# 1

# The Design of Sparse Direct Solvers using Object-Oriented Techniques

**Florin Dobrian**[1]
**Gary Kumfert**[1]
**Alex Pothen**[1,2]

ABSTRACT We describe our experience in designing object-oriented software for sparse direct solvers. We discuss *Spindle*, a library of sparse matrix ordering codes and `OBLIO`, a package that implements the factorization and triangular solution steps of a direct solver. We discuss the goals of our design: managing complexity, simplicity of interface, flexibility, extensibility, safety, and efficiency. High performance is obtained by carefully implementing the computationally intensive kernels and by making several tradeoffs to balance the conflicting demands of efficiency and good software design. Some of the missteps that we made in the course of this work are also described.

## 1.1 Introduction

We design and implement object-oriented software for solving large, sparse systems of linear equations by direct methods. Sparse direct methods solve systems of linear equations by factoring the coefficient matrix, employing graph models to control the storage and work required. Sophisticated algorithms and data structures are needed to obtain efficient direct solvers. This is an active area of research, and new algorithms are being developed continually. Our goals are to create a laboratory for quickly prototyping new algorithmic innovations, and to provide efficient software on serial and parallel platforms.

Object-oriented techniques have been applied to iterative solvers such as those found in Diffpack [LAN99, DIF] and PETSc [BGM+97, PET]. How-

[1]Department of Computer Science, Old Dominion University, Norfolk VA 23529-0162 USA. {dobrian,kumfert,pothen}@cs.odu.edu.
[2]ICASE, NASA Langley Research Center, Hampton VA 23681-2199 USA. pothen@icase.edu

ever, the application of object-oriented design to direct methods has not received the attention it deserves. Ashcraft and Liu have designed an object-oriented code called SMOOTH to compute fill-reducing orderings [AL96]. Ashcraft et. al. have created an object-oriented package of direct and iterative solvers called SPOOLES[APW+99, SPO]. Both SMOOTH and SPOOLES are written in C; we discuss SPOOLES in Sect. 1.9. George and Liu have implemented object-oriented user interfaces in Fortran90 and C++ for the SPARSPAK library, and they have discussed their design goals in [GL97].

We are interested in object-oriented design of direct solvers since we need to perform experiments to quickly prototype new algorithms that we design. We are also interested in extending our work to parallel and out-of-core computations, and to a variety of computer architectures. While several direct solvers are currently available, most of them are designed as "black boxes"—difficult to understand and adapt to new situations. We needed a robust and extensible platform to serve as a baseline, and a framework to support inclusion of new components as they are developed.

This project is the result of a balanced approach between sound software design and the "need for speed." We achieve the second goal by carefully implementing highly efficient algorithms. Our success here can be quantifiably measured by comparison to other implementations, both in terms of quality of the results and the amount of resources needed to compute it. Determining how well we achieve the first goal of good software design is much more subjective. In this arena, we argue on the grounds of *usability*, *flexibility*, and *extensibility*.

The research presented here tells the story of two separate but interrelated projects, parts of two PhD theses. The first project, Spindle, is a library of sparse matrix ordering algorithms including a variety of fill-reducing orderings. The second project, OBLIO, handles the remaining steps in solving a linear system of equations: matrix factorization and triangular solves. During development, ideas developed in one project were incorporated into the other when they were found to be suitable. As the projects matured, several design methodologies and programming techniques proved themselves useful twice. Currently the two projects have converged on most major design issues, though they are still maintained separately.

We present the work as a whole and concentrate on the commonalities of object-oriented design that have proved themselves in both projects. As this is ongoing research, we also present some equally good ideas that may not apply to both the ordering and solver codes, or that have not been incorporated into both yet. We find this convergence of design strategies somewhat surprising and most beneficial.

We explain the algorithms we chose and discuss the key features of the implementations. We also show the important tradeoffs that were made; often sacrificing more elegant applications of object-oriented techniques for very real improvements in efficiency. We include some missteps that

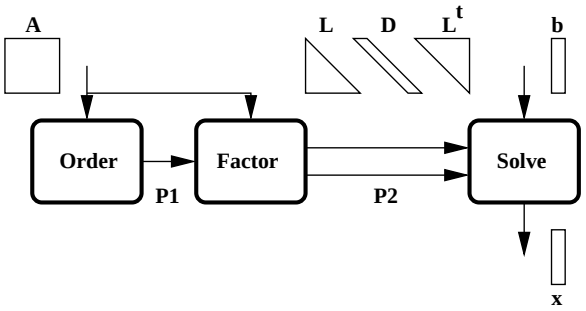



FIGURE 1.1. A "black box" formulation of a sparse direct solver.

were made in the course of our research and the lessons learned from the endeavor. Our code is written in C++.

Background information about direct methods is presented in Sect. 1.2. We identify the primary goals for our implementations in Sect. 1.3. The actual design of the software is presented in Sect. 1.4, with special emphasis on the ordering code in Sect. 1.5 and the factorization code in Sect. 1.6. We present the results of our software in Sect. 1.7. Finally, we discuss the lessons learned in course of our research in Sect. 1.9.

## 1.2 Background

The direct solution of a sparse symmetric linear system of equations $Ax = b$ can be described in three lines, corresponding to the three main computational steps: `order`, `factor` and `solve`,

$$P_1 = \text{order}(A), \tag{1.1}$$
$$(L, D, P_2) = \text{factor}(A, P_1), \tag{1.2}$$
$$x = \text{solve}(L, D, P_2, b). \tag{1.3}$$

Here $A$ is a sparse, symmetric, positive definite or indefinite matrix; $L$ is its lower triangular factor, and $D$ is its (block) diagonal factor, satisfying $A = LDL^T$; $P_1$ and $P_2$ are permutation matrices; $b$ is the known right-hand-side vector; and $x$ is an unknown solution vector. We illustrate this three-step "black box" scheme in Fig. 1.1.

Of the three steps, computing the factorization is usually the most time consuming step. For the rest of this section, we will discuss the significance of these three computational steps and their interplay.

### *1.2.1 Fill and the Elimination Forest*

The first step of the factorization of a symmetric positive definite matrix $A$ can be described by the equation

$$\begin{aligned} A \equiv A_0 &\equiv \begin{pmatrix} a_{11} & a_1^T \\ a_1 & \overline{A_1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0^T \\ l_1 & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & 0^T \\ 0 & A_1 \end{pmatrix} \begin{pmatrix} 1 & l_1^T \\ 0 & I_{n-1} \end{pmatrix}. \end{aligned}$$

By multiplying the factors, we see that $l_1 = (1/a_{11})a_1$, and $A_1 = \overline{A_1} - a_{11}l_1l_1^T$. Hence the first column of $L$ (the subdiagonal elements) is obtained by dividing the corresponding elements of $A$ by the first diagonal element; and the matrix that remains to be factored, $A_1$, is obtained by subtracting the term $a_{11}l_1l_1^T$ from the corresponding submatrix of $A$. At the end of the first step, we have computed the first column (and row) of the factors $L$ and $D$. The remainder of the factorization can be computed recursively by applying this elimination process to the first column of the submatrix that remains to be factored at each step.

Computing the factors of a symmetric indefinite matrix is a little more involved, since we have to consider the elimination of two columns (and rows) at a step if the first diagonal element of the submatrix to be factored is zero or small. We do not discuss the details of factoring an indefinite matrix here.

We observe from the equations above that the factor $L$ has nonzero elements corresponding to all subdiagonal nonzero positions in $A$; in addition, for $j > i$, if the $i$th and $j$th positions of $l_1$ are nonzero, then the $(i, j)$ element of $A_1$ is nonzero even when $A$ has a zero element in that position. These additional nonzeros introduced by the factorization are called *fill* elements. Fill increases both the storage needed for the factors and the work required to compute them.

An example matrix is provided in Fig. 1.2 that shows non-zeros in original positions of $A$ as "$\times$" and fill elements as "$\bullet$". This example incurs two fill elements. As a column $j$ of $A$ is eliminated, multiples of the $j$th column of $L$ are subtracted from those columns $k$ of $A$, where $l_{kj}$ is nonzero.

A graph model first introduced by Parter to model symmetric Gaussian elimination is useful in identifying where fill takes place. The adjacency graph of the symmetric matrix, $G = G(A)$, has $n$ vertices corresponding to the $n$ columns of $A$ labeled $1, 2, \ldots, n$. An edge $(i, j)$ joins vertices $i$ and $j$ in $G$ if and only if $a_{i,j}$ is nonzero. Since by symmetry, $a_{j,i}$ is also nonzero, the graph is undirected. It is conventional to not draw the loops $(i, i)$ corresponding to the $n$ nonzero diagonal elements $a_{ii}$.

The example in Fig. 1.2 illustrates the graph model of elimination. A sequence of elimination graphs, $G_k$, represents the fill created in each step of the factorization. The initial elimination graph is the adjacency graph of the matrix, $G_0 = G(A)$. At each step $k$, let $v_k$ be the vertex corresponding

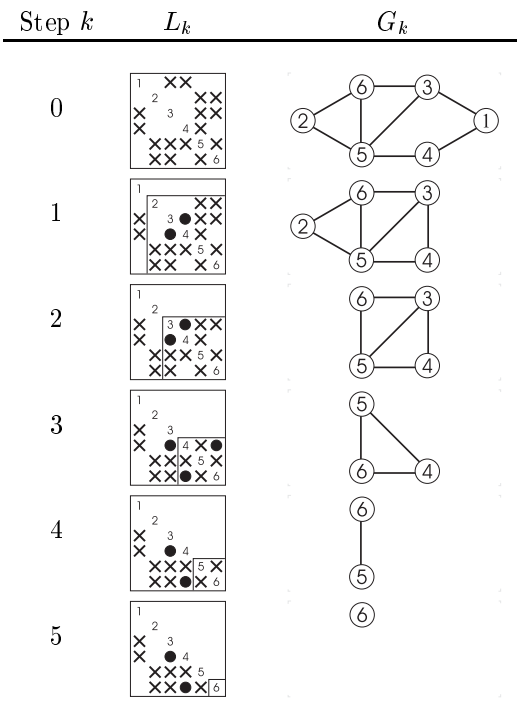



FIGURE 1.2. Examples of factorization and fill. For each step $k$ in the factorization, the nonzero structure of the factor $L_k$ and the associated elimination graph $G_k$ are shown.

to the $k^{\text{th}}$ column of $L_k$ to be eliminated. The elimination graph at the next step, $G_{k+1}$, is obtained by adding edges needed to make all vertices adjacent to $v_k$ pairwise adjacent to each other, and then removing $v_k$ and all edges incident on $v_k$. The inserted edges are *fill edges* in the elimination graph. This process is repeated until all the vertices are eliminated. We denote by $F$ the filled matrix $L + L^T$. The filled graph $G(F)$ (the adjacency graph of $F$) contains all the edges of $G(A)$ together with the filled edges.

When a column $j$ updates another column $k$ during the elimination, dependencies are created between the columns $j$ and $k$. An important data structure that captures these dependencies is an *elimination tree*, or more generally an *elimination forest*. The reader unfamiliar with elimination trees will find a comprehensive survey in Liu [LIU90]. The critical detail for our purposes is that elimination forests minimally represent the dependencies in the factorization. Each column in the matrix is represented as a node in the forest. The *parent* of a column $j$ is the smallest row index of a subdiagonal nonzero element (the first subdiagonal nonzero) in column $j$. Note that the definition of the elimination forest employs the factor and not the original matrix $A$. In our example, the parent of column three is four, and $l_{4,3}$ is a fill element.

An important concept that permits us to perform block operations in the factorization is that of a *supernode*. Supernodes are groups of adjacent vertices that have identical higher numbered neighbors in the filled graph $G(F)$. We group vertices into supernodes using the elimination forest; a supernode consists of a set of vertices that (1) forms a path in elimination forest, and (2) have the same higher numbered neighbors. In our example, the reader can verify that vertices 3 and 4, and vertices 5 and 6 form supernodes. This grouping of vertices into supernodes can then be used to define a supernodal elimination forest. In the filled matrix $F$, a supernode corresponds to a group of columns with a nested nonzero structure in the lower triangle.

