Object-oriented Numerics*

Erlend Arge Are Magnus Bruaset
Hans Petter Langtangen

September 7, 1999

Abstract

This chapter is concerned with the use of object-oriented programming techniques for numerical applications, especially in terms of the computer language C++. Through a series of examples we expose some of the strengths and possibilities of object-oriented numerics.

1 Introduction

Many fields of science rely on various types of mathematical models, typically used to describe dynamic processes in nature or for representation and analysis of information gathered from measured data sets. In most applications dealing with such models, computers are necessary tools in order to convert the researcher's intuition and experiences, via critical hypotheses and complicated equations, into numbers indicating success or failure. Throughout history, ranging from the innovative mechanical devices designed by Babbage and Newton to the turbo-charged electronical wonders of today, the need for doing numerical computations has strongly influenced the development of computing machinery. Today, we are routinely solving problems that would be impossible to attack with the help of pen, paper and brain alone. As the performance of computers increases, we will continue to stretch the limits of what is accepted to be computationally feasible enterprises. This thrilling development also puts extreme demands on quality management and robustness at all levels of the problem solving process. In particular, we need flexible but rigorous techniques and guidelines for software development.

Traditionally, numerical software development is based on use of procedural languages like Fortran and C. In large applications, the involved procedures are wrapped up in libraries, possibly also linking to other external libraries like the well-known BLAS routines for basic linear algebra operations [15]. These principles for software development have remained more or less static over the last decades, partly because of inherent limitations of simple procedural languages.

*This paper was originally published in *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pages 7–26. Birkhäuser, 1997.

Over the last few years, there has been an increasing interest in applying the paradigms of *object-oriented programming* (OOP) in numerical software development [2, 17, 18, 22]. This approach seems attractive due to well-defined mechanisms for modular design, re-use of code and for creating flexible applications that can be easily extended with respect to problem classes as well as solution methods. In short, OOP encourages computer implementations of mathematical abstractions. This statement is indeed supported by practical experiences made in the Diffpack [8] and Siscat [3] projects, respectively utilizing OOP for software development in the fields of partial differential equations and approximation of scattered data. Throughout this presentation we will use the term *object-oriented numerics* (OON) to denote the use of object-oriented programming techniques in numerical applications.

Hopefully, this chapter will reveal that OOP requires a new way of thinking about programming. At first sight, object-oriented implementations may often be more abstract and more difficult to understand than conventional codes. Usually, Fortran programmers need several months of practice before they will manage to improve the quality of their applications using OON. It is therefore natural to ask the question: Who needs object-oriented numerics? Modern scientific computing imposes two important demands on the implementation; those regarding efficiency and complexity. Efficiency typically consists of CPU-time and human time, the latter reflecting the work associated with implementation, testing and maintenance. The goal of OON is to keep the CPU-time constant while reducing the human time drastically. In addition, OON addresses complicated problems. The complexity can be on both the algorithmic and the application side. The building blocks of OON support a step-wise software development from simple problems to more complicated problems in a unified framework. The key to this goal is the reuse of already tested software components, which is a fundamental issue in OOP. Interfacing to other software packages is also made simpler by an object-oriented design. Finally, we will mention a very important requirement of software used for research, namely flexibility in the choice of problems and methods. The real advantage of OOP will in many cases be the possibility of building flexible codes for scientific experimentation. To summarize, comprehensive and complicated physical problems or numerical algorithms are candidates for object-oriented implementations. If your problems are solved by a few thousand lines of clean Fortran or C code, OON will probably not yield a substantial gain in human efficiency.

Although Fortran and C *can* be used to implement object-oriented designs [14, 19], the period of development is significantly reduced when applying a language with native support for OOP. Restricting our attention to such programming languages in general, it is observed that the lack of computational efficiency has proved to be the most important argument against object-oriented numerics. This aspect has indeed prevented serious use of OOP for numerical work until the advent of C++. This language can be seen as an extension to C, and with careful use it provides a viable alternative to Fortran and C. However, it should be emphasized that the use of certain language constructs that may be encouraged in C++ textbooks influenced by computer science,

prove to be disastrous in applications that require a high level of numerical efficiency. Based on the different, and sometimes conflicting, design criteria in the fields of computer science and scientific computing, we should keep in mind that "object-oriented numerics" and "object-oriented programming" are overlapping but not identical terms. Whenever the following presentation fails to be distinct in this respect, the phrase object-oriented programming should be related to the numerical context.

As initially mentioned, we address the main aspects of object-oriented numerics through a series of examples. However, we emphasize that this chapter is not an introduction to OOP or C++ in general. For such information we refer to available C++ textbooks, and in particular we recommend the textbook by Barton and Nackman [4] that presents OOP and C++ in a numerical setting. In the present chapter, Sections 2 and 3 use a simple matrix representation to exemplify modular design, information hiding and code extension through inheritance. Inheritance and virtual functions are used in Section 4 to construct a transparent interface between library routines and applications, before we finally focus on abstraction and top-down design in Section 5.

