OBJECT ORIENTED LINEAR ALGEBRA

A thesis submitted to the University of Manchester
for the degree of Master of Philosophy
in the Faculty of Science and Engineering

December 1999

By
Miguel Angel Luján Moreno (Mikel Luján)
Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

The weak point of traditional Linear Algebra libraries is their intellectual distance from Linear Algebra. For one matrix calculation, such as multiplication of matrices, the library provides a large number of subroutines. Each of these subroutines is an optimal implementation for a specific situation (matrix properties and storage formats). Users are forced to analyse their problems in terms of the storage formats and matrix properties supported by the library in order to get good performance (or to use the library correctly).

At present, almost all Object Oriented Linear Algebra Libraries (OOLALs) offer a simpler interface. These OOLALs are equipped with a *rule based reasoning* system for certain matrix calculations. Thus, when a method is invoked the reasoning system decides which of the different implementations (with the same functionality) is appropriate for execution. The decision is based on those situations for which the library provides efficient subroutines. The matrix calculations, for which there is no reasoning system, are offered in different ways depending on the OOLAL.

An exception is the Matrix Template Library (MTL), which combines object oriented and generic programming to reduce the number of specialised implementations for each matrix calculations. It is based on the idea of *iterators*, which support transparent access to data structures without explicitly indications.

This thesis describes an object oriented analysis and design of linear algebra that establishes a context in which various OOLALs are evaluated. The Object Oriented Linear Algebra LibrAry (OoLaLa) is a new OOLAL which arises out of this analysis and design. OoLaLa specifies an interface suitable for both expert and non-expert users. This interface covers basic matrix operations (e.g. matrix addition), and the solution of matrix equations (e.g. system of linear equations, eigenproblem) with iterative and direct algorithms. None of the reviewed OOLALs addresses such a range of numerical linear algebra functionality.

In addition, OoLaLa's design enables libraries to change the storage format of a matrix in response to changes in its matrix properties. This is a novel functionality for linear algebra libraries. OoLaLa also illustrates how matrix calculations can be implemented at storage format (traditional libraries), at iterator level (MTL approach) and at matrix abstraction level (regardless of storage format, but explicitly indicating the position to be accessed) solely using object oriented programming.

Finally, linear algebra expressions are analysed. Some of these expressions are semantically equivalent but result in different programs delivering different execution times. These expressions constitute limitations that current linear algebra libraries cannot solve efficiently. Consequently, a Linear Algebra Problem Solving Environment is proposed in which compiler techniques and OoLaLa are integrated.

Declaration

Copyright

To Agurtzane
and to my parents
Domingo and Juli

Acknowledgements

I would like to thank Professor John Gurd and Dr. Len Freeman for their support and guidance during this year. Dr. Len Freeman soon left its role of adviser to become a supervisor. This joint supervision has probed to be very helpful in this multidisciplinary thesis (mathematics and computer science). I am looking forward to continuing working towards the PhD with both of you.

During this year, I have enjoyed the company of the members of the Center for Novel Computing, specially in the tea breaks and in Rhodes. At different points, every one has provided his expertise and I want to thank you for this. In particular, Nicolas Fournier and Boby Cheng offered useful comments and discussions during the writing-up of this thesis.

This work has been supported by a research scholarship from the Department of Education, Universities and Research of the Basque Government.

Chapter 1

Introduction

1.1 Overview

Scientists and engineers describe physical phenomena in terms of mathematical models. These models are usually continuous and too complex to be solved analytically. In such cases, they are approximated with discrete mathematical models and solutions are obtained by applying numerical methods. A computer simulation of physical phenomena that follow a discrete mathematical model is called a scientific application. From a user point of view, a scientific application is a tool that enables scientists to experiment with physical phenomena and, in that way, increase their understanding. The advantage of scientific applications is that they have a limited cost and no risk. Real experiments consume products in each experiment and thus the cost is accumulative. In addition, real experiments, such as chemical reactions, can have high risk (e.g. explosions, environmental pollution) which do not exist in computer simulations. By contrast, a scientific application has a fixed cost, the software development cost, and enables scientists to experiment an unlimited number of times (assuming that the cost of running a scientific application on a computer is negligible compared with the development cost).

This thesis takes *numerical linear algebra* as an example family of scientific applications. Numerical linear algebra is a field lying between linear algebra and computer science, which generates computer programs to solve linear algebra problems.

This thesis analyses the software development process for sequential linear algebra applications in order to improve this process. The accepted development process is based on using a library. This thesis follows the library approach, but instead of using traditional libraries, it focuses on object oriented libraries. The contributions can be summarised as follows:

1. a survey and classification of object oriented linear algebra libraries is presented;

2. a new design, for the Object Oriented Linear Algebra LibrAry (OoLaLa) is developed, which spans the functionality of both traditional and object oriented libraries; and finally

3. problems, or limitations, are identified which a library approach to the development of linear algebra programs cannot solve.

Metaphorically speaking, this thesis is an intellectual journey that begins with an unsatisfactory development process for linear algebra applications based on a library approach (Section 1.2). From this departure point, the journey builds on the valid library approach and mixes it with object oriented software construction techniques (Section 1.3). A new object oriented design increases the functionality of the existing object oriented linear algebra libraries. The journey arrives at the final station where a library approach can be dispensed with, and problems that any library cannot overcome are identified (Section 1.4). The journey returns to the original departure point, and concludes that a problem solving environment approach is needed to overcome these problems. In this problem solving environment the new object oriented design will be integrated with techniques developed in other areas of computer science.

1.2 Traditional Linear Algebra Libraries

Over the last 40 years the numerical linear algebra community has developed a large number of subroutines. These subroutines have been grouped into different libraries, each library targeting a set of linear algebra calculations. A major benefit of numerical libraries is that they are a means of reusing expert knowledge in the form of code. Ideally, a numerical linear algebra program would be the declaration of data structures used by the library and a succession of calls to library subroutines. However, sometimes users' requirements (e.g. multiplication of two banded matrices) go beyond the scope of traditional libraries; and the users then have to write code themselves.

A second benefit is portability. Libraries pass a standardisation process in which the functionality to be included, realised in the form of subroutine declarations and data structures (storage formats) in a specific programming language, is determined. The implementations are not standardised, although reference ones are distributed. This enables vendors to supply an implementation optimised to a specific architecture. In this way, not only is the library portable, since the programming language itself is portable, but also the performance can be ported from architecture to architecture.

The term *traditional libraries* is applied to the libraries developed by this research community using a top-down methodology and implemented in imperative languages. The predominant language in this field is Fortran 77 and examples of these libraries are LINPACK [DBMS79], EISPACK ([SBD$^+$76],[GBDM77]) and more recently BLAS ([BLA99][1] [LHKK79], [DCHH88b], [DCHD90]) and LAPACK ([ABD$^+$95]).

Given these traditional libraries, the development process of numerical linear algebra applications can be summarised as follows:

1. describe in terms of linear algebra calculations the problem to be solved;
2. select the numerical library (or libraries) which solves the problem;
3. translate the linear algebra problem so that it is defined in terms of the specific situations (storage formats and subroutines) supported by the library (or libraries).

The third step of this development process is non-trivial. A common characteristic of traditional libraries is that they provide many implementations for one mathematical operation. Knowing information about the matrices (matrix properties) involved in a matrix calculation has enabled the numerical linear algebra community to develop optimised implementations. This means that the number of combinations of different matrix properties supported by a library is the number of different implementations of each matrix calculation. Moreover, some traditional libraries provide the facility of storing matrices in different storage formats. Hence, the number of implementations of each matrix calculation is the number of combinations of the different matrix properties together with the possible storage formats supported by a library.

[1]Draft document under community revision that will substitute the other references of BLAS.

Certain matrix calculations can be implemented using different algorithms (not developed by exploiting matrix properties) and the numerical linear algebra community is not always able to identify the situations for which each algorithm is most appropriate. This is the case for iterative and direct algorithms applied to sparse systems of linear equations (see [BBC$^+$94], [BDD$^+$95], [DER86]).

Traditional libraries do not encapsulate or hide information; subroutine names and parameters reveal implementation details. Each subroutine name describes the basic type of the matrices, the properties of matrices, the storage format and the operation. The subroutine parameters are arrays that store matrices or vectors, integer values that declare the dimensions of matrices or vectors, and string values that declare more precisely the properties of matrices.

To sum up, the program development process requires:

- analysis of the properties of matrices,
- choice of the storage formats, and
- selection of the subroutines that will deliver the best performance.

To improve the process of developing linear algebra programs, the intellectual distance from a description of the problem in terms of linear algebra to a description in terms of traditional libraries must be reduced. Following the trend in other areas of computer science, object oriented linear algebra libraries (OOLALs) are a possible way of improving the software development process for linear algebra programs. OOLALs provide abstractions closer to linear algebra and, therefore, a reduced intellectual jump.

1.3 Object Oriented Linear Algebra Libraries

In contrast with traditional libraries, there is no consensus in the community about OOLALs, possibly due to their immature state. The first paper about object oriented linear algebra [McD89] only appeared in 1989 and the first international conference dedicated to object oriented numerical applications [Rog93] was not until 1993. Several OOLALs with different designs have been developed encapsulating matrices and vectors in classes. They differ in the sets of matrix properties for which they implement optimised versions and the storage formats for each matrix property. When there is only one storage format provided, "by pure luck" users are relieved of managing the store format, but as result there is a loss of flexibility that might result in an excessive memory requirement. When there are many storage formats, users have to explicitly select the storage format.

The visible benefit for the user of OOLALs is a simpler interface than the interface of traditional libraries. The OOLALs provide for one matrix calculation one visible method. The different implementations are hidden behind the visible method. Each of these visible methods incorporates a set of rules that are able to decide the appropriated implementation. Obviously, in the cases where the numerical linear algebra community has not been able to identify which implementation is appropriate, the OOLALs have to give access to the different implementations.

The hidden implementations of matrix calculations access the representation of storage formats, as traditional libraries. This level of abstraction is referred to in this thesis as the *storage format abstraction level*.

A significantly different level of abstraction, called *iterator abstraction level* in this thesis, is used to implement the mathematical operations in the Matrix Template Library (MTL) ([SL98a], [SL98b], [SL98c], [SL99] [SLL99]). MTL combines object oriented and generic programming to reduce the number of implementations. The key for this change comes from the concept of an *iterator*. An iterator is a generic abstraction layer that provides a set of methods to traverse data structures. Each data structure implements the traversal methods in a different way, nevertheless these methods provide the same functionality. When applying iterators to linear algebra, the data structures are matrix properties with storage formats. The classes of MTL implement the iterator methods taking advantage of a given matrix property. The implementation of a matrix calculation changes from being written in terms of loop bounds to being an implementation written in terms of iterators.

Alternatively, a storage format can be considered as a mapping of element positions to memory positions. Given that every class representing a matrix implements (differently) the same methods to access (read and write) the matrix, an implementation of a matrix calculation can use these access methods and be independent of the storage formats. This level of abstraction is referred to in this thesis as the *matrix abstraction level*.

This thesis proposes a new design basis for the Object Oriented Linear Algebra LibrAry (OoLaLa). The novel characteristics of the design are the management of storage formats and the propagation of matrix properties through matrix calculations.

The library is only active when one of its subroutines (or methods) is called (or invoked). OoLaLa checks the consistency between the storage formats and the matrix properties, which are the parameters of the method invoked. If necessary, OoLaLa will change the storage formats and properties to re-establish the consistency. It might not be obvious, but when OoLaLa checks the consistency of parameters, these include both input and output parameters. Therefore OoLaLa is able to propagate matrix properties to the output parameters from the input parameters. The idea of propagation of properties in not new ([Bik96], [Mar97]), but it is a novel functionality for a linear algebra library.

1.4 Limitations of a Library Approach

At this point, it is convenient to reconsider the intellectual distance between linear algebra and OoLaLa. The distance has been reduced, but the following tasks still remain:

1. analysis of the mathematical properties of the matrices that are the inputs of a linear algebra problem,

2. parsing of linear algebra expressions to the language defined by the visible methods of OoLaLa, and

3. selection of the appropriate method.

Bik and Wijshoff ([Bik96], [BW99]) have developed efficient algorithms to automatically analyse certain matrix properties. This analysis, when included in OoLaLa, could simplify the first task.

The limitations of the library approach are a consequence of its passive role. A library is only active when a subroutine (or method) is called (or invoked). At that moment, a library is not able to look ahead to subsequent computations, and therefore the library can only offer a correct solution at that point of the program.

The second remaining task can be seen as a compilation problem. The source language is defined by expressions accepted in linear algebra and the target language is the one defined by the visible methods of OoLaLa. The parser techniques need to have access to the whole program in order to generate efficient code, but access to the whole program is incompatible with a library approach.

The third task remains an open problem. Rice and Boisvert [Ric96], among other ideas, propose expert systems or knowledge-based systems as a possible solution for this kind of problem [RB96]. They also remark that "the current state-of-the-art of knowledge-based frameworks is low-level and far from adequate for building Problem Solving Environments". A *problem solving environment* is a software system that integrates any computer science discipline in order to enable users to develop programs using the notation or language of their specific problem domain [GHR94]. The different tasks described for developing a linear algebra program constitute the description of a linear algebra problem solving environment.

1.5 Thesis Outline

The remainder of the thesis is organised as follows:

Chapter 2 introduces concepts of linear algebra and numerical linear algebra, and describes the BLAS and LAPACK designs. It is shown that the top-down design results in a complex interface. Matlab and a Sparse Compiler are introduced as alternative approaches.

Chapter 3 reviews object oriented software construction, and describes an object oriented analysis and design of linear algebra. This design is the basis of OoLaLa. Various object oriented models are proposed and used to classify several OOLALs. The design is balanced between the requirements of expert and non-experts users, and enables OoLaLa to manage the storage formats and to propagate matrix properties through matrix calculations; a novel functionality for a library. Iterator and matrix implementation abstraction levels are described as a way of reducing the number of implementations of matrix calculations.

Chapter 4 provides a high level description of the implementation issues of OoLaLa. The design of OoLaLa is adapted to the restrictions of the programming language Java. This chapter compares matrix calculations implemented at storage format, at iterator level and at matrix abstraction level.

Chapter 5 identifies limitations of a library approach in the context of linear algebra. Some of these limitations are due to in the difficulty for users to parse a linear algebra expression to an optimum set of calls to library subroutines.

Chapter 6 reviews the contributions of this thesis to the software development process of sequential linear algebra programs, and proposes future research directions.

Chapter 2

Numerical Linear Algebra

Since the mid 1950s, the numerical linear algebra community has been investigating the problem of how to write programs for matrix calculations so that the solutions are accurate and the execution times minimised. In this still open research area, numerical analysis and linear algebra are combined. The importance of numerical linear algebra resides in its applicability to important problems such as computational fluid dynamics, circuit simulations, data fitting, graph theory, etc. [AR94].

During the ensuing 40 years, important knowledge has been created in the form of algorithms and these have been made reusable as software libraries. To understand what functionality is provided, and why, as well as how the libraries are organised is the main objective of this chapter. The other important aspect is to analyse the influence on the user of the organisation and functionality of these libraries. Since the next chapter includes an object oriented analysis and design of linear algebra, this chapter can also be interpreted as a "requirements document" that summarises the domain.

The process of understanding begins with a review of the basic concepts of matrices and matrix calculations (Section 2.1). Then matrices are classified according to two criteria and the way a given matrix can be represented in different storage formats is examined (Section 2.2). The defined categories, or matrix properties, allow the creation of specialised algorithms which take advantage of certain specific matrix properties (Section 2.3). The algorithms and storage formats are combined to implement matrix calculations. Storage format abstraction level is the term used in this thesis to describe how libraries are traditionally implemented. This aspect is criticised in the next chapter by introducing another two abstraction levels. Given this knowledge, the final step is to examine how BLAS and LAPACK are organised; two examples of libraries developed by community consensus (Section 2.4). These libraries are compared with two software environments; Matlab and the Sparse Compiler. Matlab and the Sparse Compiler represent alternatives to the libraries approach of linear algebra program construction, and permit examination of the difficulties, or steps to follow, in developing linear algebra programs.

2.1 Basic Background

Numerical linear algebra is primarily concerned with matrix calculations. These calculations can be subdivided into two groups. The first group consists of basic matrix operations (e.g. transpose, addition, ...), and the second group involves more complex matrix calculations. Systems of linear equations, eigenvalue and eigenvector problems, and least squares problems are the matrix calculations of this second group. It is out of the scope of this thesis to introduce and describe all the work and state-of-the-art of this research area. Nevertheless, it is the aim of this section to familiarise the reader with the necessary notation and definitions.

2.1.1 Matrix

A matrix is defined as a rectangular array of numbers.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix}$$

The size of a matrix is described in terms of the number of rows m and the number of columns n. When $m = n$, the matrix is called a *square matrix of order* n. When $m = 1$ or $n = 1$, the matrix is called a *row vector* or a *column vector*, respectively. The general case is called a *rectangular matrix of dimension* $m \times n$ (an $m \times n$ matrix). The numbers a_{ij} that constitute the matrix are called its *elements*.

Note that this is a mathematical definition and, therefore, "array" must not be taken in its computer science sense. For computer scientists, a suggested alternative is to substitute *rectangular array* with *two-dimensional container*.

The notation (followed throughout the thesis) is

- matrices are represented by upper case letters (A, B, C, …, Z),
- column vectors are represented by lower case letters (a, b, …, z),
- scalars are represented by lower case Greek letters (α, β, …, ω).

The same letter that is used to represent the matrix, but in lower case and with two suffices represents the elements of a matrix. For example, a_{ij} represents the element which is situated in the i^{th} row and the j^{th} column of matrix A. The elements of a vector are represented with the same letter that is used to represent the vector with one suffix (e.g., x_i represents the i^{th} element of the x vector).

2.1.2 Matrix Calculations

Basic Matrix Operations

The basic matrix operations can be divided into two groups. There are operations that need only one matrix – (monadic) unary, while the others need two matrices – (dyadic) binary. This division is important when implementing the operations. Some definitions of basic matrix operations are presented in Table 2.1.

System of Linear Equations

A system of linear equations is a finite set of linear equations in the variables x_1, x_2, …, x_n and can be expressed as:

$$\begin{array}{cccccccc}
a_{11}x_1 + a_{12}x_2 + & \ldots & +a_{1j}x_j + & \ldots & +a_{1n}x_n & = & b_1 \\
a_{21}x_1 + a_{22}x_2 + & \ldots & +a_{2j}x_j + & \ldots & +a_{2n}x_n & = & b_2 \\
\vdots & \ddots & \vdots & \ddots & \vdots & = & \vdots \\
a_{i1}x_1 + a_{i2}x_2 + & \ldots & +a_{ij}x_j + & \ldots & +a_{in}x_n & = & b_i \\
\vdots & \ddots & \vdots & \ddots & \vdots & = & \vdots \\
a_{m1}x_1 + a_{m2}x_2 + & \ldots & +a_{mj}x_j + & \ldots & +a_{mn}x_n & = & b_m
\end{array},$$

Name	Notation	Definition
Vector Norms	$\|\|x\|\|_p$	$\alpha \leftarrow (\sum_i \|x_i\|)^{1/p}$
	$\|\|x\|\|_\infty$	$\alpha \leftarrow \max_i \|x_i\|$
Matrix Norms	$\|\|A\|\|_1$	$\alpha \leftarrow \max_j \sum_i \|a_{ij}\|$
	$\|\|A\|\|_\infty$	$\alpha \leftarrow \max_i \sum_j \|a_{ij}\|$
	$\|\|A\|\|_F$	$\alpha \leftarrow (\sum_{i,j} \|a_{ij}\|^2)^{1/2}$
Vector Transpose	x^T	
Matrix Transpose	A^T	
Matrix Inverse	A^{-1}	
Dot Product	$\alpha \leftarrow x^T y$	$\alpha \leftarrow \sum_i x_i y_i$
Vector Scale	$y \leftarrow \alpha x$	$y_i \leftarrow \alpha x_i$
Vector Addition	$z \leftarrow x + y$	$z_i \leftarrow x_i + y_i$
Matrix Vector Multiplication	$y \leftarrow Ax$	$y_i \leftarrow \sum_j a_{ij} x_j$
Matrix Scale	$C \leftarrow \alpha A$	$c_{ij} \leftarrow \alpha a_{ij}$
Matrix Addition	$C \leftarrow A + B$	$c_{ij} \leftarrow a_{ij} + b_{ij}$
Matrix Matrix Multiplication	$C \leftarrow AB$	$c_{ij} \leftarrow \sum_k a_{ik} b_{kj}$

Table 2.1: Definition of some basic matrix operations.

where a_{11}, a_{12}, ..., a_{mn}, b_1, b_2, ...,b_m are given constant numbers. The unknowns x_1, x_2, ..., x_n, occur linearly and do not appear as arguments for trigonometric, logarithmic or exponential functions.

The system of linear equations can be written more concisely in terms of the matrix A and the vectors x and b, as follows:

$$\boxed{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \\ \vdots \\ b_m \end{pmatrix}} \boxed{Ax = b}$$

Eigenvalues and Eigenvectors

Given an $n \times n$ matrix A, a vector x is called an *eigenvector* of A if Ax is a multiple of x and x has at least one nonzero element, i.e.

$$Ax = \lambda x$$

for some scalar λ. The scalar λ is an *eigenvalue* of A, and x is said to be the eigenvector of A corresponding to λ.

Least Square Problem

Given a linear system $Ax = b$ of m equations in n variables $n \leq m$, find a vector x that minimises

$$||Ax - b||_2.$$

2.2 Matrix Properties and their Storage Formats

2.2.1 Matrix Properties

Different criteria can be used to classify matrices. These criteria have been proposed because of the execution time benefit that results and because of their occurrence in real and important applications. Knowing the classification of a matrix, the implementation of a matrix calculation might take advantage and thereby reduce the execution time of the calculation. A second benefit might be a reduction in memory requirements for the computation. A third benefit might be that the accuracy of the results can be increased.

Two different criteria, and thereby two different classifications, are presented: nonzero elements structure and mathematical relations. The zero elements of matrices act in a particular way when added or multiplied ($a_{ij} + 0 = a_{ij}$ and $a_{ij} \times 0 = 0$). These properties enable implementations to avoid computations for which the result is already known.

The mathematical relations are relations independent of the zero elements and are expressed as operations of the matrix elements. For example, a matrix is symmetric if and only if $A = A^T$.

In general, the categories defined by these two criteria are not mutually exclusive, so that a matrix can have more than one category. For example, a matrix can be symmetric, positive definite and banded. The remainder of this section is dedicated to the definition of the categories.

From here on, the term *matrix properties* is used to refer to any category of mathematical relation or nonzero elements structure or combinations of these.

Nonzero Elements Structure Criteria

The nonzero elements structure criteria classifies matrices into *dense*, *banded*, *block* and *sparse*. Dense matrices are those matrices which have a majority of nonzero elements. At the other end of the spectrum, sparse matrices are those matrices which have a minority of nonzero elements (see Table 2.2 for dense and sparse matrix examples). A special sparse matrix is the zero matrix, $O_{m\times n}$, which has only zero elements. In the middle of the spectrum, banded and block matrices are matrices in which the nonzero elements have some structure. Both, banded and block matrices, have subcategories. Figure 2.1 presents an hierarchical view of different matrix properties derived from the nonzero elements structures.

Dense	Sparse
$\begin{pmatrix} \square & \square & \square & \square & \square & \square \\ \square & \square & & \square & \square & \square \\ \square & \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square & \square \\ \square & \square & \square & \square & \square & \square \end{pmatrix}$ 6×6	$\begin{pmatrix} & & & \square & & \square \\ \square & \square & & & & \\ \square & & \square & \square & & \\ \square & & \square & \square & & \\ & & & & & \square \\ \square & & & \square & & \end{pmatrix}$ 6×6
$\begin{pmatrix} \square & \square & \square & \square & \square & \square \\ \square & \square & & \square & \square & \square \\ \square & \square & \square & \square & & \square \end{pmatrix}$ 3×6	$\begin{pmatrix} \square & & & \square & & \\ & & \square & \square & & \\ \square & & & & & \end{pmatrix}$ 3×6

Table 2.2: Examples of dense and sparse matrices – $\square$'s represent nonzero elements and blanks represent 0.

A banded matrix is a matrix which has the nonzero elements grouped around the main diagonal. Formally, a $m \times n$ matrix A is banded if a lower bandwidth $b_l < m$ and upper bandwidth $b_u < n$ can be defined so that $a_{ij} \neq 0$ implies that $-b_u \leq i - j \leq b_l$. Different combinations of values for b_u and b_l yield different subcategories of banded matrices. For example, when $b_u = b_l = 0$ the matrix is *diagonal*. A special case of a diagonal matrix is the *identity matrix*, I_n, in which all the nonzero elements are 1. Table 2.3 presents graphical examples of banded matrices and some associated subcategories.

A matrix can be partitioned into sub-matrices A_{ij}. Since it is a partition, every element of A is in exactly one sub-matrix. Two sub-matrices which are in

Matrix 8×8	Matrix 8×8
$\begin{pmatrix} \square & \square & \square & \square & & & & \\ \square & \square & \square & \square & \square & & & \\ \square & \square & \square & \square & \square & \square & & \\ & \square & \square & \square & \square & \square & \square & \\ & & \square & \square & \square & \square & \square & \square \\ & & & \square & \square & \square & \square & \square \\ & & & & \square & \square & \square & \square \\ & & & & & \square & \square & \square \end{pmatrix}$	$\begin{pmatrix} \square & & & & & & & \\ & \square & & & & & & \\ & & \square & & & & & \\ & & & \square & & & & \\ & & & & \square & & & \\ & & & & & \square & & \\ & & & & & & \square & \\ & & & & & & & \square \end{pmatrix}$
banded $b_u = 3$, $b_l = 2$	diagonal $b_u = 0$, $b_l = 0$
$\begin{pmatrix} \square & \square & & & & & & \\ \square & \square & \square & & & & & \\ & \square & \square & \square & & & & \\ & & \square & \square & \square & & & \\ & & & \square & \square & \square & & \\ & & & & \square & \square & \square & \\ & & & & & \square & \square & \square \\ & & & & & & \square & \square \end{pmatrix}$	$\begin{pmatrix} \square & \square & & & & & & \\ & \square & \square & & & & & \\ & & \square & \square & & & & \\ & & & \square & \square & & & \\ & & & & \square & \square & & \\ & & & & & \square & \square & \\ & & & & & & \square & \square \\ & & & & & & & \square \end{pmatrix}$
tridiagonal $b_u = 1$, $b_l = 1$	upper bidiagonal $b_u = 1$, $b_l = 0$
$\begin{pmatrix} \square & \square & \square & \square & \square & \square & \square & \square \\ & \square & \square & \square & \square & \square & \square & \square \\ & & \square & \square & \square & \square & \square & \square \\ & & & \square & \square & \square & \square & \square \\ & & & & \square & \square & \square & \square \\ & & & & & \square & \square & \square \\ & & & & & & \square & \square \\ & & & & & & & \square \end{pmatrix}$	$\begin{pmatrix} \square & & & & & \square & & \\ & \square & & & & & \square & \\ \square & & \square & & & & & \square \\ \square & \square & & \square & & & & \\ & \square & \square & & \square & & & \\ & & \square & \square & & \square & & \\ & & & \square & \square & & \square & \\ & & & & \square & \square & & \square \end{pmatrix}$
upper triangular $b_u = 7$, $b_l = 0$	multi-diagonal $b_u = 5$, $b_l = 3$

Table 2.3: Examples of banded matrices – $\square$'s represent nonzero elements and blanks represent 0.

the same row (A_{ij} and $A_{i(j+1)}$) have the same number of rows. Two sub-matrices which are in the same column (A_{ij} and $A_{(i+1)j}$) have the same number of columns. Each sub-matrix can be classified as a zero matrix or a sparse matrix or a dense matrix or a banded matrix (and its subcategories).

$$A = \begin{pmatrix} A_{11} & \dots & A_{1q} \\ \vdots & \ddots & \vdots \\ A_{p1} & \dots & A_{pq} \end{pmatrix} \qquad \left(\begin{array}{|cc|ccc|c|cc|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} & a_{19} \\ \hline a_{22} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} & a_{28} & a_{29} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} & a_{39} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} & a_{48} & a_{49} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} & a_{58} & a_{59} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} & a_{69} \\ \hline \end{array}\right)$$

Having classified the sub-matrices for a given partition, a *block banded* matrix is defined as a *partitioned*, or *block*, matrix that has the nonzero sub-matrices grouped around the diagonal block (i.e. set of sub-matrices A_{ii}). Formally, a matrix A of dimension $m \times n$ and its partition in sub-matrices A_{11}, A_{12}, $\dots$, A_{pq} are block banded if a lower bandwidth $B_l < p$ and upper bandwidth $B_u < q$ can be defined so that $A_{ij} \neq 0$ implies that $-B_u \leq i - j \leq B_l$. Different combinations of values for B_u and B_l yield different subcategories of block banded matrices. For example, when $B_u = B_l = 0$ the matrix is called *block diagonal.* Table 2.4 presents examples of block banded matrices and associated subcategories.

Comparing the subcategories of banded matrices with block banded matrices, a new subcategory is found, *bordered block banded* matrices (see Figure 2.1 and Table 2.4). Given a partition $A_{11}, A_{12}, \dots, A_{pp}$ of a matrix A, the set of sub-matrices A_{ip} are called the upper border sub-matrix and the set A_{pi} of sub-matrices are called the lower border sub-matrix. A bordered block banded matrix is a matrix whose off-border sub-matrices (i.e. A_{ij} with $i \neq p$ and $j \neq p$) form a block banded matrix, and the upper and the lower border sub-matrices are nonzero matrices.

Efficient algorithms for automatic detection of nonzero elements structures have been proposed by Bik and Wijshoff [BW99]. Other algorithms for reordering matrices (i.e. interchange columns or rows of a matrix) in order to create matrices that fall into some category are described in [DER86].

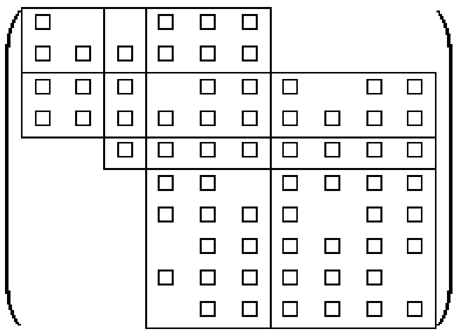
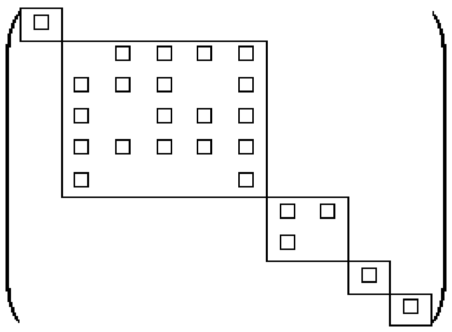
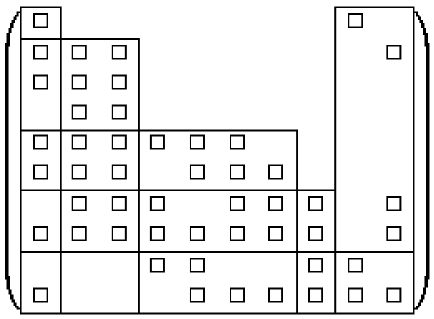
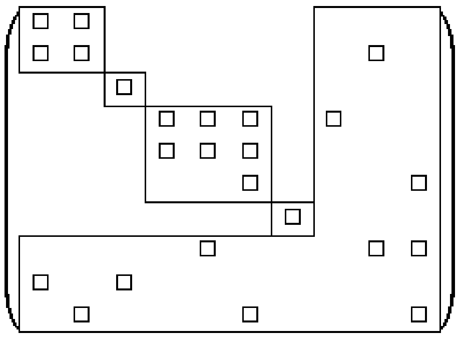
Matrix 10×10	Matrix 10×10
 block banded $B_u = 2$, $B_l = 1$	 block diagonal $B_u = 0$, $B_l = 0$
 single bordered block lower triangular $B_u = 0$ $B_l = 4$	 doubly bordered block diagonal $B_u = 0$ $B_l = 0$

Table 2.4: Examples of block matrices – $\square$'s represent nonzero elements and blanks represent 0.

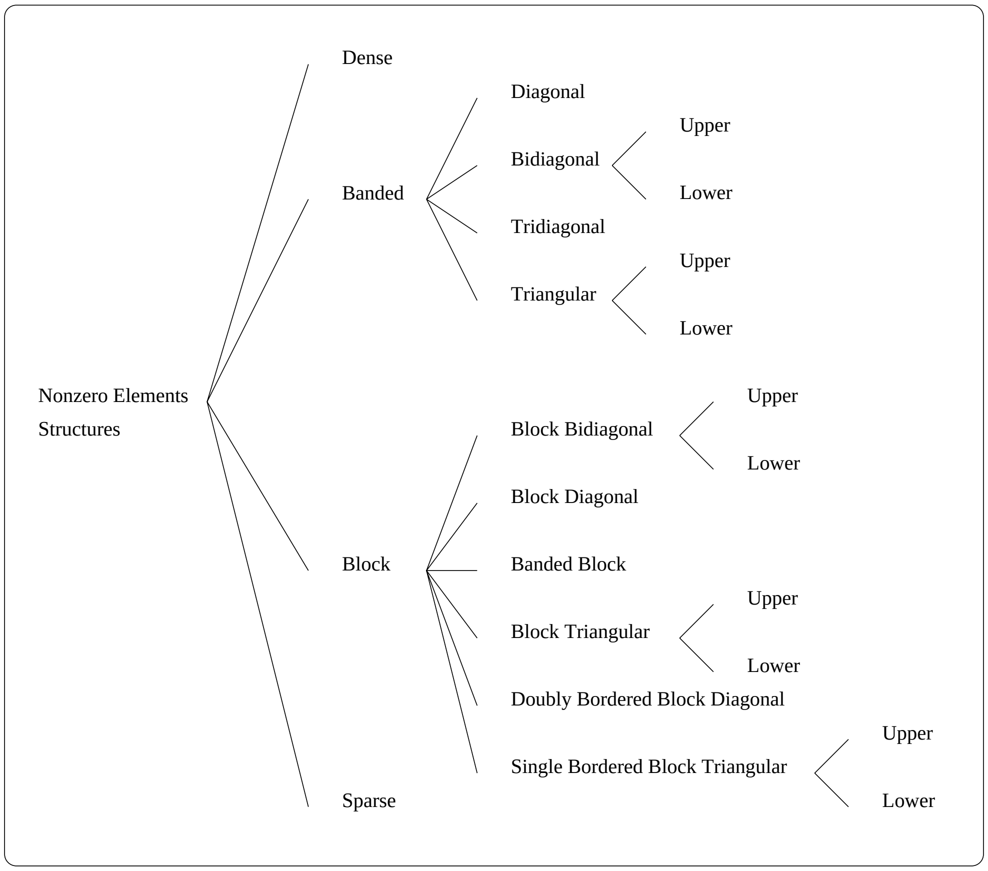


Figure 2.1: Hierarchical view of nonzero elements structures.

Mathematical Relation Criteria

The mathematical criteria, in contrast with nonzero elements structure, are not structural criteria. Loosely speaking, this means that the mathematical classification cannot be found simply by looking at the elements of the matrix. In order to verify if a matrix falls into a certain category, matrix calculations may be required. First, restrict consideration to square matrices; the following categories are used:

- *symmetric* – the matrix is equal to its transpose $A = A^T$,
- *orthogonal* – the inverse of the matrix is equal to its transpose $A^{-1} = A^T$ and therefore $AA^T = I$,
- *positive definite* – for all nonzero vectors x, $x^T Ax$ is positive, and
- *indefinite* - for some nonzero vectors x, $x^T Ax$ is positive, while for other nonzero vectors x it is negative or zero.

2.2.2 Storage Formats

Thus far, the matrices have not been represented by data structures; only mathematical notation has been used. The remainder of the section is dedicated to describing the most common data structures. The importance of this section is not simply to understand different storage formats (i.e. data structures to store matrices), but also to appreciate that a certain matrix with certain properties can be represented in a number of different storage formats.

At present, programming languages provide static and dynamic data structures. Static data structures have a predefined (compilation time) size and cannot be modified at run-time (e.g. arrays). On the other hand, a dynamic data structure can increase or decrease its size at run-time (e.g. lists, trees). Since Fortran 77 has been the dominant language for mathematicians and does not support dynamic data structures, the most commonly used storage formats are array-based. Dense, band and packed formats are presented in this section. Other storage formats for matrices can be found in [BBC$^+$94] Section 4.3 and [DER86] Chapter 2.

Note that different memory layouts to store an array have been defined and are used. For example, a two-dimensional array in C is stored by rows, whereas in Fortran it is stored by columns (see Figure 2.2). The storage formats presented in this section are organised by columns.

`A(1,1)`	`A(2,1)`	`A(3,1)`	`A(1,2)`	...	`A(3,2)`	`A(1,3)`	`A(2,3)`	`A(3,3)`

Memory for row-wise array `A(1..3,1..3)`

`A(1,1)`	`A(1,2)`	`A(1,3)`	`A(2,1)`	...	`A(2,3)`	`A(3,1)`	`A(3,2)`	`A(3,3)`

Memory for column-wise array `A(1..3,1..3)`

Figure 2.2: Row versus column-wise memory layout for arrays.

Dense Format

The most intuitive data structure to represent a matrix is a two-dimensional array. This is called *dense format*, or conventional format. The element a_{ij} of the matrix A is stored in `A(i,j)`. Figure 2.3 presents how different matrix properties can be stored in dense format. In fact, every matrix can be stored using this format.

