientific computing, developers of numerical software are faced with new possibilities and challenges. Based on experiences from an ongoing project that develops C++ software for the solution of partial differential equations, this paper has its focus on object-oriented design of iterative solvers for linear systems of equations. Special attention is paid to possible conflicts that have to be resolved in order to achieve a very flexible, yet efficient, code.

Categories and Subject Descriptors: D.2.2 [**SOFTWARE ENGINEERING**]: Tools and Techniques—*Software libraries*; G.1.3 [**NUMERICAL ANALYSIS**]: Numerical Linear Algebra—*Linear systems,Sparse and very large systems*

General Terms: Design, Performance

Additional Key Words and Phrases: iterative methods, preconditioning, object-oriented programming, C++

This research is supported by the Research Council of Norway through the strategic technology program STP 28402: *Toolkits in Industrial Mathematics* at SINTEF.

1. BACKGROUND

Over the years, the increase in computing power has enabled numerical investigations of mathematical problems at very high levels of complexity. We are today routinely solving problems that are far beyond the scope of the presently available analytical tools. Consequently, experimental mathematics is emerging as a rapidly growing research field. This development demands access to reliable and robust numerical software that can be easily adapted to different applications. Unfortunately, much of the existing numerical software is based on very restrictive programming concepts and does not provide the flexibility needed to fulfill these requirements. To overcome such limitations, it is desirable to collect knowledge and methodologies from recent developments in computer science. However, since heavy numerical computations tend to force special constraints on software design, the technology transfer should usually be made on premises of the numerical analyst.

The present paper is concerned with modern software development techniques for iterative solution of large, sparse linear systems

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n,n}$ is nonsingular and $x, b \in \mathbb{R}^n$. The solution of such systems is a fundamental, and often the most time-consuming, part of many simulation codes. In fact, systems like (1) are natural ingredients in applications found in a wide range of disciplines, e.g. structural analysis, electrical engineering, oil reservoir modelling, computer aided geometric design, atmospheric pollution, chemical engineering and economic modelling.

When using a traditional Fortran code to solve $Ax = b$, one usually calls subroutines with a large number of arguments. If A is sparse, several parameters are needed to specify the matrix entries, the storage structures and the length of various arrays. In addition, one must usually supply work arrays in the subroutine calls. Ideally, the software should be able to hide all the details of matrix storage schemes and numerical parameters specific to certain solvers and preconditioners. For convenience, this type of information should be supplied by means of some interactive user interface. In the code one would like to have three basic variables (A, x and b), form a linear system of these (possibly including a preconditioner) and apply a solver to this system. The use of such high-level variables enables programming at an abstraction level that is close to the mathematical and numerical formulation. In this paper we will present a methodology for the design of a library that offers the programmer this kind of simple high-level implementation. The program for solving $Ax = b$ with any combination of the available matrix formats, preconditioners and linear solvers is about one page in size and will be outlined later in the paper.

The purpose of the present paper is first to present a brief introduction to the basic concepts of object-oriented programming (OOP). Then we describe in more detail a specific application of OOP to a library for iterative solution of linear systems. The emphasis is put on the fundamental design ideas. These ideas are illustrated by examples taken, usually in a simplified form, from a library developed by the authors. This library is available from `netlib` [Netlib] and constitutes a part of a comprehensive software package for numerical solution of partial differential equations [Diffpack ; Bruaset and Langtangen 1995]. However, this paper is

written primarily for specialists on preconditioned iterative methods who also have an interest in and some knowledge about OOP. We also emphasize that the paper focuses on the design issues of a particular implementation and is therefore not intended to discuss or survey object-oriented design and numerical linear algebra in general.

2. OBJECT-ORIENTED NUMERICS

Object-oriented programming (OOP) is a software design technique enabling different numerical methods and implementations to be hidden behind a common interface. This interface can be closely related to the mathematics of the problem. The resulting code becomes easy to use, extend and maintain. OOP is very different from standard procedural programming in, e.g., Fortran or C. Although Fortran or C can be used to implement an object-oriented design, see Gropp and Smith [Gropp and Smith 1993; PETSc] for a C implementation of a library similar to ours, the programming effort is significantly decreased by using a language that supports OOP. Today, C++ [Ellis and Stroustrup 1990; Barton and Nackman 1994] is the only language of this kind that meets the strong efficiency requirements of numerical computations. Besides being an efficient tool for OOP, C++ simplifies programming in general by having a more rigid type checking than Fortran and classic C, and by offering a syntax that makes it easier to develop self-explanatory code.

OOP has now gained widespread use in computer science, while the interest in OOP of numerical methods is rapidly growing. Many contributions exist on the application of C++ in libraries for array computations. We refer to the proceedings [OONSKI93 1993; OONSKI94 1994] and software packages from Dyad [Dyad Software] and Rouge Wave [Rogue Wave Software] for such examples. There are only a few documented examples on sparse matrix solvers that have the flexibility of the degree that is described in this paper. One very comprehensive and impressive contribution in this field is the `SLES` library, which is the linear algebra part of the `PETSc` package developed by Gropp and Smith [Gropp and Smith 1993; PETSc]. In contrast to the `PETSc` approach, we apply C++ instead of C. Due to inherent mechanisms supporting OOP, the choice of using C++ leads to a much simpler implementation at the library level. Other impressive libraries include the `IML++` (and `SparseLib++`) software [Dongarra et al. 1994; IML], the C++ extension `Lapack++` [Dongarra et al. 1993] to the well recognized `LAPACK` library. C++ software for iterative solution of large, sparse linear systems have also been developed by Pommerell and co-workers [Heiser et al. 1991; Pommerell et al. 1992; Pommerell and Fichther 1991]. Application of object-oriented sparse matrix solvers to finite element problems are reported in [Bruaset and Langtangen 1995; Zeglinski and Han 1994]. In comparison with most other sparse matrix libraries we have put more emphasis on the design of a framework where new matrix formats, iterative solvers, preconditioners and termination criteria can easily be incorporated without affecting existing application codes. Both `PETSc` and `IML++` reach the same goal, but utilize two different implementational strategies.

We will emphasize that OOP offers the *potential* for a numerical programming environment that increases the human productivity of developing application codes and decreases the maintenance efforts. Such results rely heavily on modular and

re-usable codes that express a close connection between the program statements and a mathematical language. Simply making use of a few classes in C++ may provide improvements over traditional Fortran 77 programming, but to achieve the promises of OOP, a very careful design is required. A good numerical package that utilizes OOP must also exhibit computational efficiency that is comparable to Fortran, and this will put serious limitations on the object-oriented design. `Diffpack` represents the first step in the direction of object-oriented numerical software, where a compromise between "good" object-oriented design and computational efficiency is the main focus. That is also the central topic of the present paper.

2.1 Abstract data types

An *abstract data type* (ADT) consists of data and functions operating on the data[1]. A matrix may serve as an example of an ADT. The data include the matrix entries and the number of rows and columns. The functions are, e.g., the subscripting operator and arithmetic operations like the matrix by vector product. The members of an ADT, i.e. the data and the functions, may be regarded as protected or public. In the former case, the members are invisible to a user of the ADT, hence enabling the important concept of *data hiding*. For a matrix, the storage of the matrix size and entries is information that typically should be hidden from the user and therefore be declared as private (or protected). Functionality like subscripting and the matrix by vector product represent general mathematical operations on the matrix, and is hence a part of the public interface available to users of the ADT. In this way, the internal storage of a matrix may be changed, e.g. from an array to a list, without affecting the use of the ADT in existing code.

In C++, an ADT is represented in terms of a *class*. Consider the code segment in Figure 1 which specifies the interface of the ADT `RectMat` representing rectangular dense matrices with real-valued entries of double precision. In this example, the entries of a given matrix are stored row-by-row in the C-style two-dimensional array `entries`, while the integers `nrows` and `ncols` denote the dimensions of the current matrix object. The special member function `RectMat`, which is called the *constructor*, is used to declare a variable (instance) of the class, e.g., by the statement `RectMat A(m,n)`. Note that `m` and `n` can be variables and the memory allocation of the matrix can be performed at run-time. When the object `A` goes out of scope, the corresponding *destructor* `~RectMat` is called to clean up used storage space etc. Since the entries and dimensional parameters are invisible for the class users, public member functions are needed for operations on the data structure. The function `size` returns the current matrix dimensions. One `prod` function computes the matrix by vector product $y = Ax$, where A denotes the current `RectMat` object. The other `prod` function, representing matrix by matrix products, demonstrates the convenient concept of *function overloading*; both functions carry out a special matrix-related product and have the same name, but different arguments. The meaning of parentheses for a matrix can also be redefined by the function `operator()`, thus enabling subscripting like `A(i,j)`.

[1] The term "abstract" is perhaps misleading, since an ADT may well be a usable, concrete data type in a program, in contrast to abstract base classes in C++ [Coplien 1992].

```
class RectMat {
 protected:
   double**   entries;
   int        nrows, ncols;

 public:
   RectMat (int nrows, int ncols);
   ~RectMat ();

   void size (int& nrows, int& ncols);
   virtual void prod (const Vector& x, Vector& y);
   virtual void prod (const RectMat& x, RectMat& y);
   double& operator()(int i, int j);   // subscripting
   ...
};
```

Fig. 1. The specification of class `RectMat`.

Fig. 1

The `RectMat` class can be generalized to other types of entries than `double`. This is achieved through a parametrization of the entry type. Implementing the generic class `RectMat(Type)`, where `Type` denotes the data type of the entries, we may easily generate classes like `RectMat(double)`, `RectMat(int)` or `RectMat(Complex)`. C++ offers a special *template* construct for parameterization of class entries. However, the authors still prefer to use C macro constructions instead of the standard C++ templates since the latter lead to large compilation times and large object files, macros give to some extent more flexibility and complete control in the construction of parameterized types, and the stability of some compilers have until recently been questionable with respect to complex class hierarchy designs utilizing template constructs. We believe that this situation is due to immaturity of compilers and is likely to change in the near future. From a conceptual point of view, there is no principal difference between our current implementation of parameterized types and the C++ template construct. For simplicity, we will neglect the explicit notation of the parameterization when we discuss the matrix and vector classes.