2 A motivating example

Consider the problem of solving the linear system $Ax = b$. Assuming that this system arises from a finite element or finite difference discretization of some boundary value problem, the matrix A tends to be large and sparse. Such problems can be efficiently solved by a conjugate gradient-like iteration combined with a suitable preconditioner, see [5] and references therein. For a Fortran programmer it is then attractive to use the `ITPACK` software, which is available from Netlib. Using this package, the problem at hand can be solved by a single call to the driver subroutine `NSPCG`, e.g.

```
CALL NSPCG (PREC,ACCEL,NDIM,MDIM,N,MAXNZ,COEF,JCOEF,P,IP,U,
            UBAR,RHS,WKSP,IWKSP,NW,INW,IPARM,RPARM,IER)
```

The experienced Fortran or C programmer may look at `ITPACK` as a perfect tool for solving the given problem. This is indeed true from a purely computational point of view. However, if we look critically at the software development side of the problem solving process, the call statement above suffers from many deficiencies.

The mathematical and numerical problem can be stated in a very compact form: Solve $Ax = b$ by an iterative method. Hence there are two basic quantities in question: A matrix system and a solver. The system consists of a coefficient matrix, the right hand side, a solution vector and an optional preconditioner. The solver consists of an iterative algorithm and its governing parameters. This clear structure is not reflected in traditional design of numerical software packages like `ITPACK`. For example, the sparse matrix A in the call statement is represented by the following list of parameters,

```
COEFF, JCOEFF, N, NDIM, MDIM, MAXNZ, P, IP.
```

Moreover, the routine `NSPCG` needs some work-arrays and associated parameters,

```
WKSP, IWKSP, NW, INW,
```

that have to be supplied by the user. In total, we need 20 parameters to represent and solve a linear system! Usually, large applications have several layers of subroutines outside the final call to the solver. Consequently, to transfer the linear system and the selected solver from one subroutine to another we have to send all the 20 parameter values. The names of these parameters are not self-explanatory, i.e., the call statements need a lot of additional documentation. If we forget one of the parameters, a Fortran program will not complain until that particular information is needed. Under such circumstances, the program behavior is undefined, often leading to considerable waste of efforts on debugging errors that should be caught at compile-time.

In this example we want to solve a problem that has a compact and well defined numerical and mathematical structure. Nevertheless, the resulting Fortran code is still of high complexity, thus demonstrating the need for improved software development techniques. For much more challenging problems, where the numerical and mathematical issues *are* complicated, Fortran programs tend to become unnecessarily complex. The problem is essentially that the abstraction level in Fortran is too low.

Let us take one step back and reconsider the mathematical problem in question. We want to solve the system $Ax = b$, where A is a matrix and x and b are vectors. In order to solve this system we need a representation of the *system*, i.e. A, x and b, and we need a *method* for solving it. Obviously, we want to be able to handle a lot of different types of matrices (sparse, banded, dense, etc.) and several different methods. However, the details of the methods and the matrices should not blur the simple structure of applying a *method* to a *system*. This calls for stronger types; we need to define abstract types representing the core features of our problem. In our particular problem it is desirable to define a type `LinEqSystem` that holds A, x and b, and a method type `LinEqSolver`. Here is how we would like to program the solution of a linear system:

```
// Given Matrix A, Vector x, Vector b
LinEqSystem system (A,x,b); // declare a linear system

// Given String method (e.g. method="Orthomin")
LinEqSolver solver (method);

solver.solve (system);
// Now, x contains the solution
```

When solving the linear system, we can focus on principles, rather than the details of sparse matrix storage schemes or user allocated work arrays. In this improved example `LinEqSystem` is a user-defined type, or *abstract data type* (ADT), representing a linear system. Moreover, there is an ADT named `LinEqSolver` that represents *all* available solution methods. First we declare the two basic numerical quantities (the objects `solver` and `system`) and then we apply the solver to the system. The resulting code is self-explanatory, and

it is easy to transfer the solver and the system to other routines. However, object-oriented programming is much more than the possibility of using ADTs. Characteristic features of OOP include *inheritance* and *virtual functions* (other more technical terms are *dynamic binding* and *polymorphism*), which are vital ingredients when realizing mathematical abstractions in terms of numerical building blocks. These issues will be further discussed in the remaining examples.

3 A simple matrix class

3.1 An example

Fortran is often regarded as the ideal language for numerical computations involving arrays. Suppose we want to compute a matrix-matrix or matrix-vector product. In the mathematical language we would express this as: Given $M \in \mathbb{R}^{p,q}$ and $B \in \mathbb{R}^{q,r}$, compute $C = MB$, $C \in \mathbb{R}^{p,r}$. Similarly, the matrix-vector product is defined as: Given $x \in \mathbb{R}^q$ and $M \in \mathbb{R}^{p,q}$, compute $y = Mx$, $y \in \mathbb{R}^p$.