Although we defined the elimination forest in terms of the factor $L$, in practice it is computed *before* the factorization. It turns out that efficient algorithms can be designed to compute the elimination forest from the given matrix $A$ and an ordering $P_1$ in time almost linear in the number of nonzeros in $A$. The elimination forest is then used in symbolic factorization algorithms to predict where fill occurs in the factor. Fill entries can only occur between a node in the elimination forest and its ancestors.

The elimination forest corresponding to the example in Fig. 1.2 is a single tree shown in Fig. 1.3. It can be shown that any postordering of the elimination forest leaves the fill unchanged.

### *1.2.2 Factorization*

The example in Fig. 1.2 illustrates a *right-looking* factorization, where updates from the current column being eliminated are propagated immedi-

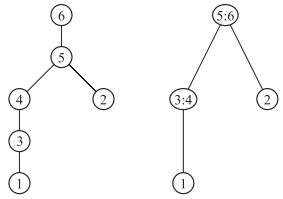



FIGURE 1.3. Examples of elimination forests. (a) An elimination forest of the example in Fig. 1.2. (b) The corresponding supernodal elimination forest, which is used in supernodal multifrontal factorization.

ately to the submatrix on the right. In a *left-looking* factorization, the updates are not applied right away; instead, before a column $k$ is eliminated, all updates from previous columns of $L$ are applied together to column $k$ of $A$. Another approach is a *multifrontal* factorization, in which the updates are propagated from a descendant column $j$ to an ancestor column $k$ via all intermediate vertices on the elimination tree path from $j$ to $k$.

OBLIO currently includes a multifrontal factorization code. In the multifrontal method, we view the sparse matrix as a collection of submatrices, each of which is dense. Each submatrix is partially factored; the factored columns are stored in the factor $L$, while the updates are propagated to the parent of the submatrix in the elimination forest. The multifrontal method and supernodal left-looking algorithms are suited to modern RISC architectures since the regular computations in the dense submatrices make better use of cache and implicit indexing.

We provide a high-level description of the multifrontal algorithm here, and the implementation details will be presented in Sect. 1.6. The multifrontal factorization is computed in post-order on the elimination forest. The algorithm creates a dense submatrix called a *frontal matrix* for each supernode. A frontal matrix is a symmetric dense matrix with columns and rows corresponding to the groups of columns to be eliminated and all the rows in which these columns have nonzeros. The frontal matrix is partially factored corresponding to the columns to be eliminated. The remainder of the frontal matrix forms a dense update matrix that will be stacked. The factored columns are copied into the sparse factor $L$. When all the children of a parent vertex in the elimination forest have been eliminated, the parent retrieves the update matrices of its children from the stack, and adds them into its own frontal matrix. The parent then has been "fully assembled"; it can then partially factor its frontal matrix, and propagate an update matrix to its parent in the elimination forest.

We illustrate the multifrontal factorization process in Fig. 1.4. At each step, $k$, we assemble the frontal matrix, $F_k$, by adding the entries in the $k^{\text{th}}$ rows and columns of $A$ and any update matrices, $U_k$. These contributions

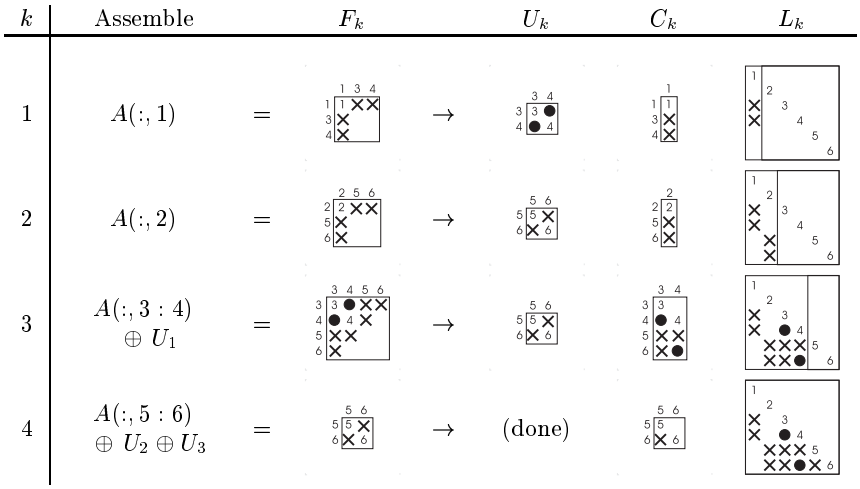



FIGURE 1.4. Example of multifrontal factorization. For each step $k$ in the factorization, the frontal matrix $F_k$ is assembled from the appropriate column(s) of $A$ and any update matrices from the children of supernode $k$ in the supernodal elimination forest. The fully assembled columns are factored; the factor columns $C_k$ are scatter-added to the sparse triangular factor $L$. Corresponding to the remaining columns in the factor matrix, a new update matrix $U_k$ is computed.

to $F_k$ vary in size, so special attention must be paid to indices; hence we call this operation *scatter-add*, and denote it by "$\oplus$". The frontal matrices then factor all columns that are "fully assembled." This looks identical to how $L_k$ was generated in Fig. 1.2, except that here, the frontal and update matrices are dense, not sparse. After the appropriate columns are factored from $F_k$, it is split into an update matrix $U_k$ and a factored column $C_k$, which is scatter-added to the sparse factor $L_k$.

### *1.2.3 Ordering*

The order in which the factorization takes place can change the structure of the elimination forest, and therefore greatly influences the amount of fill produced in the factorization. To control fill, the coefficient matrix is symmetrically permuted by rows and columns in attempts to reduce the storage(memory) and work(flops) required for factorization. Given the example in Fig. 1.2, the elimination order $\{2, 6, 1, 3, 4, 5\}$ produces only one fill element, instead of the two shown. This is the minimum amount of fill for this example.

If $A$ is indefinite, the initial permutation may have to be modified during factorization for numerical stability. This detail is captured in equation 1.2 and in Fig. 1.1 by the distinction between the two permutations, $P_1$ and $P_2$. The first ordering, $P_1$, reduces fill and the second ordering, $P_2$, is a

| Abbreviation. | Algorithm Name |
|---|---|
| MMD | Multiple Minimum Degree [LIU85] |
| AMD | Approximate Minimum Degree [ADD96] |
| AMF | Approximate Minimum Fill [ROT96] |
| MMDF | Modified Minimum Deficiency [NR97] |
| MMMD | Modified Multiple Minimum Degree [NR97] |
| AMMF | Approximate Minimum Mean Local Fill [RE98] |
| AMIND | Approximate Minimum Increase in Neighbor Degree [RE98] |

TABLE 1.1. Some of the algorithms in the Minimum Priority family.

modification of $P_1$ to preserve numerical stability. In general, any modification of $P_1$ increases fill and, hence, the work and storage required for factorization. If $A$ is positive definite, Cholesky factorization is numerically stable for any symmetric permutation of $A$.

Fill reduction is handled before the factorization step by ordering algorithms. Finding the elimination order that produces the minimum amount of fill is a well-known NP-complete problem. The heuristics used to compute fill-reducing orderings fall mainly into two classes: divide-and-conquer algorithms that recursively partition the graph and prevent fill edges joining two vertices in different partitions; and greedy algorithms that attempt to control fill in a local manner.

*Spindle* is a library of symmetric sparse matrix ordering algorithms — including most of the current "bottom-up" fill-reducing orderings. Though *Spindle* has native support for many matrix file formats, it does not implement a full matrix class and relies entirely on the graph model. Therefore while we talked about eliminating columns for the factorization, it is more natural to discuss eliminating vertices from the graph in this context. These greedy approaches choose which vertex to eliminate next based on some approximation of the fill its elimination will induce. An upper bound on the fill created by eliminating a vertex of degree $d$ is $d(d-1)/2$; hence in the minimum degree algorithm a vertex whose degree (or some related quantity) is minimum is eliminated next.

Multiple Minimum Degree (MMD) is an enhancement of the minimum degree algorithm that eliminates maximal independent sets of vertices at once to keep the cost of updating the graph low. Many more enhancements are necessary to obtain a practically efficient implementation. A survey article by George and Liu [GL89] provides all of the details for MMD. There have been several contributions to the field since this survey. A list of algorithms that we implement in our library and references are in Table 1.1.

### *1.2.4 Further Information*

This "breathless introduction" to sparse direct solvers is intended to help the reader to understand the basic computational problems in this area. For readers who desire further information, we recommend the following papers: the MMD ordering algorithm, [GL89]; the AMD ordering algorithm, [ADD96]; the multifrontal method, [LIU92]; elimination trees, [LIU90]; and symmetric indefinite solvers, [AGL99]. The books [DER86, GL81] were written earlier than these articles, nevertheless they would be helpful too.

## 1.3 Goals

Our design and implementation were guided by several principal goals we wanted to achieve. We list them briefly here, and then discuss each one in detail.

- Managing Complexity
- Simplicity of Interface
- Flexibility
- Extensibility
- Safety
- Efficiency

### *1.3.1 Managing Complexity*

Writing code for sparse direct solvers is a challenging task because the design requires sophisticated data structures and algorithms. To manage such complexity, proper abstractions must be used. Some of the earlier solvers were written in Fortran 77, which has very limited support for abstraction. As a consequence, the computation is expressed in terms of the implementation. For example, a sparse matrix is commonly represented as several arrays and the algorithms are a collection of subroutines which read from and write to these arrays. Since there is no support for abstract data types in the language, these subroutines tend to have long argument lists. Additionally, since Fortran 77 lacks dynamic memory allocation, all arrays are global, and are reused many times for different purposes. All these characteristics make the code extremely difficult to understand. Some of the more recent codes are written in C, but they tend to be influenced by the earlier codes in Fortran 77. For instance, the collection of subroutines is often the same.

In contrast to earlier codes, we wish to express the computation in terms of the mathematical formulation. This means programming in terms of vectors, trees, matrices, and permutations. This is possible in C++ because the language supports abstraction through classes, inheritance, and polymorphism. By programming at a higher level of abstraction, we are able to

implement code faster and to relegate minute details to lower layers of the abstraction.

### *1.3.2 Simplicity of Interface*

Simplicity is also achieved by good abstraction, but in a different sense. Whereas with complexity management we want to build layers of abstraction, here we focus on intuitive components with a minimal interface. Recall that the computation is formulated in terms of sparse matrices, vectors and permutations and in terms of algorithms like orderings, factorizations and triangular solves. Accordingly, we provide abstractions for such entities. The code for a solver that uses our libraries becomes then a simple driver that reads the inputs to the algorithms, runs the algorithms, and writes their outputs.

### *1.3.3 Flexibility*

The key to achieving flexibility in our codes is decoupling. We distinguish between data structures and algorithms, and design separate abstractions for structural entities such as sparse matrices, permutations and vectors on one hand, and for ordering and factorization algorithms on the other hand. One can and should expect to swap in different ordering or factorization objects in the solver in much the same way that components in a stereo system can be swapped in and out. In addition, one has the possibility of choosing the number of times different components are used. Ordering and factorization are performed only once in a series of systems with the same coefficient matrix with different right hand sides for example, while the triangular solves must be repeated for every system in the series. A similar situation occurs with iterative refinement, where triangular solves must be repeated for a single run of the ordering and factorization algorithms. It should also be possible to solve several systems of equations simultaneously, and to have them at various stages of their solution process.

Swapping components happens not only with data structures and algorithms; more generally, we want to swap smaller components within a larger one. For instance, factorization is composed of a couple of distinct phases (symbolic and numerical). Solvers for positive definite and indefinite problems differ only in the numerical part. To switch from a positive definite solver to an indefinite solver, we only have to swap the components that perform the numerical factorization. By splitting a factorization algorithm in this way we also provide the option of performing only the symbolic work for those who wish to experiment with the symbolic factorization algorithms.