2.2 Class hierarchies and inheritance

So far, we have seen that ADTs are useful in order to develop modularized code, where implementation specific details can be hidden from the user. However, OOP is more than the use of ADTs. OOP is characterized by *dynamic binding and polymorphism*. It means that an ADT chosen at run-time, for example a Krylov method, can be thought of as a more abstract ADT, e.g., an iterative solver, in the program. If the programmer just calls the *solve* functionality of such a general iterative solver, C++ will automatically call the solve function in the particular user chosen Krylov method ADT. This "magic" of OOP is accomplished by information generated by the compiler and is represented by the virtual function construct in the C++ language.

The *inheritance* concept in C++ is used to achieve the functionality outlined above. Inheritance may be used for two basic purposes, either as a convenient construction for sharing code between to ADTs or for expressing related functionality of two ADTs. The former purpose is simply a matter of convenience for saving work when writing code, while the latter is the central theme of object-oriented design. We make use of both purposes in our linear algebra package. A primitive ADT may utilize inheritance for sharing code, and ADTs at higher abstraction levels usually use inheritance to achieve a desired object-oriented functionality. Let us first give an example where inheritance is used to share code. From class `RectMat` we can derive a new class `SymmMat` for symmetric matrices, using inheritance, see Figure 2. This means that the new class `SymmMat` has access to data items and member functions from its parent class, while it still can add its own data structures and individual functionality. The derived class can also redefine the implementation of inherited member functions. Any instance of `SymmMat` will rely on the array `entries` and the dimensional parameters `nrows` and `ncols` supplied by `RectMat`. However, since the new class will omit the storage of duplicate entries due to symmetry, the algorithm for accessing individual matrix entries has changed. Consequently, `SymmMat` can not utilize the matrix by vector product implemented for `RectMat` and is therefore forced to have its own specialized implementation.

However, the member function `size` can be inherited, since it is independent of the actual storage format. Although the derivation of `SymmMat` may not be a very practical one, this example serves its purpose as an introduction to inheritance for sharing code between two ADTs.

We will now explain how inheritance is used in OOP. When working with matrices that arise from discretization of PDEs one must deal with many data formats, for example, banded matrices, diagonal matrices, sparse matrices, finite difference stencils (point operators), in addition to the common rectangular dense matrices. From a mathematical point of view, the interface to these different matrix formats should be the same. One can then create an ADT with no data structures, but with a declaration of the general mathematical interface that applies to all different matrix formats. In C++, this is called an abstract class. Here this class has the name `Matrix`. Concrete implementations of matrix formats are then represented as classes derived from `Matrix`, where data structures are added for storage and access of matrix entries. In addition, the functions in the general interface must be implemented, utilizing the special matrix storage structure. One function in the common matrix interface is the matrix by vector product function, here called `prod`. Figure 3 shows that `prod` is declared in class `Matrix` as an abstract function, while derived classes for sparse matrices (`MatSparse`) and dense matrices (`Mat`) implement their special versions of this product function. The `prod` function is *virtual*, which means that if we create a sparse matrix object of type `MatSparse` and thereafter refer to it as a `Matrix` ADT in the program, C++ will know that this `Matrix` object is actually a sparse matrix. Consequently, a call to the `prod` function of `Matrix` will then automatically invoke the `prod` function implemented for `MatSparse`. This principle, where different matrix classes can be treated as a general entity `Matrix`, is the key point of OOP, and virtual functions are the key tool for implementing OOP. The programmer can forget about details in the storage formats and instead consider all matrices as being of type `Matrix`. When programming iterative solvers, this functionality is especially advantageous since different matrix formats become completely transparent in the implementations of the solution algorithms.

One may discuss whether inheritance is a suitable model for matrices and vectors in a linear algebra package. Other packages, like `NewMat` [Davies 1993] and `M++` [Dyad Software], offer a set of stand-alone matrix classes and avoid a base class like our `Matrix`. However, the design outlined in the present paper has the flexibility of stand-alone classes as well as a unified interface for all matrix formats. The latter feature is vital when implementing iterative solvers, for which the underlying matrix format is transparent. Such data transparency can be achieved by inheritance, such as in `Diffpack` and `Lapack++` [Dongarra et al. 1993]. Alternatively, one may use class templates, e.g. as illustrated by `IML++` [Dongarra et al. 1994; IML]. On the other hand, the latter approach is less flexible than the inheritance-based design due to the static behaviour of templates. In contrast to a matrix hierarchy that allows run-time specification of the matrix format, a template-based approach can only utilize compile-time information. From our point of view, the flexibility offered by dynamic binding of matrix types is a vital ingredient in a software environment that supports experimental computations.

```
class SymmMat :  public RectMat {
 public:
  SymmMat (int n); // Symmetry, i.e., ncols = nrows = n
  ~SymmMat ();
  virtual void prod (const Vector& x, Vector& y);
  virtual void prod (const SymmMat& x, SymmMat& y);
  double& operator()(int i, int j);       // subscripting
  ...
};
```

Fig. 2. The specification of class SymmMat.

Fig. 2

Our approach requires a unified interface to matrix classes that must express the common functionality of different matrix types, and the unified interface is hence limited compared to the functionality of a specific matrix type. In some instances it is necessary to work with the specific types directly, for example, when initializing the matrix entries. Although the subscripting operator is a common functionality of all matrices, efficiency considerations prevent the practical use of a virtual function for this purpose. Looking at the `SymmMat` class, we see that the member function `operator()` is redefined in this subclass and that it is not virtual. This violates a fundamental rule in OOP [Coplien 1992], but is necessary to achieve inlining which is crucial to the computational efficiency when indexing large matrices.

Some readers may assert that if we make a `MatSparse` object and send it to a function `f(Matrix& A)` inside which we say `A(i,j)=r`, the assignment is likely to fail since `operator()` is not virtual. This does not happen in `Diffpack` since class `Matrix` does not define `operator()` at all. It is important that all functions defined in class `Matrix` are virtual (or so simple that they are not intended to be redefined in subclasses). One can make entry specific assignments to a `Matrix` object by either downward casting to `MatSparse&` and then using an efficient `operator()` function, or by calling a virtual indexing function `A.elm(int i, int j)`. The latter approach is of course slow, but avoids downward casting.

In large software packages there are usually several class hierarchies with many levels and numerous classes at each level. To fully exploit the potential of inheritance and virtual functions, say by offering run-time choice of the matrix format, it would be desirable to have virtual constructors such that instantiation of a concrete ADT could be represented as a virtual function. Unfortunately, virtual constructors are not a part of C++, but can be implemented in various ways [Coplien 1992]. Our implementation is as follows. We suggest that a *parameter class*, e.g. `prm(Matrix)`[2] is defined for each major class hierarchy. These parameter classes, which in some sense resemble the context variables used in the `SLES` package [Gropp and Smith 1993], are able to store all information needed for allocation of any object of a derived class. For example, the parameter class for the matrix hierarchy contains variables expressing the number of rows and columns, the type of storage scheme (class name) that is chosen, the bandwidth, a symmetry indicator, a pointer to a sparse matrix structure, locations of off-diagonals for the `MatStructSparse` class etc. For a given matrix (class) type, only a subset of these parameters is relevant. Typically, the parameter object is initialized by reading a data file or by reading input from a menu system. Afterwards, this entity is passed on as argument to a create function, where the wanted subclass object is allocated using the parameter object as input to its constructor. Figure 10 to be presented later in the paper, illustrates real applications of such parameter classes. This scheme makes it very simple to include new subclasses, since any modifications of existing code is local within the parameter class and the corresponding create function. The suggested parameter class approach is quite different from Coplien's [Coplien 1992] use of delegated polymorphism and virtual constructors. In contrast, the parameter class

[2] The string `prm(X)` is not a valid C++ identifier. However, we define a special preprocessor macro that translates this character sequence to `X_prm`. This macro is needed for appending the `prm` string to parameterized types like `Matrix(double)`.

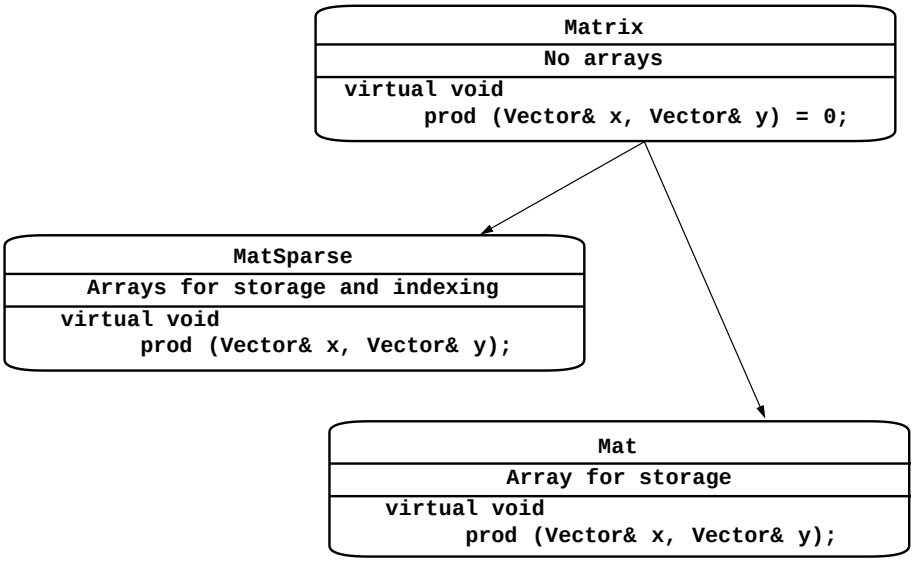

Fig. 3. The use of virtual functions permits derived classes to redefine functionality inherited from the ancestor class. Combined with abstract base classes such as `Matrix`, this allows the specification of a uniform interface that is forced on any derived matrix format.