Let us express these computations in Fortran. First, we define the relevant data structures,

```
integer p, q, r
double precision M(p,q), B(q,r), C(p,r)
double precision y(p), x(q)
```

Given these data items, we may simply call a function `prodm` for the matrix-matrix product and another function `prodv` for the matrix-vector product, i.e.,

```
call prodm (M, p, q, B, q, r, C)
call prodv (M, p, q, x, y)
```

This approach seems simple and straightforward. However, the Fortran expressions for the products involve details on the array sizes that are not explicitly needed in the mathematical formulations $C = MB$ and $y = Mx$. This observation is in contrast to the basic strength of mathematics: The ability to define abstractions and hide details. A more natural program syntax would be to declare the arrays and then write the product computations using the arrays by themselves. We will now indicate how we can achieve this goal in C++ using abstract data types. This means that we will create a new data type in C++, called `Matrix`, that is suitable for representation of the mathematical quantity *matrix*.

3.2 Abstract data types

Related to the example given above, what are the software requirements for the representation of a dense matrix? Clearly, one must be able to declare a `Matrix` of a particular size, reflecting the numbers of rows and columns. Then one must be able to assign numbers to the matrix entries. Furthermore, the

matrix-matrix and matrix-vector products must be implemented. Here is an example of the desired (and realizable) syntax:

```
// given integers p, q, j, k, r
Matrix M(p,q);          // declare a p times q matrix
M(j,k) = 3.54;          // assign a number to entry (j,k)

Matrix B(q,r), C(p,r);
Vector x(q), y(p);      // Vector is another type representing vectors
C=M*B;                  // matrix-matrix product
y=M*x;                  // matrix-vector product
```

Programmers of numerical methods will certainly agree that the example above demonstrates the desired syntax of a matrix type in an application code. We will now sketch how this user defined syntax can be realized in terms of a user defined ADT. This realization must be performed in a programming language. Using C++, abstract data types like `Matrix` are implemented in terms of a *class*. Below we list the `Matrix` class. For one who is new to C++, the syntax of the class definition may seem ugly. Moreover, one may think that the class implementation looks complicated even for something as simple as a matrix. However, the `Matrix` class is only a *definition* of a type `Matrix` that programmers can *use* in a very simple way like we have sketched above. OON in practice is not the detailed writing of matrix and vector classes, rather it is the usage of predefined basic mathematical quantities and perhaps building of new quantities in new classes where the programming is at a higher level than in a matrix class. Since the matrix class is a fundamental example on how basic mathematical ADTs are really implemented in the C++ language, we present the details of the class definition.

```
class Matrix     // Matrix M(p,q); => M is a p x q matrix
{
  private:
    double** A;    // pointer to the matrix data
    int      m,n;  // A is mxn

  public:
    //    ---  mathematical interface ---
    Matrix (int p, int q);                 // declaration: Matrix M(p,q)
    double& operator () (int i, int j);    // M(i,j)=4; s=M(k,l);
    Matrix& operator = (Matrix& B);        // M = B;
    void prod (Matrix& B, Matrix& result); // M.prod(B,C); (C=M*B)
    void prod (Vector& x, Vector& result); // M.prod(y,z); (z=M*y)
    Matrix operator * (Matrix& B);         // C = M*B;
    Vector operator * (Vector& y);         // z = M*y;
};
```

In this example, the type `Vector` refers to another class representing simple vectors in $\mathbb{R}^n$.

An ADT consists of *data* and *functions* operating on the data, which in either case are commonly referred to as *class members* in the C++ terminology. The data associated with a matrix typically consist of the entries and the size (the numbers of rows and columns) of the matrix. In the present case, we

represent the matrix entries by a standard C array, i.e., as a double pointer to a block of memory. The number of rows and columns are naturally represented as integers using the built-in C type `int`. In most cases, the internal data representation is of no interest to a user of the matrix class. Hence, there are parts of an ADT that are *private*, meaning that these parts are invisible to the user. Other possibilities for representing the matrix entries could include the use of a linked list. To maintain full flexibility, it is therefore important that the user's application program is completely independent of the chosen storage scheme. We can obtain this by letting the user access the matrix through a set of functions, specified by the programmer of the class. If the internal storage scheme is altered, only the contents of these functions are modified accordingly — without any changes to the argument sequences. That is, the syntax of any code that uses the `Matrix` class will be unaltered. Below we will give examples on alternative internal storage schemes for the `Matrix` class.