The design challenge here is that flexibility and simplicity are at odds with one another. The simplest interface to a solver would be just a black box; one throws a coefficient matrix and a right hand side vector in one

end and produces the solution out the other end of the black box. While very easy to use, it would not be flexible at all. On the other hand, a very flexible implementation can expose too many details, making the learning curve steeper for users. In general, we provide multiple entry points to our code and let the users decide which one is appropriate for their needs.

### *1.3.4 Extensibility*

Whereas flexibility allows us to push different buttons and interchange components, extensibility allows us to create new components and alter the effects of certain buttons. This software is part of ongoing research and gets regularly tested, evaluated, and extended.

The best techniques we found for ensuring extensibility in our codes were to enforce decoupling and to provide robust interfaces. These practices also encourage code reuse. In the following sections, we will show an example of how two types of dense matrices in OBLIO, *frontal* and *update*, are used in numerical factorization and inherit characteristics from a dense matrix. We also present an example of how polymorphism is used in the ordering algorithms of Spindle.

Extensibility is not an automatic feature of a program written in an object-oriented language. Rather, it is a disciplined choice early in the design. In our implementations, we have very explicit points where the code was designed to be extended. Our intent is to keep the package open to other ordering and factorization algorithms. We are interested in extending these codes to add functionality to solve new problems (unsymmetric factorizations, incomplete factorizations, etc.) and to enable the codes to run efficiently in serial, parallel (with widely differing communication latencies), and out-of-core environments.

### *1.3.5 Safety*

We are concerned with two major issues concerning safety: protecting the user from making programming mistakes with components from our codes (compile-time errors), and providing meaningful error handling when errors are detected at run-time. Earlier we argued that the simplicity of the interface enhances the usability of the software, we add here that usability is further enhanced by safety.

Compile-time safety is heavily dependent on features of the programming language. Any strongly typed language can adequately prevent users from putting square pegs in round holes. That is, we can prevent users from passing a vector to a matrix argument.

Run-time errors are more difficult to handle. Structural entities such as sparse matrices and permutations are inputs for factorization algorithms. When such an algorithm is run, it should first verify that the inputs are valid, and second that the inputs correspond to each other.

The biggest difficulty about error handling is that while we, the library writers, know very well *how* to detect the error conditions when they occur, we must leave it to the user of the library to determine *what* action should be taken. Error handling is an important and all too often overlooked part of writing applications of any significant size. Because we are writing a multi-language based application and provide multiple interfaces (Fortran77, C, Matlab, C++) we immediately rejected using C++ exception handling.

The way in which errors are handled and reported in OOLIO and Spindle differ slightly in the details, but the strategies are similar. In both codes, the classes are self-aware, and are responsible for performing diagnostics to confirm that they are in a valid state. Instances that are in invalid states are responsible for being able to report, upon request, what errors were detected.

### *1.3.6 Efficiency*

An efficient piece of software is one that makes judicious use of resources. To achieve efficiency, data structures and algorithms must be implemented carefully. Previous direct solver packages tend to be highly efficient, using compact data structures and algorithms that are intimately intertwined in the code. Decoupling the data structures from the algorithms and requiring them to interact through high-level interfaces can add significant computational overhead. Choosing robust and efficient interfaces can be quite involved. The compromise between flexibility and efficiency is determined by these interfaces.

Consider the means by which an algorithmic object accesses the data stored in a structural object: e.g., a factorization operating on a sparse matrix. A rigorous object-oriented design requires full encapsulation of the matrix, meaning that the factorization algorithm must not be aware of its internal representation.

In practice, sparse matrix algorithms must take advantage of the storage format as an essential optimization. This is often done in object-oriented libraries like Diffpack and PETSc by "data-structure neutral" programming (see [BL97] and [BGM+97] respectively). This is accomplished by providing an abstract base class for a matrix or vector, and deriving concrete implementations for each data-layout: compressed sparse row, compressed column major, AIJ (row index, column index, value) triples, blocked compressed sparse, etc. Then each concrete derived class must implement its own basic linear algebra functions. Given $t$ types of matrices and $n$ basic linear algebra subroutines for each, these libraries provide $t \times n$ virtual functions.

Our goal is somewhat narrower: provide a solver that is as fast as any other solver written in any other language, but is more usable, flexible, and extensible, because we use object-oriented design and advanced pro-

gramming paradigms. Our algorithms are more complicated than a matrix-vector multiply, and providing a set of algorithms for each possible representation of the matrix is unreasonable. Even if we designed general algorithms that operate on matrices regardless of their layout, we could not make any guarantees about their performance. We are more concerned about adding new algorithms, not adding more matrix formats. Our codes apply to general, sparse, symmetric matrices that must be laid out in a specific way for efficient computations, and we provide tools to convert to this format. Specific formats are needed because we are forced to make a tradeoff: weaken the encapsulation to increase the performance.

In terms of running time, it is also important to realize which components of the code are the most expensive and to focus on efficiently implementing them. Usually, a sparse direct solver spends most of the time in the factorization step. A triply nested loop performs the bulk of the computation. Although most of the time is spent here, the code for the factorization represents just a tiny piece of the whole program. The rest of the code does not account for much of the whole time but it is certainly more sophisticated than this kernel. We use a layered approach, in which the focus is on software design in the higher layers and on efficiency in the lower ones. To get the best speed, we use a C subset or Fortran 77 code for kernels such as the triply nested loop inside the factorization.

Finally, we restricted our use of some specific object-oriented features. For example, we avoid operator overloading for matrices and vectors because this leads to the creation of several temporary objects and unnecessary data movement[3].

## 1.4 Design

This section tackles the major issues in designing our solver. We begin by explaining the base classes in the inheritance hierarchy in Sect. 1.4.1. Then in Sect. 1.4.2, we discuss the specialization and composition of our data structure and algorithm classes. We also discuss how we use iterators to get the algorithms and data structures to interact in Sect. 1.4.3. The important problem of I/O and handling multiple (often disparate) sparse matrix file formats is discussed in Sect. 1.4.4. Then we focus on implementation details specific to each package. The discussion of Spindle details in Sect. 1.5 highlights extensibility in using polymorphic fill reducing ordering algorithms. The corresponding section for OBLIO is Sect. 1.6.

[3]The reader is invited to see the article by Velduizen [VEL99] in this book which explains some solutions to this problem in detail.

```
class Object
{
  protected:
    Error error;

  public:
    Object() {error = None;}
    Error getError(void) const {return error;}
    virtual void print(const char *description) const = 0;
};
```

FIGURE 1.5. The Object Class.

### *1.4.1 Base Classes*

We have organized most of our heavy-weight classes into a single inheritance tree. The two main branches are `DataStructure` and `Algorithm`. Each matrix, permutation, and graph class inherits from `DataStructure` a common set of state information, interfaces for validating its state, and (optionally) resources to support persistent objects. Similarly classes that perform computational services such as ordering and factorization are derived from the `Algorithm` class, which provides support for algorithmic state information and the interface for running the algorithm. Both `DataStructure` and `Algorithm` are derived from a common ancestor, the `Object` class shown in Fig. 1.5. Although this class has no physical analogy, it does provide a suite of useful services to all its descendants such as error handling and instance counting. Through inheritance from `Object`, every structural or algorithmic object remembers when the most recent error occurred, if any. This error can be retrieved on demand and communicated to other classes.

The `DataStructure` class shown in Fig. 1.6 implements the validity checking mechanism specific to all structural objects. A structural object is usually initialized as invalid. After it is fully initialized, it must be validated. Later it can be completely reset. Again, the methods for validation and resetting are pure virtual, every structural class being required to implement its own behavior.

The `Algorithm` class shown in Fig. 1.7 handles the execution of algorithmic objects. Each algorithmic object requires a certain amount of time for its execution. This information can be retrieved for later use. The execution of any algorithmic object is triggered by the same interface, only the implementation is specific. This requires the method for running algorithmic objects to be pure virtual.

There are some distinctions between *Spindle* and OBLIO in their implementations. Both have forms of object-persistence through the `DataStructure` class, though the exact mechanisms differ. *Spindle* defines a wider variety

```
class DataStructure: public Object
{
    protected:
      bool valid;

    public:
      DataStructure() {valid = False;}
      bool isValid(void) const {return valid;}
      virtual void validate(void) = 0;
      virtual void reset(void) = 0;
};
```

FIGURE 1.6. The DataStructure class.

```
class Algorithm: public Object
{
  protected:
    float runTime;

  public:
    Algorithm() {runTime = 0;}
    float getRunTime(void) {return runTime;}
    virtual void run(void) = 0;
};
```

FIGURE 1.7. The Algorithm class.

of states for `DataStructure` classes to be in, and carries a related idea of state for the `Algorithm` classes. In both packages, however, the non-trivial concrete classes are derived from this fundamental hierarchy.

### *1.4.2 Algorithms and Data Structures*

We have seen several structural classes such as `Matrix`, `Vector`, `Graph` and `Permutation`. Instances of these classes are the inputs and outputs of algorithmic objects that order, factor, and perform the triangular solves.

The ordering algorithms, for example, require only the `Graph` of a `Matrix` as input, although there are additional options that advanced users could set. After the algorithm is run, the results could be queried. In *Spindle*, all the fill-reducing ordering algorithms can produce either a `Permutation` or an `EliminationForest` (or both) upon successful execution of the algorithm.

Ordering algorithms can enter an invalid state for several reasons. The input data could have been invalid, the user could have attempted to run the algorithm without sufficient input information, a run-time error might have been detected, or there might be a problem servicing the user's output request. In all these cases, the instance of the ordering algorithm can be reset and reused with different input data.

Algorithms can also be composed to obtain more sophisticated algorithmic objects. The factorization itself is made of several components, each derived from the `Algorithm` class; we provide details in Sect. 1.6. Having these subcomputations implemented separately is useful for many reasons. We subject each component to unit testing and later combine them during integration testing. It also helps in code reuse, since the factorization algorithms for positive definite matrices and indefinite matrices share several steps.

The derivation and composition of data structures is just as rich. Multifrontal factorization requires several dense matrix types that are derived from a common `DenseSymmetricMatrix`: these are `FrontalMatrix` and `UpdateMatrix`.

### *1.4.3 Iterators*

Separating data structures from algorithms makes it necessary to create interfaces between the modules. Inspired by the Standard Template Library (STL)—which also makes a distinction between data structures and algorithms—we explored the paradigm of *iterators* for this purpose. We first define the iterator construct, and then discuss how we used it. In later sections, we show examples where the iterator paradigm was applied with much success (Sect. 1.6) and where they were misapplied and reduced performance significantly (Sect. 1.9.2).

```
bool isAdjacent( const Graph& g, int i, int j ) const
{
  for( int k = g.adjHead[i]; k < g.adjHead[i+1]; ++k ) {
    if ( g.adjList[k] == j ) {
      return true;
    }
  }
  return false;
}
```

FIGURE 1.8. A C-like function that directly accesses the data in Graph.

Definition of an Iterator.

An *iterator* is a class that is closely associated with a particular container (data structure) class. The iterator is usually a "friend" class of the container granting it privileged access. Its purpose is to abstract away the implementational details of how individual items within the container are stored.

Assume, for example, that the list of all edges in a Graph is in the array `adjList[]` and that a second array `adjHead[]` stores the beginning index into `adjList[]` for each vertex in the graph. Then to check if vertex `i` is adjacent to vertex `j` we simply run through the arrays as in the piece of code in Fig. 1.8.

This design has a flaw, in that the function assumes the layout of data in the `Graph` class and accesses it directly. Consider now a different approach where the `Graph` class creates an iterator. Conventionally, an iterator class mimics the functionality of a pointer accessing an array. The dereference operator (`operator*()`) is overloaded to access the current item in the container, and the increment operator (`operator++()`) advances the iterator to the next item in the container. Rewriting our function above, as shown in Fig. 1.9, we define `Graph::adj_begin(int)` to create an iterator pointing to the beginning of the adjacency list of a specified vertex and `Graph::adj_end(int)` to return an iterator pointing to the end of the list[4].