Fig. 3

approach allows new subclasses in the hierarchy and new governing parameters to be added without any modifications in the existing library code. Hence, the parameter classes allow great flexibility, but the construction is far from elegant. We would like to have seen elements in the C++ language that could support the basic functionality of our parameter classes and create functions.

2.3 Numerical OOP

The class concept and associated mechanisms like parameterization (templates), function overloading, inheritance and virtual functions permit close relationships between data structures and mathematical entities. This is possible since OOP offers the necessary tools for working at a high level of abstraction. Experiences with such programming environments tend to show a considerable increase of human productivity. This gain is mainly due to larger amounts of reused code and improved conditions for software maintenance. However, there are also possible losses associated with a change of programming style. In terms of human resources and economy, the trend is that more time is spent at the initial stages of a project. This is due to increased need for planning and training. Fortunately, these costs are less significant as the programmer gains experience and previously developed code is reused in new applications.

We will now discuss some of the potential problems of using C++ and OOP. Since Fortran still is the most used language for scientific computing, there are extremely large collections of Fortran library codes around. Naturally, the users of these libraries often hesitate the change of development platform if this situation implies the loss of well-known and tested software. Consequently, new environments for production of numerical codes, based on other languages than Fortran, should have a policy for integration with existing tools. This important discussion is considered to be out of scope for this paper, but we mention that C++ classes fairly easy can make use of Fortran libraries.

Given the tradition of highly optimized Fortran programs, one might ask whether the more elaborate constructs of object-oriented languages tend to slow down the execution speed. It is certainly easy to generate elegant, but inefficient C++ code. On the other hand, careful use of the language constructs results in the same efficiency as can be obtained in C. It is widely believed that Fortran compilers still generate better optimized code than C/C++ compilers. Hence, it seems to be that the increased flexibility and man-time efficiency obtained by a genuinely object-oriented library may be paid for by a slight decrease in computational efficiency. As we will show later, numerical experiments with `Diffpack` and similar well recognized, high-quality Fortran implementations indicate that the present compiler technology and a careful usage of C++ provide an efficiency that is close to that of Fortran. The recent Fortran 90 language [Metcalf and Reid 1992] merges many C++ features with traditional Fortran and seems promising both with respect to computational efficiency and programming with ADTs. However, Fortran 90 does not support OOP. The principle of dynamic binding, that is, the choice of matrix formats, solvers and preconditioners at *run-time*, is one of the practical benefits that is lost in Fortran 90. However, it will be a matter of taste whether the OOP ingredients of virtual functions and inheritance are really needed to create the desired flexibility in the resulting code.

When it comes to efficiency considerations and OOP, there are possible pitfalls. The basic rule for deriving complicated class hierarchies is to maintain a clearly layered design, where the CPU-intensive functions are implemented at the lowest levels. For instance, a specific matrix class should implement its own `prod` function for the matrix by vector product rather than rely on a more general implementation at a more abstract level. In this way, we are assured that the computations can benefit from detailed knowledge of the data format. At this level, most functions will take the form of traditional Fortran- and C-style array loops. Classes at higher levels, e.g. the representation of a linear system, should then serve as administrators that delegate the actual work to lower level classes. This approach is essentially similar to the ideas of BLAS [Anderson et al. 1992; Duff et al. 1992]. In section 5 we provide specific performance numbers on the relative efficiency of our implementations compared to Fortran and C code.

The implementation of the `prod` functions in Figure 1 could also be accomplished by *operator overloading* where the multiplication operator `*` is given a new implementation for matrices and vectors. Although tempting from an aesthetic viewpoint, it should be noted that the use of such overloaded operators may lead to inefficient code. Consider the expression `r = r + a*u`, where `r` and `u` are vectors and `a` is a scalar. C++ will first evaluate `a*u` and make a temporary vector that is thereafter added to `r`. However, we know that this expression should be evaluated in a single pass of the entries and without extra storage requirements. In case of large scale computations, temporary objects will of course be intolerable, possibly leading to low efficiency and waste of memory space. Instead, it is advisable to offer a procedural interface `add` (similar to the `prod` function in Figure 3) for calculating expressions on the form `r + a*u`. Using function overloading, one may define several `add` functions covering the most common expression types encountered in typical applications.

Special care is also needed for the assignment operator (`=`) and the copy constructor. If not specified explicitly, these member functions are given default implementations based on copying objects member by member. This may lead to unpredictable problems, e.g. when the class contains a pointer to dynamically allocated memory segments.

As a consequence of the discussion above, there are some weakesses of C++ that could be improved. The tools for code optimization are still somewhat behind Fortran on several platforms, thus indicating that additional optimization functionality is needed. However, it should be stressed that this situation has improved significantly over the last couple of years. With the present C++ language it is necessary to perform a downward casting to fix the object type before such computationally intensive loops and thereby allow inlining. Downward casting is considered to reflect a bad object-oriented design, but is sometimes needed for efficiency reasons. In `Diffpack` we have developed special tools for safe downward casting, but this could be supported in C++ directly. We also think that automatic type conversion, automatic generation of assignment operators and copy constructors, and the problems with getting a satisfactory overview of all functionality in complex class hierarchies are reasons for unsafe or complicated usage of the C++ language in practical implementations. Another weakness of C++ is the overhead associated with compound arithmetic expressions involving large data structures (see the dis-

cussion of vector expressions like `r = r + a*u` above). The suggested standard [Ellis and Stroustrup 1990] mentions new compound operators, e.g. `=+*`, that will allow optimal efficiency, yet with a mathematically attractive syntax.

3. REPRESENTATION OF LINEAR SYSTEMS

In this section we will discuss the design of class hierarchies for representation of various matrix and vector formats. We will see how these entities are combined with a hierarchy of preconditioners to form linear systems of equations. These data structures provide the basic building blocks for both direct and iterative solvers.

3.1 The matrix and vector hierarchies

In Figure 4 we have illustrated a matrix hierarchy. The unifying interface to all matrix formats is represented by an abstract base class `Matrix`. The basic data type `Mat` represents rectangular dense matrices. These are implemented using standard C matrices. However, it is advantageous to first make a very simple encapsulation of a C matrix in terms of a primitive class `MatSimplest`. This class offers only allocation of a matrix (with a parameterized entry type) and subscripting (with index check in non-optimzed compilation mode). I/O functionality is added to this simple matrix in the subclass `MatSimple`. The `MatSimplest` and `MatSimple` classes are not intended for numerical computing. Since they are parameterized, they can serve as matrices for large and complicated objects for which arithmetic operations etc. have no meaning (grid objects is an example, and the entry type should then not require `operator*` etc. to be defined). Moreover, the simple matrix classes serve conveniently as base classes for the class `Mat`, which is aimed at numerical computations. This latter class derivation is motivated by implementational issues and not mathematical structure. The derivation ensures that `Mat` relies on a generic and well tested C matrix, or in other words, the purpose of the derivation is to increase the reliability of the library and re-use code. The `Mat` class adds numerical member functions to `MatSimple`, e.g. product functions, SVD factorization, LU and Cholesky factorizations as well as corresponding forward and backward substitutions.

Based on the representation of dense matrices, `MatBand` and `MatStructSparse` are derived as implementations of banded matrices and sparse matrices with nonzeros along certain diagonals, respectively. These matrix formats rely on the rectangular array structure provided by their ancestor, but have to redefine most numerical operations to suit the actual storage scheme. The classes `MatDiag` for diagonal matrices and `MatSparse` for general sparse matrices are derived `Matrix` and utilize a vector object for representing the matrix entries. Most of the code in `MatDiag` can simply call corresponding functionality in the vector class. In order to access the entries of `MatSparse` there is an additional data structure `SparseDS` holding the index information. Presently, the chosen format is that of compressed sparse row storage, see for instance [Barrett et al. 1993]. Applications often use several sparse matrices of the same structure. Duplication or reconstruction of the structure information is of course unacceptably wasteful. By using smart pointers with reference counting, several `MatSparse` objects can share the same sparsity pattern object `SparseDS`.

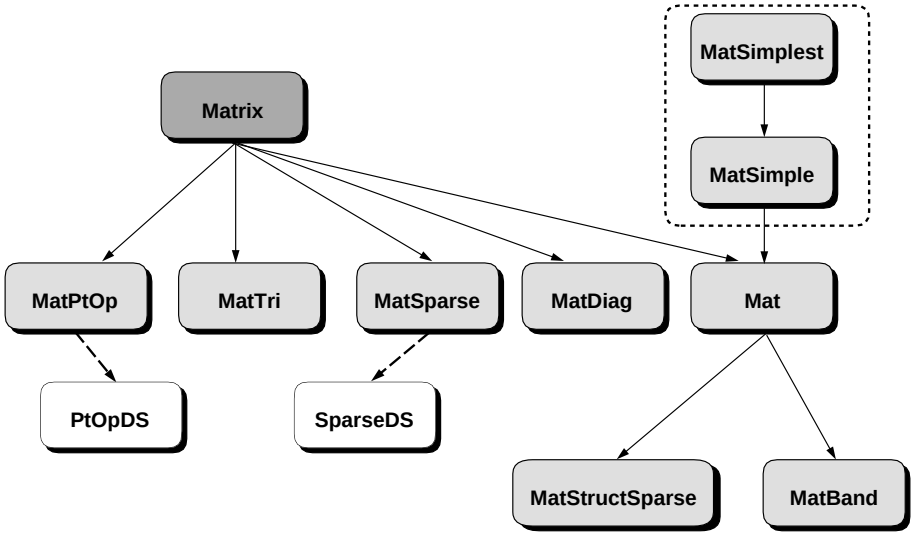

Fig. 4. The `Matrix` hierarchy. The solid arrows represent inheritance ("is a"-relationship), while the dotted arrows represent pointers to data structures ("has a"-relationship). The classes inside the dotted box are non-numerical array structures defined in the lowest library level.