Band Format

The *band format* uses a two-dimensional array to store the elements of a $n \times n$ banded matrix A. Given b_u and b_l as the upper and lower bandwidth of the matrix, the array `BAND` has $b_u + b_l + 1$ rows and n columns. The element a_{ij} is stored in `BAND`$(b_u + 1 + i - j\,,j)$ if $-b_u \leq i - j \leq b_l$. Figure 2.4 presents examples of banded matrices represented in band format. Note that the first matrix is upper triangular and its array has the same size as its array when stored in dense format (see Figure 2.3). The drawback is that the cost for accessing an element is bigger (i.e. more operations need to be done in order to calculate the memory address). Band format reduces memory requirements when b_u and b_l are less than the matrix dimensions. Dense and triangular matrices should not use this format.

Packed Format

The *packed format* uses a one-dimensional array to store symmetric and triangular matrices. Given an $n \times n$ upper triangular matrix A, the array `PACK` is of size

Matrix | Data Structure

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ & a_{22} & a_{23} & a_{24} & a_{25} \\ & & a_{33} & a_{34} & a_{35} \\ & & & a_{44} & a_{45} \\ & & & & a_{55} \end{pmatrix}$$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
	a_{22}	a_{23}	a_{24}	a_{25}
		a_{33}	a_{34}	a_{35}
			a_{44}	a_{45}
				a_{55}

$$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{54} & a_{55} \end{pmatrix}$$

a_{11}	a_{12}			
a_{21}	a_{22}	a_{23}		
	a_{32}	a_{33}	a_{34}	
		a_{43}	a_{44}	a_{45}
			a_{54}	a_{55}

Figure 2.3: Examples of matrices stored in dense format.

Matrix | Data Structure

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ & a_{22} & a_{23} & a_{24} & a_{25} \\ & & a_{33} & a_{34} & a_{35} \\ & & & a_{44} & a_{45} \\ & & & & a_{55} \end{pmatrix}$$

$b_u = 4$, $b_l = 0$

				a_{15}
			a_{14}	a_{25}
		a_{13}	a_{24}	a_{35}
	a_{12}	a_{23}	a_{34}	a_{45}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}

`BAND(1..4+0+1,1..n)`

$$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{54} & a_{55} \end{pmatrix}$$

$b_u = 1$, $b_l = 1$

	a_{12}	a_{23}	a_{34}	a_{45}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}
a_{21}	a_{32}	a_{43}	a_{54}	

`BAND(1..1+1+1,1..n)`

Figure 2.4: Examples of matrices stored in band format.

$\frac{1}{2}(n^2 + n)$ and element a_{ij} is stored in `PACK(`$i + \frac{1}{2}j(j-1)$`)` when $i \leq j$; *upper packed format*. In the case where the matrix A is lower triangular, the array size is the same but element a_{ij} is stored in `PACK(`$i + \frac{1}{2}(2n-j)(j-1)$`)` when $j \leq i$; *lower packed format*. In both cases the zero elements are not stored. A symmetric matrix has the possibility to choose if the upper triangular or the lower triangular elements are stored. Figure 2.5 presents examples of matrices in this format.

Matrix	Data Structure
$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ & a_{22} & a_{23} \\ & & a_{33} \end{pmatrix}$	$\begin{array}{\|c\|c\|c\|c\|c\|c\|}\hline a_{11} & a_{12} & a_{22} & a_{13} & a_{23} & a_{33} \\ \hline\end{array}$
$\begin{pmatrix} a_{11} & & \\ a_{21} & a_{22} & \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$	$\begin{array}{\|c\|c\|c\|c\|c\|c\|}\hline a_{11} & a_{21} & a_{31} & a_{22} & a_{32} & a_{33} \\ \hline\end{array}$

Figure 2.5: Examples of matrices stored in packed format.

The final remark concerning matrices and storage formats comes in the form of an example. Given a matrix A which is symmetric banded, the matrix can be stored in 4 different ways. First, every matrix A can be stored in dense format. Second, a banded matrix A can be stored in band format. Third and fourth, as a symmetric matrix, A can be stored in packed format, either storing the upper triangular elements or the lower triangular elements.

2.3 Exploiting Matrix Properties

Two matrix calculations are used to illustrate their implementation in traditional libraries. The first calculation, matrix-matrix multiplication, is a basic binary matrix operation. However, this operation is enough to show that for one matrix operation many algorithms can be derived. Each algorithm is specialised for certain matrix properties, taking advantage of knowledge implied by the properties.

The second example is the solution of a system of linear equations. Two families of methods can be applied to solve systems of linear equations: *direct methods* and *iterative methods*. A direct method is an algorithm that calculates

```
for $i = 1$ to $m$
 for $j = 1$ to $n$
  for $k = 1$ to $p$
   $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
  end for
 end for
end for
```

Figure 2.6: Algorithm for matrix-matrix multiplication $C \leftarrow AB$ with A and B dense matrices.

the solution in a known finite number of instructions. On the other hand, an iterative method is an algorithm that is executed repeatedly; each execution of the algorithm produces an approximate solution of the problem, and execution is stopped when the approximate solution is sufficiently accurate. The distinctive nature of the two families makes it clear that, in contrast with basic matrix operations algorithms, the different algorithms for systems of linear equations are not simple adaptations derived from the matrix properties.

The final subsection defines the storage format abstraction level; the abstraction level at which traditionally the matrix calculations are implemented. It is shown that, for each specialised algorithm when combined with storage formats for the matrix operands, different implementations are generated.

The terms *algorithm*, *storage format* and *implementation* are used in the computer science sense; i.e. that an implementation (program) is an algorithm plus storage format (data structure).

2.3.1 Matrix Matrix Multiplication

The product of a matrix A of dimension $m \times p$ with a matrix B of dimension $p \times n$ is another matrix C of dimension $m \times n$ with elements defined as

$$c_{ij} \leftarrow \sum_{k=1}^{p} a_{ik}b_{kj}.$$

When describing the algorithm, given by the above definition, three nested loops are necessary (see Figure 2.6). This algorithm assumes that both A and B are dense matrices.

The next algorithm is an example of matrix-matrix multiplication where one

```
for $i = 1$ to $m$
 for $j = 1$ to $n$
  for $k = i$ to $p$
   $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
  end for
 end for
end for
```

Figure 2.7: Algorithm for matrix-matrix multiplication $C \leftarrow AB$ with A upper triangular and B dense matrices.

```
for $i = 1$ to $n$
 for $j = 1$ to $n$
  for $k = max(i, j)$ to $n$
   $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
  end for
 end for
end for
```

Figure 2.8: Algorithm for matrix-matrix multiplication $C \leftarrow AB$ with A upper triangular and B lower triangular matrices.

of the matrices is not dense (see Figure 2.7). When A is upper triangular with dimension $m \times p$ and B is dense with dimensions $p \times n$ the algorithm can be modified (to shorten the k loop) so that the elements a_{ij} with $i \geq j$ are not used since they are known to be zero:

$$\left.\begin{array}{ll} a_{ik} \neq 0, & i \leq k \\ a_{ik} = 0, & i > k \end{array}\right\} \Rightarrow c_{ij} \leftarrow \sum_{k=i}^{p} a_{ik}b_{kj}.$$

Two more examples are given in which neither of the matrix operands is dense. For the first example, A is upper triangular and B is lower triangular, both of dimension $n \times n$. Having as a starting point the algorithm of Figure 2.7, the algorithm of Figure 2.8 is obtained. The k loop is further shortened exploiting the zeros in matrix B:

$$\left.\begin{array}{ll} b_{kj} \neq 0, & k \geq j \\ b_{kj} = 0, & k < j \end{array}\right\} \Rightarrow c_{ij} \leftarrow \sum_{k=max(i,j)}^{n} a_{ik}b_{kj}.$$

The final example multiplies two upper triangular matrices of dimension $n \times n$.

As with the previous example, the algorithm of Figure 2.7 is used as a starting point. Since B is upper triangular the elements b_{ij} with $i \leq j$ are zero:

$$\left.\begin{array}{ll} b_{kj} \neq 0, & k \leq j \\ b_{kj} = 0, & k > j \end{array}\right\} \Rightarrow c_{ij} \leftarrow \sum_{k=i}^{j} a_{ik} b_{kj}.$$

Note that for $i > j$ the elements c_{ij} are zero, i.e. C is also upper triangular. In this case it is possible to shorten the j loop (see Figure 2.9).

```
for $i = 1$ to $n$
 for $j = i$ to $n$
  for $k = i$ to $j$
   $c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$
  end for
 end for
end for
```

Figure 2.9: Algorithm for matrix-matrix multiplication $C \leftarrow AB$ with A and B upper triangular matrices.

Generalising from these examples to all the basic matrix operations, it can be observed that for each basic matrix operation many algorithms can be derived. Each algorithm is derived by exploring the knowledge implied by the matrix properties. The number of algorithms that can be derived for a unary operation has a linear relation with the number of matrix properties. The number of algorithms that can be derived for a binary operation has a square relation with the number of matrix properties. Finally, as each specialised algorithm responds to certain matrix properties, a complete decision tree can be defined for each matrix operation. This takes as inputs the properties of the matrices and determines the specialised algorithm to be used and the properties of the solution matrix.

2.3.2 Systems of Linear Equations

Direct Methods

In the case of matrix-matrix multiplication the algorithms have been presented by refining the general algorithm for each special case. In the case of a system of linear equations $Ax = b$, the specialised algorithms are described first.

The first and simplest example is a diagonal matrix A. Remembering the

```
for $i = 1$ to $n$
  $x_i \leftarrow \frac{b_i}{a_{ii}}$
end for
```

Figure 2.10: Algorithm for a system of linear equations with A diagonal

definition of diagonal matrix, when $i \neq j$ the elements a_{ij} are zero. Therefore, the solution is obtained as follows:

$$\begin{pmatrix} a_{11} & & & & \\ & \ddots & & & \\ & & a_{ii} & & \\ & & & \ddots & \\ & & & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} \Rightarrow \left\{ \begin{array}{l} x_1 \leftarrow \frac{b_1}{a_{11}} \\ \vdots \\ x_i \leftarrow \frac{b_i}{a_{ii}} \\ \vdots \\ x_n \leftarrow \frac{b_n}{a_{nn}} \end{array} \right. ,$$

which is the basis of the algorithm of Figure 2.10.

In the second example, the $n \times n$ matrix A is lower triangular. This means that the elements a_{ij} with $i < j$ are zero. The solution is obtained as follows:

$$\begin{pmatrix} a_{11} & & & & & & \\ a_{21} & a_{22} & & & & & \\ a_{31} & a_{32} & a_{33} & & & & \\ \vdots & \vdots & \vdots & \ddots & & & \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{ii} & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{ni} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} \Rightarrow \left\{ \begin{array}{l} x_1 \leftarrow \frac{b_1}{a_{11}} \\ x_2 \leftarrow \frac{b_2 - a_{21}x_1}{a_{22}} \\ x_3 \leftarrow \frac{b_3 - (a_{31}x_1 + a_{32}x_2)}{a_{33}} \\ \vdots \\ x_i \leftarrow \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}} \\ \vdots \\ x_n \leftarrow \frac{b_n - \sum_{j=1}^{n-1} a_{nj}x_j}{a_{nn}} \end{array} \right. ,$$

which is the basis of the algorithm called *forward-substitution* and presented in Figure 2.11. In a similar way, the *back-substitution* algorithm to solve an upper triangular system of linear equations can be derived.

A direct method for the solution of a general system of linear is based on the factorisation of the matrix A. Since systems of linear equations with diagonal and triangular matrices have straightforward algorithms, the interesting factorisations are those which efficiently factorise matrices into the product of matrices with these properties. Taking LU-factorisation as an example, the matrix A is factorised as $A = LU$, where L is unit-diagonal ($l_{ii} = 1$) lower triangular, and

```
for $i = 1$ to $n$
 $x_i \leftarrow b_i$
 for $j = 1$ to $i - 1$
  $x_i \leftarrow x_i - a_{ij}x_j$
 end for
 $x_i \leftarrow \frac{x_i}{a_{ii}}$
end for
```

Figure 2.11: Forward-substitution algorithm for a system of linear equations with A lower triangular.

U is upper triangular. Given this factorisation, the system of linear equations $Ax = b$ can be rewritten as $LUx = b$. Thus the system $Ax = b$ can be solved, by forward-substitution for $Ly = b$ and back-substitution for $Ux = y$. Table 2.5 presents some other factorisations developed for systems of linear equations where matrix A has particular properties. Each of these factorisation algorithms can be specialised for nonzero structures.

Pivoting is a technique that is used within factorisations to keep the error of the solutions as low as possible. It is out of the scope of this thesis to present floating point arithmetic [Gol91], demonstrate the error bounds of solutions obtained by different factorisations with and without pivoting [Hig96] and therefore the need of pivoting.

When the coefficient matrix is sparse, a factorisation creates new nonzero elements in the factor matrices where zero elements were in the coefficient matrix. Each of these new nonzero element is called a *fill-in element*. Reordering the equations and the variables can reduce the number of fill-in elements. A reordering transforms the coefficient matrix by interchanging rows and columns. The execution time is reduced by reducing the number of fill-in elements since this preserves the sparsity of the coefficient matrix and so can be exploited.

The solution of sparse systems of linear equations is divided into reordering the coefficient matrix, factorisation and solve. An ordering implementation can take in account the numerical values or simply consider the position of the nonzero elements; *sparsity pattern*. A *numerical ordering*, first kind of ordering implementations, produces a reordering which includes the pivoting and performs a factorisation. The posterior factorisation may be only used by other systems of linear equations which have a similar sparsity pattern. A *symbolic ordering*, second kind of ordering implementations, produces a reordering which does not

When A dense or banded – LU-factorisation defined as $A = LU$ where L unit-diagonal lower triangular and U upper triangular
When A symmetric positive definite – Cholesky factorisation defined as $A = U^TU$ or $A = LL^T$ where L lower triangular and U upper triangular
When A symmetric positive definite tridiagonal – LDL^T-factorisation defined as $A = LDL^T$ or $A = UDU^T$ where L is unit-diagonal lower bidiagonal, U is unit-diagonal upper bidiagonal and D is diagonal
When A symmetric indefinite – Symmetric indefinite factorisation defined as $A = LDL^T$ or $A = UDU^T$ where L is unit-diagonal lower triangular, U is unit-diagonal upper triangular and D is block diagonal with blocks of order 1 or 2

Table 2.5: Recommended factorisations for systems of linear equations with dense and banded matrices.

include pivoting and is used by posterior factorisations. A numerical ordering uses dynamic data structures to store the coefficient matrix since the number of fill-in elements is not known until it is actually performed. Consequently, posterior factorisations can use a static storage format. A symbolic ordering also uses dynamic data structures, but its posterior factorisation uses dynamic data structures to account for the fill-in elements which are produced as a consequence of the pivoting.

An ordering implementation communicates the reordering to a factorisation. Some reorderings are represented as matrices known as permutation matrices. Other reorderings are represented as trees such as elimination trees [Liu90].

The numerical linear algebra community has not yet been able to determine the matrix properties for which each ordering algorithm is adequate.

For a more detailed approach to direct methods for linear systems of equations see [Ste73], [DER86], [GvL96], and [TI97].

Iterative Methods

The algorithms classified as iterative methods are mainly used with sparse matrices. The number of iterations necessary to achieve a sufficiently accurate solution defines the cost of these algorithms. This number depends on the characteristics of matrix A. For this reason, iterative algorithms usually involve the calculation of an extra matrix, a *preconditioner*, that transforms matrix A into one with more favourable characteristics. The favourable characteristics can be seen as matrix properties but the cost of the algorithm to test these properties is comparable to the cost of solving the sparse system of equations. Thus, in practice, the choice of preconditioner and iterative algorithm cannot be determined as a function of matrix properties; it is a process determined by experimentation and testing of different combinations. For a more technical approach to iterative methods for linear systems of equations see [BBC$^+$94], [Axe94] or [Saa96].

2.3.3 Storage Format Abstraction Level

The storage format abstraction level is defined as the level of abstraction of an implemented matrix operation that knows the representations of the matrix operands and accesses these directly.

As an example, take the matrix-matrix multiplication algorithm with A upper triangular and B dense (see Figure 2.7). An implementation of this algorithm using dense format for both A and B is presented in Figure 2.12. NumType is the data type of the matrix elements (real, complex, ...). Reading the code of this implementation, it can be seen that each matrix is stored in a two-dimensional array (i.e. dense format). This means that, if A instead is stored in packed format then the implementation is no longer valid. Figure 2.13 presents an implementation of the same algorithm, but with A stored in packed format (i.e. as a one-dimensional array).

```
NumType A(m,m)
NumType B(m,n)
NumType C(m,n)

do j=1,n
 do i=1,m
  temp = 0
  do k=i,m
   temp = temp + A(i,k)*B(k,j)
  end do
  C(i,j) = temp
 end do
end do
```

Figure 2.12: Implementation of matrix-matrix multiplication $C \leftarrow AB$ with A upper triangular and B dense, both stored in dense format.

```
NumType APACK(m*(m-1)/2+m)
NumType B(m,n)
NumType C(m,n)

do j=1,n
 do i=1,m
  temp = 0
  do k=i,m
   temp = temp + APACK(i+k*(k-1)/2)*B(k,j)
  end do
  C(i,j) = temp
 end do
end do
```

Figure 2.13: Implementation of matrix-matrix multiplication $C \leftarrow AB$ with A upper triangular stored in packed format and B dense stored in dense format.

Note that an implementation is at storage format abstraction level if changing the storage format implies changing the implementation. Traditional library implementations of the matrix calculations are implemented at this abstraction level.

To summarise the contents of this section, a given matrix calculation has many specialised algorithms. For each of these algorithms there can be many implementations corresponding to different storage formats for the matrix operands. Thus there is an explosion in the number of possible implementations. The developers of these libraries have to balance the number of implementations (i.e. algorithms and storage formats) that are supported with the effort of developing the code.

2.4 Developing Numerical Linear Algebra Programs

To review the contents of preceding sections:

- matrices can be classified by different criteria and each classification is known as a matrix property;
- a given matrix can have different storage formats;
- for each matrix calculation many algorithms that take particular advantage of the matrix properties can be derived;

- for each algorithm many implementations are necessary due to the different storage formats;
- for block banded, banded and dense matrices, the implementation to use for matrix calculation can be decided as a function of the matrix properties and their storage formats;
- for sparse systems of equations, either direct or iterative methods, it is not possible automatically to select the implementation (i.e ordering implementation or combination preconditioner iterative method).

The objective of this section is to understand how these concepts are organised in traditional linear algebra libraries. The term "traditional libraries" refers to the libraries developed, in this case by the numerical linear algebra community, using top-down methodology and implemented in imperative languages, predominantly Fortran, with no programmer-defined data types. BLAS [BLA99] and LAPACK [ABD$^+$95] are chosen as examples of traditional libraries to be described. An important characteristic is the community consensus or *de facto* standardisation process which is behind their design. Other examples of libraries are LINPACK [DBMS79], EISPACK ([SBD$^+$76], [GBDM77]), LAPACK [ABD$^+$95], NAG[1], IMSL[2], SPARSPAK ([GL79], [GL81]), YSMP [EGSS82], MA28 [Duf77].

BLAS and LAPACK are compared with two alternative linear algebra environments: Matlab [Mat] and the Sparse Compiler ([Bik96], [BW96], [BW99], [BBKW98]). Rather than a theoretical discussion about the three possibilities, the matrix calculations introduced in Section 2.3 are used to illustrate the differences, advantages and disadvantages.

Matlab is a computing environment and programming language for numerical computations. Its main characteristic is that the programming language is matrix-based. Thus, a Matlab program for linear algebra resembles its mathematical form.

The Sparse Compiler parses a given dense Fortran 77 program into an equivalent sparse Fortran 77 program. A dense program means a linear algebra program that stores its matrices in dense format even if some of matrices have some nonzero elements structure. An equivalent sparse program means a linear algebra

[1]A commercial product of Numerical Algorithms Group Inc. http://www.nag.com

[2]International Mathematical and Statistical Libraries (IMSL) a commercial product of Visual Numerics Inc. http://www.vni.com

program that implements the same calculations but those matrices with nonzero elements structures are stored in advisable storage formats. The Sparse Compiler analyses the nonzero elements structure of matrices and transforms the parts of the dense program that define the matrices so detected to have certain nonzero elements structure, and the parts of the dense program that operate on these matrices. The dense program is transformed so that it uses the new storage formats selected by the compiler and exploits the nonzero elements structure of the matrices.

2.4.1 Using BLAS and LAPACK

BLAS (Basic Linear Algebra Subprograms) offers subroutines for basic matrix operation while LAPACK (Linear Algebra Package) offers subroutines for systems of linear equations, least square problems, and eigenvector and eigenvalue problems. Both libraries are implemented in Fortran 77 and are designed to provide high performance [DW95], i.e. to achieve maximum performance from a given computer.

The routines provided by the BLAS are divided into three groups:

- Level 1 BLAS – routines that require $O(n)$ floating point operations and involve $O(n)$ data items [LHKK79], e.g. dot product $x^T y$ or a vector norm $||x||_1$,
- Level 2 BLAS – routines that require $O(n^2)$ floating point operations and involve $O(n^2)$ data items ([DCHH88b], [DCHH88a]), e.g. matrix vector multiplication Ax, and
- Level 3 BLAS – routines that require $O(n^3)$ floating point operations and involve $O(n^2)$ data items ([DCHD90],[DCHD90]), e.g. matrix-matrix multiplication AB.

BLAS subroutines have been specified for dense, banded, sparse, symmetric, symmetric banded, upper and lower triangular, and upper and lower triangular banded matrices. Dense matrices are stored in dense format (GE). Banded matrices are stored in band format (GB). Symmetric matrices are stored in dense (SY) or packed format (SP). Triangular matrices are stored in dense (TR) or packed format (TP). Triangular band matrices are stored in band format (TB) and also symmetric banded (SB). Finally, sparse matrices (US) are stored in coordinate or compressed sparse column or compressed sparse row or sparse diagonal or block coordinate or block compressed sparse column or block compressed sparse row or block sparse diagonal or variable block compressed sparse row format.

Based on the case of matrix-matrix multiplication (Section 2.3), the process of developing a linear algebra program with the BLAS is described below. Given the problem description $C \leftarrow AB$ where A and B are known to be dense, the first task is to find the correct BLAS subroutine. The subroutine names follow a strict naming scheme: the first letter of the name indicates the numerical data type (REAL, DOUBLE PRECISION, COMPLEX and DOUBLE COMPLEX) of the operands; the next two letters specify the matrix properties and the storage format (in the preceding paragraph, the pairs of letters between parenthesis show the different combinations and their meanings); the final three letters indicate the matrix operation. Table 2.6 includes the specification of the different subroutines for matrix-matrix multiplication. Apart from the number of subroutines, the long lists of parameters make for an unfriendly interface.

Following the naming scheme, the xGEMM subroutine is selected. The parameters pass information about the sizes of matrix operands, the representation of the three matrix storage formats, and flags to indicate if any of the matrices have to be transposed. The functionality of xGEMM implements four matrix operations: two matrix scalings, one matrix-matrix multiplication and one matrix-matrix addition ($C \leftarrow \alpha AB + \beta C$). The reason for these extensions to the basic matrix-matrix multiplication is that all the operations can be implemented within the three nested loops of matrix-matrix multiplication and it is, thus, more efficient than separating the operations.

For the case where A is upper triangular, the appropriate subroutines are xTRMM and xTPMM. The first subroutine implements the operation using dense format, while the second uses packed format. If the first subroutine is selected, memory space might be wasted, whereas if xTPMM is selected, the user must understand and create the representation (packed format) required by the subroutine.

For the case where A and B are both upper triangular, the appropriate subroutines are again xTRMM and xTPMM. BLAS subroutines have been developed in such a way that only one of the input matrices (for binary operations) is considered to have properties others than dense. Hence, the BLAS are not complete in the sense that not all of the possible implementations are included. In this

Subroutine Specification	Functionality
xGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A) op(B) + \beta C$
xGBMM(SIDE, TRANSA, TRANSB, M, N, K, KL, KU, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A) op(B) + \beta C$ or $C \leftarrow \alpha op(B) op(A) + \beta C$ where A is banded stored in band format
xSYMM(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where A is symmetric
xSBMM(SIDE, UPLO, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where A is symmetric banded stored in band format
xSPMM(SIDE, UPLO, M, N, ALPHA, AP, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C$ or $C \leftarrow \alpha BA + \beta C$ where A is symmetric stored in packed format
xTRMM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A) B$ or $B \leftarrow \alpha B op(A)$ where A is unit-diagonal or not and upper or lower triangular
xTBMM(SIDE, UPLO, TRANSA, DIAG, M, N, K, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A) B$ or $B \leftarrow \alpha B op(A)$ where A is unit-diagonal or not and upper or lower triangular banded stored in band format
xTPMM(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, AP, LDA, B, LDB)	$B \leftarrow \alpha op(A) B + \beta C$ or $B \leftarrow \alpha B op(A) + \beta C$ where A is unit-diagonal or not and upper or lower triangular stored in packed format
xUSMM(TRANSA, K, ALPHA, A, B, LDB, BETA, C, LDC)	$B \leftarrow \alpha op(A) B + \beta C$ where A is sparse stored in a sparse format

Table 2.6: BLAS subroutines for matrix-matrix multiplication – $op(A)$ represents A or A^T and, unless indicated, matrices are stored in dense format.

case, the waste of memory space is larger since the three matrices involved are all upper triangular, and only one matrix can be stored in packed format.

LAPACK subroutines are divided into those that solve standard problems, called driver subroutines, and presented in Section 2.1, and those which compute factorisations and other calculations used by the driver subroutines. Another long list of matrix properties and storage formats is supported and is organised following the naming scheme described with BLAS.

Figures 2.14, 2.15 and 2.16 present pseudo-Fortran programs to solve the system of linear equations $ABx = c$ where A and B are $n \times n$ matrices. The programs on the left hand side of these figures follow an algorithm which first performs the matrix-matrix multiplication and then solves the system of equations. Alternatively, the programs on the right hand side of these figures follow an algorithm which first solves the system of linear equations $Ay = c$ and then the system $Bx = y$. Both algorithms are semantically equivalent, i.e. they calculate the same result assuming perfect floating point arithmetic. Figure 2.14 presents programs to solve the system of linear equations $ABx = c$ where A and B are $n \times n$ dense matrices. Figures 2.15 and 2.16 presents programs to solve the same problem, but here A and B are upper triangular matrices. The first figure uses dense format while the second figure uses packed format, whenever possible.

```
NumType A(n,n)
NumType B(n,n)
NumType D(n,n)
NumType xc(n,1)
INTEGER IPIV(n), INFO

call initialise(A,B,xc)
C D=A*B

call XGEMM('N', 'N', n, n, n,
1.0, A, n, B, n, 0.0, D, n)
C solve system Dx=xc and leave x
in xc
call XGESV(n, 1, D, n, IPIV, xc,
1, INFO)
```

```
NumType A(n,n)
NumType B(n,n)

NumType xc(n,1)
INTEGER IPIV(n), INFO

call initialise(A,B,xc)
C solve system Ay=xc and leave y
in xc
call XGESV(n, 1, A, n, IPIV, xc,
n, INFO)
C solve system Bx=xc and leave x
in xc
call XGESV(n, 1, B, n, IPIV, xc,
1, INFO)
```

Figure 2.14: Programs using BLAS and LAPACK to solve the system of equations $ABx = c$ where A and B are $n \times n$ dense matrices.

```
NumType A(n,n)
NumType B(n,n)
NumType xc(n)

call initialise_tr(A,B,xc)
C B=A*B

call XTRMM('L', 'U', 'N', 'N',
n, n, 1.0, A, n, B, n)
C solve system Bx=xc and leave x
in xc
call XTRSV('U', 'N', 'N', n,
1.0, B, n, xc, 1)
```

```
NumType A(n,n)
NumType B(n,n)
NumType xc(n)

call initialise_tr(A,B,xc)
C solve system Ay=xc and leave y
in x
call XTRSV('U', 'N', 'N', n,
1.0, A, n, xc, 1)
C solve system Bx=xc and leave x
in xc
call XTRSV('U', 'N', 'N', n,
1.0, B, n, xc, 1)
```

Figure 2.15: Programs using BLAS and LAPACK to solve the system of equations $ABx = c$ where A and B are $n \times n$ upper triangular matrices stored in dense format.

```
NumType APACK(n,n)
NumType B(n,n)
NumType xc(n)

call initialise_tr(APACK,B,xc)
C B=APACK*B

call XTPMM('L', 'U', 'N', 'N',
n, n, 1.0, APACK, n, B, n)
C solve Bx=xc and leave x in xc

call XTRSV('U', 'N', 'N', n,
1.0, B, n, xc, 1)
```

```
NumType APACK(n*(n-1)/2+n)
NumType BPACK(n,n)
NumType xc(n)

call initialise_tr(APACK,BPACK,xc)
C solve APACKy=xc and leave y in
xc
call XTPSV('U', 'N', 'N', n,
1.0, APACK, xc, 1)
C solve BPACKx=xc and leave x in
xc
call XTPSV('U', 'N', 'N', n,
1.0, BPACK, xc, 1)
```

Figure 2.16: Programs using BLAS and LAPACK to solve the system of equations $ABx = c$ where A and B are $n \times n$ upper triangular matrices stored in packed format, whenever possible.

To summarise, this process can be generalised to describe the development of linear algebra programs with traditional libraries as:

- describe the problem in terms of matrix calculations,
- analyse the matrices to determine their properties,
- select the library or libraries which support the operations and properties,
- select the subroutines which best fit the matrix properties, and
- declare the variables conforming to the storage format that is supported by the selected subroutines.

2.4.2 Using Matlab

Matlab is not only an environment for numerical linear algebra; regressions, interpolation, numerical integration, graphs, visualisation of results, etc. are integrated. Its major characteristic is that the programming language is matrix based, i.e. every variable is a matrix. For example, the multiplication of two matrices $C \leftarrow AB$ is written as `C=A*B` and the solution of a system of linear equations $Ax = b$ can be written as `x=A\ b` or `x=inv(A)*b` where `inv(A)` performs A^{-1}.

Matlab does not always exploit the matrix properties that are supported in LAPACK and BLAS, and uses only dense and compressed sparse column format for sparse matrices [GMS92].

Matlab provides LU, Cholesky, QR, Eigenvalue and Singular value factorisations. Thus, for example, the solution of a system $Ax = b$ using LU-factorisation is written as `[L,U]= lu(A); y=L\ b; x=U\ y;`. The "\" operator follows the algorithm:

- if the matrix is not square then solve least squares problem,
- otherwise, if the matrix is triangular then use back or forward substitution,
- otherwise, if it is symmetric and the diagonal elements are positive real[3] then attempt to solve with Cholesky factorisation,
- otherwise (i.e. Cholesky factorisation fails or is not symmetric with positive diagonal elements), solve with LU-factorisation.

[3]Heuristic used by Matlab to test if a matrix could be positive definite.

Figure 2.17 presents Matlab programs to compute the system of linear equations $ABx = c$ where A and B are dense matrices. Figure 2.18 presents Matlab programs to compute the same problem except that A and B are upper triangular matrices. Note that both figures present identical programs, but for the initialisation. Although transparent for users, the “\“ operator solves the system using LU factorisation for the first figure while for the second figure it uses back-substitution.

`initialise(A,B,c)`	`initialise(A,B,c)`
`D=A*B;`	`y=A\c;`
`x=D\c;`	`x=B\y;`

Figure 2.17: Matlab Programs to solve the system of equations $ABx = c$ where A and B are $n \times n$ dense matrices.

`initialisetr(A,B,c)`	`initialisetr(A,B,c)`
`D=A*B;`	`y=A\c;`
`x=D\c;`	`x=B\y;`

Figure 2.18: Matlab Programs to solve the system of equations $ABx = c$ where A and B are $n \times n$ upper triangular matrices.

The task of developing a linear algebra program with Matlab follows the steps:

- describe the problem in terms of matrix calculations,
- analyse the matrices to identify matrix properties, and
- map the problem into Matlab operators.

2.4.3 Using the Sparse Compiler

The Sparse Compiler is a source-to-source compiler that has as input dense Fortran 77 programs and as output sparse Fortran 77 programs. A dense program is a program which stores all the matrices in dense format (in Fortran 77 case `NumType A(n,m)`) and the matrix calculations are implemented using all the elements. A sparse program is a program that stores and implements the matrix calculations taking advantage of matrix properties. The compiler is divided into two phases: dense program analysis and sparse code generation.

The program analysis automatically detects the nonzero elements structure of matrices [BW99] and identifies the parts of the code that access zero elements. The user of the compiler can also provide information about the nonzero elements structure of the matrices through comments. Figure 2.19 presents the notation used in the comments to declare an upper triangular matrix.

The code generation phase takes into account the nonzero elements structure and how the matrices are accessed in order to select the storage format and automatically generate the sparse code ([BW96], [BBKW98]). In other words, the compiler changes the dense format declaration of some matrices by the declaration of the selected new storage format. It also eliminates the redundant instructions because of the nonzero elements structure found. Finally, it transforms those parts of the program that accessed matrices so that they align with the new storage formats.

The limitation of this work is that in some cases, specially hand optimised programs, the compiler fails to fully exploit the sparsity. Its second limitation is that reordering algorithms cannot be used, thus the fill-in effect, creation of nonzero elements where there were zero elements, cannot be avoided and usually the resultant sparse code could be significantly improved.

Figure 2.19 presents the Fortran 77 dense program commented for the sparse compiler to compute $ABx = c$ where A and B are upper triangular. Note that no support is provided by the compiler to develop the dense programs so usually the dense sub-set of BLAS or LAPACK would be used.

```
NumType A(n,n)
C_SPARSE(ARRAY(A), ZERO (I>J))
C_SPARSE(ARRAY(A), DENSE(I<=J)
NumType B(n,n)
C_SPARSE(ARRAY(B), ZERO (I>J))
C_SPARSE(ARRAY(B), DENSE(I<=J)
```

Figure 2.19: Sparse Compiler commented dense program to solve the system of equations $ABx = c$ where A and B are $n \times n$ upper triangular matrices.

The task of developing a linear algebra program with the sparse compiler follows the steps:

- describe the problem with matrix calculations,
- generate Fortran 77 dense program for the matrix calculations, and

- indicate the nonzero elements structure, or let the compiler give feedback on this.

2.4.4 Advantages and Disadvantages

From the user's point of view, Matlab provides the easiest way to generate a linear algebra program. The users do not need to know how the matrices are stored or how the operators are implemented. The mapping of the matrix calculation is straightforward, although it has been shown that a given matrix calculation can have different semantically equivalent programs. The main drawback is the execution time of the programs since the user does not provide information about matrix properties, and except in specific situations, the programs cannot take advantage of them.

The Sparse Compiler represents the next level of difficulty. The user has to write Fortran linear algebra programs and thereby has to know how the matrix calculations are implemented using dense format. However, the sparse compiler offers support to decide the nonzero elements structure and exploits any such structure that is found. Neither Matlab nor BLAS and LAPACK libraries provide such functionality.

BLAS and LAPACK represent the maximum level of difficulty. Matrices can be represented in different storage formats and the user has to know how to declare them. The selection of a subroutine is not a trivial process. The list of parameters is complicated and too long to remember, therefore difficult to use. The users have to know how to declare the different storage formats. The functionality is not complete, not all the possibilities of matrix properties and storage format operands are observed. On the other hand, BLAS and LAPACK subroutines deliver the minimum execution time as they utilise state-of-the-art implementations.

The user perceives the difficulty of developing a linear algebra program as the distance to jump from the problem defined in terms of matrix operations to the specific software environment expression (subroutines in traditional libraries, operators in Matlab and comments and dense program in the Sparse Compiler). This distance is represented by the tasks that need to be completed in order to develop the program. These tasks are:

- matrix properties analysis,

- selection of storage formats,
- and selection of specific environment expressions that align with the properties and storage formats.

2.5 Summary

Matrix calculations are the core of this chapter; beginning with their definitions, continuing with characterisation examples of how matrix calculations are implemented, and ending with how they are organised in libraries.

Matrix calculations have been divided into basic matrix operations and matrix equations. Due to certain matrix properties, the definition of a basic matrix operation can be specialised and thus different algorithms that exploit the matrix properties are created. Due to the different storage formats of a matrix, the set of algorithms are further extended into a set of implementations.

Matrix equations can be solved either with direct or iterative methods. Direct methods perform a factorisation and then solve the systems for the factored matrices. When the matrix equations are sparse, the matrix can be reordered to preserve the sparsity of the factored matrices. However, it is not possible to decide efficiently which of the different ordering algorithms is the adequate one. Iterative methods are usually combined with preconditioners. Some iterative algorithms are known to fail to converge to a solution for specific matrix properties. In practice, the appropriate combination of iterative method and preconditioner for a system of linear equations cannot be decided automatically.

Traditional libraries are organised by strict naming schemes. For each subroutine the naming scheme describes the matrix calculation, the matrix properties of the input matrices and their storage formats. The parameters describe how the storage formats are represented.

The comparison of the BLAS and LAPACK with Matlab and with the Sparse Compiler shows that when developing a linear algebra program the BLAS and LAPACK based programs constitute the maximum level of difficulty. The difficulty is summarised by the tasks to be completed:

- describe the problem in terms of matrix calculations,
- analyse matrices to determine their properties,

- select the library or libraries that support the matrix calculations and properties,
- select the subroutines which best fit the matrix properties, and
- declare the variables conforming to the storage format that is supported by the selected subroutines.

The information of this chapter is reused mainly to the next chapter, which reports an object oriented analysis and design of linear algebra.

Readers are referred to [Gan59a] and [Gan59b] for a more detailed, analytical, approach to Numerical Linear Algebra. Descriptions and analysis of algorithms for matrix calculations can be found in [Ste73], [GvL96] and [TI97]. Detailed study of accuracy and stability of these algorithms can be found in [Hig96].