The benefit of this second approach is that the function `isAdjacent` no longer assumes any information about how the data inside `Graph` is laid out. If it is indeed sequential as it was in the previous example, then the iterator could be simply a pointer to an integer data type. However, if the adjacency lists are stored in a different format, e.g., a red-black tree (for faster insertions and deletions), the iterator approach still applies for accessing the adjacency lists, since it assumes a suitable iterator class has

[4]Actually, it points to one past the end—a standard C++ convention.

```
bool isAdjacent( const Graph& g, int i, int j ) const
{
  for( Graph::adj_iterator it = g.adj_begin(i);
       it != g.adj_end(i); ++it ) {
    if ( *it == j ) {
      return true;
    }
  }
  return false;
}
```

FIGURE 1.9. A C++ global function that uses an iterator to search Graph.

been provided.

Application of Iterators.

Iterators provide a kind of "compile-time" polymorphism. They allow a level of abstraction between the data structure and the algorithm, but the concrete implementation is determined at compile time. This allows the compiler to inline function calls (often through several levels) and get very good performance[5].

There were a few difficulties in applying this technique to our problems. The most complicated aspect was that all STL containers are one-dimensional constructs. Many of our data structures, such as matrices and graphs, are two-dimensional. This was not a serious problem since we, as programmers, tend to linearize things anyway. In the example of iterators before, for instance, we used the iterator to iterate over the set of vertices adjacent to a particular vertex.

### *1.4.4 Input/Output*

An important problem that we often run into is sharing problems with other researchers. Whenever we agree to generate some solutions for a client (either academia or industry) we often find that we must adapt our code to a new file format. Attempts to standardize sparse matrix file formats do exist, most notably the Harwell-Boeing Format [DGL92], and the Matrix Market format [BPR+97]. However, we found it unreasonable to expect clients to restrict themselves to a small choice of formats. We found, in fact, that understanding the nature of this problem and applying object-oriented techniques is a good exercise in preparation for the harder problems ahead.

Perhaps the easiest way to handle sparse matrix I/O is to have a member

[5] C++ cannot inline virtual functions.

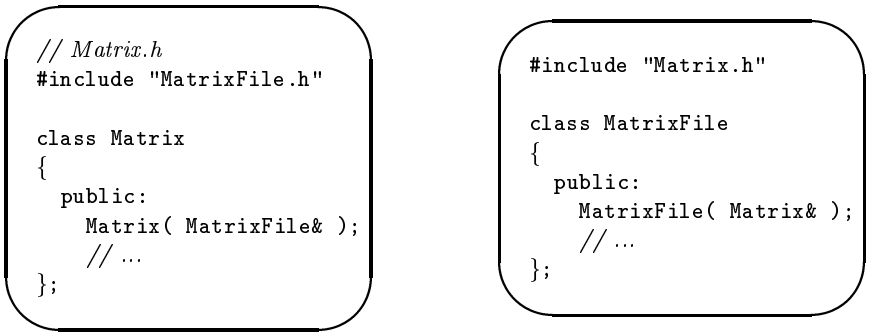



FIGURE 1.10. A first design of the Matrix and Matrixfile classes.

function of the matrix class to write a particular format and another to read. This is a simple solution, but it has a scalability problem. First, as the number of formats increase, the number of member functions grows and the matrix class becomes more and more cumbersome. Second, if separate matrix classes are needed then all of the I/O functions must be replicated.

The Chicken and Egg Problem.

One could reasonably create separate `Matrix` and `MatrixFile` classes. The immediate problem with this design is determining which begets which. One would expect to read a matrix from a file, but it also makes sense to generate a matrix and want to save it in a particular format as shown in Fig. 1.10.

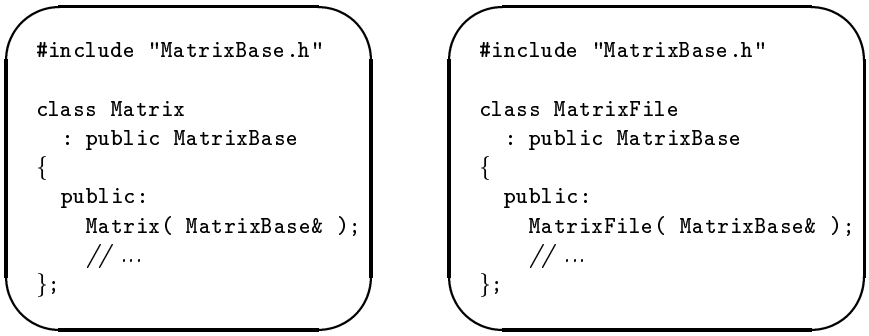
Such a design induces a cyclic dependency between the two objects, which is bad because such dependencies can dramatically affect the cost of maintaining code, especially if there are concrete classes inheriting from the cyclic dependency [LAK96, pg. 224]. This is exactly the case here, since the intention is to abstract away the differences between different file formats.

A solution is to escalate the commonality between the two classes. This has the advantage that the dependencies are made acyclic, the downside is that an additional class that has no physical counterpart is introduced for purely "software design" reasons. We will call this class `MatrixBase`, which is the direct ancestor of both the `Matrix` and `MatrixFile` classes. This second design in shown in Fig. 1.11.

Now we can derive various matrix file formats from the `MatrixFile` class, independent of the internal computer representation of the `Matrix` class. We will show later that the benefits compound when considering matrix to graph and graph to matrix conversions.

Navigating Layers of Abstraction.

It is important to understand that abstractions involved around the construct we call a "matrix" come from different levels and have different pur-



```
#include "MatrixBase.h"

class Matrix
  : public MatrixBase
{
  public:
    Matrix( MatrixBase& );
    // ...
};
```

```
#include "MatrixBase.h"

class MatrixFile
  : public MatrixBase
{
  public:
    MatrixFile( MatrixBase& );
    // ...
};
```

FIGURE 1.11. The Matrix and Matrixfile classes derived from a MatrixBase class.

poses. To define a class `Matrix` and possibly many subclasses, care must be taken to capture the abstraction correctly. It is hard to give a formula for designing a set of classes to implement an abstract concept. However, when the abstraction is captured just right, using it in the code is natural and intuitive. Often we have found that good designs open new possibilities that we had not considered.

For the matrix object, we have identified at least two dimensions of abstraction that are essentially independent, one from the mathematical point of view, one from computer science. Along the first dimension, the mathematical one, a matrix can be sparse, dense, banded, triangular, symmetric, rectangular, real or complex, rank-deficient or full-rank, etc. From a mathematical point of view, all of these words describe properties of the matrix.

From a computer science point of view, there are different ways that these two-dimensional constructs are mapped out into computer memory, which is essentially one-dimensional. Primarily, matrix elements must be listed in either row-major or column-major order, though diagonal storage schemes have been used. For sparse matrices, indices can start counting from zero or one. Layout is further complicated by blocking, graph compression, etc.

The critical question is: in all the specifications of matrix listed above, which are specializations of a matrix and which are attributes? The answer to this question determines which concepts are implemented by subclassing and which are implemented as fields inside the class.

The answer also depends on how the class(es) will be used. Rarely will a programmer find a need to implement separate classes for full-rank and rank-deficient matrices, but it is not obvious that a programmer must implement sparse and dense matrices as separate classes either. Matlab uses the same structure for sparse and dense matrices and allows conversion between the two. On the other hand, PETSc has both sparse and dense matrices subclassed from their abstract `Mat` base class.

A third dimension of complexity comes from matrix file formats, which

can be either a text or binary file, and more generally, a pipe, socket connection, or other forms of I/O streams.

Final Layout.

Our major concern was to have a flexible extensible system for getting matrices in many different formats into our program at runtime. We discuss in this section how we finally organized our solution. The inheritance hierarchy of the subsystem is shown in Fig. 1.12.

First we made non-abstract base classes `GraphBase` and `MatrixBase` which define a general layout for the data. From these, we derive `Graph` and `Matrix` classes that provide the public accessor/mutator functions; each class provides constructors from both `GraphBase` and `MatrixBase`. Furthermore, `Graph` and `Matrix` classes also inherit from the `DataStructure` class, which gives them generic data structure state, error reporting functionality, etc. In this way both can construct from each other without any cyclic dependency.

The final important piece before fitting together the entire puzzle is a `DataStream` class. This abstract base class has no ancestors. It does all of its I/O using the C style `FILE` pointers. We chose this C-style of I/O because, although it lacks the type-safety of C++ style `iostream`, it does allow us to do I/O through files, pipes, and sockets. This functionality is (unfortunately) not part of the standard C++ library.

If we try to open a file with a "`.gz`" suffix, the file object inherits from the `DataStream` class the functionality to open a `FILE` pointer that is in fact a pipe to the output of gunzip[6]. The `DataStream` class is therefore responsible for opening and closing the file, uncompressing if necessary, opening or closing the pipe or the socket, etc.; but it is an abstract class because it does not know what to do with the `FILE` once it is initialized. This class also provides the error handling services that are typical of file I/O.

To understand how all these partial classes come together to do I/O for a sparse matrix format, consider adding a new format to the library, a Matrix-Market file. To be able to read this format, we create a class `MatrixMarketFile` that inherits from `MatrixBase` and `DataStream`. This new class needs to implement two constructors based on `MatrixBase` or `GraphBase` and two virtual functions: `read( FILE * )` and `write( FILE * )` (in practice, it also implements many more accessor/modifier methods specific to the Matrix-Market format). Now we can read a Matrix-Market file, and create instances of either `Graph` or `Matrix` (or any other class that uses a `MatrixBase` in its constructor). Furthermore, from any class that inherits from `MatrixBase` or `GraphBase` we can write the object in Matrix-

[6]The "`.gz`" suffix indicates a file that is compressed with the GNU zip utility (`gzip`) and can be uncompressed by its complement, `gunzip`.

Market format. A graph-based file format, for instance Chaco [HL93], can be created using a similar inheritance hierarchy based on `GraphBase`.

## 1.5 Ordering Specifics

*Spindle* is a C++ library of sparse matrix ordering algorithms for reducing bandwidth, envelope, wavefront, or fill. We had earlier developed new envelope and wavefront reducing algorithms [KP97], but since they were implemented in C, we faced problems with complexity, scaling, and extensibility of the code. This motivated us to create *Spindle*. Although *Spindle* has several stand-alone drivers, it is intended to be used as a library, and is distributed with C, C++, Matlab, and PETSc calling interfaces[7].

In this section, we provide some examples of major concrete classes in *Spindle*, one from the `DataStructure` and one from the `Algorithm` branches of the inheritance hierarchy. In Sect. 1.5.1 we introduce the `QuotientGraph` class and describe its use. It will be revisited again in Sect. 1.9.2 when we discuss lessons learned. We also discuss the framework for implementing a family of algorithms in Sect. 1.5.2. We believe that this section provides a compelling example for how flexibility and extensibility can be achieved by proper design.

### *1.5.1 The Quotient Graph*

A *quotient graph* [GL80] is an implicit representation of the sequence of elimination graphs obtained during the factorization process. The quotient graph represents (implicitly) the elimination graph at any stage in the same amount of space as the original graph, even though the ordering algorithm creates new fill edges in the elimination graph. Thus the quotient graph is immune to the effects of fill; the price to be paid for this property is greater searching costs to determine from the quotient graph the adjacency set of a node in the elimination graph. The space efficiency helps avoid dynamic storage allocation for the ordering step, since the total fill is known only at the end of this step.

The quotient graph is an augmented graph with two distinct types of vertices: nodes and enodes. Nodes represent uneliminated nodes from the original graph, suitably grouped together; enodes represent groups of eliminated nodes that are adjacent to each other. The edges in a quotient graph join either one node to another, or a node to an enode. There are no edges connecting two enodes[8].