Fig. 4

In addition to the formats mentioned above, there is a class `MatTri` for tridiagonal matrices. In contrast to the two-dimensional array structure used by `MatBand`, this matrix class provides its own storage space in terms of three private vectors.

The final matrix type in the hierarchy of Figure 4, `MatPtOp`, refers to finite difference stencils such as the five-point representation of the discretized two-dimensional Laplacian. Instead of storing a matrix, we store the stencil itself, i.e., the involved weights and their position relative to the center node. The actual stencil representation given by the data structure `PtOpDS`, allows use of variable weights in terms of pointers to functions. Quite similar to the `Matrix` hierarchy, the common interface for vector formats is specified by `Vec`'s abstract base class `Vector`.

From an aesthetical and mathematical point of view, the matrix hierarchy could have utilized a more classical design, for example, letting a diagonal matrix be derived from a tridiagonal matrix, which is derived from a banded matrix, which then may be derived from a sparse matrix and so on. We believe that the important derivation is from class `Matrix`. The other internal derivations are based on implementational convenience and not on mathematical structure. Whether `MatDiag` is derived from `MatBand` or not is in our opinion not important when programming linear algebra applications, since the fundamental property of `MatDiag` is that it is a `Matrix`. More importantly, however, is the possible overhead in storage that may be introduced by letting `MatDiag` inherit data structures from sparse, banded and tridiagonal matrix classes (although these data structures can be made almost empty in a clever implementation).

Although our library offers several matrix and vector structures, some users may want other storage schemes, e.g., to simplify the process of porting software from another environment to ours. Plain C-arrays can be inserted into and extracted from our array types. If the programmer has a favourite matrix class, say class `MyMat`, it can easily be incorporated into our matrix hierarchy and used by iterative solvers etc. The procedure is to apply multiple inheritance and derive a new matrix class `MyNewMat` that has its object-oriented interface specified by class `Matrix`, while the implementation of this interface is accomplished by inheriting data structures and functions from class `MyMat`. In other words, `MyNewMat` has all the functionality of `MyMat`, but has in addition the interface that is required for OOP of matrix operations in our library. If the external matrix code is not present as a C++ class, it is usually possible to wrap the needed data items and associated functions and subroutines into a C++ class. When this class is implemented, it can be easily integrated with the `Matrix` hierarchy in the way we just described.

Regarding the integration of user-defined matrix classes like `MyMat` with the proposed framework, we should mention that this requires careful implementation and extensive knowledge of advanced C++ features like multiple inheritance and its implications.

Some readers may wonder if software complexity in OO designs has really been diminished, or merely moved elsewhere. For example, the conventional long parameter lists in Fortran or C may seem to be replaced by lots of complicated details about inheritance hierarchies and intricate class dependencies. The experience from the `Diffpack` project is that the code at the lowest level tend to and should be very close to that in Fortran or C. At the next level, where the basic building blocks are defined, like matrices and vectors, the complexity of the design may be significant,

at least if one aims at a high degree of flexibility. However, at higher abstraction levels (see the next subsection) the software achieves the desired simplicity from an application point of view. Long parameter lists are absent, the number of classes are few and the hierarchal structures are simple. The goal of `Diffpack` is to provide both flexible libraries *and* high level, simplified interfaces for the application programmer such that it is easy to switch numerical algorithms and storage schemes at run-time. The rest of the paper will demonstrate this strategy.

3.2 Linear systems of equations

Once we have established the abstract entities of matrices and vectors, it is natural to combine these into a structure that represents a system of linear equations. In its simplest form, a linear system consists of A, x and b. We can create a class `LinEqSystemStd` holding pointers to these three basic quantities. Preconditioned systems,

$$C_L A C_R (C_R^{-1} x) = C_L b, \tag{2}$$

typically contain the preconditioners C_L and C_R in addition to the standard system. In the context of inheritance, it is therefore natural to derive a preconditioned linear system class, `LinEqSystemPrec`, from `LinEqSystemStd`, motivated by the sharing of data structures for A, x and b.

With a linear system we also associate a set of operations that is needed in iterative solvers. For example, we need the matrix by vector product, application of the preconditioners C_L and C_R and the computation of the residuals

$$r = b - Ax, \quad s = C_L r \quad \text{and} \quad z = C_R s \tag{3}$$

given the current value of x. In our design, we introduce an abstract base class `LinEqSystem` that defines a set of useful operations on linear systems. Figure 5 shows the class relationship. One of the operations on a linear system, namely the application of the left preconditioner C_L (function `applyLeftPrec`, see Figure 5), is defined as an abstract function in `LinEqSystem`, just to indicate that with a linear system abstraction `LinEqSystem` we can call the functionality `applyLeftPrec`. In `LinEqSystemStd`, where C_L is actually the identity matrix, `applyLeftPrec` is simply implemented as a dummy function, while in the preconditioned linear system the function has a non-trivial implementation. We can then create either a standard linear system, or a preconditioned one, at run-time, and thereafter refer to the system as the abstraction `LinEqSystem`. If a linear solver then calls the application of the left preconditioner for a `LinEqSystem`, C++ will figure out the correct `applyLeftPrec` function that is to be called.

In many applications, the coefficient matrix A has a natural block structure. One way of representing such block matrices, is to build a simple matrix of smart pointers to `Matrix` objects: `MatSimplest(Handle(Matrix))`, where `Handle(X)` denotes the parameterized smart pointer class that can point to a class `X` object. The smart pointer class provides reference counting and several other features that increases the safety of pointers. For example, the risk of dangling pointers is in practice eliminated. This matrix of base class pointer construction is implemented in the class `LinEqMatrix`, which is the data type actually used by `LinEqSystem` and its derivatives. For example, when solving a system of PDEs, one PDE may be solved

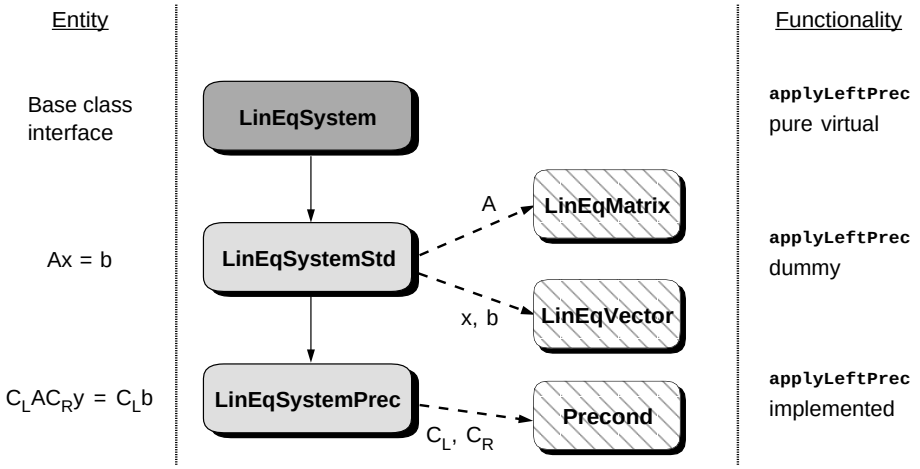

Fig. 5. Organizing matrix, vector and preconditioner objects into a system. The `LinEqMatrix` and `LinEqVector` objects are capable of holding blockwise structures. For simplicity, they can be thought of as plain `Matrix` and `Vector` data types, respectively.

Fig. 5

by finite difference methods for which the point operator matrix format may be appropriate, while another equation in the system may be solved by finite element methods where the sparse matrix format may be the most suitable choice. The `LinEqMatrix` class is flexible enough to allow a mixture of different matrix formats for the individual blocks. Corresponding to block matrices, the class `LinEqVector` implements block vectors based on the `Vector` hierarchy.

3.3 Preconditioners

As indicated above, the class `LinEqSystemPrec` combines the original system with the preconditioners in (2) through inheritance and pointers to objects derived from the base class `Precond`, see Figure 6. The preconditioner classes are typically grouped as being algebraic (`PrecAlgebraic`), procedural (`PrecProcedure`) or an inner iteration (`PrecItSolver`). The algebraic preconditioners construct a preconditioning matrix, e.g. in terms of incomplete factorization, see [Bruaset 1995] and references therein. This type of methods need matrix input to serve as basis for the construction of the preconditioner. Representing this input by a `LinEqMatrix` pointer, we have the flexibility of basing the preconditioner on the coefficient matrix A or some other relevant entity. The procedural preconditioners do not compute a preconditioning matrix. Instead they implement their action as a simple function call, e.g. a run of FFT, or even by invoking a stand-alone simulator capable of solving a simpler, yet related, problem. The third group of preconditioners rely on an inner iteration, i.e., the class `PrecItSolver` needs a `LinEqSystem` representation for the inner system and a pointer to `LinEqSolver` to hold the actual inner solver. Typically, this means running a fixed number of some simple iteration like SOR, cf. Saad [Saad 1992].