Chapter 3

Object Oriented Linear Algebra

Traditional libraries of linear algebra present two weaknesses: complex interfaces and an explosion of implementations of matrix calculations. The first weakness affects users since they find it hard to develop linear algebra programs using these libraries. The second weakness affects library developers since they have to code the many different implementations.

This chapter focuses on the analysis and design of an object oriented linear algebra library in order to overcome or reduce the two weaknesses. Object oriented software construction offers the possibility to define and use abstractions from a problem domain, in this case linear algebra. The objective is to create an object oriented model of linear algebra that hides the implementation details.

Object oriented software construction is reviewed in order to be able to create object oriented models of linear algebra (Section 3.1). The object oriented models are displayed graphically using a subset of UML notation. This notation is also introduced in Section 3.1.

Different models are proposed and used to classify several existent object oriented linear algebra libraries (Section 3.2). The object oriented model created identifies that current models do not model fully Linear Algebra. The new model constitutes the design of the Object Oriented Linear Algebra LibrAry (OoLaLa). This model enables OoLaLa to automatically manage the storage formats of matrices and propagate the matrix properties through matrix calculations. In addition, two implementation abstraction levels are described and both reduce the explosion of implementations of matrix calculations.

3.1 Object Oriented Software Construction

The process of developing software, applications or libraries, is inherently a human activity. A group of human software developers analyses certain problem, creates a model of it, and develops an implementation of that model in a programming language. As with many other problems faced by humans, the model to solve a given problem is created by dividing the problem into sub-problems repeatedly so that they eventually become trivial to solve. The model for the problem is then created as the composition of the sub-models.

> "The technique of mastering complexity has been known since ancient times: *divide et impera* (divide and rule)." Dijkstra [Dij79]

Top-down methodology, or structured programming, used by traditional linear algebra libraries, divides problems using an algorithmic decomposition, i.e. expressing what has to be done in terms of basic control structures (loops, if-then, etc.) or basic algorithms (sort, search, etc.). The basic decomposition unit is the subroutine or procedure, and thus the model is a composition of subroutine calls.

On the other hand, bottom-up methodology searches for abstractions of the problem domain, and divides the problem into an appropriate set of these abstractions. An abstraction is a key concept of the problem domain with the operations or services provided within that domain. A model of the problem is the interaction of abstractions through their defined operations, or interfaces. Special importance is given to hiding details of how the operations of the abstractions are implemented; thereby emphasis is simply placed on using the operations. In the literature, the abstractions are known as *abstract data types*.

The main advantage for software developers is that abstractions are a normal human approach to decomposing problems whereas algorithmic decomposition is an influence of what is provided by the first programming languages, such as Fortran 66, Fortran 77 or C. Using an example, it is not attractive to pass as parameters the representation of an abstraction, instead of the abstraction itself. Nowadays, few software developers would operate on an array when they want to use a stack. They would use an abstraction of the stack, often provided by modern programming languages, and use the interface (push, pop, etc.) to operate on the stack, even if it is ultimately represented as an array. Traditional linear algebra libraries are implemented accessing directly (not using an interface) the representation of the matrix, and the explosion in the number of subroutines that this provokes has been demonstrated in Chapter 2.

The objective of object oriented methodology is to propose an even more similar human approach to modelling complex problems. Object oriented methodology follows the bottom-up methodology and its basic concepts are explained in the next section. The motivation for object oriented methodology is to overcome the lack of abstraction which forces developers to always think about the problems in too much detail, thereby becoming error prone.

The remaining of the section is organised as follows. First, basic concepts (objects, classes, inheritance, client relation, etc.) of object oriented methodology (Section 3.1.1). These basic concepts are illustrated using examples from linear algebra. Some object oriented programming languages offer abstract classes and generic classes (Section 3.1.2). These are explained so as they are used in the posterior analysis and design. The next issue is to understand how the software development process is modified because of object orientation (Section 3.1.3). Finally, two design patterns and a short discussion about generic classes vs. inheritance (or how they can be simulated) are the suggested "tips" (Section 3.1.4).

3.1.1 Basic Concepts

Object oriented methodology is based on abstractions and information hiding, but includes another characteristic common of the human approach to decomposition of problems; classification. This new possibility enables software developers to create abstractions that are families of abstractions. Using object oriented terminology, the abstractions are now called *classes* and a specific member of an abstraction is called an *object*. Every object is said to be an *instance* of a class. For example, matrices might be an abstraction from the linear algebra problem domain and hence a class. A specific matrix would be an object of the class. Classes define common operations, such as "assign an element in certain position" or "access an element in certain position", and common characteristics that every object would have, such as the number of rows or number of columns. Classes have a static role since they are just definitions. Every object of a class conforms to the definitions described by the class and gives values to those definitions, also known as the state of an object. An object is dynamic since it is a run-time entity whose state can be modified.

Figure 3.1 presents a UML class diagram and object diagram of matrices. UML stands for Unified Modelling Language ([Rat97a], [Rat97b], [Mul97], [BRJ99]), which is an industrial standard notation, used to document object oriented software development. Other object oriented and structured programming notations can be found in [Wie98]. Class diagrams are used to represent classes graphically using a rectangle divided into three sub-rectangles. The first sub-rectangle contains the *class name*, the second contains the characteristics called *attributes* and the third contains the operations called *methods*, or *operations*. An object diagram is used to represent objects and is similar to the class diagram. The first sub-rectangle contains the name of the object and its class, separated by a colon and underlined. The second sub-rectangle contains the attributes that define the state of the object, and the third one contains the methods. As can be seem in the class diagram (Figure 3.1), there is a method `create` which creates objects of class `Matrix`. This method is a class method and hence appears underlined. A class method is a method that cannot be invoked in an object; it is invoked in the class. For the sake of clarity objects and classes are often represented in class diagrams and object diagrams only by their first sub-rectangle, thus not repeating known information. UML specifies how attributes and methods have to be declared in the diagrams. This thesis does not follow this specification and uses a pseudo-code based on Java syntaxes.

Once some basic UML notation has been introduced, attention is directed again to the possibility of classifying classes. Humans create hierarchies of abstractions using criteria by which each classification adds new characteristics or re-adapts existing ones. In object oriented methodology, the classes can be organised into *inheritance* hierarchies. Each class is a classification, and traversing upwards in the class hierarchy means a more general class, whereas traversing down the class hierarchy means a more specialised class. Using the example of matrices, vectors can be considered a special class of matrices, since they are matrices with either only one column or one row. In a similar way, square matrices can be considered a special class of matrices since they are matrices whose number of columns and rows has to be equal. These examples should be taken as naïve examples to illustrate the concepts. A more complete object oriented analysis and design is described in Section 3.2. Figure 3.2 presents a UML class diagram showing the inheritance relation between the class `Matrix` and the classes `ColumnVector`, `SquareMatrix` and `RectangularMatrix`. The inheritance relation is represented by an arrow which begins in the specialised class and ends in a more general class. The class `Matrix` defines the attributes, the methods

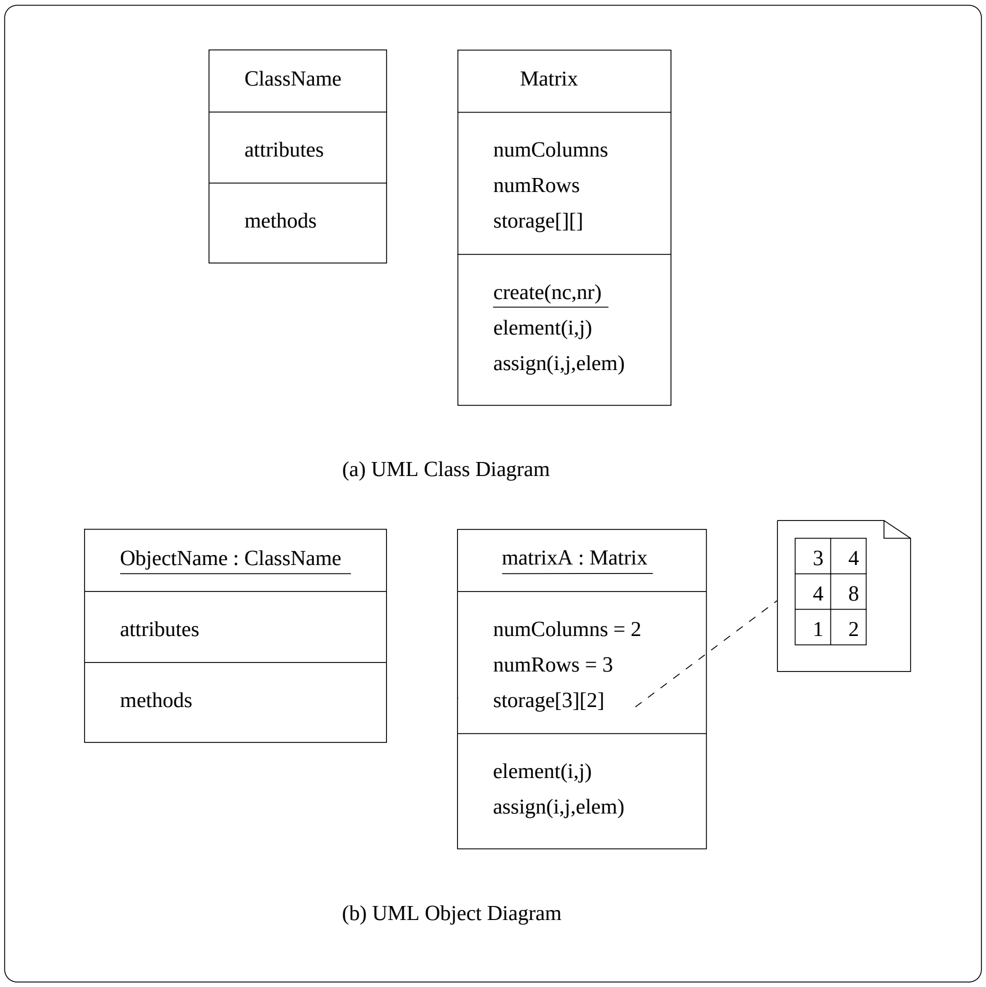


Figure 3.1: UML class diagram and object diagram for a naïve version of matrices.

and the implementations of the methods. All this is automatically inherited by the sub-classes `ColumnVector`, `SquareMatrix` and `RectangularMatrix`. A sub-class can add new methods or attributes, and also can adapt (re-implement) the methods inherited. In the class diagram, the method norm1 has been added to the class `Matrix` presented in Figure 3.1. The `norm1` method is implemented in this class following the definition for matrices ($max_j \sum_i |a_{ij}|$). However, in the class `ColumnVector` this method `norm1` is re-implemented efficiently for vectors ($\sum_i |x_i|$). The class `SquareMatrix` adds a new method `create`, which only needs one parameter for the number of rows and columns, and re-implements the inherited method `create` so that the parameters for the number of rows and columns are tested to be equal before an object is constructed.

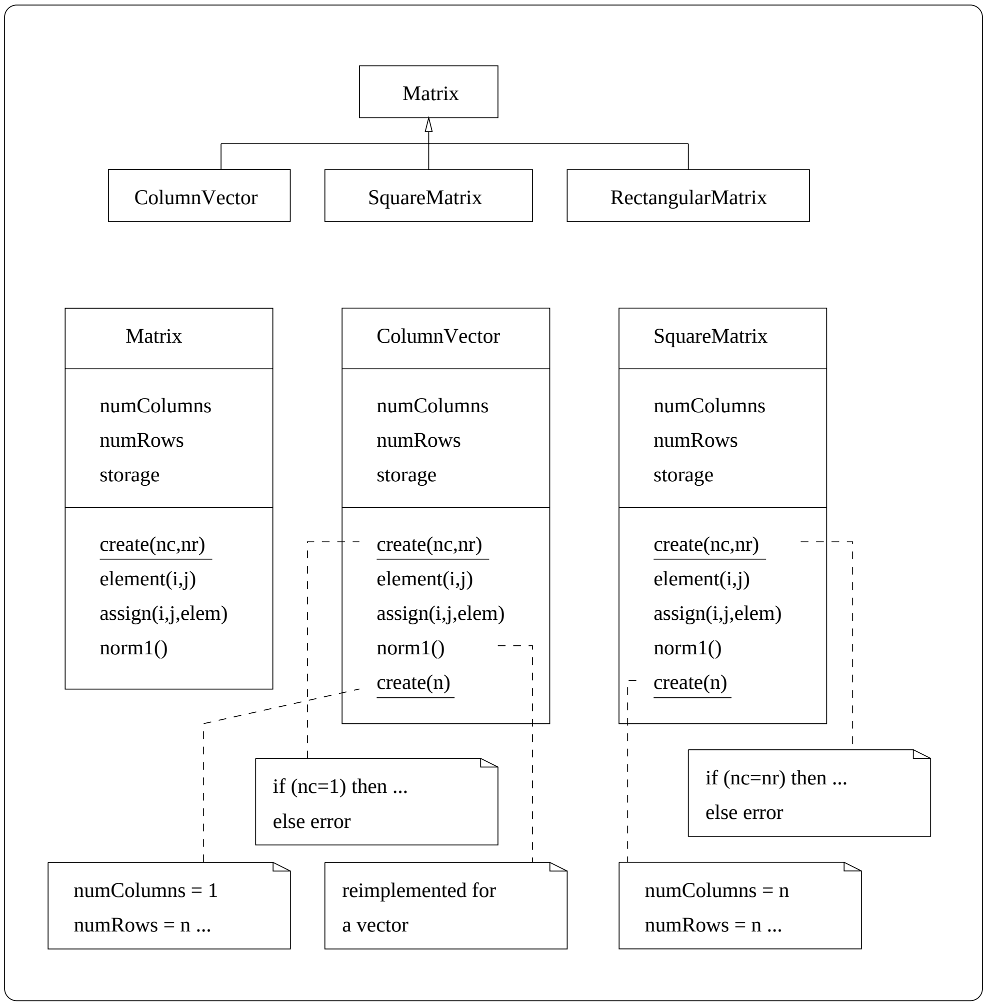


Figure 3.2: UML class diagram with a naïve inheritance hierarchy of matrices.

Classes can be seen as the data types defined by developers. The inheritance of a class `B` from a class `A` means that every object instance of class `B` is also an object of class `A`. In the case of matrices, every object of class `Vector` is always an object of class `Matrix`. A method that has as input parameter an object of class `A` accepts as valid all the objects of that class `A`. Apart from this and provided that class `B` inherits from `A`, every object instance of `B` is also an object of `A`. Hence, the method also accepts as valid the objects of `B`. In general, any object of a class that inherits directly or indirectly (i.e. inheritance through more than one class) from a class is a valid parameter. On the other hand, a second method that takes as input parameters of class `B` does not accept objects that are instances of class `A`. The feature that different objects of different classes are valid for a part of code is called *polymorphism*.

From the above paragraph and using the hierarchy introduced, every object of the classes `ColumnVector`, `SquareMatrix`, `RectangularMatrix` and `Matrix` is a valid parameter for methods that have as parameter an object of class `Matrix`. Suppose that one of these methods calls, or invokes, the method `norm1` in the parameter object of class `Matrix`. Note that the method `norm1` in the class `ColumnVector` is re-implemented while the classes `SquareMatrix` and `RectangularMatrix` inherit the implementation from the class `Matrix`. *Dynamic binding* is the mechanism which ensures that whatever valid object is passed as a parameter to the method, the correct `norm1` implementation would be executed. Dynamic binding identifies the exact class of the object and then checks if an implementation is provided in that class. Otherwise this mechanism traverses upwards through the class inheritance hierarchy, checking at each level whether or not an implementation of the method is provided. For example, when an object of class `ColumnVector` is passed as a parameter, the implementation provided in this class of `norm1` is executed. On the other hand, when an object of class `SquareMatrix` is passed as a parameter, the dynamic binding mechanism detects that its class `SquareMatrix` does not provide an implementation of `norm1`. Hence, it steps up one level to the class `Matrix` where the implementation is found and executed.

Apart from classifying, the inheritance relation between classes is a way of re-using code. Only the methods which need to be adapted to the characteristics of a more specialised class, and those methods specific to that class have to be implemented; the other implementations are simply inherited.

Multiple inheritance is a relation between one class which inherits from more than one different classes. All the explanations for inheritance are applicable to multiple inheritance, although certain problems that are caused by multiple inheritance, and omitted in this thesis, can arise during the development of object oriented software ([Mey97] Chapter 15).

An *association* between classes represents *links* between objects of these classes. Different variants of associations are defined in UML, but since only the general case (notation defined in Figure 3.3) is necessary in this thesis, the other possibilities are not discussed. The number of objects linked by an association is determined by the *cardinality* of that association. The cardinality is represented by numbers and "*" in the class diagram.

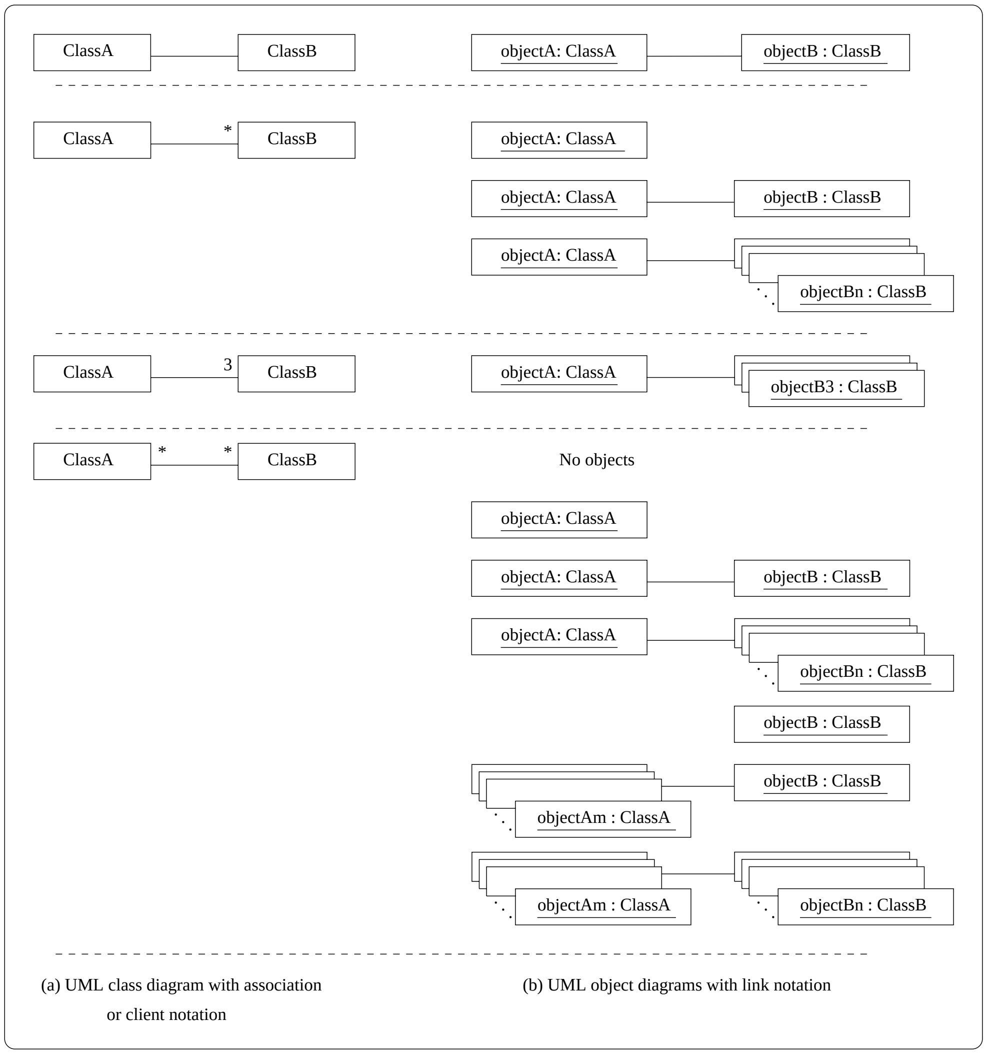


Figure 3.3: UML class and object diagrams with an association or client relation between two classes.

The association between classes represents a path through which methods are invoked by the objects so linked. This metaphoric path symbolises that a method is always invoked by an object in other object (although the other object might be itself). Meyer proposes the term *client relation* instead of association [Mey97]. The term comes from the fact that an object is using the interface of another object (the services provided by the other object) and thus they become client and supplier. The term client relation is used throughout this thesis, rather than association.

Compared with top-down decomposition, object oriented decomposition proposes a method closer to how humans approach problems. The decomposition is based on abstractions from the problem domain. These abstractions can be further abstracted creating hierarchical classifications of abstractions. The process of abstraction hides the details and enables developers to concentrate on how they interact together. The abstractions are called classes and individual members of a class are called objects. An object oriented model of a problem is a set of objects that, over a period of time, are created, destroyed and linked by client relations invoking operations (methods) from other objects. Each class knows the details of how it is implemented but does not know the details of the other's classes, just uses their services.

Object oriented concepts have been presented as an evolution towards a human-like approach to decomposition and composition of complex problems. From this perspective, it offers benefits to the developers of software. From a user perspective, the benefit depends mainly on whether the user is a user of software applications or a user of software libraries. Taking users of libraries, in particular the users of numerical linear algebra libraries, the interfaces would pass from being a list of subroutines with parameters showing the exact representation of the matrices, to operations (methods) between objects representing matrices where the representation and algorithm details are hidden from the user.

3.1.2 Implementation Related Concepts

Generic programming and *abstract classes* are two advanced concepts, which are sometimes supported by object oriented programming languages. These concepts are related to implementation aspects whereas those already explained are methodological.

In general, generic programming enables developers to write parts of programs that have as a parameter the data type of some variables. Generic programming was proposed from the observation that some algorithms could be written independently of the data types. A typical example is a sorting algorithm. The implementation of a sorting algorithm could be the same as long as the data type of the elements to be sorted has defined the comparison functions "<", ">" and "=". An early version of the Z specification language ([Abr80], [ASM80]), CLU [LAB+81], and Ada [ANS83] are the first languages that supported generic programming.

In object oriented programming languages, a class can also be generic and, thus a *generic class* is a class that has as parameters the data types or classes of some of its attributes or parameters of its methods. Generic classes are also known as template classes in the context of C++. Generic classes cannot instantiate any object since they are not complete classes. In this context, the typical examples for generic classes are the containers of elements. Lists, stacks, trees, etc. are well documented container classes that benefit from generic programming (see the Standard Template Library [LS95], [MS95], [Aus98]). Using generic classes, the containers can be defined independently from the class of the elements they will hold at run-time. In the particular case of linear algebra, the `Matrix` class might be considered a generic class whose parameter is a numerical data type. Figure 3.4 introduces the UML class diagram notation for generic classes and presents the example of class `GenericMatrix`.

Abstract classes are classes which declare methods and attributes but do not implement all the methods. The implemented methods are allowed to invoke the non-implemented methods, called also *abstract methods*. Hence, an abstract class is a completely declared but partially implemented class. No object can be instantiated from an abstract class, and only those classes that inherit from an abstract class and provide implementation for all the inherited abstract methods are not abstract classes. Figure 3.5 presents the UML notation for abstract classes and describes a class diagram for the naïve class hierarchy described in Figure 3.2. In this case, class `Matrix` does not have any attributes since some attributes become redundant for some sub-classes.

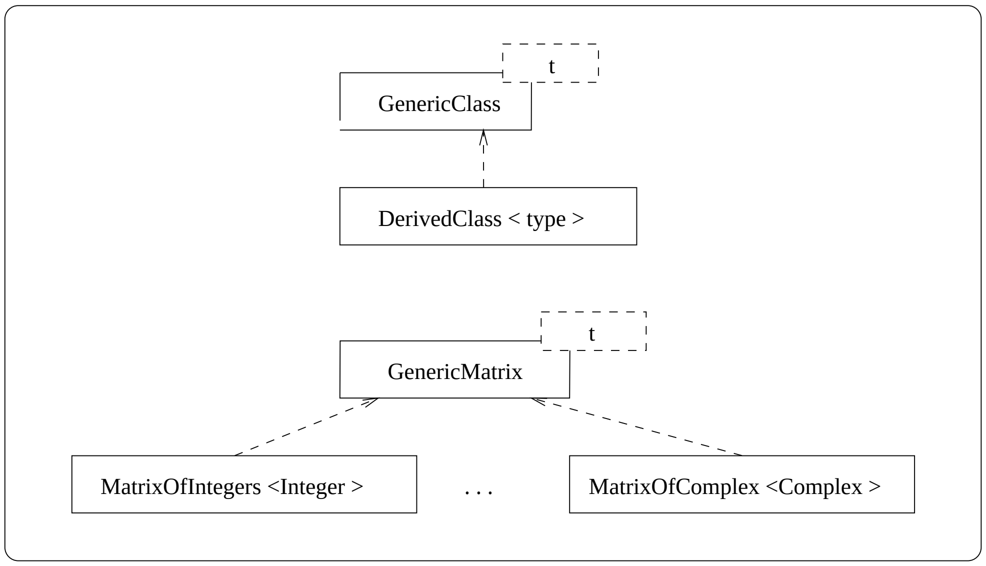


Figure 3.4: UML class diagram of a naïve generic class `GenericMatrix`.

3.1.3 The Software Development Process

Traditionally, the software development process has been divided into *analysis*, *design*, *implementation*, *testing* and *maintenance* phases using top-down decomposition. Each phase begins when the preceding phase has finished, and so the process can be seen as a linear execution. This life cycle is known as the linear sequential model, or *waterfall* model [Pre97].

Object oriented methodology does not change the abstract definition of the different phases. However, how they are carried out, and the products of each phase are different. The object oriented life-cycle is characterised by being iterative and incremental. At each iteration, object oriented analysis (OOA), object oriented design (OOD), object oriented implementation or programming (OOP) and object oriented testing are carried out increasing the part of the problem that is covered.

Of special interest for this thesis are OOA, OOD and OOP. OOA proposes classes, relations between classes, and the attributes and methods. The objective is to discover and understand the problem domain by modelling with objects and classes. OOD refines the classes by giving declarations to the classes and specification to the functionality of each method. At the same time, the model created by OOA is refined, adapting it to the restrictions of the application. The objective

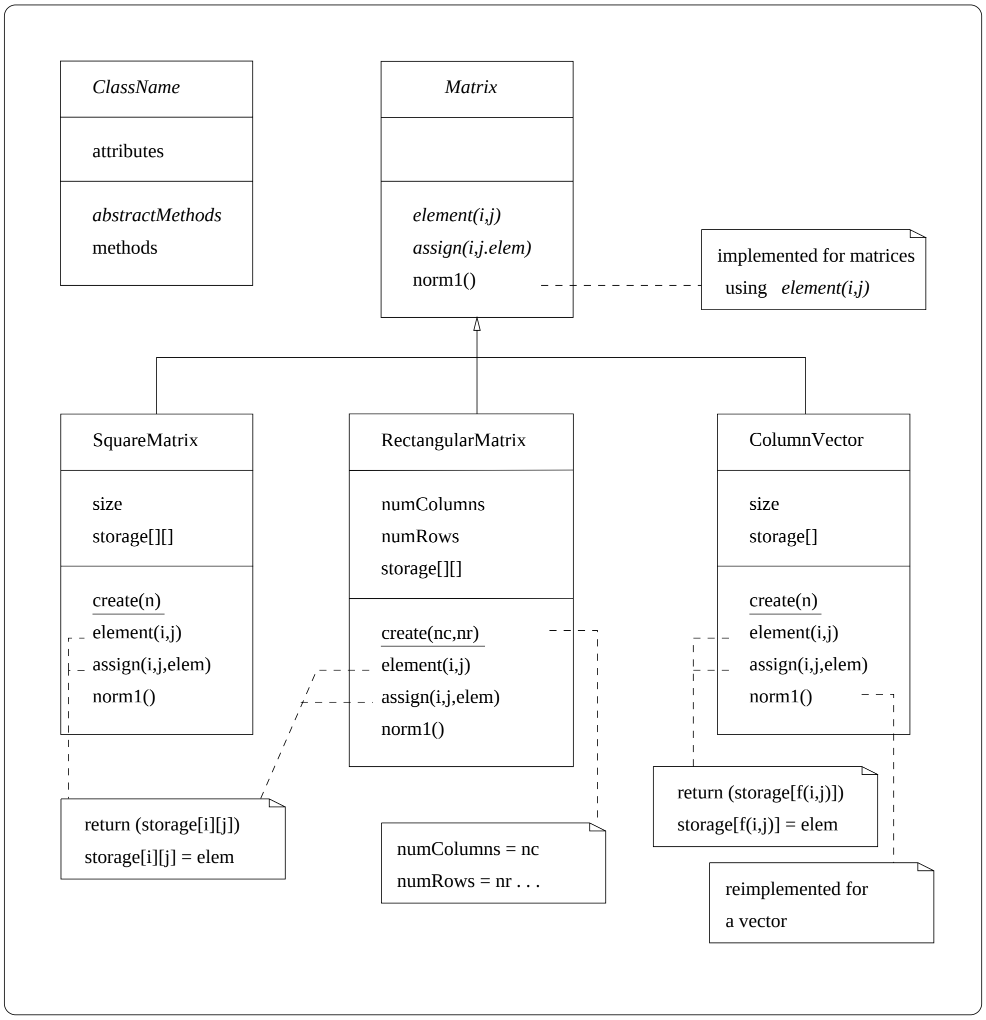


Figure 3.5: UML class diagram of a naïve abstract class `Matrix`.

is to plan how the model is going to be implemented. Finally, OOP is the implementation of the object oriented design in a given programming language. Ideally, the implementation should be made in an object oriented language; otherwise, the developers are forced to emulate the object oriented concepts. Guidelines for implementing object oriented models in non object oriented languages, such as Fortran 77 or C, are described by Meyer ([Mey97] Chapters 33 and 34) or by Decyk *et al.* ([DNS97a], [DNS97b], [DNS98]).

The division between OOA and OOD phases is fuzzy, although the focus and the products of both phases are clear. The analysis phase focuses on modelling the problem by proposing candidate classes and relations between the classes, evaluating them and rejecting the unsuitable proposals. Heuristics to find candidate classes are collected by Booch ([Boo94] Chapter 4) and Meyer ([Mey97] Chapter 22). Both authors identify as a source of candidate classes tangible things, roles, events, records of interactions, etc., from the problem domain ([SM88], [Ara89]). Also, both authors present a method based on studying a requirements document. The nouns and verbs expressing actions over them that are repeatedly used in this document become candidate classes and candidate methods [Abb83]. However, due to the complexity of natural language this approach has a limited success.

Booch and Meyer strongly disagree about the *use case* analysis formalised by Jacobson [JCJO92]. Use case analysis describes different scenarios, which are user-initiated transactions with the software. The scenarios represent the functions of the software. The analysis then takes each scenario, one-by-one, identifying possible classes and relations. In Booch's opinion, use case analysis provides an organised framework to discover the functionality required by an application and, from that, a good guide to follow. In Meyer's opinion, use case analysis is influenced by the users' vision about what the application has to do. This might lead non-expert object oriented developers to an algorithmic decomposition instead of an object oriented decomposition.

The OOD phase brings different requirements to the development process. Concurrency and synchronisation, mapping of the software onto the hardware (networks, modems, processors, etc.), and division of the object oriented model into packages, grouping related classes, are aspects that might be included during this phase [Kru95].

Following the above process, the OOA and part of the OOD for numerical linear algebra is carried out in Section 3.2. The different proposed classes and relations are used to classify current object oriented numerical libraries. Section 4.1 refines the object oriented model proposed in this chapter to accommodate the restrictions associated with the implementation programming language (Java).

3.1.4 Some Tips

The "rules" given in the literature for deciding what are the relations between classes, can be considered more as heuristics; they always end with examples of "exceptions". *Design patterns* are class structures which model problems that repeatedly appear in almost every development of software. The definition of design patterns, and a collection of them is described by Gamma *et al.* [GHJV95]. Design patterns can be considered as the heuristics extracted from the experience of expert object oriented developers. Each design pattern describes the characteristics of a repeatedly faced problem for which an "elegant" and tested solution is known. Obviously, the description of the problem and solution are in abstract terms, but real examples of the successful application are presented.

Two design patterns, the *bridge* ([GHJV95] pages 151–162) and the *iterator* ([GHJV95] pages 257–272) patterns, and a comparison between generic classes and inheritance are the "tips" suggested. These are used in the object oriented analysis and design described in the next section.

Bridge Pattern

Normally, when deciding what is the relation between classes, the client relation does not offer problems. However, it is not trivial to decide when the inheritance relation should be applied. The client relation can be semantically interpreted as a "has-a"; class `A` is client of `B` means that `A` has-a `B`. Similarly, the inheritance can be semantically interpreted as an "is-a"; class `B` inherits from class `A` means that `B` is-a `A`. For example, the problem defined by the phrase – "a person has a car" – does not offer any doubt about a client relation between a class `Car` and a class `Person`. The models `Car` is-a `Person` or `Person` is-a `Car` do not make sense. However, when adding a new phrase – "a black car is a car" – it is suggested that there are two classes: a class `Car` and a class `BlackCar` that inherits from `Car`. It is also possible to model the phrase as an object class `Car` has-an object of class `Color` and its state indicates is black. The decision depends on the problem domain and, without extra information, both models are valid.

In the case of linear algebra, the situation described in the last paragraph is repeated. The phrase to model is – "a matrix with some properties is a matrix". This phrase describing the problem suggests that class `MatrixWithProperties` is-a `Matrix`. It is also possible to model the phrase as class `Matrix` has-a `Property`. The decision and the arguments are presented in Section 3.2.1, although the bridge pattern, used to make the decision, is presented in the following paragraph.

The bridge pattern represents a problem where an abstraction can have different possibilities, only one possibility at each time, and during execution the possibility can change. The possibilities provide the same set of methods, but each possibility implements them differently. Figure 3.6 presents the class diagram of the proposed solution. A new abstract class named `Possibility` has been created where the common attributes and methods among the different possibilities is declared, but not implemented. Each possibility (`Possibility1`, ..., `PossibilityK`) is a class which inherits from the new abstract class `Possibility` and provides implementation for the inherited abstract methods. The abstraction becomes a class called `Abstraction` that is defined to be a client of the abstract class `Possibility`. This enables the client relation to be polymorphic. Figure 3.7 presents an example where the abstraction is a figure and the possibilities are circles and triangles.

Iterator Pattern

The iterator pattern presents a solution to traverse different kinds of containers with a unique interface. The iterator described by Gamma *et al.* [GHJV95] traverses and accesses the elements in sequential order and is presented in Figure 3.8. The methods `next` and `currentElement` advance one position in the container and return the current element, respectively. The method `begin` sets the iterator to the first position of the container, and the method `isFinished` tests if there are any more elements to be accessed in the container.

The Standard Template Library classifies the iterators, among others, into sequential and random access [LS95]. A random access iterator adds to class `Iterator` a new method `getElement` that returns the element in the position passed as a parameter.

The iterator pattern is used as a way to access the elements of matrices, thus enables linear algebra developers to adopt a different approach to the way that

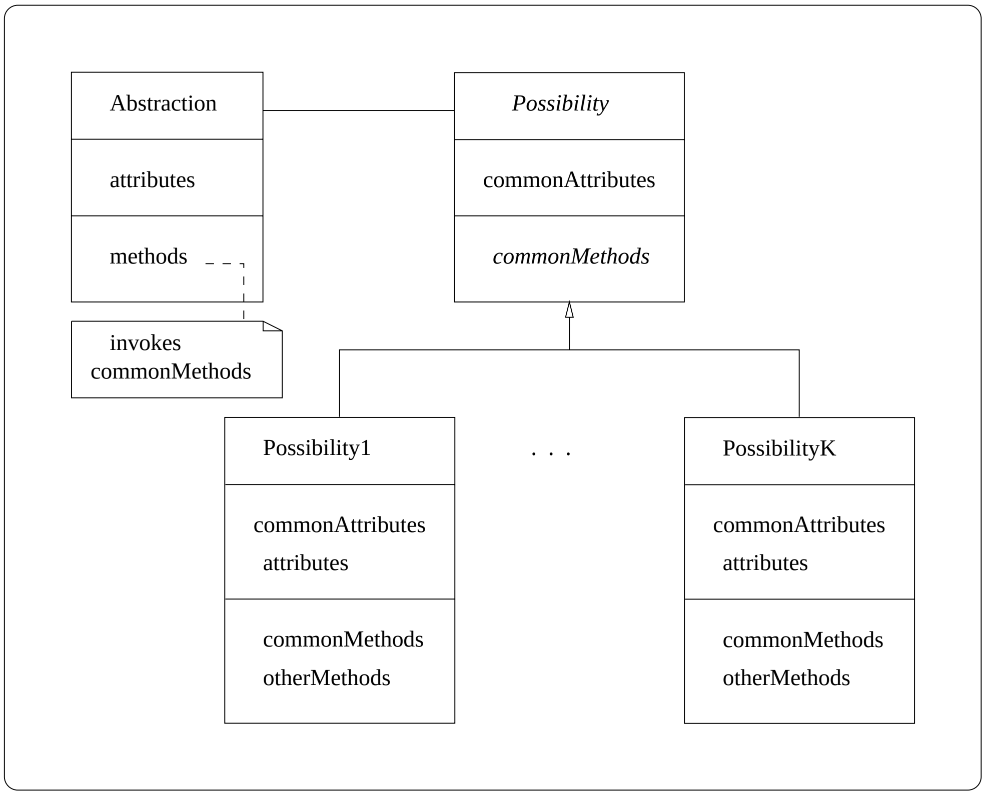


Figure 3.6: Class diagram of the bridge pattern.