[7] Fortran interfaces are also possible, but have not been implemented yet.

[8] Conceptually, an edge could connect two enodes in some intermediate state, but it would immediately be removed and the enodes would be merged into one.

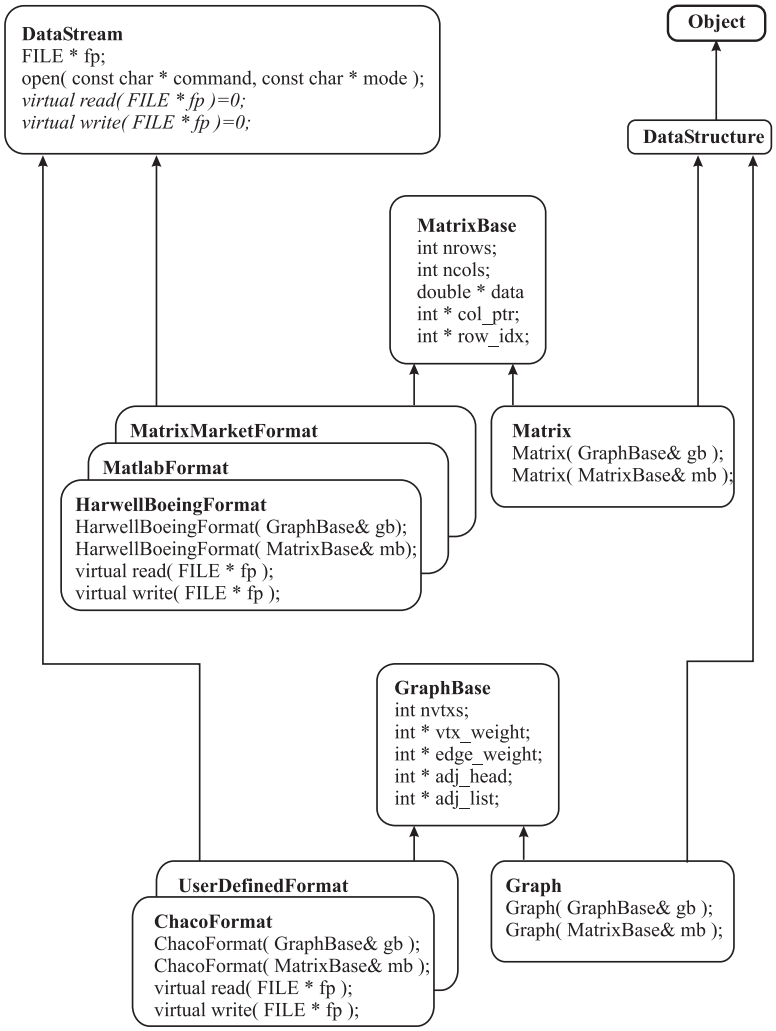
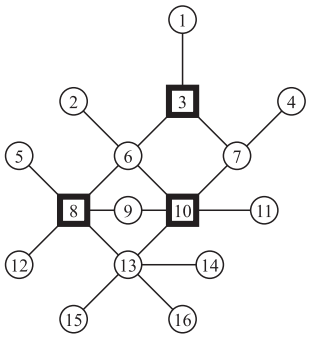



FIGURE 1.12. A fragment of the inheritance hierarchy highlighting how multiple file formats are implemented and extended. To add an additional format, create a class that inherits from `DataStream` and one of `GraphBase` or `MatrixBase`. Then implement the two pure virtual methods inherited from `DataStream`.




| node | reachable set | degree |
|---|---|---|
| 1 | 6 7 | 2 |
| 2 | 6 | 1 |
| 3 | N/A | |
| 4 | 7 | 1 |
| 5 | 6 9 13 12 | 4 |
| 6 | 1 2 5 7 9 11 12 13 | 8 |
| 7 | 1 4 6 9 11 13 | 6 |
| 8 | N/A | |
| 9 | 5 6 7 11 12 13 | 6 |
| 10 | N/A | |
| 11 | 6 7 9 13 | 4 |
| 12 | 5 6 9 13 | 4 |
| 13 | 5 6 7 9 11 12 14 15 16 | 9 |
| 14 | 13 | 1 |
| 15 | 13 | 1 |
| 16 | 13 | 1 |

FIGURE 1.13. An example of a quotient graph. The nodes are represented as circles, and enodes as boxes. The *reachable set* of a node $v$ is the union of the set of nodes adjacent to $v$, and the set of all nodes adjacent to enodes adjacent to $v$. The degree of a node in a quotient graph is the size of its reachable set, and not the cardinality of its adjacency set, as is usual in graphs.

The set of nodes adjacent to a node $v$ in an elimination graph can be computed from the quotient graph by computing the nodes that can be reached from $v$ in the latter graph. This reachable set of $v$ includes two groups of nodes: First, the set of all nodes adjacent to $v$ in the quotient graph; second, the union of the nodes that are adjacent to the enodes adjacent to $v$. The union of these two sets is the reachable set of $v$ in the quotient graph, and hence the adjacency set of $v$ in the elimination graph. Fig. 1.13 shows a sample quotient graph and the reachable sets and degrees for each node.

### *1.5.2 Polymorphic Fill Reducing Orderings*

One example where object-oriented implementation had substantial payoffs in terms of extensibility was in our ability to construct polymorphic fill reducing orderings. Recall from Table 1.1 that there are several different variants of these algorithms, many of which are quite recent. Currently there is no known library containing all of these algorithms, besides Spindle. While some of these heuristics are related, others—particularly MMD and AMD—are radically different in the ways priorities are computed, the underlying graph is updated, and in the allowed and disallowed optimizations. A fundamental distinction is that MMD allows multiple elimination, while AMD is restricted to single elimination. This means that MMD allows many vertices to be eliminated between each graph update (but the eliminated

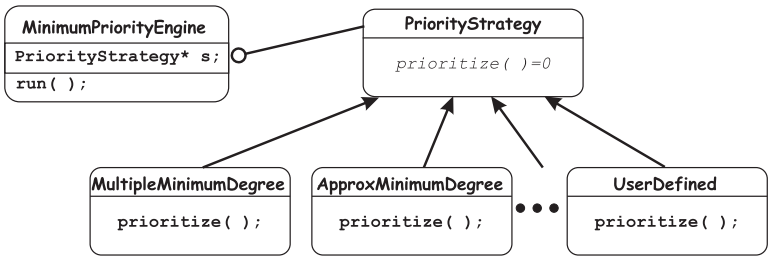



FIGURE 1.14. The Strategy Pattern.

nodes have to belong to non-adjacent supernodes). AMD does not require as much work per graph update, but the graph needs to be updated after every node is eliminated; this is a consequence of the way in which priorities are approximated in AMD.

Polymorphism was achieved using the Strategy Pattern [GHJ+95, pg. 315]. We created a complete framework for the entire family of minimum-degree like algorithms but deferred the ability to compute the (exact or approximate) degree of a node to a separate class in its own mini hierarchy. See Fig. 1.14 for the basic layout.

In this arrangement, the class `MinimumPriorityEngine` (which we will call `Engine` for short) is an algorithm that repeatedly selects a node of minimum priority from the given graph, eliminates it from the graph, and then updates the graph by adding appropriate fill edges when necessary. The catch is that it has no idea how to determine the priority of the vertices. It must rely on a `PriorityStrategy` class (`Strategy` for short), or more specifically, a specialized descendant of the `Strategy`.

This important design pattern offers several benefits. It provides a more efficient and more extensible alternative to long chains of conditional statements for selecting desired behavior. It allows new strategies to be implemented without changing the implementation of the engine. It is also more attractive than overriding a member function of the engine class directly because of the engine's overall complexity. This design also forces a separation between the work that is done and how the quality of work is measured.

There are potential drawbacks for using this pattern in general. The first drawback is that there is an increased number of classes in the library, one for each ordering algorithm. This is not a major concern, though users should be insulated from this by reasonable defaults being provided. A second drawback is the communication overhead. The calling interface must be identical for all the strategies, though individual types may not need all the information provided. A third drawback is the potential algorithmic overhead in decoupling engine and strategy. In our case, the engine could query the strategy once for each vertex that needs to be evaluated, though

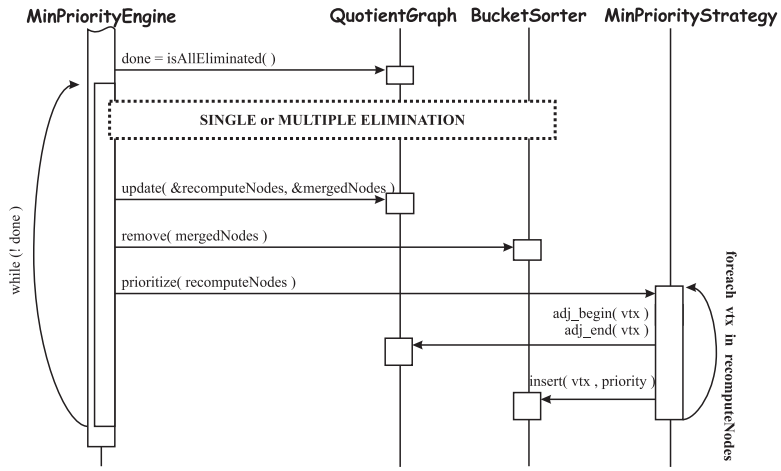



FIGURE 1.15. Interaction of the classes in the ordering step.

the virtual-function call overhead would become high. Alternatively, the engine might request all uneliminated nodes in the current quotient graph to be re-prioritized after each node is eliminated. This may result in too much work being done inside the strategy. Fortunately, in all these algorithms the only nodes whose priorities change are the ones adjacent to the most recently eliminated node. The `QuotientGraph` keeps track of this information.

For the `Engine` to work, a class must be derived from the `Strategy` abstract base class, and this class must override the pure virtual member function `computePriority`. The `Engine` is responsible for maintaining the graph and a priority queue of vertices. It selects a vertex of minimum priority, removes it from the queue, and eliminates it from the graph. The priority of all the neighbors of the most recently eliminated node is changed, so they too are removed from the priority queue for the time being. When there is no longer any vertex in the priority queue with the same priority as the first vertex eliminated at this stage, a maximal independent set of minimum priority nodes have been eliminated. The `Engine` updates the graph and gives a list of all vertices adjacent to newly eliminated ones to the `MMDStrategy` class. This class, in turn, computes the new priorities of these vertices and inserts them into the priority queue.

To make this setup efficient, we use a `BucketSorter` class to implement the priority queue and a `QuotientGraph` class to implement the series of graphs during elimination. The interaction of these four major objects is shown in Fig. 1.15. We hide the details of how single elimination and multiple elimination are handled. This too is determined by a simple query to the `Strategy` class. When the `QuotientGraph` is updated, it performs various types of compressions which may remove additional vertices or modify

the list of vertices that need their priority recomputed. When it calls the `Strategy` to compute the priorities, it provides a `const` reference to the `QuotientGraph` for it to explore the datastructure without fear of side-effects, and the `BucketSorter` to insert the vertices in.

To implement all of the ordering algorithms in Table 1.1 efficiently, we had to augment the Strategy Pattern. The `QuotientGraph` is required to behave in slightly different ways when updating for single elimination algorithms (e.g., AMD) and multiple elimination algorithms (e.g., MMD). Thus the `Engine` must query the `Strategy` what type is required and set the `QuotientGraph` to behave accordingly. This is handled in the first phase of the `run()` function that is overridden from its parent `Algorithm` class.

## 1.6 Factorization Specifics

OBLIO currently focuses on the factorization and triangular solve steps, and imports orderings from other libraries such as Spindle. Here we restrict our discussion to the factorization algorithms, which are the most complex. All abstractions associated with the factorization are built incrementally, beginning from the base classes described in Sect. 1.4.