3.3 Member functions and efficiency

In the class `Matrix` we have introduced a subscripting operator for assigning and reading values of the matrix entries. This function is actually a redefinition of the parenthesis operator in C,

```
double& operator () (int i, int j);
```

thus providing the common Fortran syntax `M(r,s)` when accessing the entry (r, s). One might argue that issuing a function call for each matrix look-up must be very inefficient. This is definitely true. To circumvent such problems, C++ allows functions to be *inlined*. That is, these functions are syntactically seen as ordinary functions, although the compiler will copy the body of inlined functions into the code rather than generating a subroutine call. In this way, the subscripting operation is as efficient as an index look-up in a C-array (e.g., `A[i][j]`). The inline function can be equipped with a check on the array indices. If this check is enclosed in C++ preprocessor directives for conditional compilation, it can be automatically included in non-optimized code and completely removed in optimized code. This use of inline functions enables the programmer to achieve the efficiency of pure C, while still having full control of the definition of the call syntax and the functionality of the index operator.

The matrix-matrix product is carried out by a member function called `prod`. The corresponding matrix-vector product function has also the name `prod`. This convenient naming scheme is available since C++ can distinguish the two `prod` functions due to different argument types. Fortran and C programmers are well aware of the problems of constructing names for a new function. In C++ one can use the most obvious names, like `print`, `scan`, `prod`, `initialize` and so on, since the compiler will use the class name and the argument types as a part of the name. Such *function overloading* will significantly reduce the number of function names employed by a package, which makes the code much easier to use.

The desired syntax for a matrix-matrix product, `C=M*B`, can be achieved by redefining the multiplication operator,

```
Matrix operator * (Matrix& B);
```

Similarly, one can redefine the addition operator to make it work for matrices, i.e.,

```
Matrix operator + (Matrix& B);
```

Such constructs permit a compact and elegant syntax for compound expressions, e.g. `Q=M*B+P*R`, where `Q,M,B,P` and `R` are matrices. Unfortunately, in C++ such expressions can lead to loss of efficiency. This is what actually happens: `P` is multiplied by `R`, the result is stored in a temporary matrix `TMP1`. Then `M` is multiplied by `B` and the result is stored in a temporary matrix `TMP2`. The `TMP1` and `TMP2` matrices are added and the result is stored in a temporary variable `TMP3`, which is finally assigned to `Q`. The temporary variables are created at run-time, and the allocation of the associated data structures (C arrays in the implementation above) can be time consuming. It should also be mentioned that the temporary objects allocated by the compiler can be a waste of storage space, since the compiler is unaware of whether algorithmic data items, say `Q`, can be used to hold the results of intermediate computations. We emphasize that most of the user-friendly, interactive matrix computation systems, like `Matlab`, `S-Plus`, `Maple` and `Mathematica`, have exactly the same efficiency problem with compound expressions although it is seldom explicitly documented.

The `prod` function stores the result in a matrix `C` that the programmer supplies, while the `operator*` function allocates a matrix and returns it. Dynamic allocation is very convenient, but we feel that full user control of matrix allocation is an important issue for efficient code. The compound expression `Q=M*B+P*R` is most effectively evaluated by a special function that implements the compound operator `=*+*`. This is already a suggested part of C++, but not generally available at the time of writing. Instead one can create a function in class `Matrix`,

```
void add (Matrix& M, Matrix& B, Matrix& P, Matrix& R);
```

that computes the desired expression. Although one sacrifices the delicate syntax, the complexity of using this function is much less than for a similar Fortran code since the matrix size is a part of the matrix type itself. In particular, there is no need for explicit transfer of separate parameters related to the internal storage scheme.

3.4 Changing the internal storage structure

Instead of using a primitive C matrix construction to hold the matrix entries in class `Matrix`, we could have used a linked list. Of course, the subscripting operator will be slow for a linked list since the list must be searched to locate the requested matrix entry. On the other hand, intensive numerical computations

like matrix-vector products implemented in `prod` can still be efficient since one can operate on the linked list directly instead of using the subscripting operator. The usage of the `Matrix` type will not be affected by the internal details in the `prod` function. In fact, all numerical operations that involve heavy computations should be implemented as member functions of the relevant class, and one should avoid extensive usage of the defined index operator (`operator()`). Such an implementation will consist of low-level C code to obtain maximum efficiency. Our view of OON is to use the sophisticated features of object-oriented techniques to hide the low-level code and create user-friendly application interfaces. For example, in the Diffpack code [8] most of the heavy numerical operations take place in low-level code where C-style arrays are traversed using standard loops that the compiler's code optimizer can recognize. The computational cost of hiding such low-level code using OOP can be made negligible.

If the programmer of class `Matrix` believes that Fortran code will improve the efficiency, it is easy to let all computations be performed in Fortran. The storage of the matrix should then be in terms of a single pointer to a memory segment, and the storage scheme must be column-wise to avoid fragmented memory accesses. For each of the `prod` and `add` functions, we would then write a corresponding Fortran version and let the C++ member functions simply call the Fortran functions. In this way it is easy to build a user-friendly C++ interface to Fortran libraries like e.g. LAPACK. For a programmer who applies class `Matrix`, it is impossible to see whether it is Fortran, C or C++ code that does the numerical work.