More sophisticated preconditioning strategies based on domain decomposition or multilevel techniques, can also be incorporated as classes in the `Precond` hierarchy. Such methods are based on grid information that can be made available through the parameter class `prm(Precond)`, or by interfacing to an external simulator with its own grid objects. Some of these preconditioning strategies, e.g. multigrid, can also be used as iterative solvers by themselves. In such cases, it may be convenient to implement the algorithm as a solver in the `LinEqSolver` hierarchy discussed in §4.1. It is then possible to use it as a stand-alone iteration as well as a preconditioner via a `PrecItSolver` derivative.

If the user desires one-sided preconditioning, i.e., $C_L = I$ or $C_R = I$, the identity preconditioner is achieved by using the dummy class `PrecNone`. By default, the preconditioner classes forces the matrix-vector product to be performed in three steps when neither C_L nor C_R is trivial. For certain factorized preconditioners, including SSOR and incomplete factorizations based on tensor-product grids, one can significantly improve the efficiency of the matrix-vector product by using "Eisenstat's trick" [Eisenstat 1981; Heiser et al. 1991]. To incorporate this functionality, we may derive a new linear system object from `LinEqSystemPrec`. This new class can then redefine the matrix-vector product function such that it detects whether Eisenstat's trick can be used. This new matrix-vector product function should call specialized low-level functions implemented for each available matrix format in order to carry out the triangular solves and diagonal matrix-vector product required in the Eisenstat procedure.

At first sight it seems that when using OOP, specialized optimizations forces more and more function primitives down into the matrix classes that are used only for one particular high level operation. Our experience is, however, that algorithmic improvements (like Eisenstat's trick) can be supported by deriving a specialized class at a high level (like a new linear system class) where only a few new statements are needed, that utilize standard mathematical operations already present in the low level classes (like triangular solves and matrix-vector products).

4. IMPLEMENTATION OF ITERATIVE METHODS

Based on the object-oriented principles introduced in §2, we will now present a flexible framework for implementation of iterative methods. The main result of this achievement is that generic code segments, e.g. memory allocation and convergence tests, are separated from the actual solver. In this way, it is possible for new solvers to take immediate advantage of existing code. In particular, any iteration that follows certain guidelines for its implementation will have access to a wide range of convergence criteria, including advanced functionality like recording of convergence history.

4.1 A hierarchy of linear solvers

In Figure 7 we present the `LinEqSolver` hierarchy, which is divided into two separate subhierarchies defined by `DirectSolver` and `IterativeSolver`. For the rest of this paper, we will concentrate on the branch for iterative algorithms and just comment that a specific direct method, e.g. naive Gaussian elimination, would be derived as a subclass of `DirectSolver`.

As indicated in Figure 7, it is convenient to differ between basic iterations like SSOR and Krylov methods like conjugate gradients. The main reason for this grouping, is that these methods have different characteristics, e.g. in terms of memory usage and possible convergence criteria. In particular, the siblings of `KrylovItSolver` have automatical access to the residual r defined in (3). Depending on the requests made during the initialization of a specific solver, say in class `ConjGrad`, the parent class may also allocate space for the preconditioned residuals s and z. On the other hand, `BasicItSolver` does not supply this automatic allocation scheme. Instead, it offers storage of the previous iterate to accommodate solution updates. If a basic iteration needs the residual at some point of the computation, this can be explicitly evaluated by the `LinEqSystem` object we are operating on. Typically, requests for such computations may be issued by a *convergence monitor*, i.e., a member of the `ConvMonitor` hierarchy discussed in §4.2. The administration of these monitors is the responsibility of the class `IterativeSolver`, while the actual solvers access convergence information only through the inherited member function `satisfied`.

A stationary iteration derived from `BasicItSolver` will typically look like the SSOR implementation in Figure 8. The member function `solve` takes a `LinEqSystem` subclass object as argument and redimensions internal vectors according to the system size. Thereafter, the function `init` is called to set up the environment provided by `BasicItSolver`. In particular, the argument value `NO_PREC` causes an error message if the supplied system is preconditioned. As mentioned above, the iteration itself is controlled by the inherited function `satisfied` which evaluates any sup-

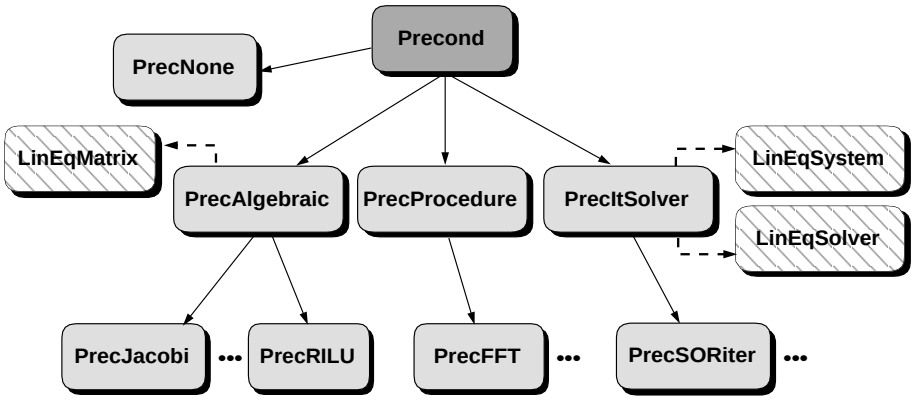

Fig. 6. Different preconditioners are branches of the `Precond` hierarchy. The mid level of this hierarchy is used to group methods that require the same type of data structure, e.g. a preconditioning matrix.

Fig. 6

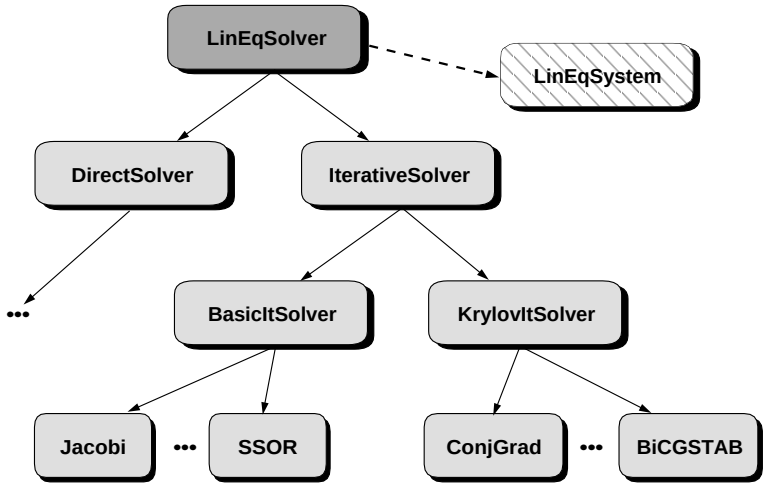

Fig. 7. Linear solvers are organized in a class hierarchy, where the abstract base class `LinEqSolver` provides a transparent interface to all implemented methods, direct as well as iterative ones.

Fig. 7

plied convergence criteria. Inside the iteration, a single SSOR step is performed by calling functionality associated with the coefficient matrix. In this way, we are assured that this operation is efficiently performed at the lowest level of the `Matrix` hierarchy, while the matrix format is completely transparent in the `solve` function.

Turning to the conjugate gradient method (CG) implemented in class `ConjGrad`, we have a typical iteration based on the attributes of `KrylovItSolver`, see Figure 9. Inside the iteration, still controlled by `satisfied`, we issue calls to standard vector operations like `inner` and `add`, to the matrix by vector product `matvec` and to the preconditioning operation `applyPrec`. Due to being virtual functions, these calls are automatically redirected to low-level computations that fully exploit the structure of the involved vector, matrix and preconditioner objects. It is emphasized that this code is efficient even when applied to the original problem $Ax = b$ in terms of a `LinEqSystemStd` object (or a `LinEqSystemPrec` object equipped with "preconditioners" of type `PrecNone`). In this case, the framework offered by `KrylovItSolver` knows that the residuals r, s and z in (3) are identical. Since these entities are represented by (smart) pointers `r`, `s` and `z`, storage space is allocated only for the unpreconditioned residual `r` while `s` and `z` are pointing to the contents of `r`. Consequently, the call to `applyPrec` will under these circumstances do nothing at all since `z` and `r` refer to the same vector object. Similar actions are also taken when using one-sided preconditioning.

In Figure 10 we demonstrate the flexibility of the suggested framework in a `Diffpack` program. This relatively compact code segment permits the user *at run-time* to combine any available matrix format with any linear solver, choosing any preconditioner. Due to consistency checks in the library code, illegal combinations are detected and result in an error message. Note that the details of the mathematics of this code segment are unknown at compile-time, except for the high-level principal tasks.

As previously discussed, the numerous parameters entering storage schemes and details of iterative solvers, preconditioners and termination criteria are collected in parameter objects. These entities can easily be manipulated by a menu system, either interactively or in batch. In practice, most of these parameters can take on their respective default values, thus limiting the amount of input that has to be provided by the user. The menu system is a part of `Diffpack` that enables the user to initialize objects in a simple way. Most class objects have a `defineStatic` function that defines a set of menu items, usually corresponding to internal variables in the object that need to be initialized by input data from the user. In addition, the class objects have a `scan` function that can read the user given answers on the menu. In this way, all objects are completely responsible for their own input and initialization and can make use of a generic menu class for communicating with the user. The interface to the menu system can be chosen at run-time and includes command line options, command statements in a file or a standard graphical box-based dialog.

The code segment in Figure 10 shows a high-level interface to our linear algebra package. Many readers will find it illustrative to see a more concrete example where we operate with real instances and not only base class pointers whose bindings depend on answers in a menu system. Figure 11 displays a similar program where we work with a tridiagonal matrix (class `MatTri(real)`, a conjugate gradient solver

```
BooLean SSOR::  solve (LinEqSystem& system) {
  redim (system);  // Redim internal data structures based on system
  init(NO_PREC);   // No preconditioner allowed
  while ( (niterations < max_iterations) && (!satisfied()) ) {
    // Should check omega inside iteration to capture adaptive parameters
    ++niterations;
    *prev_x = msys→x();
    msys→A().SSOR1it(msys→x(),*prev_x,msys→b(),omega);
    updateCommBlk(); // Update communication
  }
  exit();
  return convflag;
}
```

Fig. 8. Implementation of the SSOR method in class `SSOR`. The `msys` variable is a smart pointer (`Handle(LinEqSystem)`) to `system`, and `prev_x` is a smart pointer to the previously computed approximation to the solution vector x.