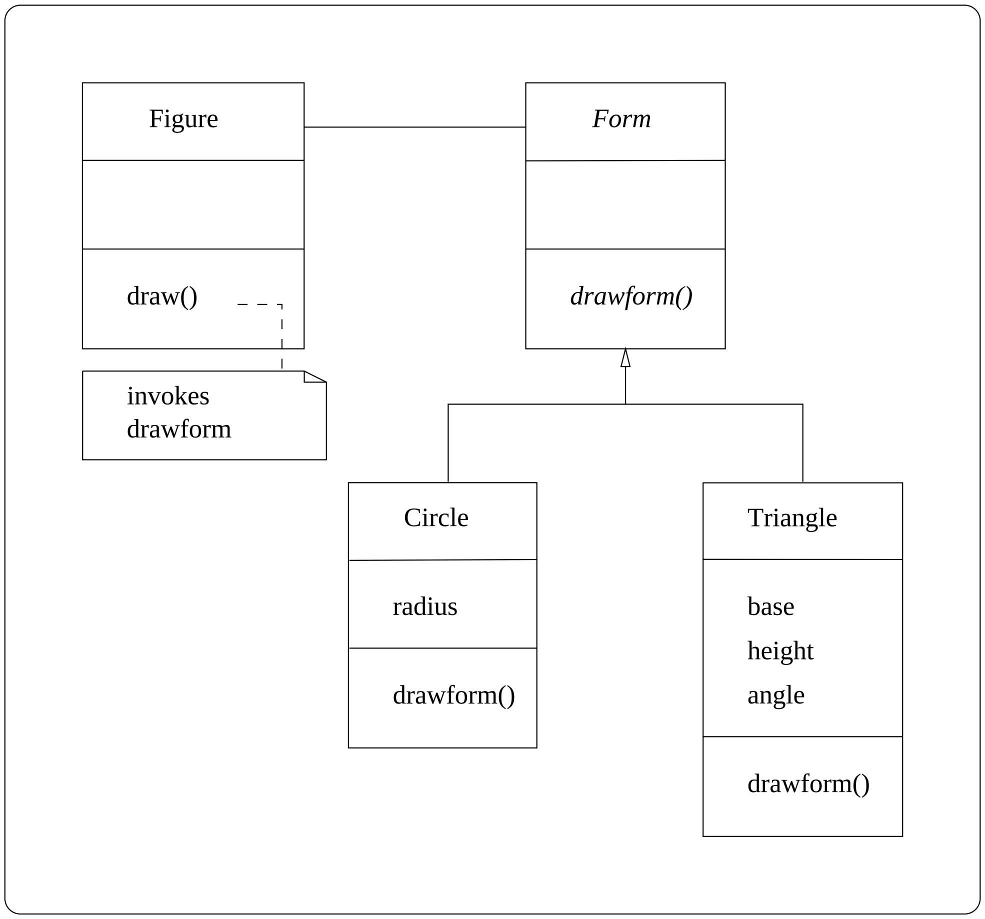


Figure 3.7: Class diagram of an application of the bridge pattern.

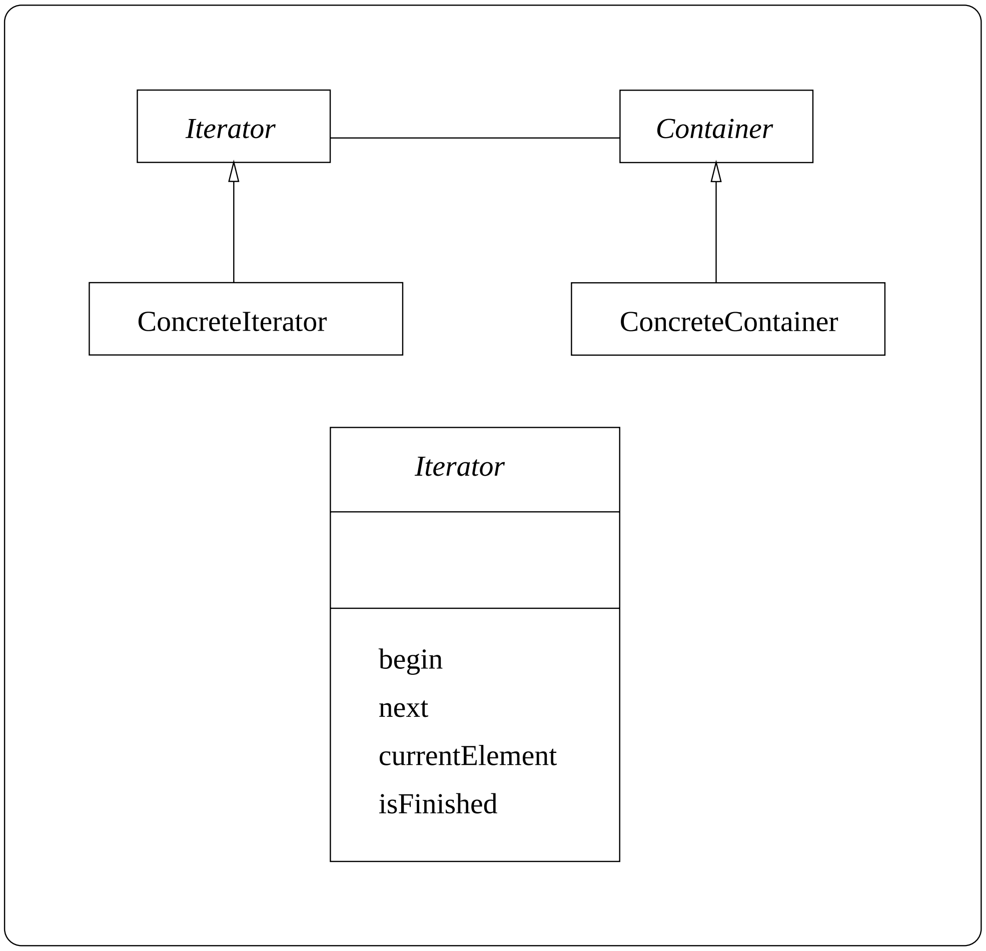


Figure 3.8: Class diagram of the iterator pattern.

matrix calculations can be implemented.

Simulation of Generic Classes

Generic classes are not supported by every object oriented language. In their absence, the developers may have to code, by hand, each of the different possible derived classes from the generic class. The number of classes that have to be written is linearly proportional to the number of different valid parameters of the generic class. Figure 3.9 presents a class diagram for a generic class `GenericMatrix` whose parameter is the class of the elements.

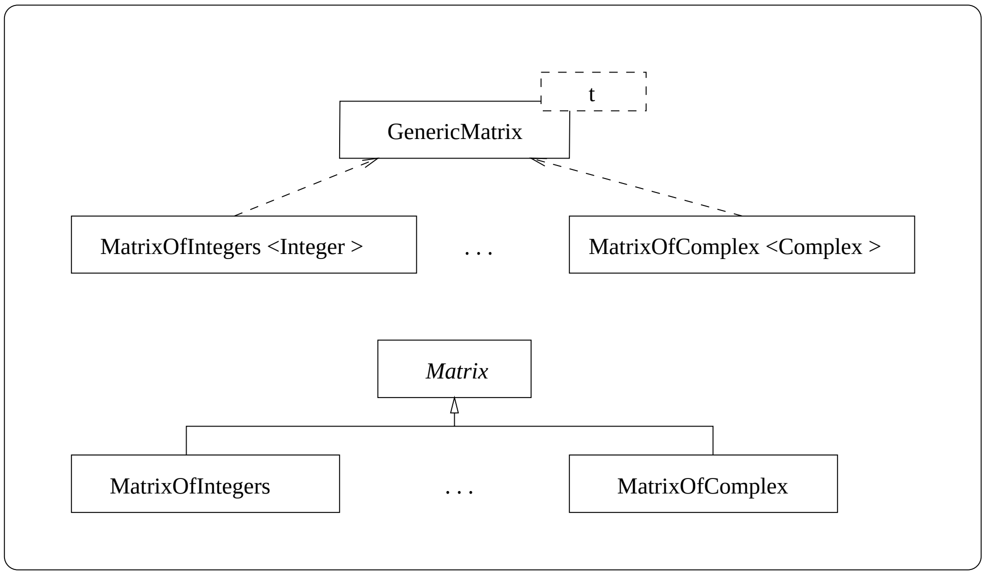


Figure 3.9: Class diagram emulating generic classes by hand code.

Alternatively, developers can simulate a generic class using a class with a polymorphic client relation. Each of the different valid parameters of generic class is made to inherit from a new abstract class. The class that simulates the generic class is a client of the new abstract class. Figure 3.10 presents the pertinent class diagram using the generic class `GenericMatrix`. Class `SimulatedGenericMatrix` simulates the class `GenericMatrix` by being a client of the abstract class `Number`.

`GenericMatrix` and `SimulatedGenericMatrix` class structures represent polymorphism. In the case of class `GenericMatrix`, the polymorphism is resolved at compile-time since its sub-classes resolve the polymorphism when choosing one class for the elements. In the other case, the polymorphism is resolved at run-time since every object of class `Number` or sub-classes might be assigned at any time. The generic class creates an object matrix that only can store one class of objects. However, the class `Matrix` creates an object matrix that can store any object of the hierarchy `Number` (bridge pattern). Nevertheless, it is also possible that only objects of one class are stored and thus simulate the generic class.

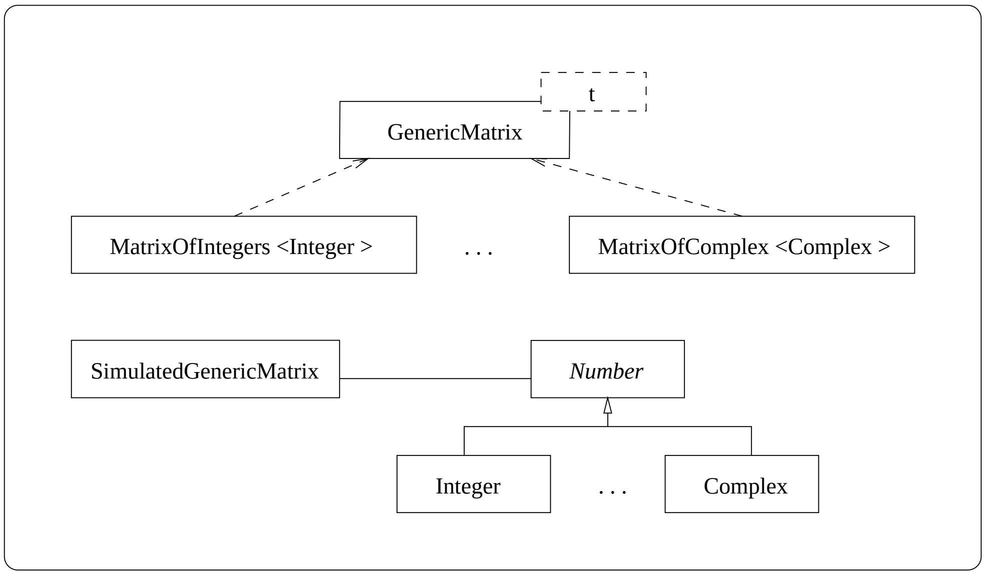


Figure 3.10: Class diagram of generic classes simulated by inheritance and client relation.

Developers simulating generic classes with polymorphism find, unless the compiler implements an aggressive algorithm, that generic classes are faster. In the case of generic classes, the dynamic binding mechanism is not necessary because the polymorphism has been resolved at compile-time. However, the emulation of generic classes needs the dynamic binding mechanism. In this case, an aggressive compiler would be able to resolve the polymorphism only if it can prove that only one class of objects is assigned.

3.2 Analysis and Design of OoLaLa

Object oriented analysis and design is the part of the software development process where an object oriented model of the problem to be solved is created. Key abstractions (classes) from the problem domain are identified and relations (client or inheritance) between these classes are proposed. The nature of this process is iterative and incremental; different models are created and evaluated against parts of the problem domain until the parts are properly described, and then a new iteration begins, including new parts of the problem domain.

This section is dedicated to a review of different designs of object oriented linear algebra libraries. In order to present clear diagrams and discussions, the following aspects have been omitted: the class of the elements of matrices; methods that create objects, and methods that query the state (attributes).

An initial step is carried out modelling matrices, matrix properties and storage formats simply including the access methods of matrices (Section 3.2.1). This initial step provides the basic design which is extended, firstly, to allow sections of matrices to be matrices and matrices formed by merging other matrices (Section 3.2.2). Secondly, the iterator pattern is modified for the purpose of traversing linear algebra matrices (Section 3.2.3). Finally, basic matrix operations and matrix equations solved with direct or iterative algorithms are given a representation (Section 3.2.4). At each stage, different solutions are proposed. These are used to classify some object oriented linear algebra libraries (see Table 3.1). When selecting a solution, two user groups are kept in mind: numerical linear algebra experts and non-experts. The obvious differences between these two groups force the library to be as simple as possible for non-expert users, but also to provide as many tuning details as possible for expert users. However, these tuning details do not reveal how they are implemented.

3.2.1 Initial Analysis

A *matrix* is a two-dimensional container of numbers. The *dimensions* of a matrix are the number of rows (`numRows`) and number of columns (`numColumns`). The basic operations are to obtain an *element* of the matrix, a_{ij}, and to *assign* a value to an element of the matrix, e.g. $a_{ij} \leftarrow 32$. An element is determined by its (unique) position; number of row i and column j. Given two integers i and j, they determine an element if both are greater or equal than 1 and if they are less or equal than `numRows` and `numColumns`, respectively. In other words, every matrix has two methods to access the elements: `assign` and `element`. The `element` method needs two integers, i and j, and returns the element in the i^{th} row and j^{th} column, whereas `assign` needs the same two integers and a number to assign to the element in the i^{th} row and j^{th} column. Figure 3.11 presents a `Matrix` class according to the above description.

Library	References
LAPACK++	[DPW93a], [DPW93b], [DPW96], [LAP]
SparseLib++ and IML++	[DLN+94], [PRL96], [DLPR96], [Spa], [IML]
Paladin	[GJ95], [GJP96]
JLAPACK	[BC98], [BC99], [JLA]
OwlPack	[BKP98], [BK99b], [BK99a], [Owl]
MTL and ITL	[SL98b], [SL98c], [SLL99], [SL98a], [SL99], [MTL], [ITL]
PMLP	[BBV+99], [BPB+99], [PML]
Diffpack	[BL97], [Dif]
ISIS++	[ACMW99], [ISI]
Sparspak++ or Sparspak90	[GL99]
Oblio and Spindle	[DKP99], [DKP98], [KP98]
JAMA	[JAMb]
Jampack	[Ste99], [Jama]
BPKIT	[CH96], [CH98], [BPK]

Table 3.1: Object oriented linear algebra libraries.

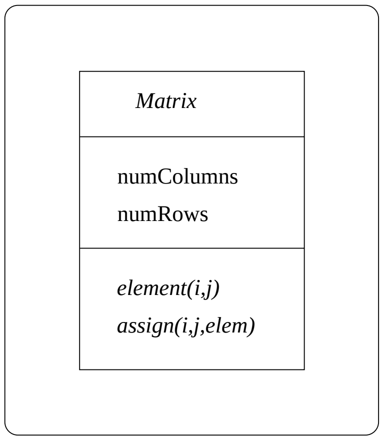


Figure 3.11: A simple `Matrix` class.

With this description of a matrix as a starting point, the discussion is organised around a set of different proposals. Each proposal differs in the organisation or relations between matrix, matrix properties and storage formats. For each proposal two class diagrams are presented. The first class diagram presents the general structure (generalised class diagram) without using real properties or storage formats. The second class diagram applies the generalised structure to dense, banded, symmetric, symmetric banded and symmetric positive definite matrix properties, and to dense and band storage formats (concrete class diagram).

Proposals

The first proposal, `Matrix` version 1 (see Figures 3.12 and 3.13), is based on the inheritance relation. The combinations of matrix properties and storage formats are considered to be sub-classes of `Matrix`. The class `Matrix` is on the first level of the inheritance hierarchy. On the second level, the `Matrix` class has been specialised by the matrix properties; a band matrix is always a matrix. The third level specialises the matrix properties by combining the properties of the second level and thus creating properties such as symmetric banded. The fourth level specialises the matrix properties by giving them a storage format. Only the fourth level classes are not abstract classes.

The second organisation, `Matrix` version 2 (see Figures 3.14 and 3.15), introduces the client relation between classes. A new abstract class called `Storage-Format` is created and every storage format inherits from it. The same two methods, `element` and `assign`, are included for the `StorageFormat` class, thereby creating a unified interface for all the storage formats. The class `Matrix` has a client relation with the class `StorageFormat`. The matrix properties classes inherit from the class `Matrix`, as in `Matrix` version 1, but they are not abstract classes any more.

The third organisation, `Matrix` version 3 (see Figures 3.16 and 3.17), introduces a new abstract class called `Property`. The matrix properties that can be represented in different storage formats inherit from `Property` while the other properties are attributes of `Property`. The class `Matrix` has a client relation with `Property`, which also has a client relation with `StorageFormat`.

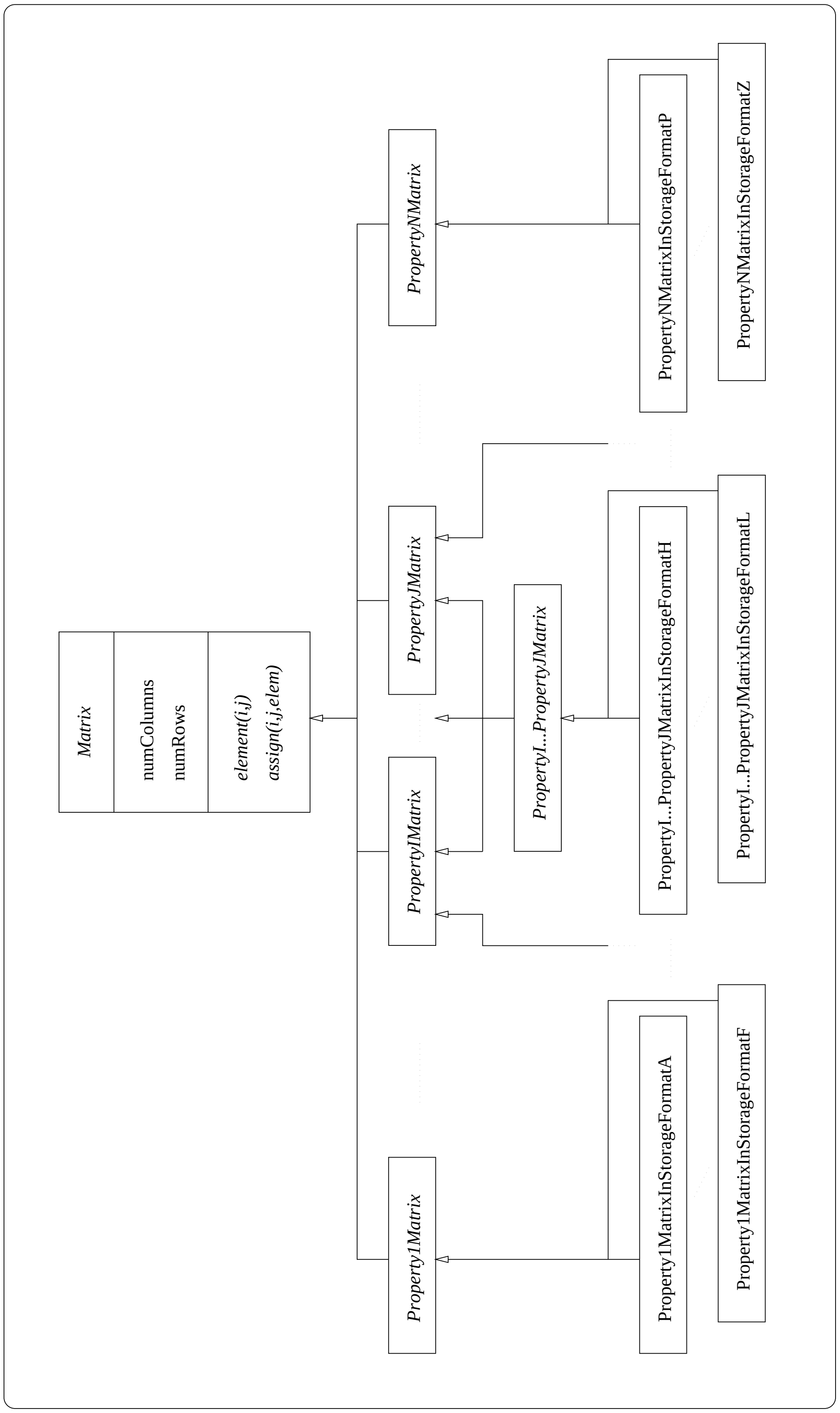

Matrix
numColumns
numRows
element(i,j)
assign(i,j,elem)
Property1Matrix
PropertyIMatrix
PropertyJMatrix
PropertyNMatrix
PropertyI...PropertyJMatrix
Property1MatrixInStorageFormatA
Property1MatrixInStorageFormatF
PropertyI...PropertyJMatrixInStorageFormatH
PropertyI...PropertyJMatrixInStorageFormatL
PropertyNMatrixInStorageFormatP
PropertyNMatrixInStorageFormatZ

Figure 3.12: Generalised class diagram of `Matrix` version 1.

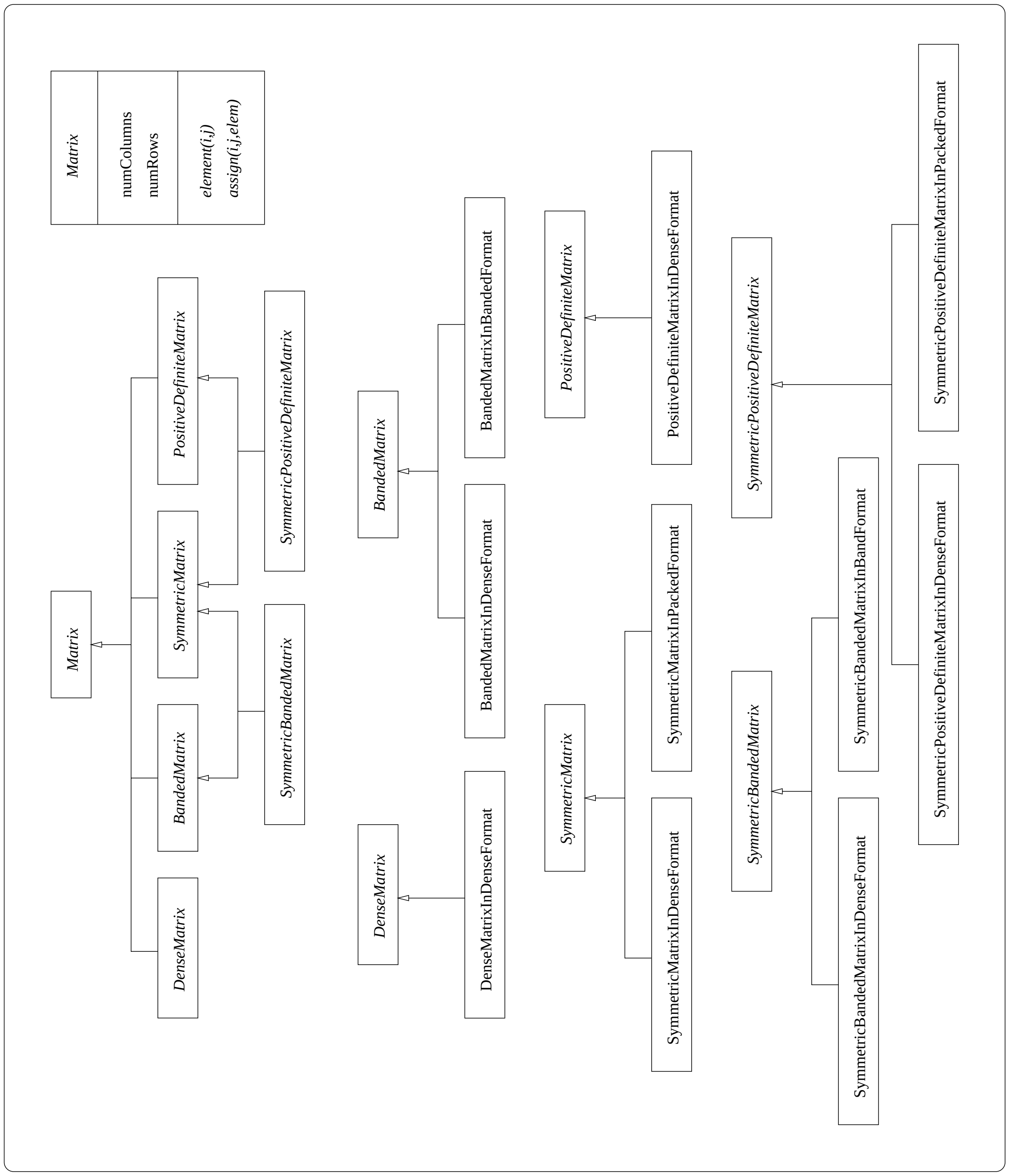

Matrix
numColumns
numRows
element(i,j)
assign(i,j,elem)
Matrix
DenseMatrix
BandedMatrix
SymmetricMatrix
PositiveDefiniteMatrix
SymmetricBandedMatrix
SymmetricPositiveDefiniteMatrix
DenseMatrix
DenseMatrixInDenseFormat
BandedMatrix
BandedMatrixInDenseFormat
BandedMatrixInBandedFormat
SymmetricMatrix
SymmetricMatrixInDenseFormat
SymmetricMatrixInPackedFormat
PositiveDefiniteMatrix
PositiveDefiniteMatrixInDenseFormat
SymmetricBandedMatrix
SymmetricBandedMatrixInDenseFormat
SymmetricBandedMatrixInBandFormat
SymmetricPositiveDefiniteMatrix
SymmetricPositiveDefiniteMatrixInDenseFormat
SymmetricPositiveDefiniteMatrixInPackedFormat

Figure 3.13: Concrete class diagram of `Matrix` version 1.

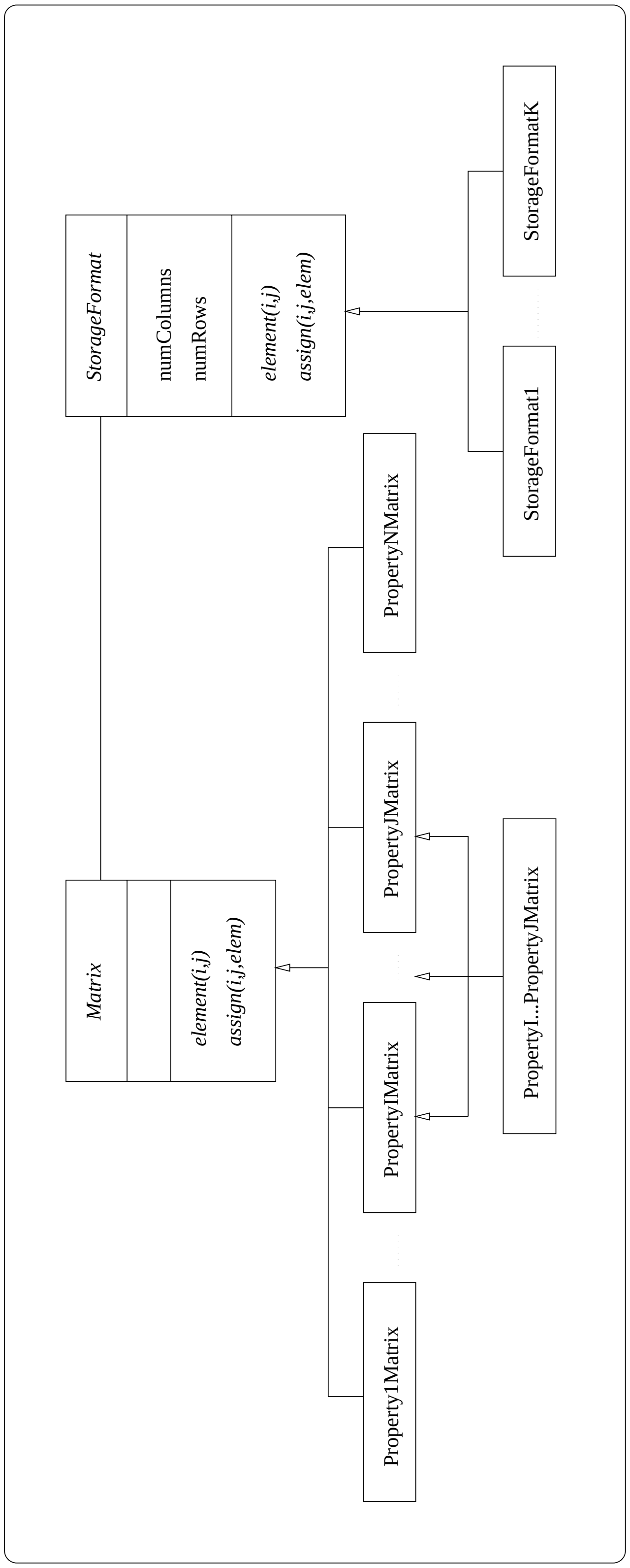


Figure 3.14: Generalised class diagram of `Matrix` version 2.

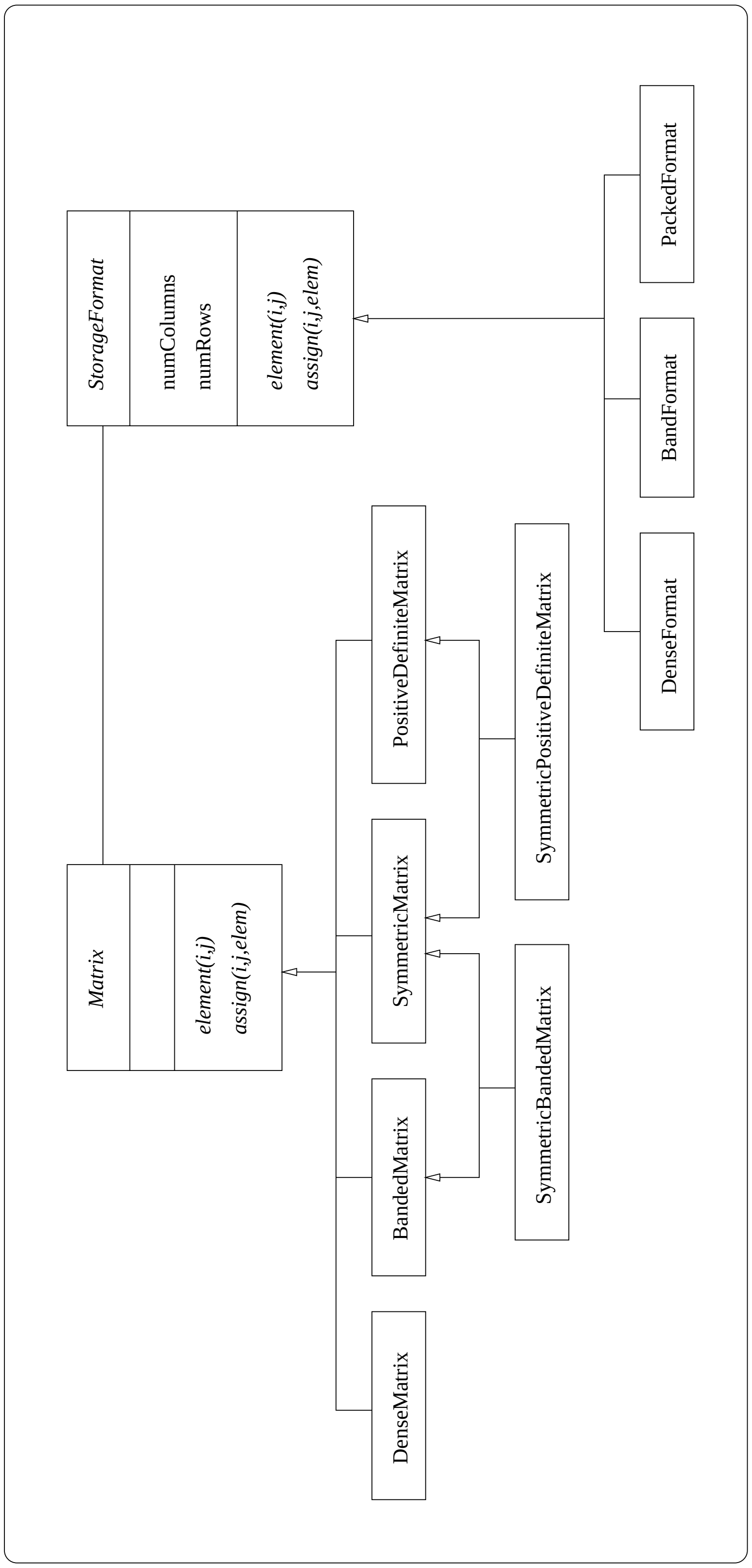


Figure 3.15: Concrete class diagram of `Matrix` version 2.

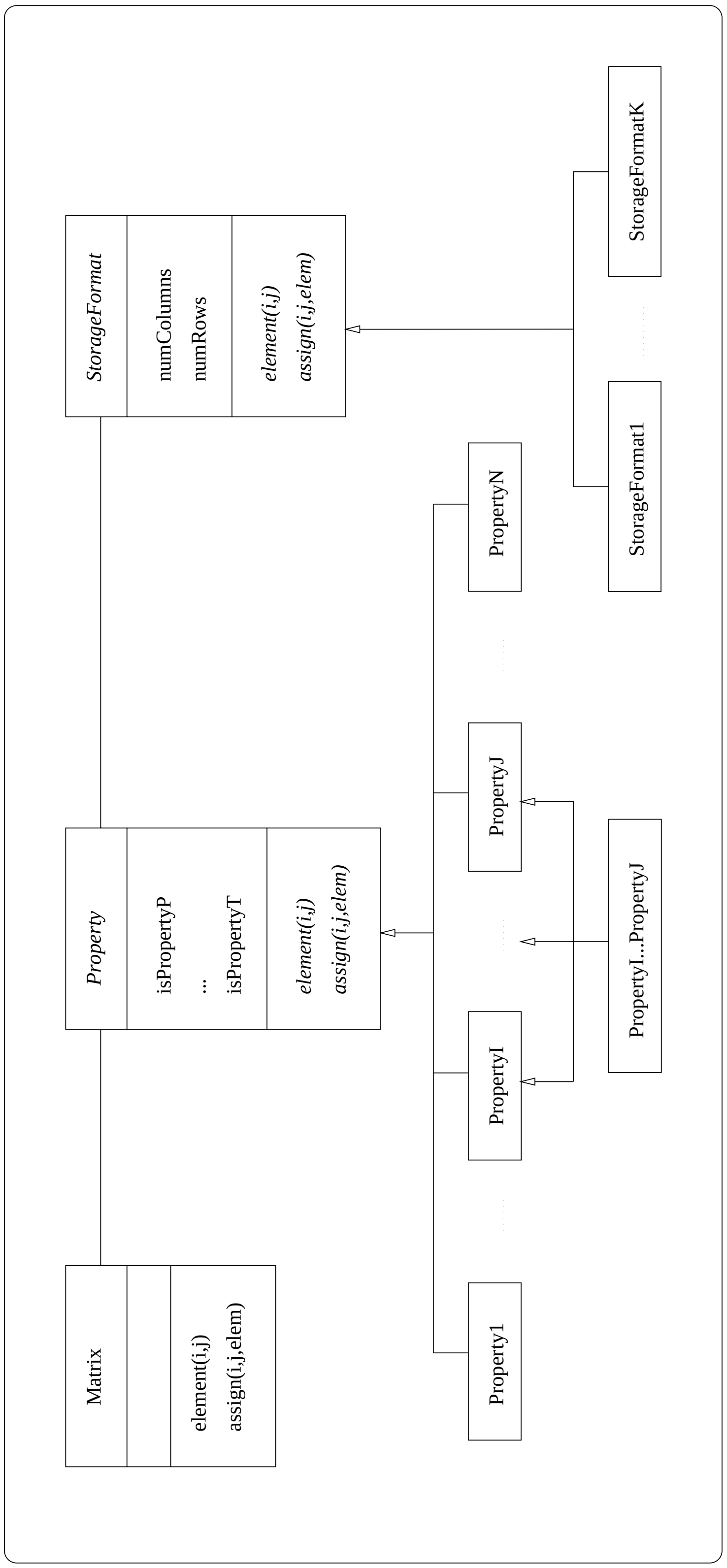


Figure 3.16: Generalised class diagram of `Matrix` version 3.