3.5 Extension to sparse matrices

Some readers will say that hiding the matrix size is advantageous, but not very critical. Nevertheless, it is easy to make a small extension of our matrix example to show that the Fortran code grows in complexity while the C++ version does not. Consider a sparse matrix. From a mathematical point of view, the expression for the matrix-vector product remains the same as for a dense matrix. However, in the implementation it is important to take advantage of the fact that only the nonzero entries of the sparse matrix need to be stored and used in the algorithm. The interface of class `Matrix` should be the same regardless of the storage format. However, the internal data structure will be much more complicated for a sparse matrix since we need information on the location of the nonzero entries. To exemplify, a well-known storage scheme referred to as compressed row storage will need the following data structure,

```
class Matrix
{
  private:
    double* A;     // the nonzero matrix entries stored in a long vector
    int*    irow;  // indexing array
    int*    jcol;  // indexing array
    int     m, n;  // A is (logically) m x n
    int     nnz;   // number of nonzeroes
```

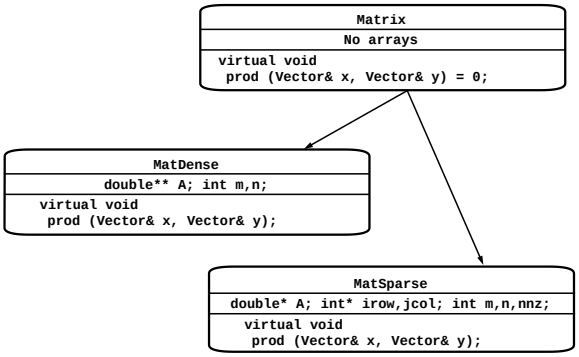

Figure 1: When organizing multiple matrix formats into a hierarchy, the application programmer relates to the abstract base class `Matrix` that defines a generic interface to matrix operations. The correct versions of member functions like `prod` are automatically chosen at run-time due to the C++ concept of dynamic binding. We refer to the remaining examples for further discussions of inheritance and virtual functions.

```
  public:
    Matrix (int p, int q, int nnz, int* irow, int* jcol); // constructor
    // exactly the same functions as in the example above
};
```

For a *user* of the class, the interface is the same. Hence an application code will not change when employing a sparse matrix instead of a dense matrix. As soon as we have *declared* a matrix, and given the information on whether it is dense or sparse, the storage details are inherent parts of the matrix representation and the programmer does not need to be concerned with this when using the matrix object. Refining this approach, the dense and sparse matrix formats could be organized together with other matrix implementations as a class hierarchy [6, 7]. Introducing a common base class that defines a generic interface to matrix operations, the user would then be able to dynamically switch between different storage schemes without changes to the application code, see Figure 1.

At this point, it is appropriate to discuss how the extension from dense to sparse matrices affects the implementation in Fortran. We will then have to explicitly transfer the sparse matrix storage structure to all functions operating on the matrices. For example, the code segment

```
integer p, q, nnz
integer irow(p+1), jcol(nnz)
double precision M(nnz), x(q), y(p)
...
call prodvs (M, p, q, nnz, irow, jcol, x, y)
```

is needed for a sparse matrix-vector product. In C++, this computation will still read `M.prod(x,y)`, where `M` is a sparse matrix object.

What about efficiency in this case? The implementation of the `prod` function would consist of running through the internal array `A` and multiplying each entry

by an indirectly indexed entry in x. Using a low-level C implementation based on pointer arithmetics or typical "do-loops" that are recognized by the compiler, this function will be as efficient as can be achieved in C. Usually, such layered designs can preserve all the advantages of data abstraction and simple syntax in C++, and still have the efficiency of pure low-level C programs. Performance comparisons of C versus Fortran are more delicate since each compiler's code optimizer plays a critical role. Some practitioners assert that Fortran compilers seem to produce better optimized code than what is obtained from C and C++ compilers. The authors have performed extensive comparisons of Fortran, C and C++ implementations of basic linear algebra operations on workstations [1]. The general conclusion is that the speed is the same for all these three languages. Even vendor-optimized BLAS [15] implementations in Fortran never decrease the execution time by more than 50 percent compared to straightforward C++ code. This observation suggests that the increased flexibility of genuinely object-oriented software results in an efficiency that is comparable to Fortran and only slightly less efficient than machine-optimal code. Fortran 90 is by many held as an alternative to C++ since it offers the computational strengths of Fortran bundled with some C++ features, see [16]. However, Fortran 90 does not support OOP, although the language permits the programmer to use ADTs. We refer to [1] for a more comprehensive treatment of computational efficiency

So far we have outlined a possible object-oriented design of dense matrices and looked briefly at its extension to sparse matrix structures. A more detailed discussion on matrix and vector implementations in the framework of OOP can be found in [6, 7]. In particular, these papers deal with hierarchical implementations of multiple storage schemes, as well as the use of *templates*, i.e., parameterization with respect to the data type used for the matrix entries. For other discussions and alternative views of object-oriented linear algebra tools, see for instance [9, 10, 11], as well the documentation of existing commercial packages like M++ [12] and LAPACK++ [21]. Several authors have also discussed software abstractions for numerical linear algbra in a finite element setting, see for instance [6, 14, 19, 23].