Fig. 8

```
BooLean ConjGrad::  solve (LinEqSystem& system) {
  redim (system);  // Redim internal data structures based on system
  init(ANY_PREC);  // No, left, right, split preconditioning allowed
  // Note that x and r correspond to A·x=b, while z=Cright·Cleft·r
  p = *z;
  rho = inner(*z,*r);
  while ( (niterations < max_iterations) && (!satisfied()) ) {
    ++niterations;
    msys→matvec(p, u);
    alpha = rho/inner(p,u);
    add (msys→x(), msys→x(), alpha, p);
    add (*r, *r, -alpha, u);
    msys→applyPrec(*r,*z);
    rho1 = inner(z,r);
    beta = rho1/rho;
    rho = rho1;
    add(p,*z,beta,p);
    updateCommBlk(); // Update communication
  }
  exit();
  return convflag;
}
```

Fig. 9. Implementation of the conjugate gradient method in class `ConjGrad`. The function works for any of the implemented matrix formats, preconditioners and termination criteria.

```
#include <createLinEqSolver.h>
#include <LinEqSolver.h>

void main (int nargs, const char** args) {

  // standard Diffpack initialization procedures...
  // set up a menu for the user (global_menu is a menu object)
  prm(Matrix(double))::defineStatic(global_menu,SUB);
  prm(LinEqSolver)::   defineStatic(global_menu,SUB);
  prm(Precond)::       defineStatic(global_menu,SUB);

  prompt user for menu answers

  // allocate matrix and vectors in terms of parameter class
  // objects that are initialized through the menu system

  prm(Matrix(double)) mat_prm;
  mat_prm.scan(global_menu);   // load user's answers from the menu
  prm(Vector(double)) vec_prm;
  vec_prm.fill(mat_prm);     // make sure that vec_prm will match mat_prm

  Handle(Matrix(double)) A = createMatrix(double)(mat_prm);
  Handle(Vector(double)) x = createVector(double)(vec_prm);
  Handle(Vector(double)) b = createVector(double)(vec_prm);

  // ...
  // Initialize A and b, e.g., by a finite element assembly process
  // ...

  LinEqSystemPrec system(*A,*x,*b);

  // Attach preconditioner (left or right)

  prm(Precond) prec_prm;  // description of preconditioning method
  prec_prm.scan(global_menu);
  system.attach(prec_prm,*A);  // the preconditioner is based on matrix A

  // Create a LinEqSolver and solve the system

  prm(LinEqSolver) slv_prm;
  slv_prm.scan(global_menu);

  Handle(LinEqSolver) method = createLinEqSolver(slv_prm);
  method→solve(system);

  // the solution is now accessible as system→x() (or directly in *x)
}
```

Fig. 10. This generic application allows any combination of available matrix formats, preconditioners and linear solvers *at run-time*. The user input is supplied by the `Diffpack` menu system.

Fig. 10

(class `ConjGrad`), SOR preconditioning (class `PrecSOR`) and a stopping criterion based on a relative residual (class `CMRelResidual`). Note that the program must be edited and recompiled if the user wants to change one of the methods. Hence, in an application code, like a finite element solver, the interface used in Figure 10 gives much more flexibility for the user to play with different solution strategies.

4.2 Monitoring the convergence history

The iterations shown in Figures 8 and 9 are controlled by the function `satisfied`, which is inherited from the abstract class `IterativeSolver`. This function will at each call evaluate all active convergence criteria using information from the latest iteration and signal whether convergence is reached. In order to offer a wide selection of criteria, all convergence tests are organized as members of the class hierarchy `ConvMonitor`. As indicated by the term "convergence monitor", all subclasses in this hierarchy offer the possibility of recording the evolution of the observed entity as a function of the iteration number. Moreover, each `ConvMonitor` sibling can operate as a convergence criterion, possibly causing the iteration to halt, or merely as a monitor with no other purpose than accumulating the convergence history. To facilitate combinations of different convergence criteria, or possibly using one particular convergence test while monitoring other measures, it is convenient to organize these objects as a list represented by the class `ConvMonitorList`. The function `satisfied` will then traverse this list and call each list member's own function `satisfied`. The result obtained for all monitor objects acting as a convergence criterion is accumulated to give the value of the compound convergence test. Associated with each object there is also a relational operator, e.g. `AND` or `OR`, which defines how the accumulated result should be combined with the value obtained from this particular criterion. For simplicity and efficiency, the `ConvMonitorList` is evaluated from left to right without any change of precedence. For this reason, it should be possible to insert monitor objects at either end of the list, depending on the user's instructions.

Internally, most classes derived from `ConvMonitor` are grouped as *absolute*, e.g.

$$\|r^k\| \leq \varepsilon, \quad r^k = b - Ax^k,$$

or *relative*, such as

$$\|r^k\|/\|r^0\| \leq \varepsilon.$$

As for most other hierarchies, the initialization of a convergence monitor is based on a parameter object of type `prm(ConvMonitor)`. The actual norm $\|\cdot\|$ used to measure convergence is identified by the contents of this parameter block. Moreover, it is possible to specify other details, e.g. whether a relative criterion should be based on the (preconditioned) right-hand side rather than the initial (preconditioned) residual, or if the monitor should utilize inner products or eigenvalue information extracted from the iteration itself.

Most Krylov methods compute the original residual $r^k = b - Ax^k$, and possibly the preconditioned counterparts $s^k = C_L r^k$ and $z^k = C_R s^k$, by recursive updates. For badly conditioned systems, these recursions may lead to inaccuracies due to loss of orthogonality. In such cases, or in the case of basic iterations without residual storage, it may be necessary to force explicit computation of one or more

```
#include <ConjGrad.h>
#include <MatTri_double.h>
#include <PrecAlgebraic.h> // defines PrecSOR
void main (int nargs, const char** args) {
  standard Diffpack initialization procedures....
  const int n = 10; // no of equations
  MatTri(double) A(n);
  Vec(double) x(10), b(10); // solution and right hand side
  A.fill (0.0);
  for (int i = 1; i <= n; i++) {
      A(i,0) = 3; b(i) = 3*i; x(i) = 0;
  }
  prm(Precond) prec_prm;
  prec_prm.relax_SSOR = 1.3; // relaxation parameter for SOR and SSOR
  prec_prm.left = dpTRUE; // left preconditioning
  prec_prm.prec_tp = "PrecSOR";
  PrecSOR prec (prec_prm); // make preconditioner according to parameters
  LinEqSystemPrec system(A, x, b);
  system.attach(prec, A); // extend system with a precond.  based on A
  prm(LinEqSolver) slv_prm;
  slv_prm.basic_method = "ConjGrad";
  slv_prm.max_iterations = 10;
  CMRelResidual term_crit (1.0e-5); // alternative:  init w/prm(ConvMonitor)
  ConjGrad solver (slv_prm);
  solver.attach (term_crit);
  solver.solve (system);
}
```

Fig. 11. Example of a `Diffpack` program where the specific choices of matrix format, linear solver, preconditioner and termination criteria are made at compile/time.

Fig. 11

residuals by performing a matrix by vector product and subsequent preconditioning operations. Special versions of the residual-based convergence monitors are able to do this. However, since the requested extra operations are expensive, it is convenient to have the possibility of *delayed evaluation*. That is, two or more convergence monitors are combined, e.g. one using the recursive updates for r^k and one involving explicit computation of $b - Ax^k$. However, the expensive evaluation is delayed until a reasonable level of convergence is reached in terms of the recursively updated residual. This approach may allow a reliable convergence test at relatively low cost. In addition to the residual-based criteria, there are several other possibilities, e.g. by monitoring changes from one iterate to the next, or by observing changes to certain iteration parameters, see for instance Bruaset [Bruaset 1995] and references therein.

4.3 Communication between modules

As indicated in the previous section, certain information computed as part of the iterative algorithm may be useful for other parts of the framework, e.g. for a convergence monitor. First of all, residual-based convergence criteria need efficient access to the vectors r, s and z which are updated in each iteration. To meet this requirement, we introduce the concept of a *communication block* implemented as the class `LinEqCommBlk`, see Figure 12. This data structure have pointers to the dynamically allocated residual vectors, which are set as part of the initialization provided by the framework. When a certain convergence monitor is initialized, it establishes a connection to the communication block and directs its own private pointer to address the relevant vector. The benefit of a communication scheme based on shared memory of this type, is that any residual update is immediately available to the convergence monitor without the cost of extra memory space or data copying. This information is also available to any other external module linked through the same `LinEqCommBlk` object. However, since the communication is administered within the framework for iterative methods, the data flow is restricted to the relevant objects cooperating with the chosen solver.

Some iterations compute inner products internally, that may be useful for testing the convergence. In order to avoid unnecessary computations, such values can be shared between the actual solver and the convergence monitor in a way similar to the communication of residual vectors. In `LinEqCommBlk` we may have pointers to real values. If these pointers are set to point at the location of an evaluated inner product, say the result of (r, r), the convergence monitor may access this value directly. Certain methods compute special inner products, e.g. the product (r, z) used by the conjugate gradient iteration. Since this value is special to a given solver, the method class `ConjGrad` should use a specialized communication block, say `LECommBlkCG`, derived from the generic `LinEqCommBlk`. In this way the concept of communication blocks is extended by inheritance. Similarly, methods like conjugate gradients may use convergence tests based on iteration parameters, see for instance Kaasschieter [Kaasschieter 1988]. In this case the tailored structure `LECommBlkCG` may accommodate storage of these entities. However, since we may need all preceding iteration parameters in order to build a tridiagonal matrix, the relevant values have to be copied and stored as part of the communication block.