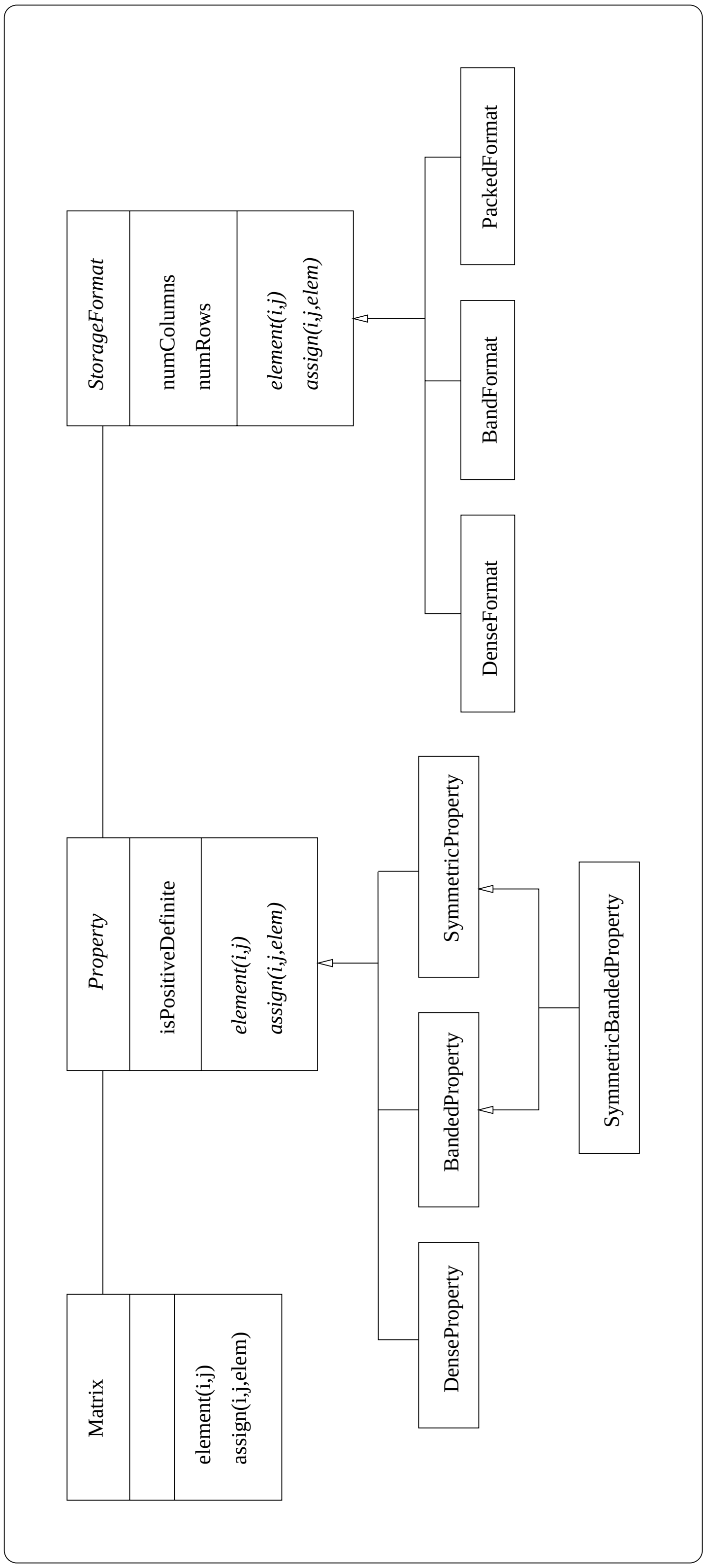


Figure 3.17: Concrete class diagram of `Matrix` version 3.

Discussion

In `Matrix` version 1, the classes at the bottom of the hierarchy can be seen as a possible combination of matrix properties and a storage format. Comparing these classes with the BLAS naming scheme, described in Section 2.4.1, for each two letters that represent matrix properties and a storage format (e.g. GE dense matrix in dense format or TP triangular matrix in pack format), a class is created.

LAPACK++, SparseLib++, Paladin, OwlPack, Diffpack, ISIS++, Spindle and Oblio, Jampack libraries (Table 3.1) are examples of `Matrix` version 1.

Since an object of any of the sub-classes of `Matrix` encapsulates the storage format, the number of rows and columns and the properties, a method `multiply`, with parameters of class `Matrix` can substitute for the BLAS subroutines `XGEMM`, `XGBMM`, etc. An implementation strategy for the method is to test the properties of the matrices and storage format and then decide which of the BLAS subroutines to call. The benefit for the user is that only one method, whenever possible, is offered for a matrix calculation. Section 3.2.4 returns to this point in more detail.

The benefit for the developer of the library is that a second implementation strategy is to use the unified access interface to every class in order to implement the methods. Hence, the number of implementations is reduced since the interface offers a way of accessing matrices that is independent of storage format. Figure 3.18 presents a naïve implementation of the method `element` for `DenseMatrixInDenseFormat`, `BandedMatrixInDenseFormat`, and `BandedMatrixInBandFormat`. Each implementation is adapted to the specific properties and storage format so that the correct element is returned. In a similar way, the method `assign` can be implemented and thus the unified access interface of every class is completed.

`Matrix` version 1 has a problem related to the number of classes that have to be implemented. For each matrix property, a matrix can be represented in many storage formats; therefore the number of required classes is of the order of the number of matrix properties multiplied by the number of storage formats.

`Matrix` version 2 uses the client relationship, or more precisely the bridge pattern, in order to reduce the number of classes of `Matrix` version 1. The class diagram can be read as "a matrix, with whatever properties, has a storage format". The storage format can be any of those in the hierarchy and can vary at run-time. The effect is that all the classes on the fourth level of the hierarchy of `Matrix` version 1 (Figure 3.12) are eliminated, and new ones encapsulating the storage format appear. The abstract class `StorageFormat` has the same two

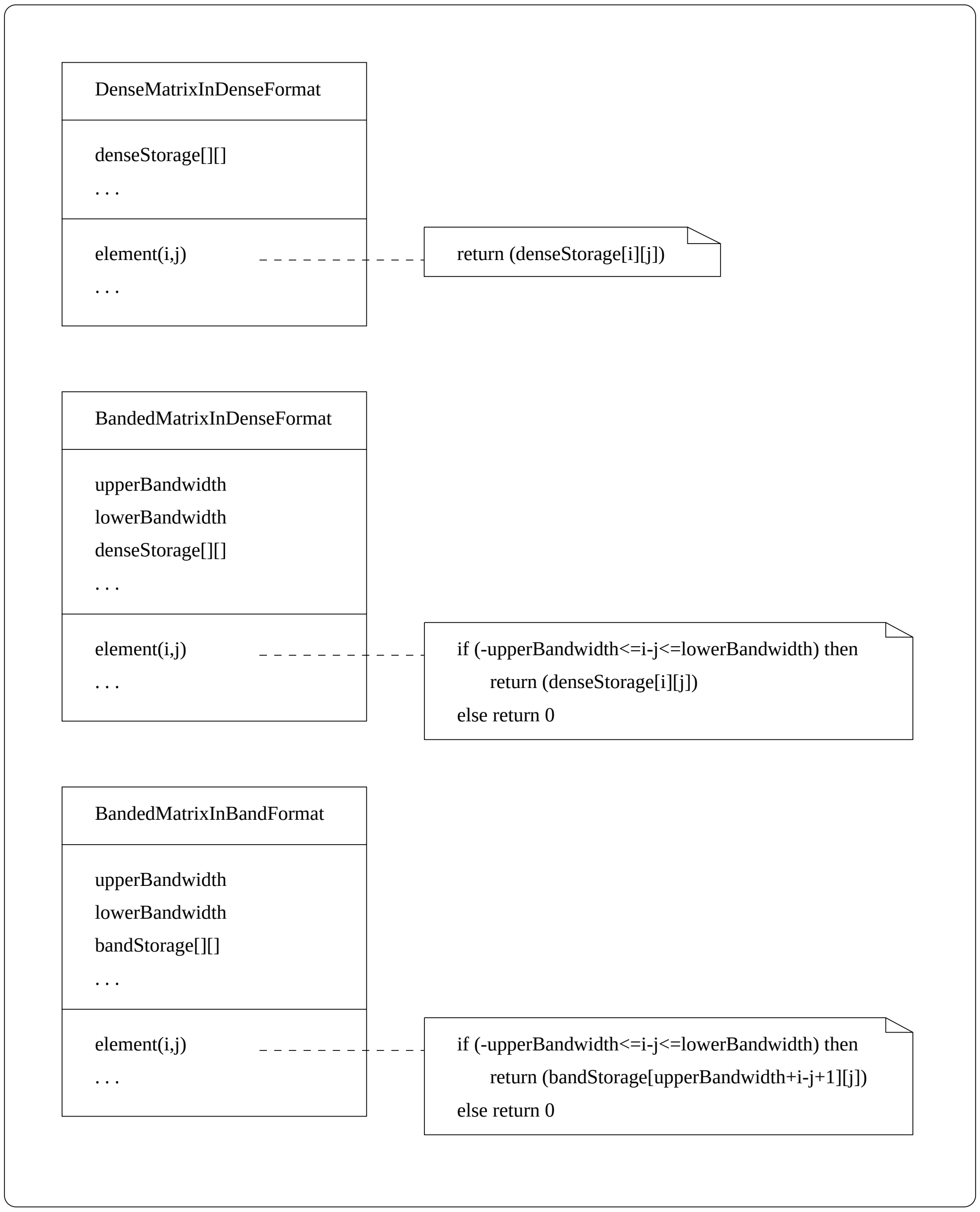


Figure 3.18: Implementation of the method `element` in `DenseMatrixInDense-Format`, `BandedMatrixInBandFormat` and `BandedMatrixInDenseFormat` classes – `Matrix` version 1.

methods as `Matrix`; `element` and `assign`. This creates a unified access interface and, thus, the sub-classes of `Matrix` do not need to know in which storage format they are represented in order to access the storage format. Figure 3.19 presents naïve implementations of the method `element` for the `DenseFormat`, `BandFormat`, `DenseMatrix` and `BandedMatrix` classes. These implementation only access to the storage format when the element cannot be implied from the matrix property. Since storage formats are created omitting those elements that can be implied from the matrix properties, these implementations of `element` are independent of the storage format.

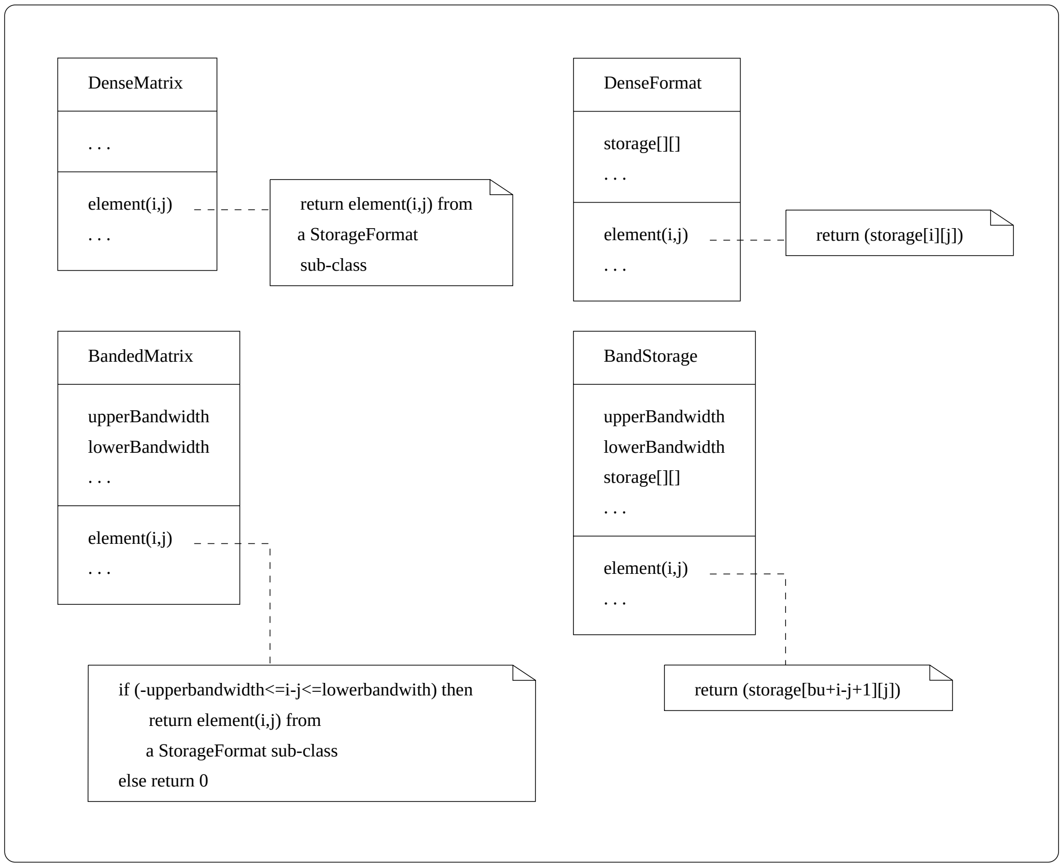


Figure 3.19: Naïve implementation of the method `element` in `DenseMatrix`, `BandedMatrix`, `DenseFormat` and `BandFormat` classes – `Matrix` version 2

Using generic programming, class `Matrix` can become a generic class with as its parameter a subclass of `StorageFormat`, and a similar model is obtained. PMLP and MTL (including other options as parameters, such as column-wise or row-wise arrays) libraries propose this variation to `Matrix` version 2.

PMLP and MTL face a common problem. Users do not need to know how a storage format is represented because it is encapsulated in the sub-classes of `StorageFormat`. The problem now is that users can create inadvisable combinations, such as a dense matrix stored in any sparse storage formats, or impossible combinations, such as a dense matrix stored in packed format.

Having identified this problem, and without a solution provided by any of the aforementioned libraries, an option is to hide the list of possible storage formats from users and rely on the library to decide which storage format to use. A second option is to allow users to define the storage format when an object of class `Matrix` is created and leave to the library to check the coherency between the matrix properties and the storage format specified.

The first option addresses the requirements of non-expert users of the library, who are relieved from having to know that a matrix can be represented in different storage formats and which one is advisable for their cases. However, this option is not satisfactory for expert users who wish to test different storage formats in order to determine the best for their needs (execution time, memory size, etc). The second option will satisfy the expert user as long as most storage formats are supported in the library. However, it has the disadvantage that adding a new storage format implies changes to the code that controls the coherence between storage format and matrix properties.

In the author's opinion, a combination of the two proposed options addresses the necessities of both user groups. The class organisation of `Matrix` version 2 does not need to be changed; the effect of accepting the two proposed options is reflected in the implementation of each subclass of `Matrix`.

The main change that the reader needs to understand is that the storage format, with any of the options for `Matrix` version 2, is a possible way to reduce execution time or memory requirements and not a restriction because the library does not support a combination of storage format for a determined matrix operation. This can be achieved because the implementations are able to use the unified access interface of `Matrix` inheritance hierarchy. Matrix calculations implemented using this interface are said to be implemented at *matrix abstraction level*. Nevertheless matrix operation can still be implemented at storage format abstraction level.

`Matrix` version 3 introduces the possibility that some matrix properties are not represented by classes. The positive definite property is a property that does not give chances to represent a matrix in different storage formats; it is just a factor that influences the implementation of a matrix calculation. Furthermore, it can be combined with any other property, but the combinations do not change the advisable storage formats of the original properties. The positive definite property is represented as an attribute of `Property`, and is thus inherited by every sub-class producing all the combinations. The rule to apply in general to determine if a matrix property is represented as a class or as an attribute is whether or not the property enables a matrix to be represented in different storage formats.

The second modification introduced in version 3 is that class `Matrix` becomes a client of `Property` from which the different matrix properties inherit. The class `Property` follows the same unified access interface of `Matrix` and no changes are needed for the sub-classes representing matrix properties. The interface of class `Matrix` includes methods, `setProperty`, so that the properties of a matrix can be declared. The following example of a linear algebra calculation presents a situation that `Matrix` version 2 and version 1 both fail to model, and which motivates version 3. Suppose that $B \leftarrow AB$ is the desired linear algebra calculation, where A is a dense matrix and B is a banded matrix. Mathematically speaking, this calculation is correct as long as both A and B are square matrices of order n. In other words, a matrix calculation is correct as long as it conforms to its definition and the properties of the matrices do not interfere. However, using `Matrix` version 1 or version 2 this matrix calculation is not accepted or is performed incorrectly. A sensible program which uses version 2 creates an object `a` of class `DenseMatrix` and an object `b` of class `BandedMatrix`. After executing a method that assigns to `b` the product, two different problems may arise. The first problem is that, if the object `b` had-an object of class `BandFormat`, an exception should be raised informing that a dense matrix cannot be stored efficiently in band format; the solution to this problem is to allow the library to change the storage format. The second problem arises from the change of properties of `b`; although the library can change the storage format it is impossible for it to change `b` to be an object of class `DenseMatrix`, because `DenseMatrix` is not a sub-class of `BandedMatrix`. Consequently, `b` is an object of the class `BandedMatrix` that should be an object of class `DenseMatrix`. Access to elements outside the bandwidths would return zero, although the result is known to be different.

The characteristic of this example is that, during run-time, an object that represents a matrix can vary its properties when it is operated upon. Using again the bridge pattern, the class `Matrix` has a client relation with `Property` under which the matrix property classes can be found. The class diagram of `Matrix` version 3 can be read as – "a given matrix can have different matrix properties and, in function of these properties, can be represented in different storage formats". The properties and storage formats are not fixed, this means that, when operated on, the properties and storage format of an object of class `Matrix` can be changed. The properties and storage format have to change in a way that the combination of both is advisable. The model created by `Matrix` version 3 allows to control these changes.

Through the different proposals, the functionality that the library provides has been increased. The interface has been adapted so that non-expert users can rely on the library to manage the properties and the storage formats for the matrices. The interface also offers expert users the possibility to create a matrix with a specific storage format supported. The library checks the coherency of the combinations determined by users and through calculations. The `StorageFormat` inheritance hierarchy unifies access to the different storage formats represented as its sub-classes. The `Property` inheritance hierarchy determines which elements are known owing to the properties. Otherwise, the storage format is accessed. The `Matrix` class is the user interface that encapsulates how properties and storage formats are implemented and enables the library to change them transparently for users.

None of the object oriented libraries reviewed in this section can be classified as `Matrix` version 3 (see Table 3.2); nor do they provide support for checking the coherency of matrix properties and storage formats. In order to provide this functionality, the library has to be able to propagate the properties of matrices from the operands to the results. A simple version of how to implement this new functionality is presented in Section 4.4. The `Matrix` version 3 and the functionality discussed above are the basis of a new library known as Object Oriented Linear Algebra LibrAry (OoLaLa). This design is refined in the next sections so as to enable users to use sections of matrices (rows, columns, sub-matrices) as if they were matrices, and to merge a set of matrices into one (Section 3.2.2). The second refinement includes the abstraction of *iterators* in order to traverse matrices and allowing a different abstraction level of implementation (Section 3.2.3). Finally, matrix calculations are included in the library (Section 3.2.4).

Library	Class Structure
LAPACK++	version 1
SparseLib++ and IML++	version 1
Paladin	version 1
JLAPACK	–
JAMA	–
Jampack	version 1
OwlPack	version 1
MTL and ITL	generic version 2
PMLP	generic version 2
Diffpack	version 1
ISIS++	version 1
Sparspak++ or Sparspak90	–
Oblio and Spindle	version 1
BPKIT	version 1

JLAPACK, JAMA and Sparspak++ or Sparspak90 do not offer enough information to be classified and "–" has been used to represent it.

Table 3.2: Class structure of various object oriented libraries.

3.2.2 Different Views of Matrices

Sometimes, applications need to work on sections of matrices as if they were matrices. For example, subroutines of LAPACK partition the matrices into blocks and work on these blocks independently. The transpose of a matrix can be treated as a section that is accessed by interchanging the indexes. An LU factorisation can store the L and U matrices in the matrix A, assuming that A is stored in dense format. This implementation of LU factorisation is called *in place factorisation*. The subsequent phase of solving the triangular systems with coefficient matrices U and L accesses only the upper triangular section or the lower triangular section. On other occasions, applications need to merge matrices to create a new matrix; for example, a block matrix can be created by merging its blocks.

Examples of matrix sections are a row or a column of an $m \times n$ matrix, which can be viewed as a row vector of size n or a column vector of size m, or three consecutive rows, which can be viewed as a $3 \times n$ matrix. A block lower triangular can be formed by merging its blocks and zero matrices. *View* is the term used to refer to either sections or merged matrices.

A simple solution for sections of matrices is to provide methods that create a new object of class `Matrix` with a corresponding new object of class `Property` and an advisable new object of class `StorageFormat`. The elements of the original matrix are copied into this new object of class `StorageFormat`. This solution does not modify the class structure of `Matrix` version 3. This is valid for applications that do not need to reflect in the original matrix the modifications made to the new section matrix. However, other applications need both matrices to reflect the modifications made to any of them. Hence, this solution becomes inefficient since applications need to copy back the elements in the original matrix (or section matrix) at the same time the section matrix is modified (or original matrix). A similar argument can be made for a matrix formed by merging other matrices.

In cases where the new section matrix and the original matrix need to keep a consistency (i.e. objects of class `Matrix` need to share an object of class `Storage-Format`) new classes have to be included. Among other solutions which share a common problem, Figure 3.20 presents a class diagram with one of these solutions. New abstract classes, called `Section` and `Merged`, are introduced. Their subclasses replicate those of the `Property` inheritance hierarchy and, thus, a view can also have properties. Since the position of an element of a view is based on the viewed matrices, this is reflected with `Section` and `Merged` being clients of `Matrix` and having as attributes information such as the row and column base or the list of merged matrices. The numbers in these client relations indicate that an object of class `Merged` merges at least two matrices. They also indicate that an object `Section` is a section of only one matrix. Their final indication is that an object of class `Matrix` can have as many sections, or be part of as many merged matrices, as wanted. The three hierarchies inherit from a new abstract class called `ViewOrProperty`. This solution is part of a family of solutions that has the major drawback that the `Property` inheritance hierarchy is triplicated.

Reviewing the role of the `Property` inheritance hierarchy, it is defined to determine whether an element is known independently of the way it is stored. In other words, the `Property` inheritance hierarchy is independent of elements being stored in sections of matrices or in sections of different matrices or in a storage format; its function is just to determine if an element is known. For example, when A is an upper triangular matrix the elements a_{ij} with $i > j$ are known to be zero elements independently of their storage format. Figure 3.21 presents a class diagram following this criterion. The classes `Section`, `StorageFormat` and `Merged`

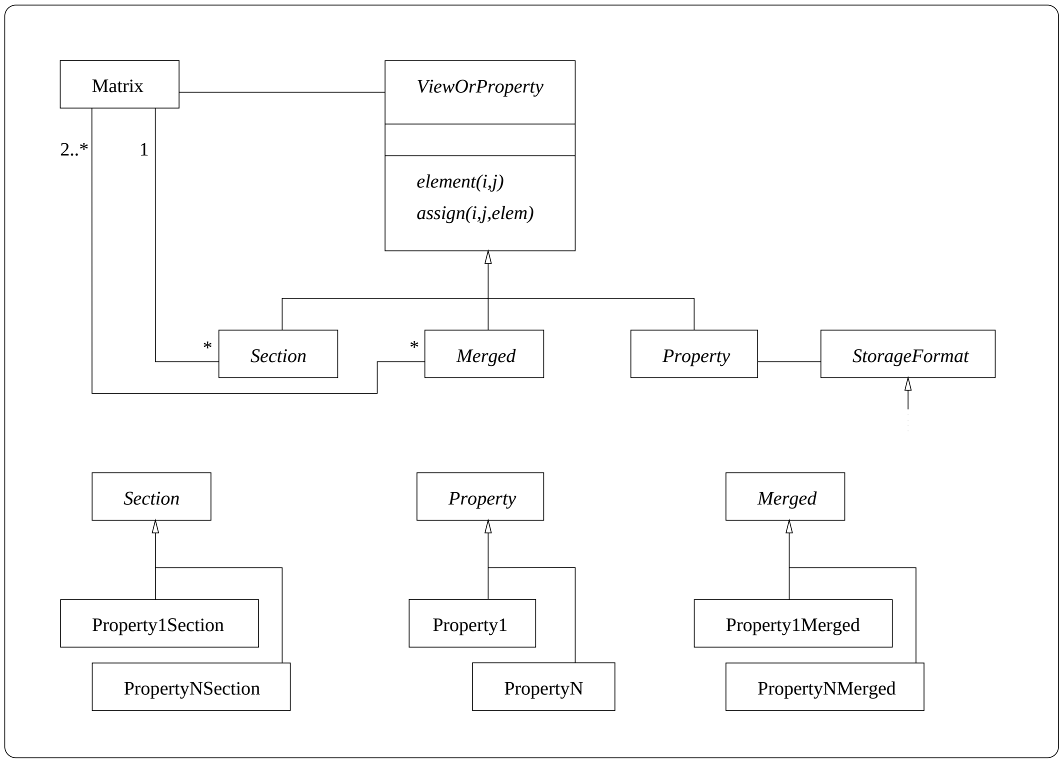


Figure 3.20: Class diagram of `Matrix` version 3 – first attempt to include different views of matrices.

are sub-classes of `ViewOrStorageFormat`. The class `Property` changes to have a client relation with `ViewOrStorageFormat` rather than with class `Storage-Format`. The classes `Section` and `Merged` keep their client relations with `Matrix`.

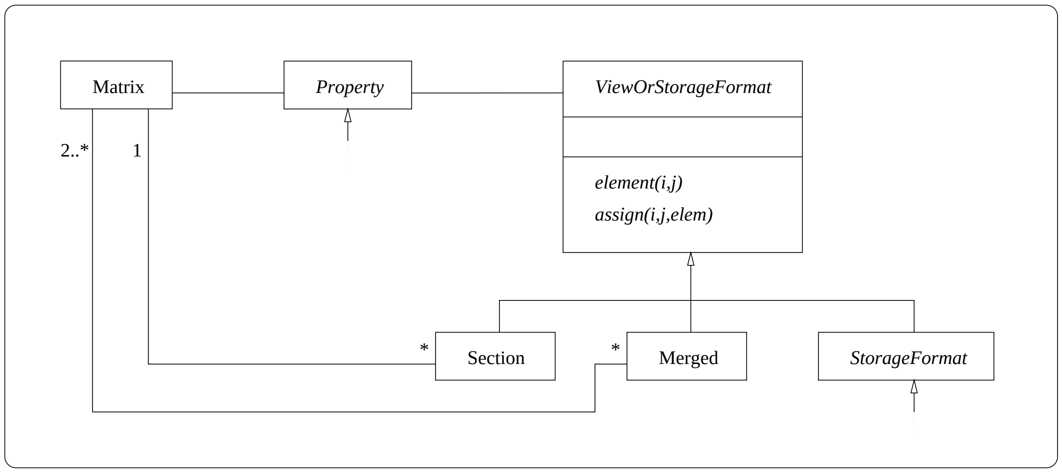


Figure 3.21: Class diagram of `Matrix` version 3 – second attempt to include different views of matrices.

Some current object oriented libraries provide views of matrices without replicating the elements. However, these libraries only allow the views to be dense matrices. Table 3.3 presents various object oriented libraries and how they support views of matrices.

3.2.3 Including Iterators

The iterator pattern, introduced in Section 3.1.4, is now extended to cover two-dimensional containers and, more specifically, linear algebra matrices. The iterator is redefined to traverse the elements of the matrices skipping those known to be zero.

Figure 3.22 presents the class `MatrixIterator`. The methods `setColumnWise` and `setRowWise` indicate how an object of class `MatrixIterator` traverses a matrix; column-wise or row-wise. The method `begin` places the object in the first column and first row. The method `beginAt` places the object in the position passed as a parameter. The class `MatrixIterator` considers a vector either as a column or as a row of the matrix and, thus, a matrix is traversed by passing through each vector of the matrix. The method `nextVector` increases an index of the current position and modifies the other index so that it points to the

Library	Sections	Merged Matrices
LAPACK++	(nce)	–
SparseLib++ and IML++	–	–
Paladin	(nce)	–
JLAPACK	(nce)	–
JAMA	(ce)	–
Jampack	(ce)	(ce)
OwlPack	–	–
MTL and ITL	(nce)	–
PMLP	–	–
Diffpack	–	–
ISIS++	–	–
Sparspak++ or Sparspak90	–	–
Oblio and Spindle	–	–
BPKIT	–	(nce)

The libraries that support views only allow them to be dense matrices or vectors. Only, BPKIT offers merged matrices whose blocks can be any kind of matrix. However, BPKIT's merged matrices are dense matrices. When a library does not support views it is represented as "–". When a library supports views copying elements it is represented as "(ce)". When a library supports views without copying elements it is represented as "(nce)".

Table 3.3: Support of views of matrices in various object oriented libraries.

first position. Which index is increased or modified depends on how the matrix is traversed. The method `isMatrixFinished` tests if there are more vectors to traverse in the matrix. A vector is traversed using the methods `nextElement` and `isVectorFinished`. The method `nextElement` searches for the next nonzero element within the vector while `isVectorFinished` tests if there are more nonzero elements in the vector. An element is accessed by the method `currentElement` that returns the current element and the row and column indexes.

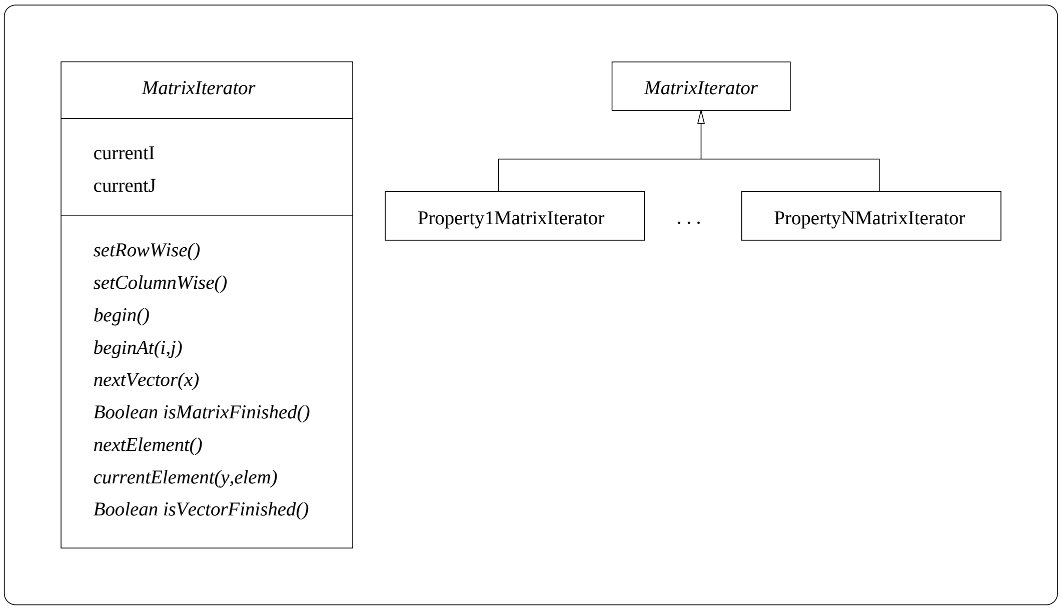


Figure 3.22: Class diagram of `MatrixIterator`.

Once the iterator pattern has been adapted to the requirements of linear algebra matrices, the next step is to integrate it with the class structure of OoLaLa. The `MatrixIterator` can be seen as an iterator for sequential access whereas `Matrix` and `Property` can be seen as iterators for direct access. The `MatrixIterator` interface can be integrated with the interface of `Matrix` and `Property` (see Figure 3.23) and, thus, the inheritance hierarchy of `Property` would not be replicated for `MatrixIterator`, if this class was included. The class structure is not modified with this integration.

Since an iterator traverses matrices skipping the zero elements, iterators constitute a new abstraction level at which matrix calculations can be implemented. Until now, an implementation of a matrix calculation was a set of nested loops defined in terms of explicit bounds which vary depending on the matrix properties.

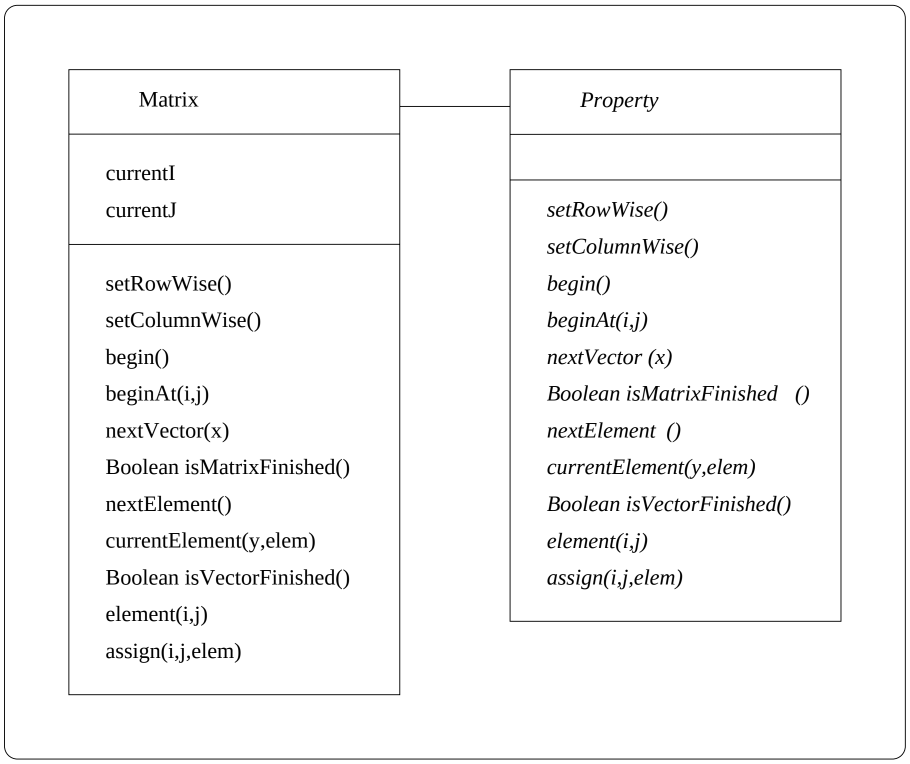


Figure 3.23: Class diagram of classes `Matrix` and `Property` including the methods of `MatrixIterator`.

In other words, an implementation of a matrix calculation traversed matrices by indicating explicitly which elements to access and by avoiding explicitly those elements that are known to be zero. On the other hand, an iterator expresses implicitly the elements to be accessed. An implementation of a matrix calculation using the interface of `MatrixIterator` implicitly changes the elements to be accessed when properties of the matrices are changed. This reduces the number of implementations of a matrix calculation.

MTL and PMLP use iterators, but with contradictory results. MTL has reported performance results on a Sun UltraSPARC for dense matrix-matrix multiplication and sparse matrix-vector multiplication, which are comparable to the highly optimised libraries ATLAS [WD98] and Sun Performance Library. On the other hand, PMLP declares [BBV+99]:

> "Iterators in PMLP provide a convenient means for users to iterate over elements in vectors and matrices, regardless of their internal data storage format. They also provide a storage format independent means for writing functions that access elements in objects using disparate storage formats. Since iterators are not an efficient mechanism for accessing elements in sparse matrices, much of the core functionality in PMLP is written using data access mechanisms specific to particular storage formats."

Note that no reference is given to justify their affirmation about the inefficiency of iterators or even a criterion to decide which functionality is written using what. The paper also does not reference MTL.

This thesis has introduced three implementation abstraction levels; storage format, matrix and iterator abstraction levels. The reader can either jump to Section 4.5 which compares the code of matrix calculations at the different abstraction levels for different matrix properties, or move to the next section that gives representations of matrix calculations in OoLaLa.

3.2.4 Including Matrix Calculations

The analysis has focused on modelling matrices, matrix properties and storage formats with respect to the access operations and matrix calculations. Access operations have been represented as methods (`assign` and `element`) of class `Matrix` while matrix calculations have been left without representations. The focus is now directed towards the representation of matrix calculations in OoLaLa. From the design of other object oriented libraries, matrix calculations can be represented as follows:

(a) as methods of class `Matrix`, or an equivalent name in each library,

(b) as classes, sometimes grouped into inheritance hierarchies, with the parameters transformed into attributes and the operation performed through a method `execute`, or

(c) as methods of a *utility class*, where related operations are grouped together.

The following description makes the above representations concrete using the addition of matrices as an example. The first representation includes a method called `add` in class `Matrix`. This method takes as a parameter an object of class `Matrix` and returns a new object of class `Matrix`. This new object is the addition of the parameter object and the object in which the method `add` has been invoked.

The second representation creates a class `Add`. This class has three attributes of class `Matrix`, and, when the method `execute` is invoked, two of these attributes are added to form the third one. By using classes, related operations can be grouped into inheritance hierarchies, such as `MatrixOperation`; every matrix operation inherits from `MatrixOperation`.

Finally, the third representation includes a method called `add` in a utility class. A utility class is a class that, despite being fully defined and implemented, cannot be instantiated. A utility class is a similar concept to a library of subroutines. In this case, the utility class could be named `MatrixOperation`. The method `add` is declared to have three parameters of class `Matrix`; two inputs and one output. Figure 3.24 presents graphically each of the described representations.

The implementations of matrix calculations have different features. These features divide the calculations into basic matrix operations and solvers of matrix equations (direct and iterative). The remainder of this section examines the features of the calculations in order to decide which representation should be used. The objective is to provide a simple and consistent interface. At the same time, the interface has to satisfy the requirements of both user groups; experts and non-experts.

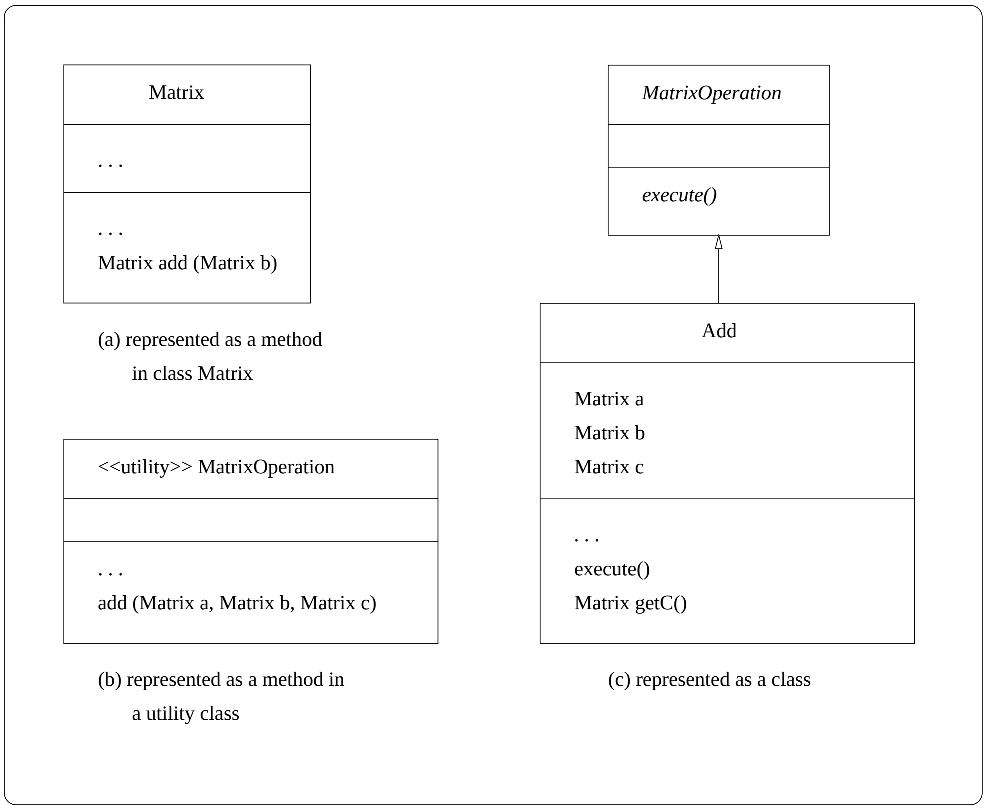

Matrix
. . .
. . .
Matrix add (Matrix b)
(a) represented as a method in class Matrix
<<utility>> MatrixOperation
. . .
add (Matrix a, Matrix b, Matrix c)
(b) represented as a method in a utility class
MatrixOperation
execute()
Add
Matrix a
Matrix b
Matrix c
. . .
execute()
Matrix getC()
(c) represented as a class

Figure 3.24: Different representations of matrix addition.