### *1.6.1 The First Structural and Algorithmic Classes*

The initial abstractions result naturally from the high level formulation of the factorization, illustrated in Fig. 1.1. We need to describe coefficient matrix objects, permutation objects and factor objects, and for these OBLIO provides the following three classes: `SparseSymmMatrix`, `Permutation`, and `SparseLwTrMatrix`.

A coefficient matrix is described by the `SparseSymmMatrix` class. Internally, the matrix data is stored using the compressed column format: for each column there is a set of numerical values and a set of row indices that correspond to these values. In addition, an index array contains the location of the first element in each column in the value and row index arrays. The `Permutation` class describes permutations by storing two maps: a map from old to new indices and another from new to old indices. The `SparseLwTrMatrix` class describes both the triangular and diagonal factors. While this abstraction may look awkward, it is done with a specific intent. Note that the diagonal elements of $L$ do not have to be stored explicitly, which allows the elements of $D$ to replace them. Since the factors are usually manipulated together, this unified implementation is more efficient. The storage of the data is similar to the one used in the `SparseSymmMatrix` class, with the difference that row indices are compressed according to the supernodal structure of the factors.

OBLIO provides a separate abstraction for each factorization algorithm.

```
class PosDefMultFrtFactor: public Algorithm
{
  private:
    const class SparseSymmMatrix *a;
    class Permutation *p;
    class SparseLwTrMatrix *l;
    // ...
  public:
    PosDefMultFrtFactor(const class SparseSymmMatrix,
                        class Permutation,
                        class SparseLwTrMatrix);
    // ...
};
```

FIGURE 1.16. An algorithm connects to its inputs and outputs by means of pointers to the corresponding data structures.

Currently, it supports only multifrontal factorizations, for both symmetric positive definite and indefinite problems. In the indefinite case we make use of the Duff-Reid [DR83] pivoting strategy now, but we plan to extend the library with other strategies too, for example Aschcraft-Grimes-Lewis [AGL99]. Pivoting strategies are discussed in detail in the latter paper. Also, the library can be easily extended with other factorization algorithms, such as the left-looking algorithm. The current factorization classes are `PosDefMultFrtFactor`, and `IndefMultFrtFactor`.

### *1.6.2 Connecting Data Structures and Algorithms*

Data structures are inputs to and outputs for algorithms. We associate data structures with algorithms by using pointers to structural objects inside algorithmic classes. The inputs to a factorization algorithm are the coefficient matrix and the sparsity preserving permutation. The outputs are the factors and the permutation that provides stability. We make the choice of composing the two permutations, so the output permutation replaces the input permutation after the factorization algorithm is run. Fig. 1.16 describes the connection between a factorization algorithm and its input and output data structures.

Inputs and outputs can be associated with an algorithm through the constructor of the algorithm class, or at a later stage (not shown in Fig. 1.16). This provides the flexibility to change them at any time.

### *1.6.3 The Factorization in More Depth*

More abstractions are needed deeper inside the factorization algorithms. First, there is the elimination forest, which guides both the symbolic and

```
class PosDefMultFrtFactor: public Algorithm
{
  private:
    // ...
    ElimForest f;
    CompElimForest elm;
    SymFactor sym;
    PosDefMultFrtNumFactor num;
  public:
    // ...
};
```

FIGURE 1.17. Composition within a multifrontal factorization algorithm.

the numerical phases of the factorization. The forest stores parent, child and sibling pointers for every node. This way it can be traversed both bottom-up and top-down. The symbolic and numerical factorization proceed bottom-up (postorder) but require child and sibling information. The parent, child and sibling pointers are stored for both nodal and supernodal versions of the forest. There are also three major algorithms inside the factorization: algorithms to compute the elimination forest, the symbolic factorization, and the numerical factorization. Positive definite and indefinite solvers differ only in the numerical part, so separate numerical factorization abstractions are needed. Thus we have the classes: `ElimForest`, `CompElimForest`, `PosDefMultFrtNumFactor`, `IndefMultFrtNumFactor` and `SymFactor`.

A factorization class is based on these new classes. To define it we use composition, as shown in Fig. 1.17.

By splitting the factorization algorithm into its three algorithmic components, we give the user the possibility of running these components individually. This flexibility is needed in many situations. When some types of nonlinear equations are solved by successive linearization, the zero-nonzero structures remain constant while the numerical values change from one iteration to the next. In this case, only the numerical factorization algorithm needs to be run at each iteration. This splitting also aids code reuse since the algorithms for computing the elimination forest and the symbolic factorization are implemented only once but are used by both positive definite and indefinite factorizations.

Three more abstractions are required within the numerical factorization. They correspond to the frontal and update matrices and to the update stack. Since both frontal and update matrices are dense we first provide a general abstraction for dense matrices. In addition to the numerical values, frontal and update matrices must also store two types of maps: the map from local to global indices and vice versa. The corresponding classes are `DenseSymmMatrix`, `FrontalMatrix`, `UpdateMatrix`, and `UpdateStack`.

Composition is used again within the numerical factorization algorithms,

```
class PosDefMultFrtNumFactor: public Algorithm
{
  private:
    // ...
    FrontalMatrix fm;
    UpdateMatrix um;
    UpdateStack u;
  public:
    // ...
};
```

FIGURE 1.18. Composition within a multifrontal numerical factorization algorithm.

as shown in Fig. 1.18. Also, the update stack is composed of update matrices.

Figs. 1.19 and 1.20 summarize our discussion above. They provide a high level description of the OBLIO classes associated with the factorization from two perspectives, showing inheritance and composition relationships.

### *1.6.4 The Interaction between Data Structures and Algorithms*

Because of the conflict between high level abstractions and efficiency, the interaction between data structures and algorithms is a major concern in OBLIO. We have made several tradeoffs here rather than adopt a uniform solution in all such situations. There are cases when efficient interfaces make algorithmic classes aware of the internal representation of structural classes. In other situations we were able to achieve more abstraction. The code in Fig. 1.21, which is the kernel of a numerical factorization algorithm, is representative from this perspective. The loop traverses the supernodal elimination forest in postorder using iterators. For each supernode the frontal matrix assembles original numerical values from the coefficient matrix; for a non-leaf supernode, update matrices from the update stack are also assembled into the frontal matrix. Partial factorization is then performed on the frontal matrix, eliminated columns are saved in the factors, and the update matrix is pushed onto the stack.

Note that most of the execution time is spent inside the partial factorization of the frontal matrices. Accordingly, we had to make sure that this operation is performed efficiently. The factorization is a triply nested loop which needs to be carefully written. To make sure that we get the best performance, we have written it in Fortran 77 in addition to C++, since Fortran 77 compilers tend to generate more efficient code than C++ compilers.

In terms of efficiency, note also that we perform storage allocation for

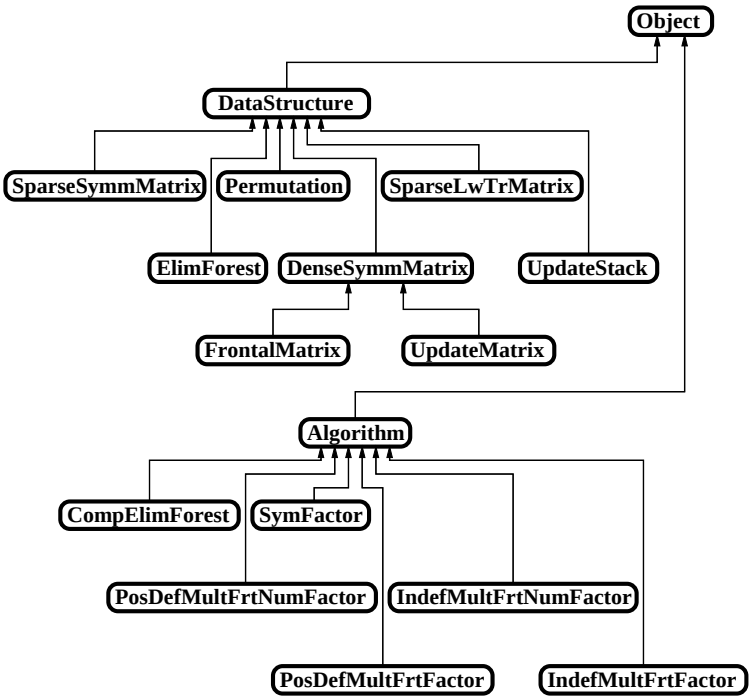



FIGURE 1.19. Inheritance relationships.

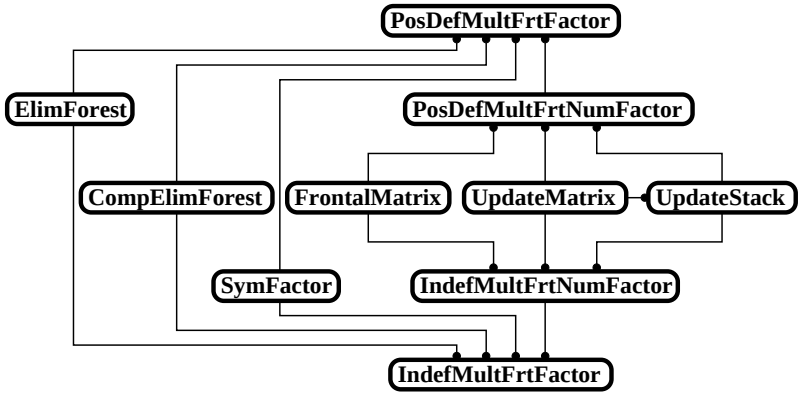



FIGURE 1.20. Composition relationships.

```
for (ElimForest::constSupPostIter it = f->supPostBegin();
     it != f->supPostEnd(); it++)
{
  fm.init(*l, u, *it); // Initialize the frontal matrix.
  fm.clear(); // Zero out all its values.

  fm.assembleOrg(*a, *p); // Assemble original values.

  for (int childCnt = it.getSupChildCnt();
       childCnt > 0; childCnt--)
  {
    u.pop(um); // Pop the update matrix out of the stack.
    fm.assembleUpd(um); // Assemble update values.
  }

  fm.factor(); // Perform partial factorization.

  fm.saveElm(*l); // Save eliminated values.

  um.init(fm, u, *it); // Initialize the update matrix.
  fm.saveUpd(um); // Save update values.

  u.push(um); // Push the update matrix into the stack.
}
```

FIGURE 1.21. The core of the multifrontal numerical factorization.

the update matrices only once. We actually allocate a big chunk of memory for the update stack and the update matrices use that storage. Of course, in the indefinite case we have to increase the size of the stack when needed.

## 1.7 Results

We report results obtained on a Sun Ultra I (167 MHz) workstation with 128 MB of main memory and 512 KB of cache, running Solaris 2.5. We compiled the C++ source files with g++ 2.8.1 (-O) and the Fortran 77 source files with f77 4.2 (-fast).