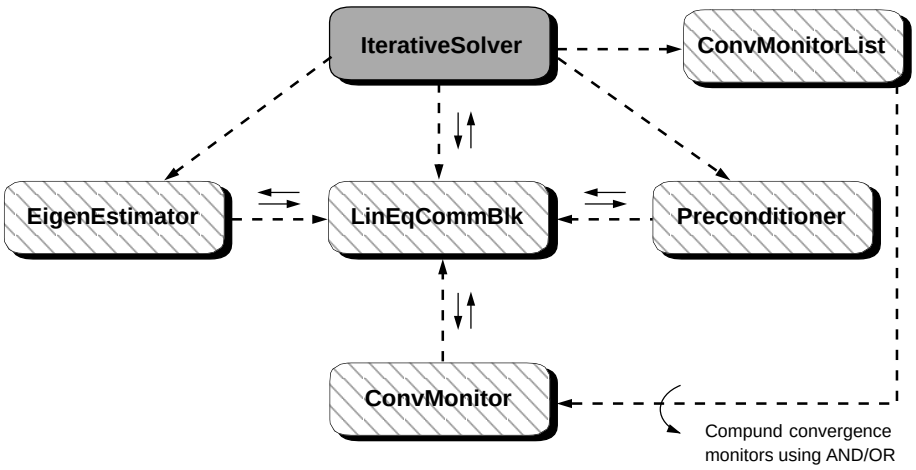

Fig. 12. All methods derived from class `IterativeSolver` have the possibility of communicating with other modules offered by the framework through a `LinEqCommBlk`. In this way, different modules as `ConvMonitor`s and `EigenEstimator`s may share information with each other as well as with the iterative solver itself.

Fig. 12

This action should be implemented in the member function `updateCommBlk`, which is part of the solver's interface, see Figures 8 and 9.

For some Krylov subspace methods, such as GMRES [Saad and Scultz 1986] and QMR [Freund and Nachtigal 1991], the residual vector is not explicitly available. Of course, when implementing such algorithms we should avoid the direct computation of the residual for every iteration. Instead, we may use tailored convergence measures utilizing other types of information that can be extracted from each iteration at low cost. Such convergence monitors are easily implemented by use of specialized communication blocks derived from the general-purpose class `LinEqCommBlk`. In fact, this approach may even be useful for solvers that do supply the residual vector in each iteration. For instance, in the conjugate gradient method we may choose to monitor one of the inner products computed as an integral part of the algorithm.

Given the possibility of transmitting iteration parameters to external objects like a convergence monitor, it is possible to interface to other utilities as well. In particular, it is now quite simple to implement external classes for estimation of spectral properties, i.e., the class `EigenEstimator` shown in Figure 12. Since the actual implementation of such estimators depend on the chosen solver, tailored versions should be derived as subclasses, e.g. `EigenEstimatorCG` that is capable of handling information extracted from a `ConjGrad` object via the link provided by a `LECommBlkCG`. Since the computed estimates may be needed by other parts of the environment, also the results should be accessible through the communication block. This exchange of data opens possibilities for easy implementation of adaptive preconditioners or adaptive relaxation methods based on spectral information. That is, the data flow may go back to the iterative solver, to a preconditioner via a `LinEqSystem` derivative, or to a convergence monitor.

5. COMPUTATIONAL EFFICIENCY

As mentioned in Section 2.3, there are several pitfalls when implementing CPU intensive algorithms within an object-oriented framework. It is hence interesting to evaluate the computational efficiency of a generic, object-oriented code like `Diffpack`, compared to specialized, hand-tuned C and Fortran implementations. For this purpose we have conducted a series of numerical experiments involving, for example, basic linear algebra operations and full finite element based simulators. Here we will present some results for basic linear algebra operations where high-level `Diffpack` code is compared to low-level, hand-tuned C code as well as vendor optimized Fortran implementations taken from the well recognized `BLAS` library. The `Diffpack` code applies dynamic memory management and the high-level abstract data types `LinEqMatrix` and `LinEqVector` to represent matrices and vectors. The Fortran and C implementations, on the other hand, make use of the primitive, static array types built into the language. The vendor optimized Fortran codes are, to our knowledge, based on a generic `BLAS` library and not on special assembly language implementations. For a full report on these comparisons, as well as tests run by full-scale simulators, we refer to Arge et al. [Arge et al. 1996].

We will report the results of two basic level 1 `BLAS` operations: (*i*) `DAXPY`: double precision vector update, i.e., $y \leftarrow \alpha x + y$; (*ii*) `DDOT`: double precision inner product, i.e., $dot \leftarrow x^T y$. Here, x and y are vectors of length n, α is a scalar, and $\rightarrow$ denotes

the assignment operator. Moreover, we also report the results of two matrix-vector products: (*iii*) `DGEMV`: double precision dense matrix-vector product, i.e., $y \leftarrow Ax$, where A is a dense $m \times m$ matrix; (*iv*) `SparseMatVec`: double precision sparse matrix-vector product, i.e., $y \leftarrow Bx$, where B is a sparse matrix with m rows and approximately 9 non-zero entries per row (B was generated from a finite element discretization of the two-dimensional Laplace operator). Note that `DAXPY`, `DDOT` and `DGEMV` appear in the `BLAS` library, while `SparseMatVec` has been coded directly in Fortran, using the storage scheme of the `Diffpack` class `MatSparse`.. The numerical experiments have been conducted on widely used workstations. More details about the hardware platforms are given in Table I.

Identifier	Model	Operating system	CPU/MHz	Memory (Mb)
IBM	IBM RS6000/590	AIX 3.2.5	Power2/66	256
SGI	Silicon Graphics Indigo2	IRIX 5.3	R4400/200	128
HP	Hewlett-Packard 9000/735	HP-UX 9.05	PA-RISC/99	144
SUN	Sun Sparcstation 10/512	SunOS 5.4	SuperSPARC/50	64

Table I. *The hardware platforms used for the numerical experiments.*

The results of the `DAXPY` and `DDOT` operations appear in the Figures 13 and 14. Since the sensitivity to n is small, the results for the largest n value, $n = 10^6$ are presented. We have divided the CPU times on each platform by the CPU time required by the Fortran code. The resulting quantity is referred to as normalized CPU time in the figure captions. All tests have been run 1,000 times to obtain reliable efficiency measures. It is seen from Figure 13 that in the `DAXPY` test Fortran gives the best performance, with C very close on HP and SUN computers. Among all the level 1 `BLAS` tests we have performed, also including the norm (`DNRM2`), vector copy (`DCOPY`), and vector swap (`DSWAP`), Fortran was most superior in terms of efficiency in the `DAXPY` operation. On the average, the results obtained for the `DDOT` operation seem to be more representative. Here the differences between C++, C and Fortran are small. The dense matrix-vector product (`DGEMV` operation) with $m = 1000$ gave similar results, as shown in Figure 15. For the sparse matrix-vector product (`SparseMatVec` operation) the results were even closer; in this example CPU times of the C++, C and Fortran implementations were practically equal and the plot is therefore omitted.

We believe that the current performance numbers indicate the somewhat surprising result that there are hardly any significant loss in programming with high-level abstractions like class `LinEqMatrix` and `LinEqVector`, making extensive use of OOP, compared to standard Fortran `BLAS` code. The reason for this conclusion is clear. Since OOP is limited to class management and high-level constructions, the actual CPU time consuming statements take place in member functions of the subclasses in the matrix and vector hierarchies where the numerical algorithms are implemented. In these member functions there are low-level, plain, long loops with standard array manipulations. It seems that the present compiler technology is capable of recognizing such primitive loops, regardless of whether they are programmed in Fortran, C or C++, and make use of optimal procedures for code optimization.. We mention that we have experienced significantly worse results with