Basic Matrix Operations

The main feature of basic matrix operations is that, given the storage format and the matrix properties the implementation has already been decided. In other words, a set of "if-then" rules can be defined. These rules test the matrix properties and storage format of the operands and decide the corresponding implementation. The set of rules define a *rule based reasoning system*, or a *complete decision tree*.

Since an object of class `Matrix` encapsulates its matrix properties and its storage format, the reasoning system can be hidden behind the representation of each basic matrix operation. In this way, users have the impression that there is only one implementation of each basic matrix operation, although internally there may be multiple implementations. The interface is simplified in comparison with the BLAS because the number of visible subroutines for a matrix operation is reduced to only one visible representation. Moreover, the parameters of a basic matrix operation representation are no longer each detail of how the operands are stored, they are simply objects of class `Matrix`.

Due to the close relation between basic matrix operations and matrices, it is logical to represent them as methods of class `Matrix`. For example, the addition of matrices is an operation with domain and range matrices; it takes two matrices and produces a third matrix. On the other hand, to represent a basic matrix operation as a class is artificial, since such an operation is not an obvious abstraction from numerical linear algebra. Finally, a matrix operation can be represented as a method of a utility class. This utility class would resemble the BLAS and thereby users familiar with the BLAS would benefit. This benefit might be seen as an advantage over the first representation, but it is actually a signal expressing that this is not an object oriented form.

OoLaLa represents basic matrix operations as methods of class `Matrix`. In order to reduce execution time and memory requirements two syntaxes (or two methods) are discussed for a given basic matrix operation. For example, the matrix addition $C \leftarrow A+B$ can be represented by `c=a.add(b)` or `c.addInto(a,b)`, where `a`, `b` and `c` are objects of class `Matrix`. The method `Matrix add(Matrix b)` (`c=a.add(b)`) takes an object `b` as a parameter and performs the addition with the object in which `add` is invoked. This method returns a new object of class `Matrix`, i.e. also a new object `Property` and a new object `StorageFormat`. On the other hand, the method `void addInto(Matrix a, Matrix b)` performs the same operation, but does not return anything. This method performs the addition in the object in which the method has been invoked. This enables the method to create new objects only if it is “strictly necessary”. More details about how the storage formats and properties are managed in OoLaLa are described in Section 4.4.

Figure 3.25 presents the interface of class `Matrix` including a unary operation `norm1` ($||A||_1$) and two binary operations; `addInto` ($C \leftarrow A + B$) and `multiplyInto` ($C \leftarrow AB$). Table 3.4 offers a list of how matrix operations are represented in different object oriented linear algebra libraries.

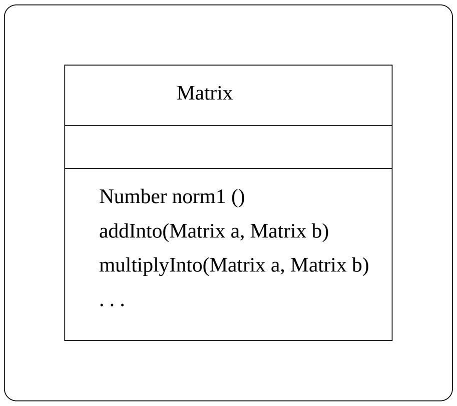


Figure 3.25: Class diagram of class `Matrix` including matrix operations as methods.

Solvers of Matrix Equations

In contrast with matrix operations, object oriented libraries disagree about how the operation of solving matrix equations should be represented. Some libraries represent these operations as methods (`solveLinearSystem`, `solveLeastSquares`, and `solveEigenproblem`) of class `Matrix` or as methods of a utility class. These methods have a parameter representing the solver as an object of class `LinearSystemSolver`, `LeastSquareSolver`, or `EigenProblemSolver`. Other libraries represent the matrix equation itself as a class (`LinearSystemEquation`, `LeastSquareEquation` or `EigenProblemEquation`) with attributes that are the matrices defining an equation, and the solver as another class (`LinearSystemSolver`, `LeastSquareSolver`, or `EigenProblemSolver`) with a client relation of class `LinearSystemEquation`, `LeastSquareEquation` or `EigenProblemEquation`. The

Library	Representation
LAPACK++	(a) and (b)
SparseLib++	(a) and (b)
IML++	(a)
Paladin	(a)
JLAPACK	(b)
JAMA	(a)
Jampack	(b) or (c)
OwlPack	(a)
MTL and ITL	(b)
PMLP	(a)
Diffpack	(a)
ISIS++	(a)
Sparspak++ or Sparspak90	–
Oblio and Spindle	–
BPKIT	(a)

Basic matrix operations represented as methods of a class `Matrix` are denoted with "(a)". Basic matrix operations represented as methods of a utility class are denoted with "(b)". Basic matrix operations represented as classes are denoted with "(c)". Basic matrix operations not supported by the library are denoted with "–". Note 1 – IML++ does not provide matrix operations, however it needs a library that provides them represented as (a). Note 2 – Jampack represents each matrix operation as a unique utility class and a method similar to `execute`. This method instead of using the attributes uses its parameters. Depends on the personal interpretation to decide between (b) or (c).

Table 3.4: Representation of basic matrix operations in various object oriented libraries.

operation of solving a matrix equation is represented by a method `solve` in the class representing the solvers. Finally, some other libraries have the same classes representing solvers but they do not have the classes representing the matrix equations.

Among these descriptions, the common point is that a solver is presented as a class. Each solver has different phases and for each phase different algorithms have been proposed by the numerical linear algebra community. The bridge pattern [1] can be applied again given the structure shown in Figure 3.26.

From an object oriented point of view, there is no argument against representing matrix equations as classes and the operation of solving a matrix equation as a method of these classes. Linear algebra defines matrix equations in terms of basic matrix operations. Hence, it is reasonable to represent them in a different way, as long as the model remains correct. However, from a consistency point of view, it can be argued that the operation of solving matrix equations should also be a method (`solveLinearSystem`, `solveLeastSquares`, and `solveEigenproblem`) of class `Matrix`. In order to keep the interface simple for non-expert users, these methods would have a solver as a parameter only if it is necessary. The solvers would be represented as a class inheriting from `MatrixEquationSolver`. OoLaLa represents the operation of solving a matrix equation in this way. Table 3.5 presents the representation of matrix equations and the operation of solving them in various object oriented libraries. Table 3.6 presents the matrix equations supported by these object oriented libraries.

Once it has been decided how the operation of solving a matrix equation is represented, the next requirement is to clarify when it is necessary to include a solver as a parameter, and to model the different kind of solvers: direct and iterative.

Direct Solvers of Matrix Equations

Direct solvers have different phases and characteristics depending on the properties of the coefficient matrix. In this discussion, structured matrices (dense, banded, block banded, block triangular) are distinct from sparse matrices.

A direct solver of a matrix equation with a structured coefficient matrix is composed of two phases. The first phase performs a factorisation of the coefficient

[1] The bridge pattern when applied to classes representing algorithms is known as the *strategy pattern*([GHJV95] pages 315–324).

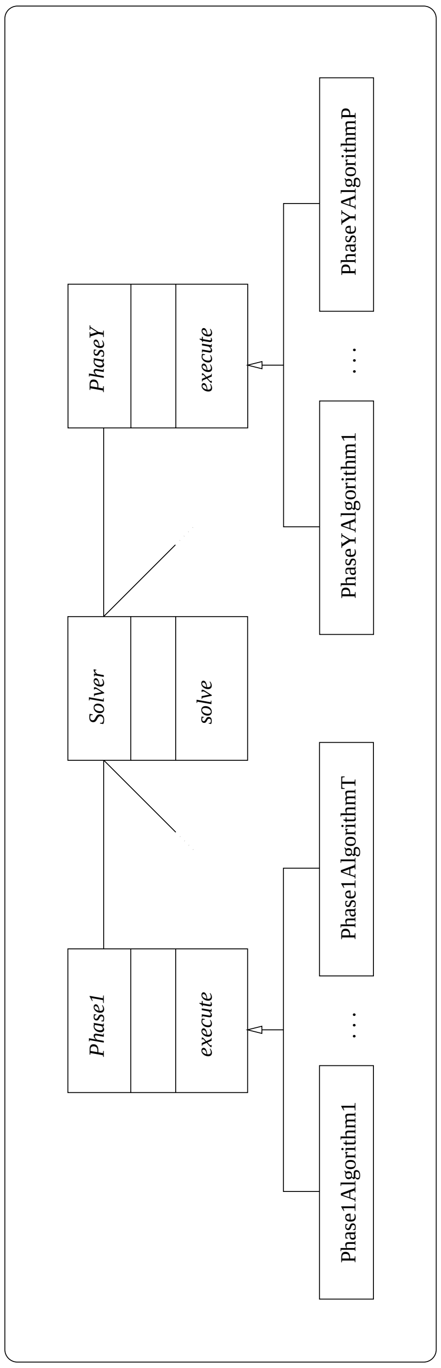


Figure 3.26: Class diagram of general `Solver` of matrix equations.

Library	Operation	Matrix Equation
LAPACK++	method in utility class	parameters of the method
SparseLib++ and IML++	method in utility class	parameters of the method
Paladin	method in `Matrix`	parameters of the method
JLAPACK	method in a utility class	parameters of the method
OwlPack	method in `Matrix`	parameters of the method
MTL and ITL	method in a utility class	parameters of the method
PMLP	method in `Solver`	attributes of `Solver`
Diffpack	method in `MatrixEquation`	class `MatrixEquation`
ISIS++	method in `MatrixEquation`	class `MatrixEquation`
Sparspak++ or Sparspak90	method in `Solver`	class `MatrixEquation`
Oblio and Spindle	method in `Solver`	attributes of `Solver`
JAMA	method in `Matrix` and `Solver`	parameters of the method or attributes of `Solver`
Jampack	method in utility class	parameters of the method
BPKIT	–	–

BPKIT provides block preconditioners and an interface to be used by iterative algorithms. However, BPKIT does not report how the iterative algorithms are represented.

Table 3.5: Representation of matrix equations and the operation of solving them in various object oriented libraries.

Library	Direct Solvers		Iterative Solvers
	Structured Matrix	Sparse Matrix	
LAPACK++	(a), (b) and (c)	–	–
SparseLib++ and IML++	–	–	(a)
Paladin	(a)	–	–
JLAPACK	(a)	–	–
OwlPack	–	–	–
MTL and ITL	(a)	–	(a)
PMLP	–	–	(a)
Diffpack	–	–	(a)
ISIS++	–	–	(a)
Sparspak++ or Sparspak90	–	(a)	–
Oblio and Spindle	–	(a)	–
JAMA	(a), (b) and (c)	–	–
Jampack	(a), (b) and (c)	–	–
BPKIT	–	–	see below

Systems of linear equations are represented as “(a)”. Least square problems are represented as “(b)”. Eigenproblems are represented as “(c)”. Kinds of matrix equations that are not supported by the library are denoted by “–”. BPKIT provides block preconditioners and an interfaces to be used by iterative algorithms. However, BPKIT does not report what iterative solvers are supported.

Table 3.6: Solvers of matrix equations provided by various object oriented libraries.

matrix, unless it is trivial and efficient to solve (e.g. diagonal matrix or triangular matrix). The second phase solves the matrix equation using the factorisation. According to the properties of the coefficient matrix and its storage format a factorisation and its specialised implementation can be selected. In other words, a set of "if-then" rules can be defined. These rules test the matrix properties and storage format of the operands and determine the corresponding implementation. This set of rules define another *rule based reasoning system.*

As with matrix operations, behind the methods `solveLinearSystem`, `solve-LeastSquares`, and `solveEigenproblem` the existence of different factorisations and their specialised algorithms can be encapsulated. In this way, users have the impression that there is only one implementation, although internally there are multiple implementations.

In general, the factorisation phase can be characterised as pivoting or no-pivoting. This characteristic distinguishes between a factorisation that needs to check the stability or not. Hence, using method overloading, a method with different parameters but same name (`solveLinearSystem`, `solveLeastSquares`, and `solveEigenproblem`) is included. The parameters are the same, except for an object of class `MatrixEquationSolver` that will indicate the characteristic of pivoting or no-pivoting. Table 3.6 presents various object oriented libraries that provide direct solvers for structured matrix equations.

A direct solver of a linear system of equations with a sparse coefficient matrix has three different phases. The first phase produces a new ordering of the coefficient matrix in order to conserve the sparsity. The second phase factorises the re-ordered matrix, and then, the third phase solves the linear system.

The ordering phase can take into account the numerical values of the elements of a matrix and simulate a factorisation. The ordering algorithms that take into account the numerical values are called numerical ordering. Other ordering algorithms that take into account structure but not specific numerical values are called symbolic ordering. The factorisation phase after a numerical ordering does not perform pivoting since it has already been calculated. This factorisation phase knows exactly the fill-in elements and, therefore, the factorisation phase can use a static storage format. However, the factorisation phase after a symbolic ordering needs to perform pivoting, possibly creating an unknown number of new fill-in elements, and therefore a dynamic data structure is necessary.

Table 3.6 presents Spindle and Oblio, and Sparspak++ or Sparspak90 as object oriented libraries that provide direct solvers for sparse systems of linear equations. Spindle and Oblio are two complementary libraries as Spindle provides ordering algorithms (minimum degree algorithms) and Oblio provides factorisations and for symmetric matrices. Sparspak++ and Sparspak90 are object oriented wrappers, C++ and Fortran 90 respectively, of the Sparspak library ([GL79], [GL81]).

Figures 3.27, 3.28, 3.29 and 3.30 present the classes `LinearSystemDirectSolver`, `GeneralFactorisation`, `KindOfPhase` and `Ordering`. `LinearSystemDirectSolver` has two sub-classes, `LinearSystemDirectSolverStructuredMatrix` and `LinearSystemDirectSolverSparseMatrix`, since the phases of solving a linear system are different for structured matrices and sparse matrices. `LinearSystemDirectSolverStructuredMatrix` is client of class `Factorisation` which represents the phases of solving a linear system with a structured matrix. `LinearSystemDirectSolverSparseMatrix` is a client of class `KindOfPhase`. `KindOfPhase` distinguishes between numerical ordering and factorisation represented as its sub-class `NumericalOrderingAndFactorisation`, and symbolic ordering and factorisation represented as `SymbolicOrderingAndFactorisation`. Since there is a dependence between the ordering phase and the factorisation, based on how the ordering is represented, `NumericalOrderingAndFactorisation` and `SymbolicOrderingAndFactorisation` are further specialised. Each of these classes is a client of two classes that represent the factorisation of a sparse matrix and the ordering. Class `Ordering` is specialised into `SymbolicOrdering` and `NumericalOrdering` and then further to take account of the data structure that represents the ordering. Class `GeneralFactorisation` is specialised into `Factorisation` and `SparseMatrixFactorisation`. `SparseMatrixFactorisation` is specialised for the structure in which the ordering is represented. Class `GeneralFactorisation` has as an attribute a boolean flag which indicates if pivoting is to be performed.

Iterative Solvers of Matrix Equations

An iterative solver of matrix equations comprises two phases that are repeatedly executed. The first phase is the algorithm itself, while the second phase is a termination test. The first phase usually requires preconditioning matrices. These matrices are created from the coefficient matrix in an attempt to make the algorithm converge in fewer iterations.

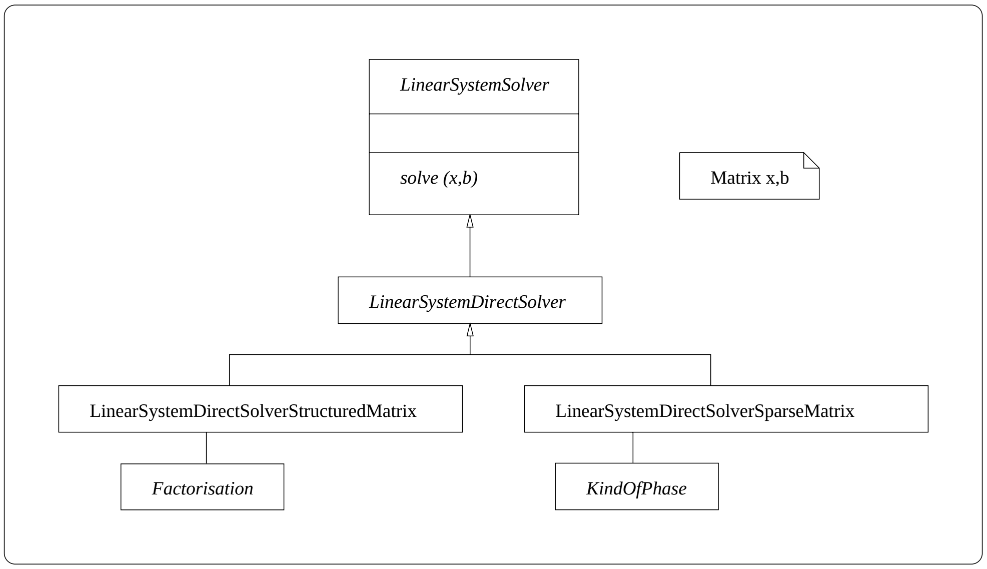


Figure 3.27: Class diagram of class `LinearSystemSolver` for direct solvers.

Some iterative algorithms are known not to converge for certain matrix properties. The best combination of a preconditioner and an iterative algorithm cannot be chosen, practically, only given the properties of the coefficient matrix. Users need to be able to select the iterative algorithm, the preconditioner, and the termination test to be used.

Figure 3.31 presents class `LinearSystemIterativeSolver`. A specific iterative algorithm is represented as a class inheriting from `LinearSystemIterativeSolver`. This class is a client of class `TerminationTest`. A termination test algorithm is represented as a class that inherits from `TerminationTest`. A method `test` that returns a `Boolean` is included in `TerminationTest`. The `create` method of sub-class of `LinearSystemIterativeSolver` takes as parameter the matrix defining the linear system of equations. When the algorithm can be preconditioned another method with the same name `create` but with other parameters the preconditioning matrices is included in the class.

A preconditioning matrix is the output of an operation that takes an input matrix and returns another matrix. Hence, a preconditioner operation is represented as a method in class `Matrix` having as parameter an object of class `Preconditioner`. Each kind of preconditioner operation is represented as a class inheriting from `Preconditioner`.

Table 3.6 presents various object oriented libraries that provide iterative solvers

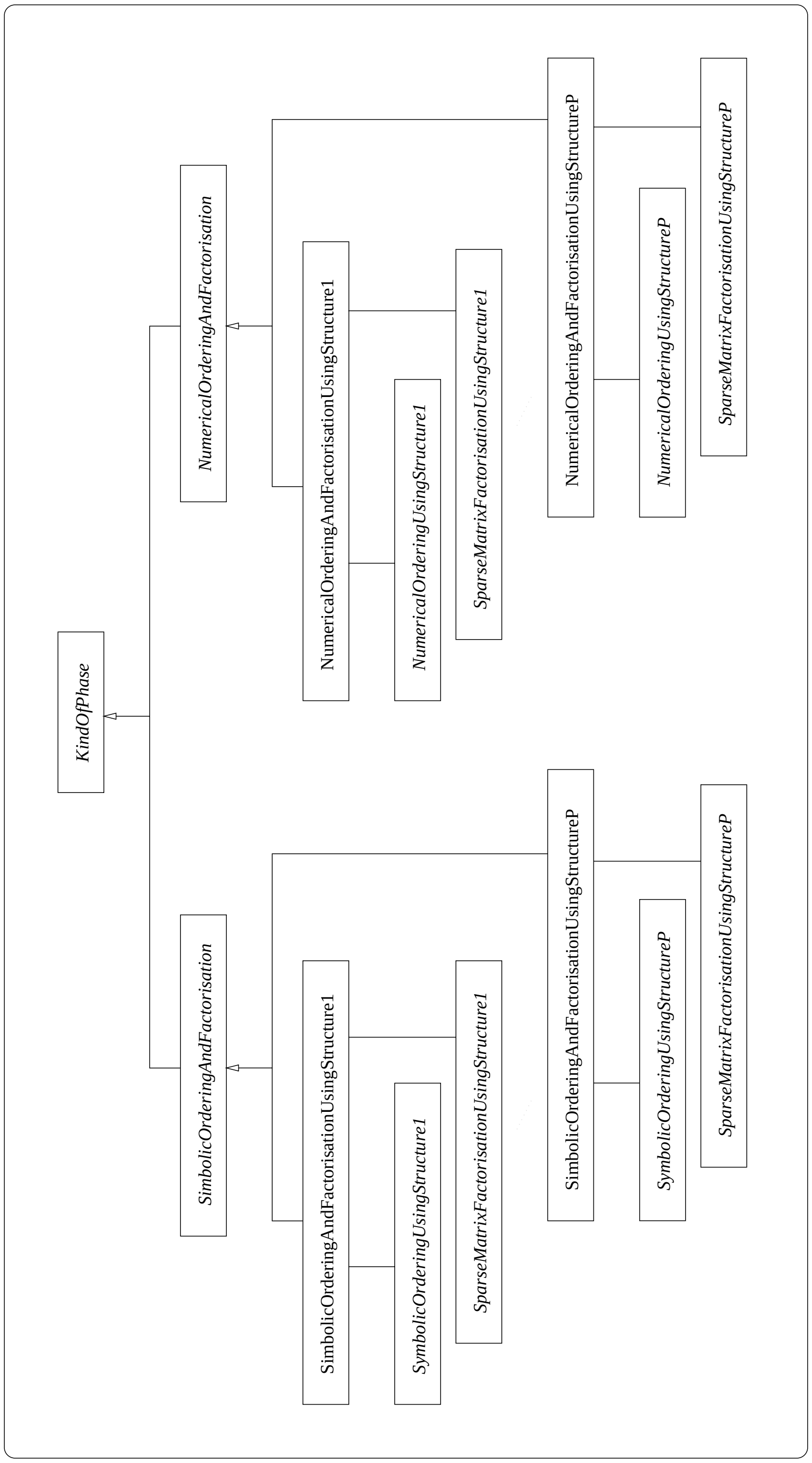

KindOfPhase
NumericalOrderingAndFactorisation
SimbolicOrderingAndFactorisation
NumericalOrderingAndFactorisationUsingStructure1
NumericalOrderingUsingStructure1
SparseMatrixFactorisationUsingStructure1
NumericalOrderingAndFactorisationUsingStructureP
NumericalOrderingUsingStructureP
SparseMatrixFactorisationUsingStructureP
SimbolicOrderingAndFactorisationUsingStructure1
SymbolicOrderingUsingStructure1
SparseMatrixFactorisationUsingStructure1
SimbolicOrderingAndFactorisationUsingStructureP
SymbolicOrderingUsingStructureP
SparseMatrixFactorisationUsingStructureP

Figure 3.28: Class diagram of class `KindOfPhase` for direct solvers.

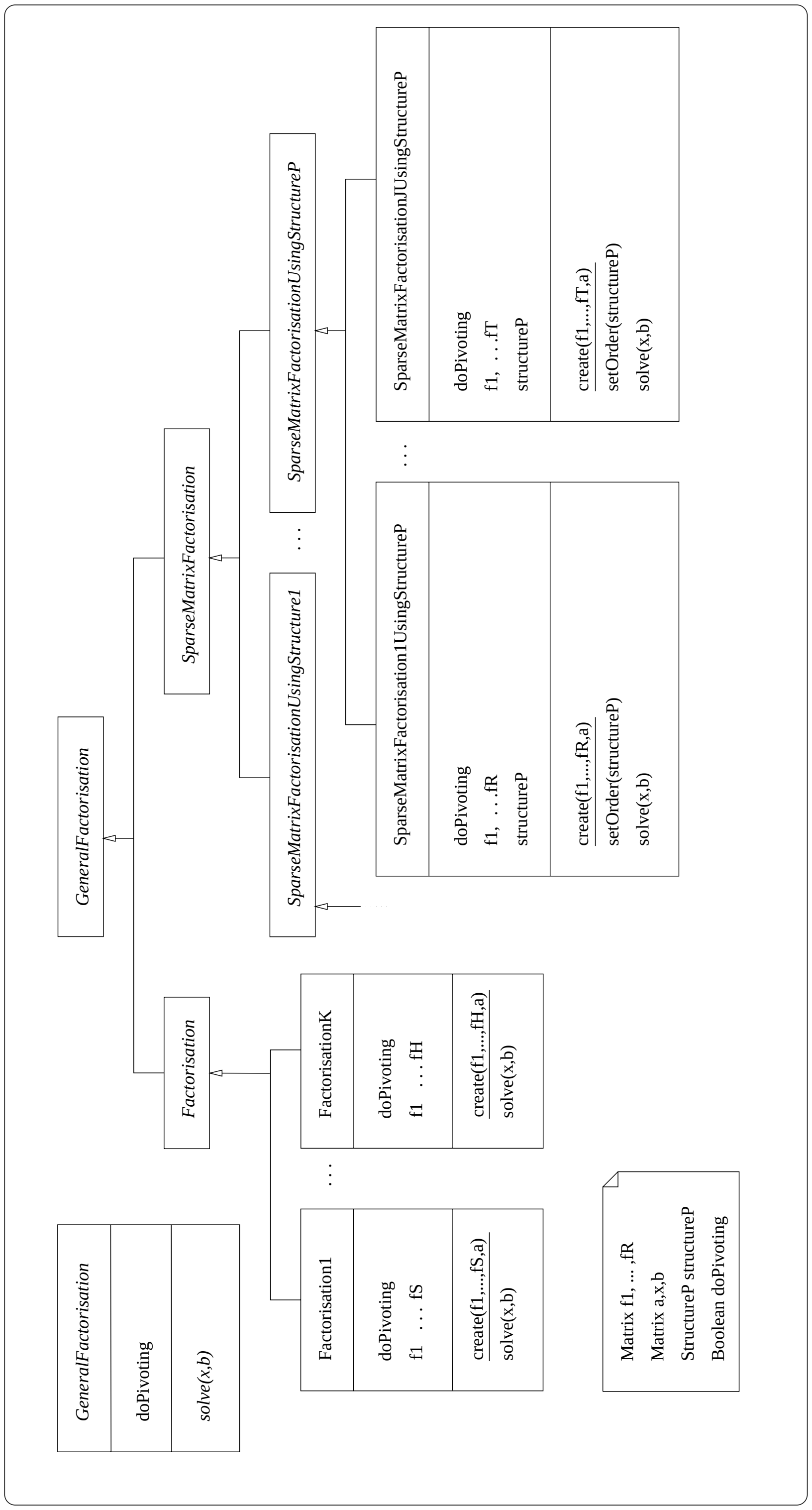


Figure 3.29: Class diagram of class `GeneralFactorisation` for direct solvers.

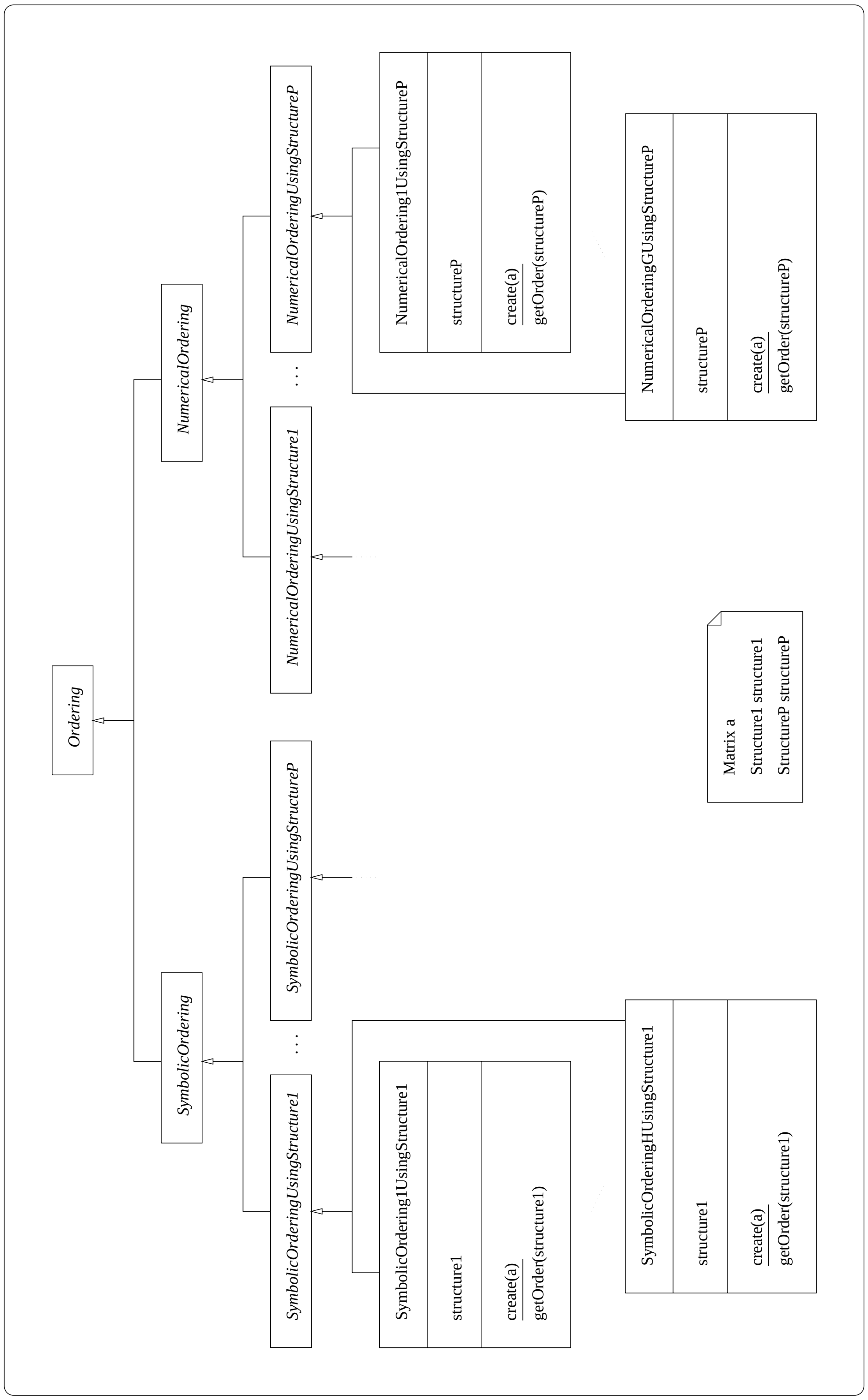

Ordering
NumericalOrdering
SymbolicOrdering
NumericalOrderingUsingStructure1
NumericalOrderingUsingStructureP
SymbolicOrderingUsingStructure1
SymbolicOrderingUsingStructureP
NumericalOrdering1UsingStructureP
structureP
create(a)
getOrder(structureP)
NumericalOrderingGUsingStructureP
structureP
create(a)
getOrder(structureP)
SymbolicOrdering1UsingStructure1
structure1
create(a)
getOrder(structure1)
SymbolicOrderingHUsingStructure1
structure1
create(a)
getOrder(structure1)
Matrix a
Structure1 structure1
StructureP structureP

Figure 3.30: Class diagram of class `Ordering` for direct solvers.

for matrix equations.

3.3 Summary

The analysis and design of an object oriented linear algebra library is the core of this chapter. Object oriented software construction is proposed and reviewed as a way of improving the development of linear algebra programs.

The analysis and design has kept in mind the requirements of both users (experts and non-experts) and library developers. Traditional libraries provide users with complex interfaces, and library developers are faced with an explosion of matrix calculation implementations.

From the fact that matrix calculations are defined in terms of matrices and their dimensions and not in terms of matrix properties and storage formats, current object oriented libraries' designs (Tables 3.1 and 3.2, and Figures 3.12, 3.13, 3.14 and 3.15) do not fully model linear algebra. These libraries do not allow a matrix to vary its properties during execution time. Consider, for example, the $B \leftarrow A+B$ matrix calculation, where A and B are square matrices of order n, but A is a dense matrix while B is an upper triangular matrix. After the calculation is performed, B becomes a dense matrix. A new class structure (Figures 3.16 and 3.17) has been designed that enables a library to manage the storage formats and to propagate the matrix properties; this is a novel functionality for linear algebra libraries. In this way, matrices can vary their properties and storage formats transparently.

The class structure has been extended so that sections of matrices and matrices formed by merging other matrices can be created without the need to replicate matrix elements and can be used like any other matrix. Hence, the new matrices (sections and merged) can have any property. By contrast, the reviewed object oriented libraries (Table 3.1) consider these new matrices to be dense matrices (Table 3.3).

From the set of reviewed object oriented libraries (Tables 3.1, 3.4 and 3.5), and from the analysis and design reported in this chapter, the following guidelines support the creation of simpler interfaces:

- matrices are represented by classes that encapsulate the way they are stored;
- a matrix calculation is represented as a unique visible method, although

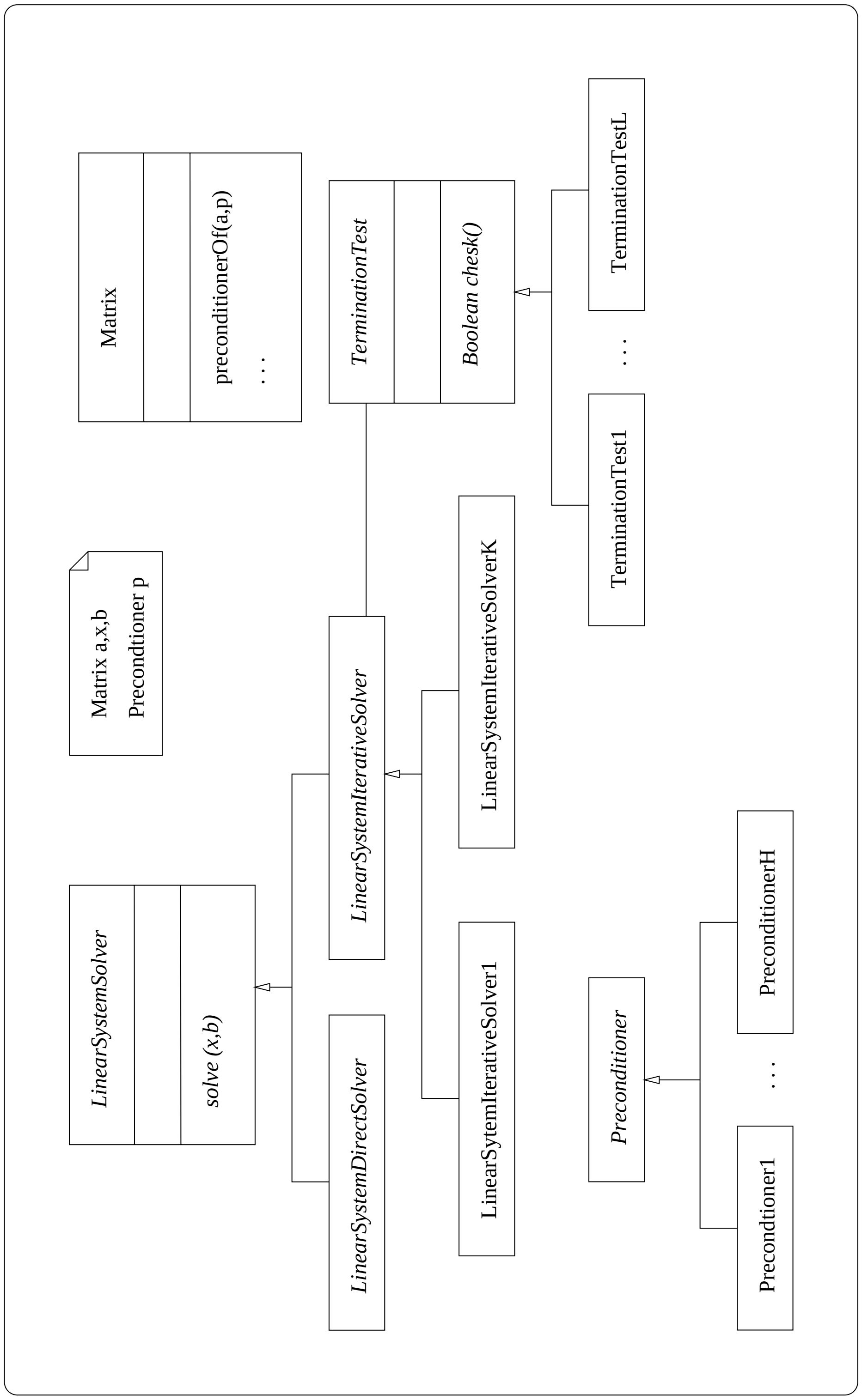


Figure 3.31: Class diagram of class `LinearSystemSolver` for iterative solvers.

different implementations and a rule based reasoning system that selects the adequate implementation are hidden behind the visible method; and

- when the reasoning system cannot be defined, the different algorithms, and not the implementations, are presented as classes and objects of these classes are passed as parameters.