4 Flexible user interfaces

When writing application programs based on general purpose numerical libraries, a standard problem is the conversion between the data formats used in the application program and the formats used in the libraries. In the procedural setting, the library routines will require specific formats for the input and output, and it is only by chance that these formats are the same as in the application program. Thus, the application programmer is forced to do the tedious job of writing chunks of code for shuffling data from one format to another. This is time and storage consuming, makes the code less readable and is a source of errors. And last but not least, the implementation of data-shuffling is perhaps one of the most boring tasks a programmer may face.

The crux of the matter is the fact that no knowledge of the application

program data formats is available at the library level. In the procedural setting this gives the library programmer no choice but selecting a rather general format to which the application programmer has to relate. This motivates the need for language constructs supporting interface definitions at compile-time of the application development. In the domain of OOP such constructs are available through the concepts of class inheritance and virtual functions. As an example, assume that the library should contain functionality for evaluating some function of the form

$$f(x,y) = f(x,y \mid p^1, p^2, \ldots, p^n)$$

at the point $(x,y) \in \mathbb{R}^2$ given the n points $p^i = (x_i, y_i, z_i) \in \mathbb{R}^3$ as parameters. For example, one can think of the `f(x,y)` function as some interpolation algorithm that finds a value of a field at an arbitrary point when the field is known at some scattered points. In the application programs, the points p^i will typically be stored in some array or list structures, but the exact format is not known. By invoking the concepts of inheritance and virtual functions in the framework of C++ one can implement the function f independent of the actual storage format used in a specific application. To do this we define a generic interface to the points via an abstract class, say

```
class Points3D {
  public:
    virtual int    getN () =0;      // get number of parameter points
    virtual double getX (int i) =0; // get x-value of i'th point
    virtual double getY (int i) =0; // get y-value of i'th point
    virtual double getZ (int i) =0; // get z-value of i'th point
};
```

The member functions of class `Points3D` are specified to be *virtual* which implies that they can be overridden by functions in derived classes. Moreover, the qualifier '`=0`' makes the corresponding functions *pure virtual* which means that there is no associated code implemented at the level of the `Points3D` class. The code, together with the actual storage format, is expected to be provided at a derived class level, typically implemented by the application. The interface to the function f, using the abstract `Points3D` class, could then take the form

```
double f(double x, double y, Points3D* p);
```

The code of this function will access the parameters only through the interface defined by `Points3D`. Any instance of a class derived from `Points3D` will now be a valid third argument to `f`. Thus, the application programmer only needs to provide access to the applications storage formats by deriving specific classes from `Points3D`, e.g.

```
class MyPoints3D : public Points3D
{
  protected:
    double* p;
    int n;
```

```
  public:
    MyPoints3D(double* xyz, int nn) {
     p = xyz;
     n = nn;
    }

    virtual int    getN () {return n};
    virtual double getX (int i) {return p[i];}
    virtual double getY (int i) {return p[n+i];}
    virtual double getZ (int i) {return p[2*n+i];}
};
```

The internal data structure[1] assumes that the application stores the parameters in a single array of the form $(x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n)$. At the site of the call, the implementation would look like this:

```
Points3D* p = new MyPoints3D(xyz,n);
double val = f(x,y,p);
```

It is apparent that no copying of the data is necessary when using this technique. However, the generality of using inheritance and virtual functions is offered at the expense of using virtual functions which cannot be made inline. Thus, in the above example every access to the parameters implies the cost of a call to a virtual function. In certain applications this technique might be too expensive, and one has to rely on less general implementations. One possibility is to use the function overloading mechanism in C++, and write several versions of the function `f`, each corresponding to a specific parameter format.

5 Abstraction and top-down design

The design process is often recognized as the most important phase of OOP. In general this process consists of identifying and grouping logical entities, i.e., making useful abstractions of the problem at hand. In the context of numerics, the logical entities are often readily available in the form of mathematical abstractions. Thus, large parts of the design are already inherent in the underlying mathematical structures. In effect, numerical problems are often well suited for formulation in an object-oriented framework. This is not only true for the design of overall system structures; also at the algorithmic level concepts from OOP lend themselves naturally to the design.