We use the two sets of problems listed in Tbl. 1.2. We read only the sparsity pattern for the first set and generate numerical values for the nonzero elements (both real and complex) that make the coefficient matrices positive definite. The problems in the second set are indefinite and we read the original numerical values also. To compare ordering algorithms we use the first set of problems. Tbl. 1.3 shows ordering and numerical factorization times for several ordering algorithms available in *Spindle*, for real-valued coefficient matrices.

| problem | order | nonzeros in $A$ (subdiag.) | description |
|---|---|---|---|
| commanche | 7,920 | 11,880 | helicopter mesh |
| barth4 | 6,019 | 17,473 | computational fluid dynamics |
| ford1 | 18,728 | 41,424 | structural analysis |
| barth5 | 15,606 | 45,878 | computational fluid dynamics |
| shuttle_eddy | 10,429 | 46,585 | model of space shuttle |
| ford2 | 100,196 | 222,246 | structural analysis |
| tandem_vtx | 18,454 | 117,448 | helicopter mesh (tetrahedal) |
| pds10 | 16,558 | 66,550 | multicommodity flow |
| copter1 | 17,222 | 96,921 | helicopter mesh |
| ken13 | 28,632 | 66,586 | multicommodity flow |
| tandem_dual | 94,069 | 183,212 | dual of helicopter mesh |
| onera_dual | 85,567 | 166,817 | dual of ONERA-M6 wing mesh |
| copter2 | 55,476 | 352,238 | helicopter mesh |
| shell | 4,815 | 100,631 | model of venous stent |
| rev7a | 9,176 | 143,586 | ,, |
| nok | 13,098 | 806,829 | ,, |
| e20r0000 | 4,241 | 64,185 | Driven cavity (Stokes flow) |
| e30r0000 | 9,661 | 149,416 | ,, |
| e40r0000 | 17,281 | 270,367 | ,, |
| helmholtz0 | 4,224 | 19,840 | Helmholtz problem (acoustics) |
| helmholtz1 | 16,640 | 74,752 | ,, |
| helmholtz2 | 66,048 | 283,648 | ,, |

TABLE 1.2. Test problems.

| problem | | MMD | MMMD | AMD | AMIND | AMMF |
|---|---|---|---|---|---|---|
| commanche | o | 0.28 | 0.33 | 0.55 | 0.55 | 0.58 |
| | f | 0.24 | 0.22 | 0.25 | 0.23 | 0.26 |
| barth4 | o | 0.23 | 0.25 | 0.33 | 0.33 | 0.35 |
| | f | 0.27 | 0.28 | 0.28 | 0.28 | 0.27 |
| ford1 | o | 0.89 | 1.07 | 1.31 | 1.38 | 1.44 |
| | f | 0.89 | 0.84 | 0.91 | 0.95 | 0.95 |
| ken13 | o | 2.26 | 2.72 | 6.01 | 7.44 | 7.32 |
| | f | 1.61 | 1.79 | 1.61 | 1.86 | 1.85 |
| barth5 | o | 0.93 | 1.06 | 1.17 | 1.21 | 1.27 |
| | f | 0.93 | 0.83 | 0.94 | 0.88 | 0.88 |
| shuttle_eddy | o | 0.44 | 0.51 | 0.62 | 0.66 | 0.65 |
| | f | 0.98 | 0.87 | 0.98 | 0.87 | 0.87 |
| ford2 | o | 6.67 | 7.86 | 8.05 | 8.78 | 8.81 |
| | f | 12.19 | 9.01 | 11.41 | 9.15 | 8.57 |
| tandem_vtx | o | 2.25 | 2.45 | 1.82 | 2.11 | 2.17 |
| | f | 31.64 | 20.95 | 30.02 | 19.05 | 16.34 |
| pds10 | o | 89.52 | 103.74 | 3.88 | 4.46 | 4.43 |
| | f | 41.60 | 41.50 | 40.86 | 39.66 | 42.51 |
| copter1 | o | 2.50 | 2.62 | 2.01 | 2.38 | 2.80 |
| | f | 42.15 | 34.98 | 39.50 | 31.81 | 31.18 |
| tandem_dual | o | 11.34 | 12.32 | 9.84 | 11.26 | 11.15 |
| | f | 256.46 | 205.72 | 224.28 | 146.66 | 121.28 |
| onera_dual | o | 10.78 | 11.70 | 8.98 | 10.35 | 10.55 |
| | f | 344.38 | 214.66 | 261.60 | 191.71 | 136.83 |
| copter2 | o | 11.74 | 12.27 | 7.85 | 9.32 | 9.45 |
| | f | 413.02 | 259.38 | 364.55 | 234.47 | 218.37 |

TABLE 1.3. Time (seconds) needed to compute some fill-reducing orderings from the MMD family (o), and to factor the reordered matrices (f), for positive definite real-valued problems.

| problem | nonzeros in $L$ (subdiagonal) | real-valued perf. (Mfl/s) | real-valued error | complex-valued perf. (Mfl/s) | complex-valued error |
|---|---|---|---|---|---|
| commanche | 70,308 | 12.1 | 6e-17 | 37.0 | 1e-16 |
| barth4 | 110,054 | 21.6 | 4e-17 | 58.3 | 8e-17 |
| ford1 | 313,855 | 28.6 | 2e-17 | 72.6 | 3e-17 |
| barth5 | 364,751 | 28.8 | 5e-17 | 75.2 | 1e-16 |
| shuttle_eddy | 390,020 | 37.2 | 4e-17 | 89.6 | 7e-17 |
| ford2 | 2,463,669 | 32.2 | 2e-17 | 75.5 | 2e-17 |
| tandem_vtx | 2,708,953 | 36.2 | 9e-17 | 69.3 | 1e-16 |
| pds10 | 1,634,148 | 27.2 | 2e-17 | 53.0 | 3e-17 |
| copter1 | 2,470,144 | 32.0 | 2e-16 | 69.8 | 2e-16 |
| ken13 | 334,470 | 16.3 | 1e-17 | 42.0 | 6e-18 |
| tandem_dual | 11,796,630 | 34.5 | 1e-16 | 69.9 | 2e-16 |
| onera_dual | 12,028,652 | 34.7 | 1e-16 | 69.5 | 2e-16 |
| copter2 | 14,977,651 | 33.8 | 7e-17 | 67.1 | 1e-16 |

TABLE 1.4. Factorization results obtained with the positive definite code.

We focus next on one particular ordering, MMD, and look at several other properties. Tbl. 1.4 lists, for the first set of problems, the number of off-diagonal nonzero entries in the factors, the numerical factorization performance and the backward error for both real and complex-valued problems. Note that performance results must be interpreted with care for sparse matrix computations, since an ordering that increases arithmetic work tends to increase performance as well. Thus one can achieve a higher performance by not preordering the coefficient matrix by a fill-reducing ordering. Performance reports are useful when the same ordering is used on different architectures, or as in this case, when looking at both real and complex valued problems. Note that while the arithmetic work of complex-valued problems is roughly four times the work in real-valued problems, performance improves only by a factor of two. The relative residual $(\|(b - Ax)\|/(\|A\| \, \|x\|))$ is computed to report backward error.

We switch now to the second set of problems. The first two groups contain real-valued structural analysis and computational fluid dynamics (Stokes) problems. The third group contains Helmholtz problems, which are complex-valued. We order the coefficient matrices with MMD. Tbl. 1.5 reports the number of off-diagonal nonzero entries in the factors, numerical factorization time, performance during numerical factorization, and the relative residual. Performance is still meaningful because we are interested in how it changes for problems within the same category. It is expected to increase with increasing number of nonzeros in the factor, and that is what we see for structures and the Stokes problem. Yet, performance decreases with increasing size for the Helmholtz problems. This is likely caused by increased paging due to the larger storage requirements of these problems

| problem | nonzeros in $L$ (subdiagonal) | time (s) | perf. (Mfl/s) | error |
|---|---|---|---|---|
| shell | 139,643 | 0.33 | 19.7 | 1e-17 |
| rev7a | 580,599 | 2.95 | 26.7 | 7e-17 |
| nok | 2,346,692 | 17.74 | 34.5 | 1e-19 |
| e20r0000 | 369,843 | 2.47 | 22.1 | 2e-18 |
| e30r0000 | 1,133,759 | 10.30 | 24.0 | 1e-18 |
| e40r0000 | 2,451,480 | 27.11 | 27.3 | 1e-18 |
| helmholtz0 | 129,525 | 0.56 | 66.3 | 3e-17 |
| helmholtz1 | 619,292 | 4.64 | 58.5 | 4e-17 |
| helmholtz2 | 3,106,177 | 39.45 | 66.9 | 5e-17 |

TABLE 1.5. Factorization results obtained with the indefinite code.

relative to the available memory. We have observed a performance increase with size for these problems on machines with more memory (such as a Sun Ultra 60 workstation with 512 MB of memory and an IBM RS6000 workstation with 256 MB of memory). These results show that for large problems an out-of-core solver is needed to obtain high performance.

There is a detail that needs to be explained here. The indefinite numerical factorization is based on a sparsity/stability threshold which can take values between 0 and 1. A large threshold enhances stability but also generates more swapping among the columns within each supernode, and more delaying of columns from child to parent supernodes, which leads to reduced sparsity and performance. Sparsity can be preserved and performance increased by lowering the threshold, but that decreases stability. The optimal choice depends on the problem and on what tradeoffs are made. Stability can usually be reinstated from an unstable factorization by few steps of iterative refinement. The results in Tbl. 1.5 are obtained with a threshold value of 0.1, and no iterative refinement.

## 1.8 Comparisons with Existing Work

While we were completing this article, we learned about another object-oriented library called SPOOLES that contains direct solvers. SPOOLES (SParse Object-oriented Linear Equations Solver) is a recent package developed by Cleve Ashcraft and colleagues at the Boeing Company [SPO]. The code is publicly available without any licensing restrictions.

SPOOLES contains solvers for symmetric and unsymmetric systems of equations, and for least squares problems. It supports pivoting for numerical stability; it has three ordering algorithms (minimum degree, nested dissection, and multisection). Implementations on serial, shared parallel (threads), and distributed memory parallel (message-passing using MPI) environments are included. It is a robust, well documented package that is

written in C using object-oriented techniques. We have learned much from its design and from our continuing interactions with Cleve Ashcraft.

SPOOLES distinguishes between data classes, which can have some trivial functionality, and algorithm classes, which handle the more sophisticated functionality. However, it does not enforce this paradigm strictly, as it has ordering classes but only factorization and solve methods.

SPOOLES employs a left-looking factorization algorithm instead of the multifrontal algorithm that we have implemented. However, with the functionality provided in SPOOLES, a multifrontal method can be quickly implemented. It employs a one-dimensional mapping of the matrix onto processors during parallel factorization, and a two-dimensional (2-D) mapping during the parallel triangular solves. It would be harder to support pivoting with a mapping of the matrix. The factorization algorithms are not as scalable as parallel algorithms that employ the 2-D mapping.

SPOOLES covers more ground than we have done, as it has solvers for symmetric and unsymmetric systems of equations, as well as overdetermined systems. It does not have all of the minimum priority orderings in Spindle.

SPOOLES is implemented in C, which is generally more portable than C++. The C programming language is a procedural language and not designed to support object-oriented semantics of inheritance, automatic construction and destruction of objects, polymorphism, template containers, etc. These can be emulated in C, but it falls upon the library developer and the user to adhere to certain programming styles with no help or enforcement from the compiler.

## 1.9 Lessons Learned

### *1.9.1 General Comments*

It is difficult to create general solutions to software design problems. Often a solution that seems good in a particular context becomes unsatisfactory when the context is extended. For example, the initial design of OOLIO did not decouple algorithms from data structures. It was initially implemented in C instead of C++. When OOLIO was ported to C++, the structs were converted to classes and algorithms were made methods of these classes. This introduced several difficulties. First, the association was not natural. Is an ordering algorithm supposed to be associated with a matrix or with a permutation? Second, this solution was neither flexible nor extensible. These problems led us later to the natural and general solution in which we introduced algorithmic classes.

One area where we continue to struggle for the "right" solution is with shared objects. This happens, for example, with an ordering algorithm and a coefficient matrix since the matrix exists outside the algorithm but it

also has a copy inside the algorithm, as the algorithm's input. Since the matrix is built before the ordering algorithm is run, the input may just be a pointer to the matrix, an external object. What happens with the output of the ordering algorithm (the `Permutation` object) is trickier to manage.

Currently, we have a limited approach for the ownership of dynamically allocated data inside structural objects. It is not desirable to allocate separate copies for different copies of the same entity. We have used two approaches until now. In the first, the output object also exists before the algorithm is run (it can be empty) and the algorithm packs the empty class with information. The second approach lets algorithm classes construct the output classes internally and spawn ownership (and cleanup responsibilities) of only the outputs that the user requests.

The preallocation scheme is simple and makes responsibility of instance reclamation obvious. The downside of this is that an algorithm's output has to be allocated and assigned to it before the algorithm is run. It is less intuitive than running the algorithm and querying the results. *Spindle*'s fill-reducing orderings construct the parent pointers for the elimination forest in the normal course of computing the ordering. One user may indeed want a permutation, but others may want the elimination forest. It is inelegant to force users to pre-initialize output classes that they are not even interested in.