The reviewed object oriented linear algebra libraries (Table 3.1) provide basic matrix operations, and solution of matrix equations with iterative and direct algorithms. However, none of them support all these matrix calculations (Tables 3.4 and 3.6). Following the above guidelines, a library interface that accounts for all these matrix calculations has been proposed. This class structure, the novel functionality and the proposed library interface constitute the design of a new library known as the Object Oriented Linear Algebra LibrAry (OoLaLa).

Developers of traditional libraries have benefited from two abstraction levels at which matrix calculations can be implemented. These abstraction levels reduce the number of implementations. Matrix abstraction level enables matrices to be represented and accessed independently of their storage formats. Iterator abstraction level is an implicit way of traversing matrices. That is, a matrix is traversed without explicitly expressing the positions of the elements that are accessed. A matrix iterator is defined so that it accesses only the elements that can be implied to be nonzero from the matrix properties.

The next chapter adapts OoLaLa to a specific object oriented programming language, Java. The implementation of the novel functionality and the implementation of matrix calculations using the abstraction levels are also illustrated. Chapter 5 describes the problems or limits in developing linear algebra programs based on libraries, either traditional or object oriented.

Readers interested in acquiring more background on object oriented software construction are recommended to look at [Mey97], [Boo94] and [GHJV95], and, as introductions to object oriented scientific programming, at [Dub97] and [Nor96]. Modelica ([MEO98], [FE98], [Mod]), and MathObject ([FVHF92], [FEV93], [FA93], [FVHF95], [AF95], [Obj]) offer an object oriented mathematical language that allows users to represent equation-based model directly. The projects Overture ([BHQ98], [BCHQ97], [BDH+98], [BHQ99], [OVE]), Pooma ([HKBR98], [HRC+98], [KCC+98], [CCH+99], [HC99], [POO]), Cogito ([Ran95], [Åhl95], [MOT97], [TMO+97], [Cog]), Diffpack ([BL97], [Lan99] , [Dif]) and PETSc ([BGMS97], [BGMS99], [PET]) have focused on object oriented partial differential equations.

Other object oriented linear algebra libraries, such as SLES a library of iterative solvers of systems of linear equations that is part of PETSc, SMOOTH ([AL96], [SMO]) an ordering library of sparse matrices, and SPOOLES ([AG99], [SPO]) a library of direct solvers for sparse linear equations, have not been reviewed since they are implemented in C (a non object oriented language) and, therefore, their designs are limited. Other object oriented linear algebra libraries have a different design objective. For example, LAKe ([NE99]) focuses on using the same code for sequential and parallel iterative solvers, and Cactus ([McD89]) focuses on finite dimensional vector spaces instead of matrix algebra. Other object oriented linear algebra libraries that have not been reviewed in this thesis include TNT ([Poz97], [TNT]) and some others listed at http://oonumerics.org/oon.

Chapter 4

Implementation of OoLaLa

The previous chapter has reported an analysis and design of an object oriented linear algebra library. The library, OoLaLa, has been designed independently of any programming language. OoLaLa offers a novel functionality for libraries: propagation of matrix properties and management of storage formats. OoLaLa also enables library developers to implement matrix calculations at two abstraction levels: matrix and iterator abstraction levels. These abstraction levels reduce the number of implementations of a given matrix calculation.

Matrix abstraction level is independent of the storage format in which matrices are represented. A given matrix element is mapped automatically to the position of its storage format. Iterator abstraction level, apart from also being independent of the storage format, traverses matrices without explicitly indicating the positions of the elements that are accessed. A matrix iterator is defined so that only the nonzero elements of matrices are accessed.

The objective of this chapter is to describe how OoLaLa is adapted and implemented in Java. At the same time, example programs are presented to show users how to develop programs using OoLaLa.

Firstly, OoLaLa is adapted to the specific characteristics of Java (Section 4.1). An example program that declares, creates and initialises matrices, illustrates how these are implemented using UML object diagrams (introduced in the previous chapter) and UML sequence diagrams (introduced in this chapter) (Section 4.2). Two more example programs show how views (i.e. sections of a matrix or matrices formed by merging other matrices) are created and how they are implemented (Section 4.3). The management of storage formats is presented in conjunction with the propagation of properties (Section 4.4). Finally, matrix calculations are implemented at matrix and iterator abstraction levels (Section 4.5).

4.1 Adapting OoLaLa to Java

Java is a clean and strongly typed object oriented language. Unlike other languages (C++, Ada95, ...) which have evolved from their procedural subsets (C, Ada83, ...), Java was designed to be an object oriented language. Java offers built in parallelism, a powerful set of classes to develop graphical interfaces and makes network based applications easy to program.

Java programs are compiled into an intermediate language known as *bytecode*. A Java Virtual Machine (JVM) is an interpreter of bytecodes. The JVMs enable Java programs to be written once and run on any computer (as long as an implementation of a JVM exists for it). Both the language and the JVM have been fully specified, leaving no details to the discretion of compiler developers.

The Java Grande Forum[1] (JGF), an open forum to academia, industry or government, was formed under the belief that "Java has potential to be a better environment for Grande Applications development than any previous languages such as Fortran and C++" [Jav98]. The term Grande Application is also defined "as an application of large-scale nature, potentially requiring any combination of computers, networks, I/O, and memory". Numerical linear algebra libraries are the kernels of most of these applications.

However, Java has some poor characteristics for implementing OoLaLa:

- Java does not support multiple inheritance;
- Java does not support generic classes, nor complex numbers as a language data type, nor light-weight classes; and
- Java specifies a multidimensional array as an array of arrays.

The following paragraphs discuss the problems that these characteristics of Java cause and the decisions taken to overcome them.

Multiple inheritance has been used in the class structure of OoLaLa to model matrix properties that result from composing other matrix properties. For example, class `SymmetricBandedProperty` inherits from `SymmetricProperty` and

[1]Java Grande Forum web site at http://www.javagrande.org/

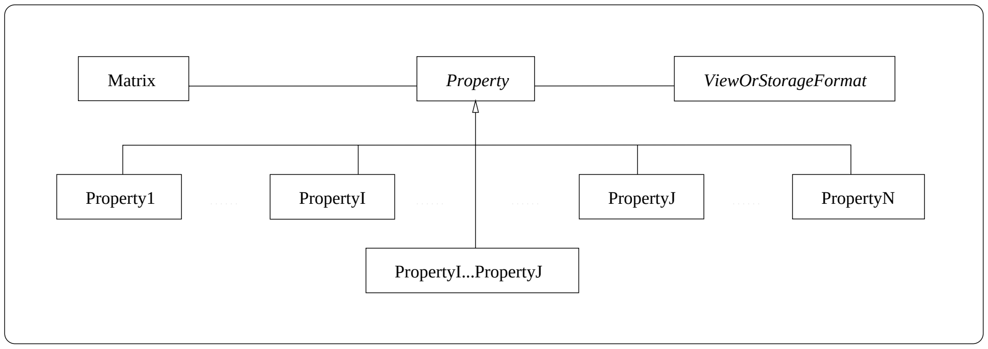


Figure 4.1: Class diagram of class `Property` and its sub-classes adapted to Java.

`BandedProperty`. Since multiple inheritance is not available, every class representing matrix properties simply inherits from the class `Property`. Figure 4.1 presents the changes to the `Property` class inheritance hierarchy.

Ideally, generic classes would be used to develop only one version of OoLaLa independent of the data type of the matrix elements. Users would choose the data type of the matrix elements and the compiler would generate automatically the version of OoLaLa. Section 3.1.4 described how generic classes can be emulated using inheritance and client relation. The OwlPack linear algebra library ([BK99b], [BK99a], [BKP98]) has been implemented emulating generic classes by an equivalent class `Matrix` having a client relation with an abstract class `Number` from which `Float`, ..., `Complex` classes inherit. OwlPack also has been implemented by writing one version of the library for each data type. It is reported that the version emulating generic classes is between 4 times and 100 times slower than writing one version for each data type depending on the benchmark. In order to close the gap, Budilimć and Kennedy ([BK99b], [BK99a]) propose interprocedural and interclass compiler optimisation, which are only possible if the compilation strategy of Java is changed. Currently, the compiler can only consider one class at the time. On the other hand, JGF proposes the inclusion of light-weight classes. A light-weight class is a class whose objects are treated by the compiler and the JVM as variables of a language data type. In response to this proposal, Sun (owner of Java specification) plans to write a proposal for light-weight classes [Jav99]. Other projects have experimented with generic classes in Java ([AFM97], [BML97], [OW97]). Given current circumstances, OoLaLa is implemented by developing a version for each data type.

Java multidimensional arrays are specified to be an object array that has objects array. This specification creates a very powerful data structure. Given a two-dimensional Java array, each of its one-dimensional arrays can be substituted with different arrays of different sizes. However, this structure does not ensure that the objects array are continuous in memory and this might result in a poor memory locality. This structure also needs to perform bound and null object checks for each dimension since both checks are compulsory in the Java language specification. This array structure and the precise exception model do not allow all the compiler optimisation techniques developed for fixed size multidimensional arrays. The Java exception model specifies that an exception must appear in strict program order. The above motivated the JGF Numerics Working Group to develop a Java package with multidimensional arrays mapped into one-dimensional Java arrays. This package was developed by the IBM's Ninja group[2] [MMG99], which at the same time developed compiler techniques to identify exception-free code sections. For these exception-free code sections, compiler optimisation techniques developed for Fortran and C can be applied [MMG98]. The experiments reported with the array package show an improvement of 15% in MFlops performance when multiplying two matrices using two-dimensional arrays from the package compared with two-dimensional language arrays [MMG99].

Blount and Chatterjee ([BC99],[BC98]) in their JLAPACK library also store matrices by mapping them into one-dimensional language arrays. They optimised this approach by noting that most matrix calculations are implemented with sequential, stride *inc*, access to the matrices. This enables the next position in a one-dimensional language array to be calculated as $lastposition + inc$ instead of as $i - 1 + (j - 1) * n$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Note that a Java language array has its first element in index 0 and that two-dimensional arrays are mapped column-wise (as in Fortran). It is reported that LU factorisation on a Pentium II is around 1.5 times faster and on a Ultra Spark around 3 times faster than a version obtained using the f2j translator [DDS99] which uses language two-dimensional arrays ([BC98], [BC99]).

OoLaLa represents two-dimensional arrays by mapping them to one-dimensional language arrays in a column-wise form. In order to exploit Blount and Chatterjee's array access observation, new methods, `incIndexColumn` and `incIndexRow`,

[2]Ninja group address http://www.research.ibm.com/ninja

are included in `Matrix`. This constitutes the final design modification due to particular features of Java.

4.2 Declare and Access Matrices

The first step in writing a program is to declare variables. In numerical linear algebra the variables are mainly matrices. Using OoLaLa, users declare objects of class `Matrix`. These objects are then given dimensions and properties. The next step in writing a program is to initialise the variables. In OoLaLa, users access objects of class `Matrix` using mainly `element` and `assign`. Figure 4.2 gives an example program to declare and initialise matrices.

Figure 4.3 introduces UML sequence diagram notation. A sequence diagram is a way of representing the life (creation, invocations of methods, and destruction) of objects over time. Objects are represented by rectangles in which their names and class names are written underlined. A method invocation is represented as an arrow with solid head from the object that invokes the method to the object where the method is invoked. An object (in sequential execution) becomes active when a method is invoked in it. The time that an object is active is represented by a thin rectangle under the object. An object remains active while an invoked method remains unfinished. This does not mean that the flow of control is in this object. The flow of control is transferred to another object when a method is invoked in this other object. The flow of control returns when the method is finished. The arrows represent the transfer of control flow (in sequential execution).

Figure 4.4 presents the sequence diagram for the statements labelled as action 1 and action 2 in the example program of Figure 4.2. The first statement declares an object of class `Matrix` and the second statement sets the dimension and property. Users only perceive what is on the left of the object `a` in Figure 4.4; the methods invoked and objects on its right are not visible to users. Figure 4.5 presents the object diagram after every object `Matrix` has been declared and properties have been set. Note that only the object `e` has requested a specific storage format. The other storage formats have been selected automatically, see Section 4.4 for details.

Finally, Figure 4.6 presents the sequence diagram for the statements labelled as action 3 and action 4 in the example program of Figure 4.2. These are invocations to `assign` and `element` methods. Again, users only perceive what is on

```
class DeclareAndAccessMatrices
{
  public static void main(String args[])
  // how to declare and set properties
  {
    // begin declare matrices
    Matrix a = new Matrix(); // action 1
    Matrix b = new Matrix(); Matrix c = new Matrix();
    Matrix d = new Matrix(); Matrix e = new Matrix();
    // end declare matrices
    double temp;

    // begin set matrices properties
    a.setDenseMatrix(10,15); // action 2
       // numRows=10 and numColumns=15
    b.setBandedMatrix(20,30,2,1);
       // numRows=20, numColumns=30,
       // numUpperBandwidth=2 and numLowerBandwidth=1
    c.setSymmetricMatrix(15);
       // numRows=15 and numColumns=15
    d.setSymmetricBandedMatrix(15,3);
       // numRows=15, numColumns=15,
       // numUpperBandwidth=3 and numLowerBandwidth=3
    e.setBandedMatrix(100,100,50,65,OOLaLaStorageFormat.denseFormat());
       // numRows=100, numColumns=100
       // numUpperBandwidth=50 and numLowerBandwidth=65
       // requested dense format
    // end set matrices properties

    // begin access matrices
    a.assign(8,6,3.14159); // action 3
    temp=a.element(8,6); // action 4
    // end access matrices
  }// end main
}// end class DeclareAndAccessMatrices
```

Figure 4.2: Example program of how to declare and access matrices using OoLaLa.

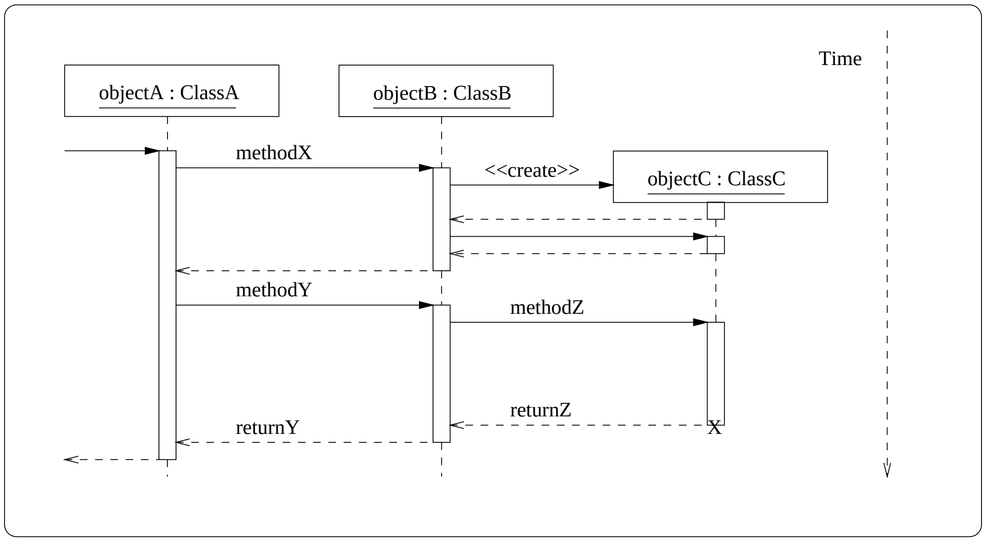


Figure 4.3: UML sequence diagram notation.

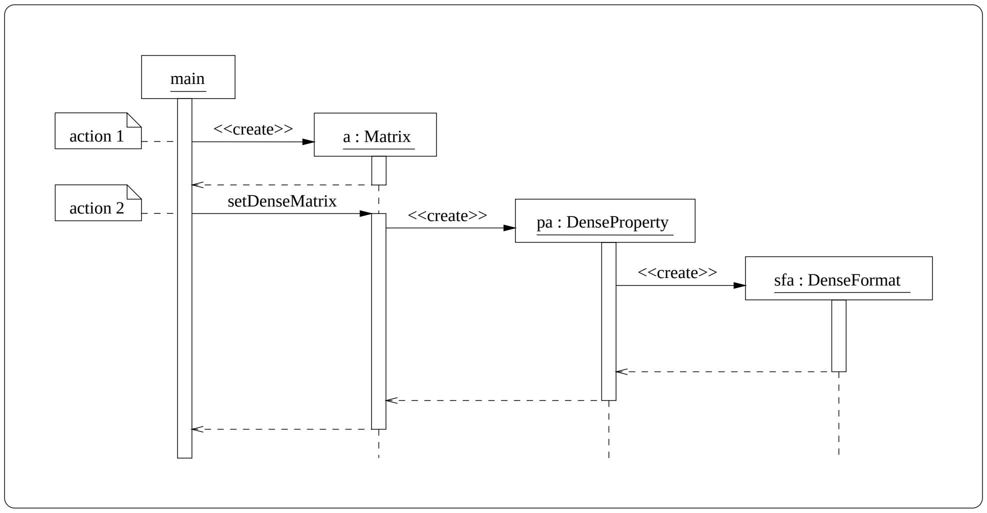


Figure 4.4: Sequence diagram for declaring a dense matrix using OoLaLa.

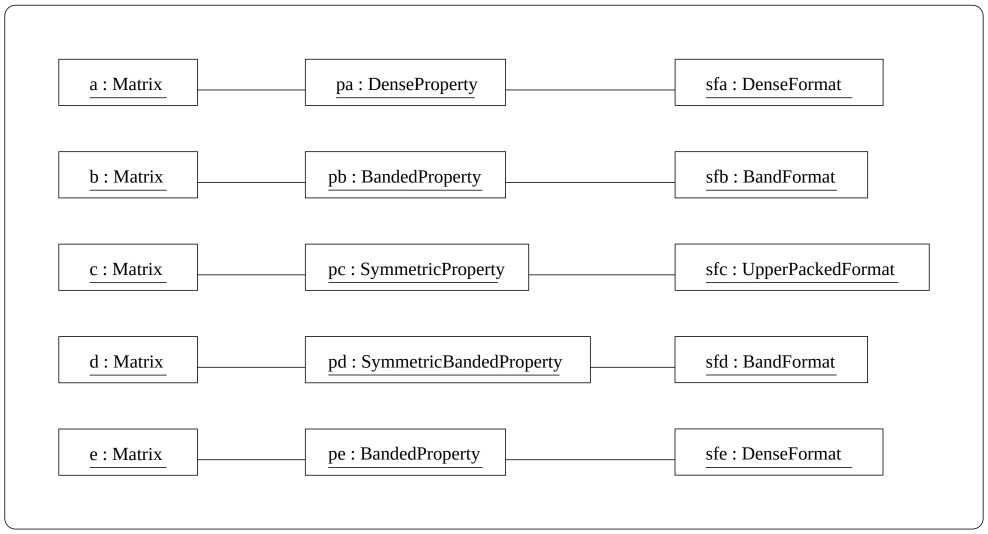


Figure 4.5: Object diagram after declaring and setting properties of matrices.

the left of object `a` in Figure 4.6, the rest occurs transparently.

4.3 Create Views

A view can be either a section of a matrix or a matrix formed by merging other matrices. Figure 4.7 presents an example program showing how different sections of matrices can be created. In this program, a 5×5 dense matrix A is represented by an object `a` of class `Matrix`. Three matrices represented by three objects (`section1`, `section2` and `section3`) of class `Matrix` are created as sections of `a`. These three matrices do not replicate the matrix elements. Figure 4.8 presents the sections of the matrix A for each object `Matrix`. Figure 4.9 presents the sequence diagram for the program and Figure 4.10 presents the object diagram after all the sections have been created. Each object of class `Matrix` has its own properties, but they share the object of class `DenseFormat`. This shared object stores the elements of the matrix A and, consequently, the elements of the defined sections of A.

A merged matrix is formed by merging other matrices. Figure 4.11 presents an example program which creates a 5×5 block diagonal matrix from its block sub-matrices. The objects `zero1_2`, `zero2_1` and `zero2_2` of class `Matrix` represent zero matrices with different dimensions. The objects `diag1`, `diag2` and `diag3` of class `Matrix` represent the block sub-matrices which are on the diagonal of

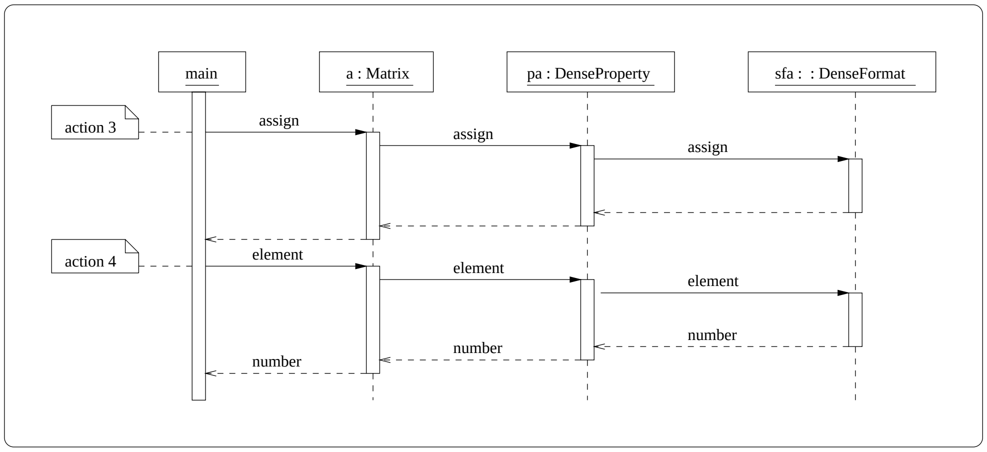


Figure 4.6: Sequence diagram for access methods.

```
class CreateSections
{
  public static void main (String args[])
  {
    // begin declare matrices
    Matrix a= new Matrix();
    Matrix section1= new Matrix();
    Matrix section2= new Matrix();
    Matrix section3= new Matrix();
    // end declare matrices

    a.setDenseMatrix(5,5); // set properties
    // begin create sections
    a.getSubMatrix(section1,3,5,3,5);
    a.getTranspose(section2);
    a.getUpperTriangularSection(section3);
    // end create sections
  }// end main
}// end CreateSections
```

Figure 4.7: Example program of how to create sections of matrices using OoLaLa.

$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$ `Matrix a`	$\begin{pmatrix} a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{pmatrix}$ `Matrix section1`
$\begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} & a_{51} \\ a_{12} & a_{22} & a_{32} & a_{42} & a_{52} \\ a_{13} & a_{23} & a_{33} & a_{43} & a_{53} \\ a_{14} & a_{24} & a_{34} & a_{44} & a_{54} \\ a_{15} & a_{25} & a_{35} & a_{45} & a_{55} \end{pmatrix}$ `Matrix section2`	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ & a_{22} & a_{23} & a_{24} & a_{25} \\ & & a_{33} & a_{34} & a_{35} \\ & & & a_{44} & a_{45} \\ & & & & a_{55} \end{pmatrix}$ `Matrix section3`

Notation: blanks represent zero elements which cannot be modified.

Figure 4.8: Graphical representation of the sections of matrices and matrices created in Figure 4.7.

matrix A. Matrix A is represented by an object `a` of class `Matrix` formed after the execution of the statement labelled as action 1. Figure 4.12 describes the object structure after this statement has been executed. Figure 4.13 presents the matrices that each object of class `Matrix` represents.

The object `a` represents a block diagonal matrix. Looking at the object diagram (see Figure 4.12), the block diagonal matrix is stored as a set of objects of class `StorageFormat`. Each object is used for certain block sub-matrices of A. In general, any matrix can be partitioned into block sub-matrices. Each block can have different properties and therefore different advisable storage formats. The class structure of OoLaLa enables users to operate transparently with a matrix that is stored by its blocks, and each block is stored in any advisable storage format.

Moreover, since the object `a` is of class `Matrix`, sections of the matrix represented by `a` can be also created regardless of `a` being stored by its blocks. The statement labelled as action 2 in Figure 4.11 makes the object `section` a section of the matrix represented by `a`. The object `section` represents a diagonal matrix (see Figure 4.13). Hence, the object diagram in Figure 4.14, presents the object `section` linked to an object of class `DiagonalProperty`. Efficient algorithms for

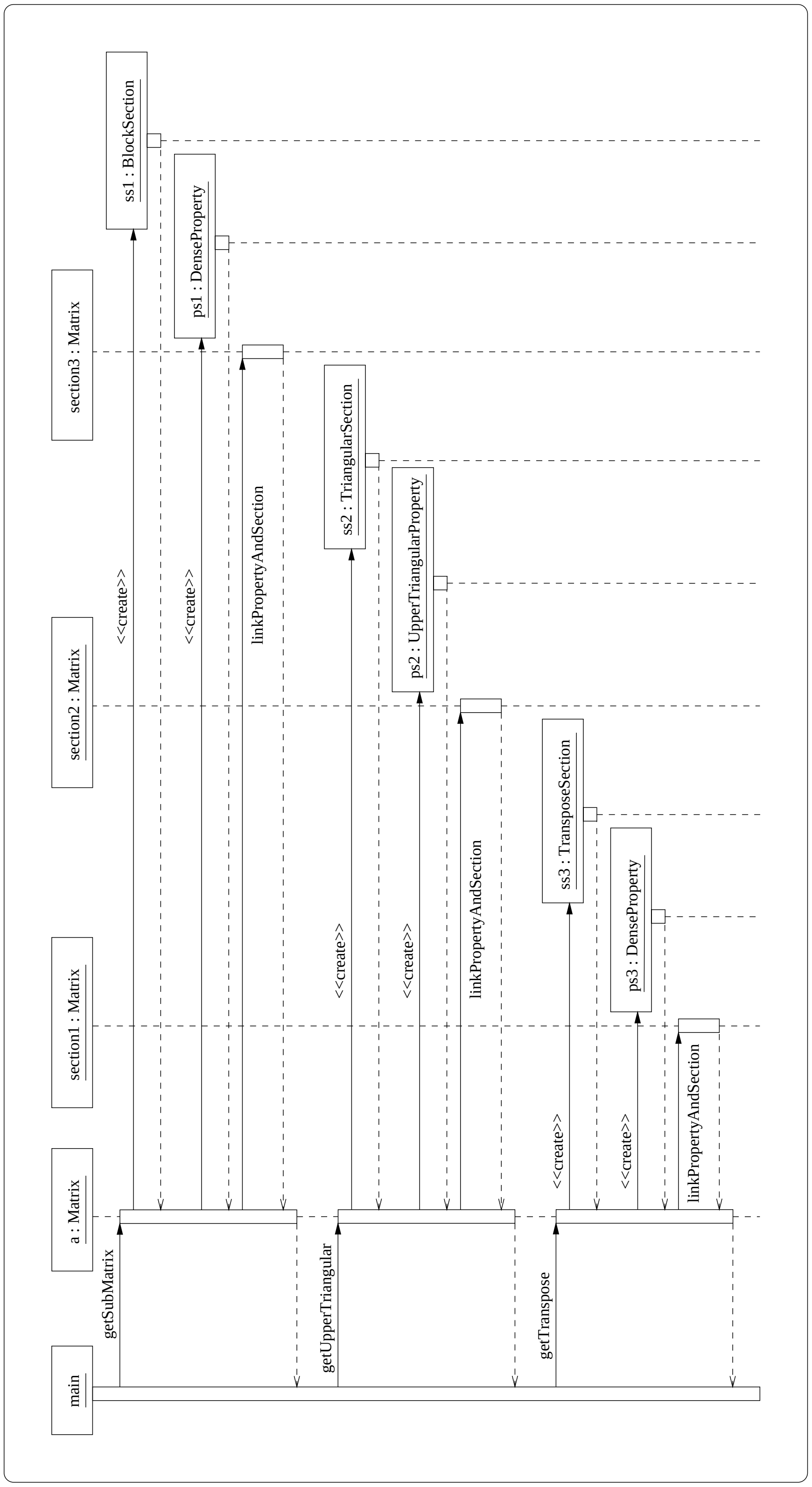


Figure 4.9: Sequence diagram for the sections created in Figure 4.7.

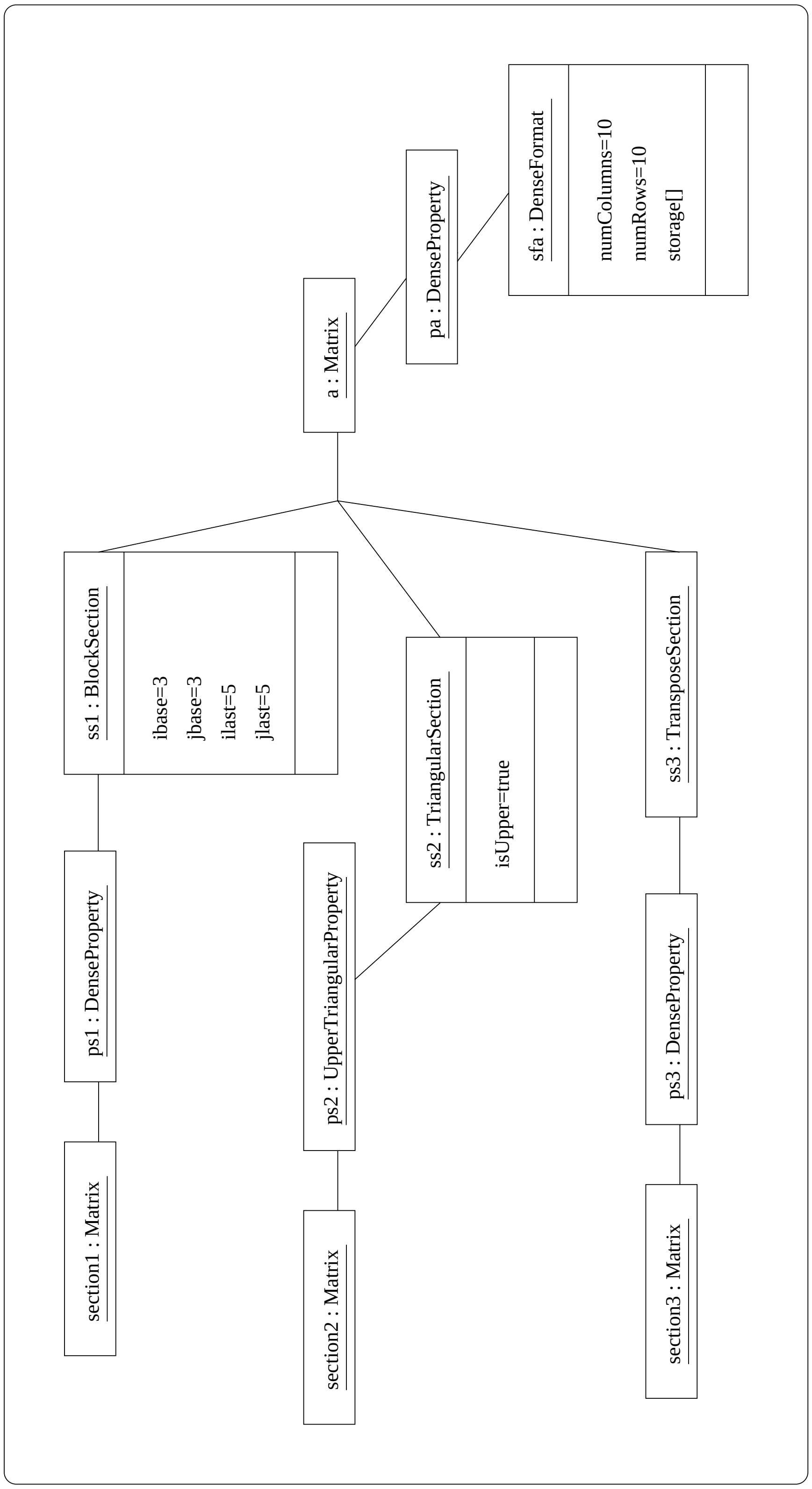


Figure 4.10: Object diagram after the sections have been created in Figure 4.7.

```
class CreateAMergedMatrix
{
  public static void main (String[] args)
  {
    //begin declare matrices
    Matrix a= new Matrix();
    Matrix zero1_2= new Matrix();
    Matrix zero2_1= new Matrix();
    Matrix zero2_2= new Matrix();
    Matrix diag1= new Matrix();
    Matrix diag2= new Matrix();
    Matrix diag3= new Matrix();
    Matrix section = new Matrix();
    //end declare matrices
    Matrix array={{diag1,zero2_1,zero2_2},
                  {zero1_2,diag2,zero1_2},
                  {zero2_2,zero2_1,diag3}};

    // begin set properties
    diag1.setDenseMatrix(2,2);
    diag2.setDenseMatrix(1,1);
    diag3.setLowerTriangularMatrix(2,2);
    zero1_2.setZeroMatrix(1,2);
    zero2_1.setZeroMatrix(2,1);
    zero2_2.setZeroMatrix(2,2);
    // end set properties

    // create a matrix by merging matrices
    a.merge(array); // action 1
    // create a section of matrix
    a.getSubMatrix(section,2,4,2,4); // action 2
  }// end main
}// end CreateAMergedMatrix
```

Figure 4.11: Example program of how to create a matrix by merging matrices using OoLaLa.

determining the nonzero elements structure, described in [BW99], enable the library to identify the matrix as being diagonal. In this way, a section of matrix A or of a set of matrices (block sub-matrices) can be created and used transparently as a matrix.

4.4 Management of Storage Formats

In all the examples that have been presented, the programs have not specified the class of storage format, except once (see Figures 4.2, 4.7, and 4.11). However, the

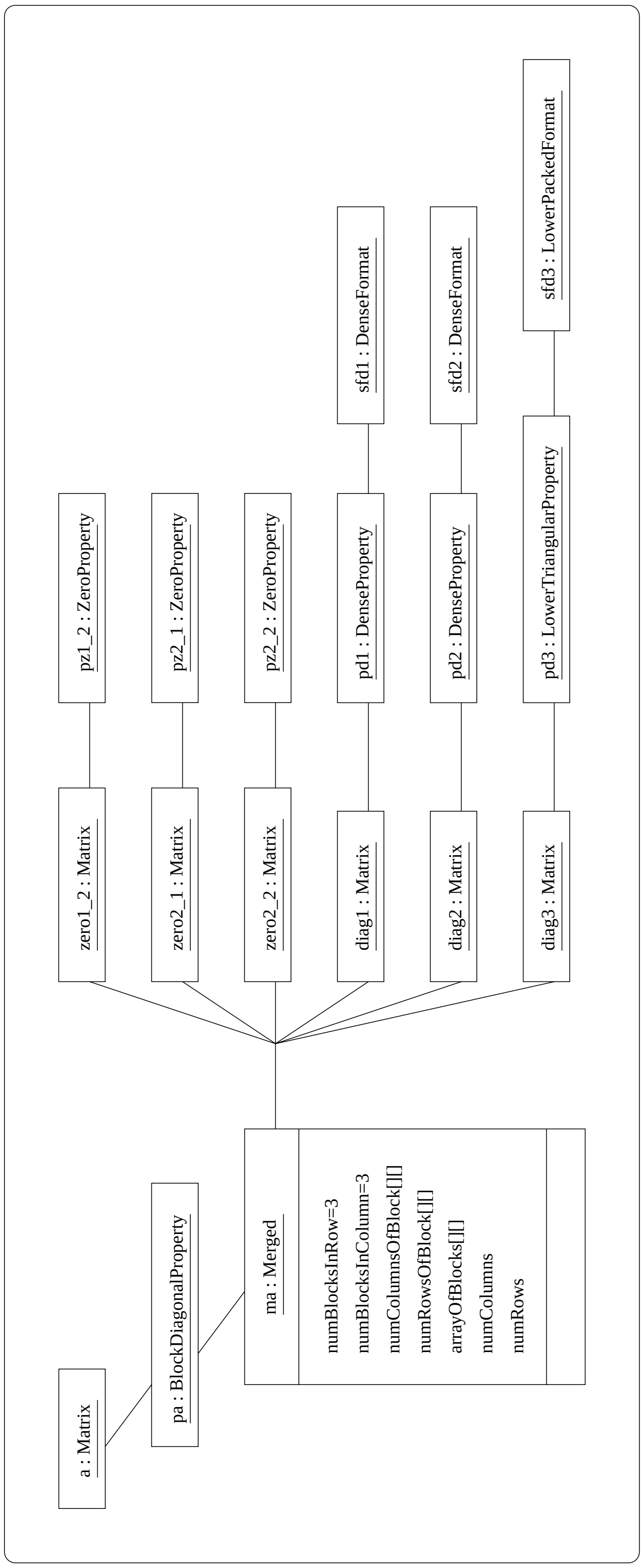


Figure 4.12: Object diagram after a matrix has been created by merging matrices from example program in Figure 4.11.