In the following example we will demonstrate how these techniques can be exploited for the construction of triangulated surfaces, i.e., surfaces composed of surface patches defined on triangles in the plane. The basic operations needed are to build the triangulation from a set of n points

$$p^i = (x_i, y_i, z_i) \quad i = 1, 2 \ldots, n,$$

[1] The keyword `protected` indicates that the internal storage format (here given by the array `p` and its length `n`) is hidden from users of the class `MyPoints3D`. However, in contrast to the previously used directive `private`, protected class members can be freely accessed inside new classes derived from `MyPoints3D`.

and to evaluate the constructed surface at an arbitrary point (x, y). We want the program to read a collection of spatial points and corresponding field values, compute a triangulation and other necessary data structures, and then evaluate the field at some user given (arbitrary) points.

There are numerous triangulation algorithms. However, at the conceptual level the contents of them remain the same. We want our program to be independent of the particular triangulation algorithm that is actually used. This suggests an interface to triangulation algorithms defined through an abstract class, say

```
class Triangulation
{
  public:
    virtual void insert(Points3D* p) =0;
    virtual double operator () (double x, double y) const =0;
};
```

In other words, with a triangulation one should be able to insert points and evaluate the surface (defined up by a triangulation over the current point set). We have here used the abstract class `Points3D`, defined above to be an interface to the points in a triangulation. The two listed member functions define the main outside view of this triangulation. Of course, other member functions may be needed, but these will be specific for each derived class level.

From a top-down point of view, let us now construct a class called `SimplestDelaunay` which will create a Delaunay triangulation according to the specified interface. We will focus on simplicity rather than efficiency for this basic implementation. However, given the basic structure, the implementation of each task may be refined in different ways to achieve an acceptable level of efficiency. The suggested class interface reads as follows,

```
class SimplestDelaunay : public Triangulation
{
  protected:
    List(Triangle)     tlist;
    virtual void       insert (Node* node);
    virtual Triangle*  getTriangle (double x, double y) const;
    virtual double     eval (Triangle* triangle) const;

  public:
    virtual void       insert (Points3D* nodes);
    virtual double     operator () (double x, double y) const;
};
```

A natural data structure for holding the collection of triangles is a list of triangle objects. Each triangle object is in the implementation an ADT of type `Triangle`. We will not go into further details on the implementation of the `List` class used in this example.

A simple implementation of the member functions could be like the following:

```
void SimplestDelaunay :: insert (Points3D* nodes) {
  for (int i=1; i<=nodes->getN(); i++)
```

```
    insert(new Node(nodes->getX(i), nodes->getY(i), nodes->getZ(i)));
}

Triangle* SimplestDelaunay :: getTriangle (double x, double y) const {
  int i = 0;
  while (i<tlist.getN() && !tlist[i]->inside(x,y)) i++;  // search in list
  if (i==tlist.getN()) return NULL; else return tlist[i];
}

Triangle* SimplestDelaunay :: eval (Triangle* triangle) const {
  if (!triangle)
    return 0;
  else  // average of the corner values (=> piecewise const. surface)
   return ((triangle->getN1())->getZ() +
           (triangle->getN2())->getZ() +
           (triangle->getN3())->getZ())/3;
}

double SimplestDelaunay :: operator () (double x, double y) const {
  return eval( getTriangle(x,y) );
}
```

The Delaunay algorithm considers one point at a time, a behavior that is reflected by the way the member function `insert(Points3D*)` is implemented. The implementation of `insert(Node*)`, which is the most complicated member function, is not shown here. Moreover, the implementation of the evaluator is over-simplified. First, a search, possibly involving all triangles, is needed to find the one containing the point (x, y), see `getTriangle`. This simple strategy will work, but is of course extremely inefficient for large data sets. Thereafter, the value returned is just the average of the vertices of the corresponding triangle, see `eval`. This gives a piecewise constant surface, which for most applications is too crude. However, these two tasks are singled out as member functions and can later be overridden by more sophisticated implementations.

The most complicated task in class `SimplestDelaunay` is to insert nodes, or put more generally, to manipulate the structure of triangles. To design a convenient data structure, note that the basic entities forming a triangulation are the following:

- `Triangulation` - a collection of triangular patches;
- `Triangle` - a triangular patch, bounded by three edges;
- `Edge` - a line segment connecting two nodes;
- `Node` - a point in $\mathbb{R}^3$.

These entities obey natural containment relationships expressed through a `Node` being a part of an `Edge`, an `Edge` being a part of a `Triangle` and a `Triangle` being a part of a `Triangulation`. From the point of view of object-oriented design, such relations translate to class descriptions and encapsulation.