The approach of having shared objects and explicitly transferring ownership can be complicated to use and is more prone to programmer error. This is, however, much easier to implement than another method for sharing large data structures: reference counting.

Full reference counting requires an interface class for each class that can be shared. An additional layer is needed to provide a counter that counts the number of interfaces that access the current instance. This gives more flexibility in assigning inputs and outputs (outputs can actually be built before or later). Changes are then seen by all interfaces and memory deallocation occurs correctly when the last interface is destroyed. The problem of unwanted changes by another interface is eliminated by performing "deep-copy" on write.

### *1.9.2 Judicious Application of Iterators*

One disappointing endeavor was to provide an iterator class to traverse the *reachable set* of the `QuotientGraph` class. The idea was to provide an adjacency iterator for an `EliminationGraph` class, a reachable set iterator for the `QuotientGraph` class, and a collection of minimum priority algorithms that were totally unaware of whether it was operating on an elimination graph or a quotient graph. Although we were successful in implementing the `ReachableSetIter` class, its performance was so poor, that its general use was abandoned. This required the minimum priority algorithms to be specific to the `QuotientGraph` class and iterate directly over the enode and

```
void MinimumDegree::prioritize( const QGraph& g, List& l,
                                PriorityQueue& pq )
{
  for( List::iterator it = l.begin(); it != l.end(); ++it ) {
    int i = *it;
    int degree = 0;
    for( QGraph::reach_iterator j = g.reach_begin(i);
         j != g.reach_end(i); ++j ) {
      degree += g.getNodeWeight( *j )
    }
    pq.insert( degree, i );
  }
}
```

FIGURE 1.22. Computing the degree of all the nodes in `List` the elegant (but inefficient) way using reachable set iterators. The innocuous looking `++it` hides a cascade of `if-then-else` tests that must be performed at each call.

supernode adjacency lists. This increased the complexity of the interface and the coupling between data-structure and algorithm, but also significantly improved the performance. Here we explain why this idea looked good on paper, why it did not work well in practice, and why this problem is unavoidable.

Ideally, one would like to provide a class that iterates over the reachable set so that the priority computation can be implemented cleanly. In Fig. 1.22 this class is `typedef`'ed inside the `QuotientGraph` class as `reach_iterator`. We add the additional detail that a weight might be associated with each supernode, so the degree computation sums the weights of the nodes in the reachable set.

Internally, we expected the reachable set iterator to have much in common with the C++ standard `deque::iterator`. A `deque` is a doubly ended queue that is commonly implemented as a vector of pointers to pages of items. The `deque` can easily add or remove items from each end by checking the first or last pages, and adding or deleting pages as necessary. The iterator of a `deque` need only advance to the end of a page, then jump to the next page. In the context of a quotient graph, an enode is much like a page with each having a list of items; namely supernodes.

There are, unfortunately, some critical differences. In a `deque`, the size of each page is known *a priori*, which is not true for the supernodes adjacent to an enode. Furthermore, the same supernode may be reachable through two different enodes. The reachable set need not be traversed in sorted order (as presented in Fig. 1.13), but it cannot allow the same supernode to be counted twice through two different enodes. Furthermore, the reachable set does not include the node itself.

We were able to implement such an iterator, but there is a hidden overhead that causes the code fragment in Fig. 1.22 to be too expensive. A reachable set is the union of several non-disjoint sets; and therefore the iterator must test at each iteration if there are any more items in the current set, if there are any more sets, and use some internal mechanism to prevent double visiting the same node in different sets.

Most of the details for the `ReachableSetIter` are not difficult, but the increment operator is excessively tedious. The problem is that the increment operator must redetermine its state at each call: Is it already at the end of the adjacency list? Are there more nodes or enodes in the current list? Once a next node has been located, has it been marked? The alternative is to evaluate the reachable set by iterating over sets of *adjacent* nodes and enodes of the quotient graph manually. This is shown in Figure 1.23 which is functionally equivalent to the code in Figure 1.22. In this case, the state information is implicit in the loop and not confined to the increment operator, which allows for the entire process to execute more efficiently.

The lesson learned here is to be judicious in the use of fancy techniques. The coding benefits of using a reachable set iterator are far outweighed by the increase in performance in manually running through the adjacency lists. The latter scheme makes the critical assumption that every node has a "self-edge" to itself in the list of adjacent enodes. This convenient assumption also increases coupling between the `QuotientGraph` class and the descendants of the `MinimumPriorityStrategy` class.

### *1.9.3 Conclusions*

There is no inherent conflict between object-oriented design and efficient direct solvers in scientific computing. Some features of C++ such as function inlining, templates, encapsulation, and inheritance suffer no performance penalties and can be used aggressively. Other features such as virtual functions, excessive temporary objects that can result from operator overloading, and hidden copy construction can significantly degrade performance if left unchecked. The target of object-oriented design must be the higher layers of the software. With direct solvers they are the most sophisticated, taking most of the code, but just a small fraction of the execution time. There is not much room for abstraction in low level loops, so the focus there must be on performance.

We have learned that good design requires tradeoffs. There is no perfect solution, unless we are talking about artificial problems typically constructed for pedagogical use. Real-life applications have many constraints, many that conflict with each other, and it is difficult to satisfy all of them. The solution is to prioritize the constraints and to satisfy those with high priorities. An example is decoupling, which introduces an overhead since objects have to communicate through well defined interfaces. By decoupling we can localize potential changes in the code, and this is one of the proper-

```
void MinimumDegree::prioritize( const QGraph& g, List& l,
                                PriorityQueue& pq )
{
  for( List::iterator it = l.begin(); it != l.end(); ++it ) {
    int i = *it;
    int degree = 0;
    int my_stamp = nextStamp();// get new timestamp
    g.visited[ i ] = my_stamp;    // Mark myself visited
    for( QGraph::enode_iterator e = g.enode_begin(i);
         e != g.enode_end(i); ++e ) {
      int enode = *e;  // for all adjacent enodes
      for( QGraph::node_iterator j = g.node_begin(e);
           j != g.enode_end(e); ++j ) {
        int adj = *j;  // for all adjacent nodes
        if ( visited[ adj ] < my_stamp ) {
          // if not already visited, mark it and add to degree
          visited[ adj ] = my_stamp;
          degree += g.getNodeWeight( adj );
        }
      }
    }
    pq.insert( degree, i );
  }
}
```

FIGURE 1.23. Computing degree without using the reachable set iterators. This piece of code is not as elegant as the equivalent fragment in Figure 1.22, but in terms of efficiency , this method is the clear winner. In this case, the `visited` array is part of the `MinimumPriorityStrategy` base class along with the `nextStamp()` member function that always returns a number larger than any previous number. If the next stamp is the maximum integer, then the `visited` array is reinitialized to all zeros.

ties that make object-oriented software appealing. To avoid overheads, we have weakened the encapsulation in performance-critical parts of the code.

Well-designed object-oriented software that is implemented in C++ leads to libraries that are easier to use, more flexible, more extensible, safer, and just as efficient as libraries implemented in C or Fortran. Designing and implementing a "good" object-oriented library is also significantly harder. Implementing sparse direct solvers cleanly and efficiently using object-oriented techniques is an area where challenging research issues remain.

*Acknowledgments:* We thank our colleague, David Hysom, who worked on the integration of our codes into PETSc, provided user feedback, and participated in our spirited debates. We are grateful to Cleve Ashcraft for many engaging technical conversations and Socratic dialogues. Thanks also to the PETSc team: Satish Balay, Lois McInnes, Barry Smith and Bill Gropp, for being so helpful—even with non-PETSc specific issues. Finally, we thank David Keyes for his enthusiasm and support.

This work was supported by U. S. National Science Foundation grants CCR-9412698 and DMS-9807172; by the U. S. Department of Energy by subcontract B347882 from the Lawrence Livermore Laboratory; by a GAANN fellowship from the Department of Education; and by NASA under Contract NAS1-19480 while the third author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE).

## 1.10 References

[ADD96] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.

[AGL99] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20:513–561, 1999.

[AL96] C. Ashcraft and J. W. H. Liu. *SMOOTH: A software package for ordering sparse matrices*, November 1996. www.cs.yorku.ca/ joseph/SMOOTH.html.

[APW+99] C. Ashcraft, D. Pierce, D. K. Wah, and J. Wu. *The Reference Manual for SPOOLES, Release 2.2*, February 1999.

[BGM+97] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern Software Tools for Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, (eds.), Birkhäuser, 1997.

[BL97] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in DIFFPACK. *ACM Transactions on Mathematical Software*, 23:50–80, 1997.

[BPR+97] R. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. Dongarra. Matrix Market: a web resource for test matrix collections. In *Numerical Software: Assessment and Enhancement*, R. Boisvert, (ed.), pp. 125–137. Chapman and Hall, London, 1997.

[DKP98] F. Dobrian, G. Kumfert, and A. Pothen. Object-oriented design of a sparse symmetric solver. In *Computing in Object-oriented Parallel Environments*, Lecture Notes in Computer Science 1505, D. Caromel et al (eds.), pp. 207–214, Springer Verlag, 1998.

[DGL92] I. S. Duff, R. G. Grimes, and J. G. Lewis. *Users' Guide for the Harwell-Boeing Sparse Matrix Collection*, Oct 1992.

[DIF] Diffpack homepage: `www.nobjects.com/Diffpack`.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. on Math. Software*, 9(3):302–325, 1983.

[DER86] I. S. Duff, A. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley Longman, 1995.

[GL80] A. George and J. W. H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Trans. on Math. Software*, 6:337–358, 1980.

[GL81] A. George and J. W. H. Liu. *Computer Solution of Large, Sparse, Positive Definite Systems*. Prentice Hall, 1981.

[GL89] A. George and J. W. H. Liu. The evolution of the minimum degree algorithm. *SIAM Rev.*, 31(1):1–19, March 1989.

[GL97] A. George and J. W. H. Liu. An object-oriented approach to the design of a user interface for a sparse matrix package. Technical Report, Department of Computer Science, York University, Feb. 1997.

[HL93] B. Hendrickson and R. Leland. *The Chaco User's Guide*. Sandia National Laboratories, Albuquerque, NM 87815, Oct 1993.

[KP97] G. Kumfert and A. Pothen. Two improved algorithms for envelope and wavefront reduction. *BIT*, 37:559–590, 1997.

[KP98] G. Kumfert and A. Pothen. An object-oriented collection of minimum degree algorithms: design, implementation, and experiences. In *Computing in Object-oriented Parallel Environments*, Lecture Notes in Computer Science 1505, D. Caromel et al (eds.), pp. 95–106, Springer Verlag, 1998.

[LAK96] J. Lakos. *Large-Scale C++ Software Design*. Professional Computing Series. Addison-Wesley, 1996.

[LAN99] H. P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Springer-Verlag, 1999.

[LIU85] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. on Math. Software*, 11:141–153, June 1985.

[LIU90] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, 1990.

[LIU92] J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Rev.*, 34(1):82–109, 1992.

[NR97] E. G. Y. Ng and P. Raghavan. Performance of greedy ordering heuristics for sparse Cholesky factorization. Technical Report, Computer Science Department, University of Tennessee, Knoxville, 1997.

[PET] PETSc homepage: `www.mcs.anl.gov/petsc/petsc.html`.

[RE98] E. Rothberg and S. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal. Appl.*, 19(3):682–695, July 1998.

[ROT96] E. Rothberg. Ordering sparse matrices using approximate minimum local fill. Preprint, April 1996.

[SPO] SPOOLES homepage: `www.netlib.org/linalg/spooles/spooles.2.2.html`.

[VEL99] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Modern Software Tools in Scientific Computing*, A. M. Bruaset, H. P. Langtangen and E. Quak (eds.), pp. xxx-xxx, Springer-Verlag, 1999.