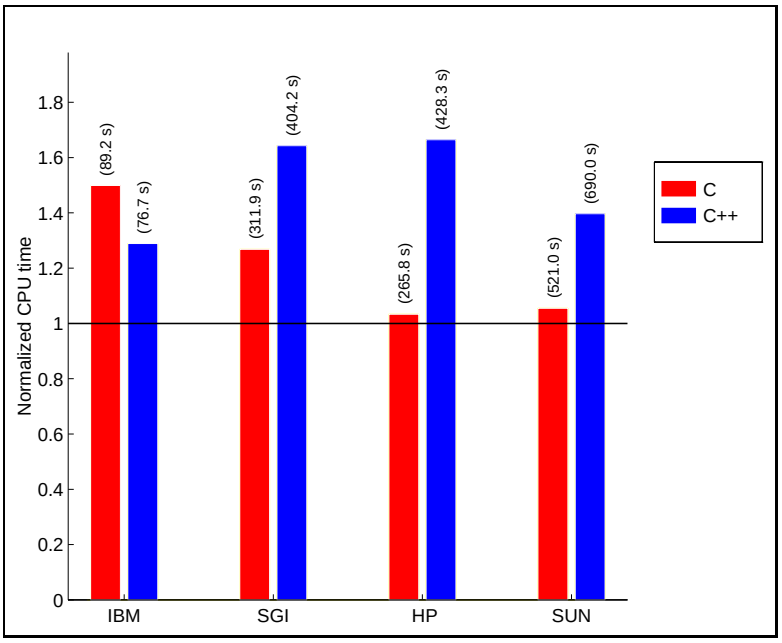

Fig. 13. *The normalized CPU time of the* `DAXPY` *operation.*

Fig. 13

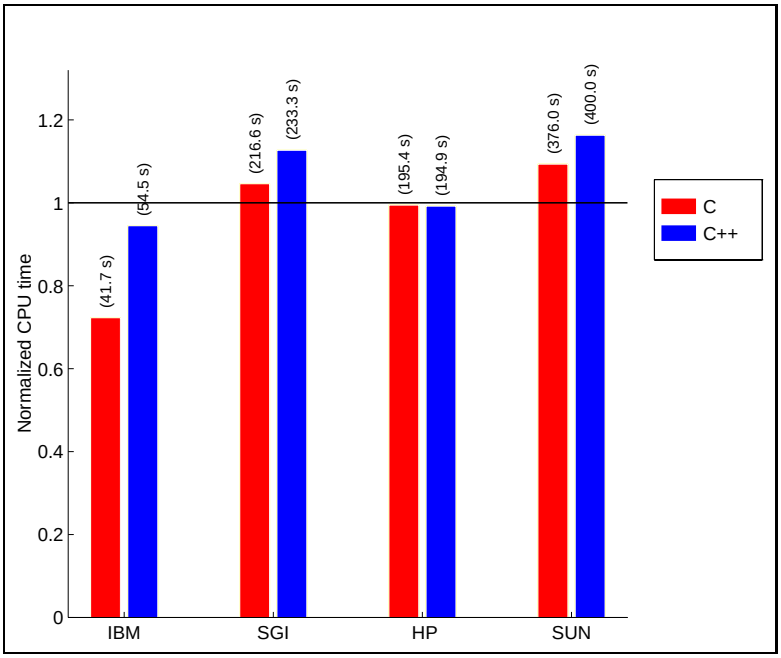

Fig. 14. *The normalized CPU time of the* `DDOT` *operation.*

Fig. 14

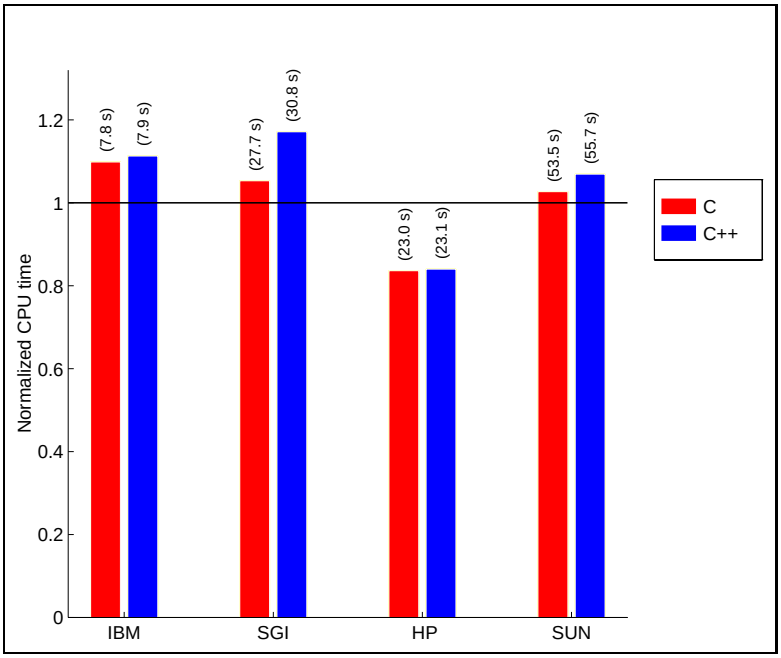

Fig. 15. *The normalized CPU time of a dense matrix by vector product.*

Fig. 15

older C++ compilers on some platforms, thus pointing out the recent development of mature C++ language implementations.

6. CONCLUDING REMARKS

In this paper we have described an application of modern object-oriented programming techniques for developing improved numerical software for preconditioned iterative methods. The improvement consists of creating modules with interfaces close to the mathematical operations in the problem, while hiding details related to data structures and algorithms. By grouping modules with a common interface in hierarchies, utilizing the techniques of object-oriented programming, we have obtained a library with very flexible building blocks. For example, basic modules for matrices, vectors, linear systems, preconditioners, iterative solvers and convergence criteria can be chosen by the user of a program at run-time and combined in almost any manner. The particular choices of matrix format, convergence criteria and preconditioners are completely transparent in an iterative solver such as the conjugate gradient method. Of course, such flexibility can be programmed using Fortran and standard procedural techniques. However, the object-oriented approach simplifies the task of creating the flexibility, mainly by offering tools for easy combination of existing modules into new modules, and by avoiding scattered if-tests for determining which modules that were actually chosen at run-time. Another advantage is that new modules can easily be implemented without affecting the existing code.

Our suggested design separates different parts of an algorithm that traditionally have been implemented as a contiguous code segment, e.g., by implementing an iterative solver and its convergence test as individual modules. This separation is essential to the flexibility. In order to avoid redundant storage and achieve high efficiency, it may be necessary to construct tailored communication modules that serve as a switchboard between the cooperating objects.

Although object-oriented methodologies may lead to very powerful applications, it is important to maintain a clearly layered design to achieve maximum efficiency. Typically, CPU-intensive computations should be implemented at the lowest levels where we are able to exploit private details of the data structures. The functionality at higher abstraction levels should then concentrate on preparing input and organizing calls to low-level functionality.

Most of the ideas presented in this paper are implemented in the the program system `Diffpack` which is available from `netlib` under a public access license. Further information can be obtained by accessing the `Diffpack` home page [Diffpack] on World Wide Web.

References

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1992. *LAPACK Users' Guide*. SIAM.

Arge, E., Bruaset, A. M., Calvin, P. B., Kanney, J. F., Langtangen, H. P., and Miller, C. T. 1996. On the numerical efficiency of c++ in scientific computing. In M. Dæhlen and A. Tveito, Eds., *Numerical Methods and Software Tools in Industrial Mathematics* (1996). (To be published by Birkhäuser Boston).

Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. A. 1993. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM.

Barton, J. J. and Nackman, L. R. 1994. *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley.

Bruaset, A. M. 1995. *A Survey of Preconditioned Iterative Methods, Pitman Research Notes in Mathematics Series* vol. 328. Longman.

Bruaset, A. M. and Langtangen, H. P. 1995. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, Eds., *Numerical Methods and Software Tools in Industrial Mathematics* (1995). (In preparation).

Coplien, J. O. 1992. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley.

Davies, R. B. 1993. Writing a matrix package in C++. In *Proceedings of OON-SKI'93* (1993).

Diffpack. The Diffpack WWW home page. (Use Mosaic or another WWW browser to load the URL `http://www.oslo.sintef.no/avd/33/3340/diffpack`).

Dongarra, J., Lumsdaine, A., Niu, X., Pozo, R., and Remington, K. 1994. Sparse matrix libraries in C++ for high performance architectures. In *Proceedings of OON-SKI'94* (1994).

Dongarra, J. J., Pozo, R., and Walker, D. W. 1993. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of OON-SKI'93* (1993).

Duff, I. S., Marrone, M., and Radicati, G. 1992. A proposal for user level sparse BLAS. Technical Report TR/PA/92/85, CERFACS, Toulouse, France. (SPARKER Working Note # 1).

Dyad Software. M++ Class Library. User's Guide. Dyad Software.

Eisenstat, S. C. 1981. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput. 2*, 1–4.

Ellis, M. A. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley.

Freund, R. W. and Nachtigal, N. M. 1991. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math. 60*, 315–339.

Gropp, W. and Smith, B. 1993. Simplified linear equation solvers users manual. Technical report ANL-93/8-REV 1, Argonne National Laboratory, Mathematics and Computer Science Division.

Heiser, G., Pommerell, C., Weis, J., and Fichtner, W. 1991. Three dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Transactions on Computer-Aided Design 10*, 1218–1230.

IML. The IML WWW home page. (Use Mosaic or another WWW browser to load the URL `http://gams.nist.gov/acmd/Staff/RPozo/iml++.html`).

Kaasschieter, E. F. 1988. A practical termination criterion for the conjugate gradient method. *BIT 28*, 308–322.

Metcalf, M. and Reid, J. 1992. *Fortran 90 Explained.* Oxford Science Publications.

Netlib. The Netlib WWW home page. (Use Mosaic or another WWW browser to load the URL `http://www.netlib.org`).

OONSKI93. 1993. *OON-SKI'93. Proceedings of the First Annual Object-Oriented Numerics Conference* (1993). Sunriver, Oregon.

OONSKI94. 1994. *OON-SKI'94. Proceedings of the Second Annual Object-Oriented Numerics Conference* (1994). Sunriver, Oregon.

PETSc. The PETSc WWW home page. (Use Mosaic or another WWW browser to load the URL `http://www.mcs.anl.gov/petsc/petsc.html`).

Pommerell, C., Annaratone, M., and Fichtner, W. 1992. New mapping and coloring heuristics for distributed-memory parallel processors. *SIAM J. Sci. Stat. Comput. 13*, 194–226.

Pommerell, C. and Fichther, W. 1991. PILS: An iterative linear solver package for ill-conditioned systems. In *Proceedings of Supercomputing '91* (1991). pp. 588–599. (IEEE-ACM, Albuquerque NM, November 991).

Rogue Wave Software. LAPACK.h++ User's Guide. Rogue Wave Software.

Saad, Y. 1992. Highly parallel preconditioners for general sparse matrices. Technical Report 92-087, Army High Performance Comput. Res. Center, University of Minnesota, Minneapolis.

Saad, Y. and Scultz, M. H. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7*, 856–869.

Zeglinski, G. W. and Han, R. S. 1994. Object oriented matrix classes for use in a finite element code using C++. *Int. J. Numer. Meth. Engrg. 37*, 3921–3937.