Furthermore, each triangle object should contain three edge objects, and each edge object should contain two node objects. In addition, a natural way to express neighbor relations between triangles is to make the edges know which triangles they belong to. Here are the above relations expressed in C++:

```
class Node
{
  protected:
    double x,y,z;           // the 3D point (x,y,z)

  public:
    ...
};

class Edge
{
  protected:
    Node *n1, *n2;          // end points of edge
    Triangle *left, *right; // the triangles it belongs to

  public:
    ...
};

class Triangle
{
  protected:
    Edge *e1, *e2, *e3;     // edges bounding the triangle

  public:
    ...
};
```

From this point on, we are ready to work out the interface functions needed in each of the above classes. Some of these are already specified in the member functions of class `SimplestDelaunay`. For example, `SimplestDelaunay::eval` requires that `Triangle` implements the members for extracting the nodes belonging to the triangle. In turn this implies that the `Edge` must offer similar member functions. The implementation of `SimplestDelaunay::insert(Node*)` will uncover the need for several member functions belonging to the various classes. Also new data members may be necessary. Thus, the thread of implementation and design goes the way from `SimplestDelaunay` down to the basic classes `Node, Edge` and `Triangle`. Of course, this top-down approach will uncover problems that might imply changes at higher levels of the design, but the main direction of design will be top-down.

When the class `SimplestDelaunay` is finished and working, we might want to expand the functionality and enhance the efficiency of some member functions. This can be done by adding more member functions, using function overloading or overriding the member functions in derived classes.

We should mention here that more sophisticated examples on OON are provided in [17, 18, 22]. A nice example on an object-oriented framework for a general ODE solver library is also described by Gustafsson [13].

6 Concluding remarks

Using a few selected examples, we have exposed some of the strengths and possibilities of object-oriented numerics. The concepts offered by object-oriented languages for data hiding, inheritance and virtual functions permit a close relationship between the implementation and the underlying mathematical abstractions. Using these tools to organize and implement numerical methods as parts of class hierarchies, we can derive very flexible building blocks for development of large-scale applications. A particularly attractive and user-friendly feature of such software is the ability to flexibly combine different numerical algorithms and data structures without disturbing the principal mathematical steps in the calling code with uninteresting implementational details. The resulting code is naturally divided into relatively small modules corresponding to the entities in the class hierarchies. This structured approach suggests that each module should be tested separately, thus promoting a high level of robustness. Compared to traditional procedure oriented programming in Fortran or C, object-oriented programming simplifies the development of the code, increases the robustness and reliability as well as decreases the maintenance efforts dramatically. The results are in practice measured as a significant increase in the human productivity.

Although object-oriented designs can be realized in terms of procedural languages like Fortran and C, the code development is considerably simplified by the use of a truly object-oriented language. Within this group of programming languages, C++ stands out as the only competitive alternative for numerical applications due to efficiency considerations. Even when using this tool, special care should be taken to avoid constructs that result in loss of efficiency or waste of storage space. The main rule proves to be that algorithms expected to carry out heavy computations should be implemented as member functions of the relevant classes using low-level C functionality. The sophisticated features of C++ are then used at higher abstraction levels for implementation of application interfaces and software administration, typically by organizing calls to the low-level functions.

References

[1] E. Arge, A. M. Bruaset, P. B. Calvin, J. F. Kanney, H. P. Langtangen, C. T. Miller. On the Numerical Efficiency of C++ in Scientific Computing. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 91–118, Birkhäuser, 1997.

[2] E. Arge, A. M. Bruaset and H. P. Langtangen, editors. *Modern Software Tools for Scientific Computing Methods.* Birkhäuser, 1997.

[3] E. Arge and Ø. Hjelle. Software tools for modelling scattered data. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 45–60, Birkhäuser, 1997.

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

[5] A. M. Bruaset. Krylov Subspace Iterations for Sparse Linear Systems. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 255–280, Birkhäuser, 1997.

[6] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. To appear in *ACM Trans. Math. Software*, 1996.

[7] A. M. Bruaset and H. P. Langtangen. Basic tools for linear algebra. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 27–44, Birkhäuser, 1997.

[8] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 61–90, Birkhäuser, 1997.

[9] R. B. Davies. Writing a matrix package in C++. In [17].

[10] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. Sparse matrix libraries in C++ for high performance architectures. In [18].

[11] J. Dongarra, R. Pozo, and D. W. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In [17].

[12] Dyad Software. *M++ Class Library. User's Guide.* Dyad Software.

[13] K. Gustafsson. Object-oriented implementation of software for solving ordinary differential equations. *Scientific Programming*, 2:217–225, 1993.

[14] W. Gropp and B. Smith. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. Preprint MCS-P356-0393, Argonne National Laboratory, Mathematics and Computer Science Division, 1993. (To appear in *Scientific Programming*).

[15] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–329, 1979.

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

[17] *OON-SKI'93. Proceedings of the First Annual Object-Oriented Numerics Conference*, 1993. Sunriver, Oregon.

[18] *OON-SKI'94. Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994. Sunriver, Oregon.

[19] The PETSc Package WWW home page.
(URL: `http://www.mcs.anl.gov/petsc/petsc.html`).

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

[21] Rogue Wave Software. *LAPACK.h++ User's Guide*. Rogue Wave Software.

[22] The SciTools'96 WWW home page.
(URL `http://www.oslo.sintef.no/SciTools96`).

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