$\begin{pmatrix}0\\0\end{pmatrix}$ `Matrix zero2_1`	$\begin{pmatrix}0 & 0\end{pmatrix}$ `zero1_2`
$\begin{pmatrix}0 & 0\\0 & 0\end{pmatrix}$ `Matrix zero2_2`	$\begin{pmatrix}d1_{11} & d1_{12}\\d1_{21} & d1_{22}\end{pmatrix}$ `Matrix diag1`
$\begin{pmatrix}d2_{11}\end{pmatrix}$ `Matrix diag2`	$\begin{pmatrix}d3_{11} & 0\\d3_{21} & d3_{22}\end{pmatrix}$ `Matrix diag3`
$\left(\begin{array}{\|cc\|c\|cc\|}\hline d1_{11} & d1_{12} & 0 & 0 & 0\\ d1_{21} & d1_{22} & 0 & 0 & 0\\ \hline 0 & 0 & d2_{11} & 0 & 0\\ \hline 0 & 0 & 0 & d3_{11} & 0\\ 0 & 0 & 0 & d3_{21} & d3_{22}\\ \hline\end{array}\right)$ `Matrix a`	$\begin{pmatrix}d1_{22} & 0 & 0\\0 & d2_{11} & 0\\0 & 0 & d3_{11}\end{pmatrix}$ `Matrix section`

Figure 4.13: Graphical representation of the matrices created in Figure 4.11.

object diagrams have always presented objects of a sub-class of `StorageFormat` (see Figures 4.5, 4.10, and 4.12). OoLaLa chooses automatically and statically a storage format for every matrix. Before invoking any matrix calculation that changes the matrix elements, OoLaLa decides whether the storage format and properties need to be changed. Consider, for example, the addition $C \leftarrow A + B$ where A and B are tridiagonal matrices and C is a bidiagonal matrix. After performing the addition C also becomes tridiagonal. A program using OoLaLa would create objects `a`, `b`, `c` of class `Matrix` and set the correspondent properties of each matrix. The program would continue with the statement `c.addInto(a,b);`; this method invoked in `c`, would change its linked object of class `BidiagonalProperty` by one of class `TridiagonalProperty`. Depending on the class of the linked object that represents the storage format, different action could be taken. Suppose the actual storage format is large enough to store the extra elements that will be created and it is advisable to have the result matrix in this storage format, then no action is needed. This would be the case if `c` were linked with an object of class `DenseFormat`. Otherwise (either the storage format is not large enough or it is not advisable to store the result matrix in that storage

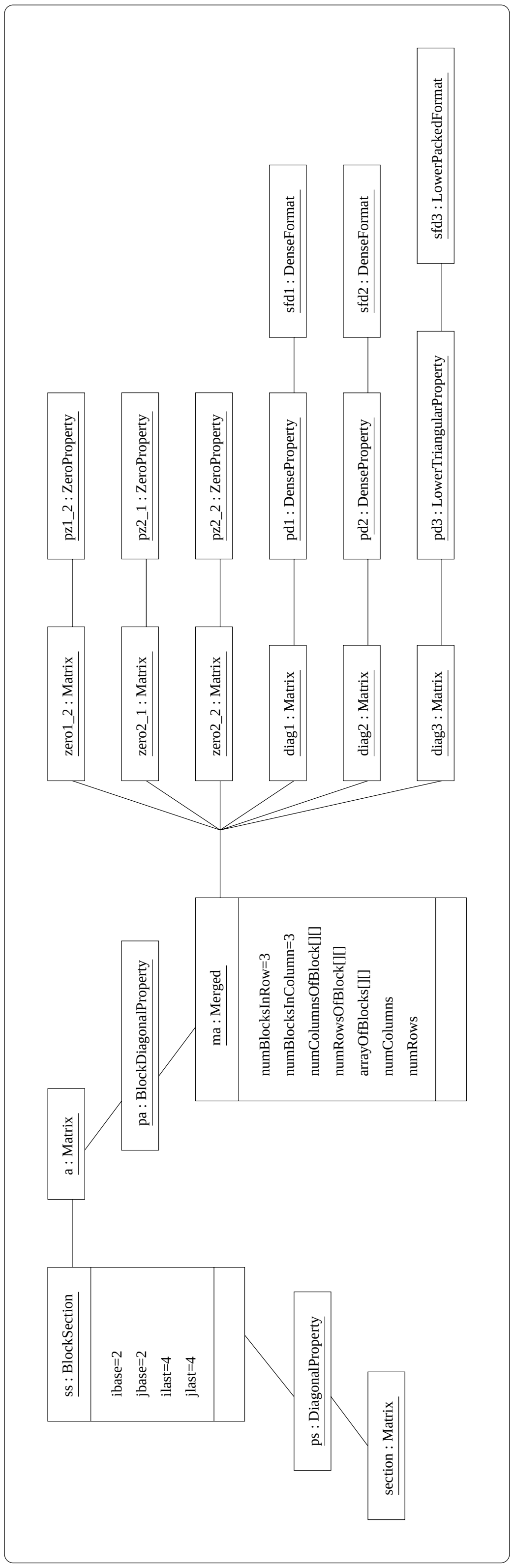


Figure 4.14: Object diagram after a section of matrix, which has been created by merging matrices, is created – example program in Figure 4.11.

Property	Storage Format
de	df
ba	df or bf
sy	upf
sb	upf or bf
ut	upf
lt	lpf
ub	upf or bf
lb	lpf or bf

Table 4.1: Storage format selected for each matrix property.

format), the linked object representing the storage format would be changed.

The following paragraphs explain how to select a storage format for a certain property, how to detect inconsistency between properties and storage formats, and how consistency is recovered. The description is limited to a set of properties (dense (de), banded (ba), symmetric (sy), symmetric banded (sb), upper triangular (ut), and lower triangular (lt) properties) and a set of storage formats (dense (df), band (bf), upper packed (upf) and lower packed (lpf) formats). These properties and storage formats are those supported in BLAS ([DCHH88b], [DCHD90]) and were described in Section 2.2.

The first question to answer is how OoLaLa chooses a storage format for a matrix with certain properties. Table 4.1 presents recommended storage formats for each matrix property as a set of static "if-then" rules. These rules do not have an explicit representation in OoLaLa; they are included in the code of each method `setPropertyMatrix`. For example, inside the code of `setDenseMatrix` there is a part that creates an object of class `DenseFormat`. The rules select, whenever possible, a storage format which uses the least memory space. Note that some rules can choose between band format and another format. The band format is selected when the upper bandwidth and lower bandwidth are less than half the number of columns and number of rows, respectively. This condition is an initial guess that needs validation with experiments.

The second question is how to detect inconsistency between matrix properties and storage formats. An inconsistent situation can only arise when a user sets a property and an inconsistent storage format, or when a matrix calculation results in a change of property. The first case is easier to solve. Table 4.2 presents

	df	bf	upf	lpf
de	√			
gb	√	√		
sy	√		√	
sb	√	√	√	
ut	√		√	
lt	√			√
ub	√	√	√	
lb	√	√		√

Table 4.2: Consistency between storage formats and matrix properties.

the advisable combinations. When a user sets a property and a storage format (e.g. `a.setDenseMatrix(10,10,OOLaLaStorageFormat.bandFormat());`) OoLaLa checks the combination against Table 4.2 and raises an exception of class `NonAdvisablePropertyAndStorageFormatCombination` when necessary.

The second case, when a matrix calculation results in a property change, requires the prediction of the new property of the matrix and the identification of the circumstances under which each matrix calculation triggers a property change. Note that a property is considered to be changed even if the property is the same but some characteristic of the property has been changed. For example, a banded matrix may remain a banded matrix but with a reduced or increased bandwidth.

Table 4.3 presents the matrix property of the result matrix from an analysis of the operand properties for matrix addition. Table 4.4 presents equivalent information for matrix-matrix multiplication. These tables are constructed assuming no knowledge of the numerical values of the elements apart from that implied by the properties of their matrices.

The prediction of matrix properties for matrix-matrix multiplication and matrix addition is fully determined; the tables represent static "if-then" rules. These rules use the properties of the operands to decide the property of the result matrix. OoLaLa implements them as internal tables that are consulted by the codes of `addInto` and `multiplyInto`.

Having explained how to detect inconsistent combinations of matrix properties and storage formats, it is now described how to recover consistency; i.e., the storage format needs to be changed in order to be consistent with the matrix property. Table 4.5 presents the selection of the new storage format. In the first

A \ B	de	ba	sy	sb	ut	lt	ub	lb
de	0	0	0	0	0	0	0	0
ba	0	1	0	1	1	1	1	1
sy	0	0	2	2	0	0	0	0
sb	0	1	2	3	1	1	1	1
ut	0	1	0	1	4	0	4	1
lt	0	1	0	1	0	5	1	5
ub	0	1	0	1	5	1	6	1
lb	0	1	0	1	1	4	1	7

```
0 →  c.setDenseMatrix(a.numRows(), a.numColumns())
1 →  if (Math.max(a.upperBandwidth(), b.upperBandwidth())
     == a.numColumns()-1 && Math.max(a.lowerBandwidth(),
     b.lowerBandwidth()) == a.numRows()-1)
     { c.setDenseMatrix(a.numRows(), a.numColumns()) }
     else { c.setBandedMatrix(a.numRows(), a.numColumns(),
     Math.max(a.upperBandwidth(), b.upperBandwidth()),
     Math.max(a.lowerBandwidth(), b.lowerBandwidth())) }
2 →  c.setSymmetricMatrix(a.numRows())
3 →  c.setSymmetricBandedMatrix(a.numRows(), a.upperBandwidth())
4 →  c.setUpperTriangularMatrix(a.numRows(), a.numColumns())
5 →  c.setLowerTriangularMatrix(a.numRows(), a.numColumns())
6 →  c.setSymmetricMatrix(a.numRows())
7 →  c.setUpperTriangularBandedMatrix(a.numRows(),
     a.numColumns(), Math.max(a.upperBandwidth(),
     b.upperBandwidth()), 0)
8 →  c.setLowerTriangularBandedMatrix(a.numRows(),
     a.numColumns(), 0, Math.max(a.lowerBandwidth(),
     b.lowerBandwidth()))
```

Table 4.3: Rules for determining the properties of the result matrix C for the addition of matrices $C \leftarrow A + B$.

B A	de	ba	sy	sb	ut	lt	ub	lb
de	0	0	0	0	0	0	1	1
ba	0	1	0	1	0	0	0	0
sy	0	0	0	0	0	0	0	0
sb	0	1	0	2	0	0	1	1
ut	0	0	0	0	3	0	3	1
lt	0	0	0	0	0	4	1	4
ub	0	1	0	1	3	1	5	1
lb	0	1	0	1	1	4	1	6

```
0 →  c.setDenseMatrix(a.numRows(), b.numColumns())
1 →  if (a.upperBandwidth() + b.upperBandwidth() >=
     b.nomColumns()-1 && a.lowerBandwidth() + b.lowerBandwidth()
     >= a.numRows()-1)
     { c.setDenseMatrix(a.numRows(), b.numColumns()) }
     else {c.setBandedMatrix(a.numRows(), b.numColumns(),
     Math.min(a.upperBandwidth() + b.upperBandwidth(),
     b.nomColumns()-1), Math.min(a.lowerBandwidth() +
     b.lowerBandwidth(), a.numRows()-1)) }
2 →  if (a.upperBandwidth()==0 && b.upperBandwidth()==0 &&
     a.lowerBandwidth()==0 && b.lowerBandwidth()==0)
     { c.setSymmetricBandedMatrix(a.numRows(), 0) }
     else { c.setBandedMatrix(a.numRows(), b.numColumns(),
     Math.min(a.upperBandwidth() + b.upperBandwidth(),
     b.nomColumns()-1), Math.min(a.lowerBandwidth() +
     b.lowerBandwidth(), a.numRows()-1)) }
3 →  c.setUpperTriangularMatrix(a.numRows(), b.numColumns())
4 →  c.setLowerTriangularMatrix(a.numRows(), b.numColumns())
5 →  c.setUpperTriangularBandedMatrix(a.numRows(),
     b.numColumns(), Math.min(a.upperBandwidth() +
     b.upperBandwidth(), b.nomColumns()-1))
6 →  c.setLowerTriangularBandedMatrix(a.numRows(),
     b.numColumns(),Math.min(a.lowerBandwidth() +
     b.lowerBandwidth(), a.numRows()-1))
```

Table 4.4: Rules for determining the properties of the resultant matrix C for the matrix-matrix multiplication $C \leftarrow AB$.

	df	bf	upf	lpf
de		df	df	df
ba		bf or df	bf or df	bf or df
sy		upf		upf
sb		bf or upf		bf or upf
utr		upf		upf
ltr		lpf	lpf	
utb		bf or upf		upf
ltb		bf or lpf	lpf	

Table 4.5: Storage formats transitions triggered by a new matrix property.

row the current storage format of the matrix is specified, while the new matrix properties are specified in the first column. In some cases, two different storage formats can be selected: band format or some other. As before, the band format is selected when the upper and lower bandwidths are less than half the number of columns and rows, respectively.

Since views of matrices do not have an explicit storage format, they are treated as special cases. Views are matrices that are sections of other matrices, or matrices formed by merging other matrices. When a section matrix is operated on and its property is changed, the property of the matrix of which it is a section might change and, consequently, its storage format also. These changes are performed when such situations arise. However, when the matrix that has a section is operated on, a lazy algorithm is implemented. This algorithm updates the matrix and leaves a signal for the section matrix that is not updated. Only when the section matrix is used again is its new property updated.

For a matrix formed by merging other matrices, either the matrices or the merged matrix can be operated on and their properties changed. When a merged matrix changes its properties, every matrix of which it is formed has to be adapted. However, when a matrix that forms part of the merged matrix changes its properties, a lazy implementation can again be used. The merged matrix is only updated when it is subsequently used.

Other matrix calculations, and techniques for dealing with sparse and block matrices, are implemented similarly, but following the structure predictions described in [Gil94] [Coh99].

4.5 Matrix Calculations

A matrix calculation is divided into four phases: select among the implementations with the appropriate functionality, check correctness of parameters, predict property of the result matrix, and the specialised implementation of the matrix calculation. In earlier chapters, the storage format (Section 2.3.3), matrix (Section 3.2.1) and iterator (Section 3.2.3) abstraction levels have been introduced. Matrix and iterator abstraction levels have been introduced as a way of traversing matrices, but examples of how to implement operations with them have yet to be presented. These abstraction levels (matrix and iterator) enable library developers to code fewer implementations, compared with the storage format abstraction level, but still deal with the same storage formats and matrix properties.

Matrix calculations are only defined for certain (conformable) matrices. For example, the addition of matrices is only defined for matrices that have the same numbers of rows and columns. Similarly, the matrix-matrix multiplication $C \leftarrow AB$ is only defined for A being a $n \times k$ matrix and B a $k \times m$ matrix. These tests are straightforward to implement and are, therefore, omitted in the following description. Simply note that the test is performed before any of the instructions of the matrix calculation implementation are executed. When the test is not successful an exception is raised and the matrices are left unmodified.

In traditional libraries, users have to select a subroutine that represents an implementation of the matrix calculation for matrices with certain properties and certain storage formats. Since OoLaLa encapsulates, whenever possible, the different implementations behind a unique method offered to its users, a selection algorithm is necessary. The selection algorithm varies with the abstraction level at which the matrix calculations are implemented.

4.5.1 Implementing at Different Abstraction Levels

Matrix-matrix multiplication is used to describe how matrix and iterator abstraction levels can reduce the number of implementations compared with storage format abstraction level. Among the different combinations of storage formats and matrix properties, the following combinations are selected to illustrate the abstraction levels for the operation $C \leftarrow AB$:

- A and B are both dense matrices stored in dense format,

- A is an upper triangular matrix and B is a dense matrix, both stored in dense format, and
- A is an upper triangular matrix stored in packed format and B is a dense matrix stored in dense format.

Figures 4.15 and 4.16 present implementations at storage format abstraction level. Note that these implementations use two-dimensional language arrays for the sake of clarity (on the right hand side of Figure 4.15 the same implementation is presented but the arrays are mapped into one-dimensional language arrays). Since implementation at this abstraction level requires access to the representation of the storage format, there is a different implementation for each storage format.

The matrix abstraction level is independent of the storage formats. Figure 4.17 presents the implementations for the three combinations. Only two implementations are necessary corresponding to A dense or A upper triangular, and the only difference between the implementations is the bound on the inner `i` loop.

The iterator abstraction level is also independent of the storage formats and of the nonzero element structures. Figure 4.18 presents an implementation in which the elements are accessed through the method `currentElement` and `nextElement`. Depending on the property of the matrix `nextElement` accesses different matrix positions.

4.5.2 Selecting an Implementation

The selection algorithm for implementations at storage format abstraction level checks the properties and storage formats of the matrices involved in an operation. The selection algorithm for implementations at matrix abstraction level simply checks the matrix properties. Finally, the selection algorithm for implementations at iterator abstraction level checks the mathematical relation matrix properties (symmetric, positive definite, etc.).

Recall that only certain matrix calculations have a complete decision tree and, therefore, not all matrix calculations have a selection algorithm implemented. In these cases, users pass the selection as a parameter (see direct and iterative solvers of matrix equations, in Section 3.2.4).

Some object oriented libraries follow the guidelines of BLAS and LAPACK and implement matrix calculations taking into account the properties and storage

```
double a[n][k];
double b[k][m];
double c[n][m];
double temp;
int j,l,i;

for (j=0; j!=m; j++)
{
  for (i=0; i!=n; i++)
  {
    c[i][j]=0.0;
  }// end for
}// end for
for (j=0; j!=m; j++)
{
  for (l=0; l!=k; l++)
  {
    temp=b[l][j];
    if (temp!=0.0)
    {
      for (i=0; i!=n; i++)
      {
        c[i][j]+=a[i][l]*temp;
      }// end for
    }// end if
  }// end for
}// end for
```

```
double a[n*k];
double b[k*m];
double c[n*m];
int j,l,i;
int column_c=0;
int column_a=0;
int ind_b=0;
int ind_a, ind_c;

for (ind_c=0; ind_c!=n*m; ind_c++)
{
  c[ind_c]=0.0;
}// end for
ind_c=0;
for (j=0; j!=m; j++)
{
  column_a=0;
  for (l=0; l!=k; l++)
  {
    temp=b[ind_b];
    if (temp!=0.0)
    {
      ind_a=column_a;
      ind_c=column_c;
      for (i=0; i!=n; i++)
      {
        c[ind_c]+=a[ind_c]*temp;
        ind_c++;
        ind_a++;
      }// end for
    }// end if
    column_a+=n;
    ind_b++;
  }// end for
  column_c+=n;
}// end for
```

Figure 4.15: Implementation of matrix-matrix multiplication $C \leftarrow AB$ at storage format abstraction level where A and B are dense matrices stored in dense format.

```
double a[n][k];
double b[k][m];
double c[n][m];
double temp;
int j,l,i;

for (j=0; j!=m; j++)
{
  for (i=0; i!=n; i++)
  {
    c[i][j]=0.0;
  }// end for
}// end for
for (j=0; j!=m; j++)
{
  for (l=0; l!=k; l++)
  {
    temp=b[l][j];
    if (temp!=0.0)
    {
      for (i=0; i!=l+1; i++)
      {
        c[i][j]+=a[i][l]*temp;
      }// end for
    }// end if
  }// end for
}// end for
```

```
double ap[(n*k)*(n*k)/2+k/2];
double b[k][m];
double c[n][m];
double temp;
int j,l,i;

for (j=0; j!=m; j++)
{
  for (i=0; i!=n; i++)
  {
    c[i][j]=0.0;
  }// end for
}// end for
for (j=0; j!=m; j++)
{
  for (l=0; l!=k; l++)
  {
    temp=b[l][j];
    if (temp!=0.0)
    {
      for (i=0; i!=l+1; i++)
      {
        c[i][j]+=ap[i+l*(l-1)/2]*temp;
      }// end for
    }// end if
  }// end for
}// end for
```

Figure 4.16: Implementations of matrix-matrix multiplication $C \leftarrow AB$ at storage format abstraction level where A is an upper triangular matrix stored in packed format (right) or dense format (left) and B is a dense matrix stored in dense format.

```
Matrix a=new Matrix();
Matrix b=new Matrix();
Matrix c=new Matrix();
double temp;
int j,l,i;

a.setDenseMatrix(n,k);
b.setDenseMatrix(k,m);
c.setDenseMatrix(n,k);

for (j=0; j!=m; j++)
{
  for (i=0; i!=n; i++)
  {
    c.assign(i,j,0.0);
  }// end for
}// end for
for (j=0; j!=m; j++)
{
  for (l=0; l!=k; l++)
  {
    temp=b.element(l,j);
    if (temp!=0.0)
    {
      for (i=0; i!=n; i++)
      {
        c.assign(i,j,c.element(i,j)
           +a.element(i,l)*temp);
      }// end for
    }// end if
  }// end for
}// end for
```

```
Matrix a=new Matrix();
Matrix b=new Matrix();
Matrix c=new Matrix();
double temp;
int j,l,i;

a.setUpperTriangularMatrix(n,k);
b.setDenseMatrix(k,m);
c.setDenseMatrix(n,k);

for (j=0; j!=m; j++)
{
  for (i=0; i!=n; i++)
  {
    c.assign(i,j,0.0);
  }// end for
}// end for
for (j=0; j!=m; j++)
{
  for (l=0; l!=k; l++)
  {
    temp=b.element(l,j);
    if (temp!=0.0)
    {
      for (i=0; i!=l+1; i++)
      {
        c.assign(i,j,c.element(i,j)
           +a.element(i,l)*temp);
      }// end for
    }// end if
  }// end for
}// end for
```

Figure 4.17: Implementations of matrix-matrix multiplication $C \leftarrow AB$ at matrix abstraction level where A is dense matrix (left) or upper triangular matrix (right) and B is a dense matrix.

```
Matrix a=new Matrix();
Matrix b=new Matrix();
Matrix c=new Matrix();
double atemp, btemp;
// set properties

b.setColumnWise();
a.setColumnWise();
b.begin();

while (!b.isMatrixFinished()) // for (j= ... )
{
  while (!b.isVectorFinished()) // for (l= ... )
  {
    b.nextElement();
    b.currentElement(l,btemp);
    if (btemp!=0.0)
    {
      a.beginAt(1,l);
      while (!a.isVectorFinished()) // for (i= ... )
      {
        a.nextElement();
        a.currentElement(i,atemp);
        c.assign(i,j,atemp*btemp);
      }// end while
    }// end if
  }// end while
  b.nextVector(j);
}// end while
```

Figure 4.18: Implementation of matrix-matrix multiplication $C \leftarrow AB$ at iterator abstraction level.

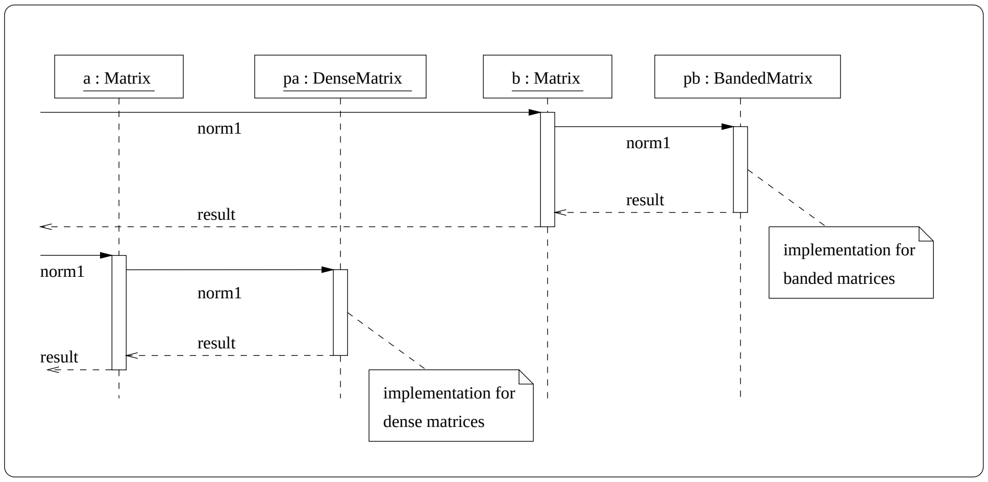


Figure 4.19: Sequence diagram of dynamic binding as a selection of `norm1` implementations.

format of only one matrix. These libraries can implement the selection algorithm implicitly using the dynamic binding mechanism provided by most object oriented languages. A simpler unary example $||A||_1$ is represented by `r=a.norm1();` where `a` is an object of class `Matrix`. The object `a` is linked with an object of a subclass of `Property`. The implementation of the selection algorithm is simply to invoke the method `norm1` in the linked object representing the property. The dynamic binding mechanism would check the class of this object and select the method that is implemented in this class. Figure 4.19 presents a sequence diagram where the method `norm1` is invoked in two matrices with different properties.

In a more general case, where binary operations are implemented using the properties and storage formats of both matrices, the selection algorithm has to be implemented explicitly.

Java offers the possibility of calling subroutines written in other languages using its *Java Native Interface*. OoLaLa can implement a selection algorithm that checks if a traditional library supports the combination of matrix properties and storage formats and then call the subroutine. Even when the combination of matrix properties and storage format is not supported it remains possible to always call traditional library subroutines. In this way, OoLaLa becomes simply a wrapper for traditional libraries; users of OoLaLa benefit from a simpler interface, and library developers can save their legacy code and concentrate on new functionality. Experiences with the Java Native Interface accessing Fortran

BLAS and LAPACK have reported similar performance to that achieved by the Fortran libraries ([BG97], [BC98], [BC99], [GFHM98], [GGMS99]).

4.6 Summary

The implementation of OoLaLa is the core of this chapter. The OoLaLa's design has been modified so that:

- the `Property` inheritance class hierarchy does not use multiple inheritance to model composed properties, such as symmetric banded;
- a version of OoLaLa is created for each numerical data type; and
- two-dimensional arrays are implemented by mapping them to one-dimensional Java language arrays.

Example programs have been presented showing how matrices and views are created and initialised. Users can specify the storage format for each matrix and can also rely on the library which can automatically select the storage format according to the matrix properties. UML object diagrams and sequence diagrams illustrate the implementations.

The management of storage formats requires the propagation of properties, and is implemented by checking the consistency between the property and storage format of a given matrix. Consistency is checked when matrices are created (users having specified a storage format) and when matrices are operated on as their properties may vary.

A matrix calculation in OoLaLa is divided into checking correctness of parameters, propagating the properties, selecting an implementation and implementing the matrix calculation. The implementation of the matrix calculation can be at storage format, matrix or iterator abstraction levels. Matrix abstraction level reduces the number of implementations, since this abstraction level is independent on storage formats. However, this abstraction level remains dependent on the matrix properties because the implementations indicate explicitly the matrix position of the elements to be accessed. The iterator abstraction level defines a matrix iterator that traverses a matrix (column-wise or row-wise) accessing the nonzero elements. A matrix iterator does not declare explicitly the elements to be accessed. Thus, for this abstraction level, the number of implementations is reduced to the number of mathematical matrix properties.

Obviously, the selection algorithm varies depending on the abstraction level at which matrix calculations are implemented. OoLaLa can become an object oriented interface, or a wrapper, of traditional libraries if the selection algorithm always selects subroutines of these libraries. OoLaLa can be also a hybrid library where some matrix calculations are implemented at iterator abstraction level while others are implemented at storage format abstraction level.

The next chapter analyses circumstances under which the propagation of properties and the management of storage formats can fail, or be inefficient. It also presents situations where users have to choose among semantically equivalent matrix calculations with different execution times. These situations are either not solved by libraries or it is unusual for libraries to address them.

Experiences of matrix properties propagation and of automatic storage format management by compilers have been reported in [Bik96], [BW96], [Mar97] and for Matlab in [GMS92]. Comparisons between implementations at iterator abstraction level and implementations at storage format abstraction level have been reported in [SL98b], [SL98c], [SL98a], [SL99], [SLL99] for MTL, which is written in C++.

Chapter 5

Limits of the Library Approach

This thesis has followed a library approach as the way of improving the development of linear algebra programs. An object oriented library that:

- encapsulates storage formats and matrices in classes,
- selects the appropriate implementation of certain matrix calculations given the properties and storage formats of the matrix operands, and
- is able to manage the storage formats and to propagate matrix properties (a novel functionality for libraries)

has been designed and described how the library could be implemented.

The objective of this chapter is to investigate the difficulties in developing the program with minimum execution time; linear algebra libraries, both traditional and object oriented, cannot solve this challenge.

The difficulties can be in one of two forms. Firstly, different semantically equivalent matrix expressions that can be implemented yielding different programs, and the execution times of these programs may be different. The term *semantically equivalent* is used since it is only when perfect floating point arithmetic is assumed that the programs are really equivalent. The equivalent expressions are obtained from the mathematical properties of the matrix operations. The commutative property of matrix addition (Section 5.1), the associative property of matrix multiplication (Section 5.2), and the distributive property of matrix multiplication are discussed.

The second difficulty is directly related to the novel functionality. Examples are presented to illustrate where a library approach cannot propagate efficiently the properties through matrix calculations (Section 5.4). It is also described the problem of selecting the best storage format for each matrix of a program (Section 5.5).

Finally, the chapter provides an overview of a software environment for the development of linear algebra programs,i.e. a problem solving environment, that merges the library approach with techniques to address the difficulties identified (Section 5.6).

5.1 The Best Order Problem

The commutative property of matrix addition states that

$$A + B = B + A. \tag{5.1}$$

When adding 3 matrices, the commutative property yields the following identities:

$$\begin{aligned} A + B + C &= A + C + B \\ &= B + A + C \\ &= B + C + A \ . \\ &= C + A + B \\ &= C + B + A \end{aligned}$$

the number of different ways of representing the addition of 3 matrices is $3 \times 2 = 6$.

When the number of matrices is increased up to 4, the commutative property yields $4! = 24$ different representations (ordering of the additions). In general, when adding n matrices the commutative property yields $n!$ different representations. Users who want to develop a program that calculates the addition of n matrices can develop $n!$ different programs; each program corresponds to a different order of addition. For example, the addition $A+B+C$ can be programmed as `R=(A+B)+C` or `R=(B+C)+A` or `R=(C+B)+A` or $\ldots$, all being semantically equivalent programs. However, the execution time of each program varies depending on the order of addition and the properties of A, B and C.

For example, suppose that A and B are diagonal matrices and C is a dense matrix, and all of them are $m \times m$ matrices. A specialised program that implements `R=(A+B)+C` would use $2m$ floating point addition instructions, $3m + m^2$ memory read instructions and $2m^2$ memory write instructions. On the other

	$(A+B)+C$			$(C+A)+B$		
	`R=A+B` $\diagdown + \diagdown$	`R=R+C` $\diagdown + \blacksquare$	Total	`R=C+A` $\blacksquare + \diagdown$	`R=R+B` $\blacksquare + \diagdown$	Total
# add	m	m	$2m$	m	m	$2m$
# read	$2m$	m^2+m	m^2+3m	m^2+m	m^2+m	$2(m^2+m)$
# write	m^2	m^2	$2m^2$	m^2	m^2	$2m^2$

Table 5.1: Number of instructions for programs implementing $A+B+C$ and $C+A+B$, where A and B are $m \times m$ diagonal matrices ($\diagdown$) and C is a $m \times m$ dense matrix ($\blacksquare$).

hand, another specialised program which implements `R=(C+A)+B` would use the same number of instructions except that the number of memory read instructions becomes $2(m+m^2)$ (Table 5.1 shows how these counts are obtained). Assuming constant execution time for memory access, the program implemented as `R=(A+B)+C` would be faster as it executes m^2-m fewer memory read instructions.

The *best order problem* is defined as the search for the program that has minimum execution time to calculate an expression of n elements which are combined by the same commutative binary operation.

The addition of n matrices constitutes a *best order problem*, and so a search space of $n!$ possible solutions characterises the addition of n matrices.

In this case, the best order problem can be solved by first selecting the two matrices which, when added, produce a matrix with the minimum number of nonzero elements. When more than one pair of matrices produce a matrix with the minimum number of nonzero elements, the pair that collectively the smallest number of nonzero elements is selected. In this way the best order problem for n matrices is solved recursively in terms of the best order problem for $n-1$ matrices. The base case occurs when $n=2$.

This algorithm needs a mechanism to predict the number of nonzero elements for the result matrix. Table 4.3 presented the rules when dense and banded matrices are added. Different prediction algorithms can be used when sparse matrices are considered. The simplest algorithm makes the worst-case prediction, that the number of nonzero elements as the sum of the numbers of nonzero elements of the two added matrices. More sophisticated algorithms would need to exploit the specific structures of the matrices.

Note that, the best order problem cannot be solved by a library unless a subroutine (or method) is provided which implements the addition of n matrices.

This is not the usual case.

5.2 The Best Association Problem

The associative property of matrix multiplication states that

$$(AB)C = A(BC). \tag{5.2}$$

When 4 matrices are multiplied, the associative property yields

$$\begin{array}{rcl} ((AB)C)D & = & (A(BC))D \\ & = & A(B(CD)) \\ & = & A((BC)D) \\ & = & (AB)(CD) \end{array}.$$

Each representation is formed dividing the 4 matrix multiplication into two subsets by introducing parenthesis (e.g. $(AB)(CD)$ or $(A)(BCD)$). When a subset has only one or two matrices, that subset is a base case. Otherwise, the subset is recursively subdivided until a base case subset is found.

Let $ANI(n)$ be the number of ways of representing the multiplication of n matrices (i.e., the association of the $n-1$ matrix multiplications). It is straightforward to show that $ANI(3) = 2$, $ANI(4) = 6$ and, in general, $ANI(n) = \sum_{i=1}^{n-1} ANI(i)ANI(n-i)$. $ANI(n)$ is known as the catalan number ([Slo73], [PB85]). Other examples of catalan numbers are $ANI(5) = 14$ and $ANI(15) = 2674440$.

Each representation is the basis of a different program, and all such programs are semantically equivalent. However, the execution time of these programs varies. The variation is due to matrix dimensions and matrix properties. For example, consider the matrix multiplication ABC where A and B are $n \times n$ dense matrices and C is a $n \times 1$ dense matrix. The association $(AB)C$ performs one matrix-matrix multiplication ($O(n^3)$ floating point operations) and one matrix-vector multiplication ($O(n^2)$ floating point operations). On the other hand, the association $A(BC)$ performs two matrix-vector multiplications ($2O(n^2)$ floating point operations).

The *best association problem*, also referred to as the chain multiplication problem [God73], is defined as the search for the program to calculate an expression of n elements which are combined by the same binary associative and non-commutative operation.

The multiplication of n matrices constitutes a best association problem and so a search space of $ANI(n)$ (catalan numbers) possible solutions characterises the multiplication of n matrices. Algorithms to solve the best association problem can be found in [HS82] [HS84] [Coh99].

A library can only solve the best association problem if a subroutine (or method) is provided which implements the multiplication of n matrices. Again, this is not the usual case.

5.3 The Maximum Common Factor Problem

The distributive property of matrix multiplication states that

$$A(B + C) = AB + AC. \tag{5.3}$$

The right hand side of Equation 5.3 implies that the implementation would require two matrix multiplications and one addition. On the other hand, the left hand side of Equation 5.3 implies that the implementation would require one multiplication and one addition. The execution times would be significantly different and the left hand side of Equation 5.3 would be faster.

The distributive property can be generalised as

$$A(B_1 + B_2 + \cdots + B_h) = AB_1 + AB_2 + \cdots + AB_h,$$

where $A, B_1, B_2, \ldots, B_h$ are matrices or combinations of matrix calculations that produce a matrix. With this generalisation in mind, the *maximum common factor problem* is defined as finding the matrix A, so that the expression $A(B_1 + B_2 + \cdots + B_h)$ has no further common factors. That is, there is no matrix X, different from the identity matrix, such that $B_i = XY_i$ and $i = 1, 2, \ldots, h$.

Assuming a language that allows a matrix to be a variable, the maximum common factor problem can be solved applying standard compiler techniques. In the first phase, forward substitution is applied to replace variables by their current expression. This facilitates common subexpression elimination; the common expression is replaced by an appropriately initialised new variable. Finally,

```
A=C*D*H
B=C*D*J
R=A+B
```

(a) original code

```
R=C*D*H+C*D*J
```

(b) after forward substitution

```
TEMP=C*D
R=TEMP*H+TEMP*J
```

(c) after common subexpression elimination

```
R=TEMP*(H+J)
```

(d) after strength reduction

Figure 5.1: Example of applying standard compiler optimisations in order to solve the maximum common factor problem.

strength reduction optimisation exploits the distributive property of matrix multiplication to replace an expensive operation with an equivalent, but less expensive, operation. Figure 5.1 presents the effects of forward substitution, common subexpression elimination, and strength reduction in a program where the variables are matrices. The compiler optimisations above described are presented in more detail by Aho, Sethi and Ullman [ASU85].

A library can never solve the maximum common factor problem since its solution requires knowledge about the data flow in a program.

Similar situations arise when $AB^{-1}C$ or $A + B^{-1}C$ or $B^{-1}C$ need to be computed, where A, B and C are matrices or combinations of matrix calculations that produce matrices. Calculation of $B^{-1}C$ by forming the inverse matrix is known to be more time consuming than solving the system of linear equations $BX = C$ for X. The solution follows exactly the steps defined for solving the maximum common factor problem, except that the strength reduction rule is different.

A further example is the system of linear equations $A_1 A_2 \ldots A_p x = b$ where A_1, A_2, $\ldots$, A_p are square matrices. Instead of carrying out the chained matrix multiplication, with a cost of $2n^3 + O(n^2)$ floating point operations for each multiplication, each matrix can be LU-factorised ($A_1 = L_iU_i$ and $i = 1, 2, \ldots, p$) at a cost less than or equal to $\frac{2}{3}n^3 + O(n^2)$ floating point operations per factorisation.

The work of Marsolf ([Mar97], [MGG97]) in the Falcon project uses transformation patterns for interactively restructuring Matlab's programs. Users define patterns to be found in a Matlab program and specify how the code matched with a pattern should be restructured ([Mar97] Chapter 4). These transformation patterns enable the Falcon environment to apply traditional restructuring compiler transformations ([Mar97] Chapter 5), such as loop unrolling [BGS94], and basic algebraic transformations ([Mar97] Chapter 6). Among other basic algebraic transformations, Marsolf presents a limited solution to the multiplication of n matrices (best association problem Section 5.2) and a solution to the example, where the inversion of a matrix is avoided by solving a system of linear equations, as presented above. Marsolf's solution to the multiplication of n matrices identifies the vectors and multiplies these first. However, transformation patterns cannot implement the algorithm presented in [Coh99] for the general best association problem. This algorithm uses information related to the number of nonzero elements in rows and columns and this information is not represented by the transformation patterns. Transformation patterns are able to perform the strength reductions presented in this section, but Marsolf does not show how forward substitution or common subexpression elimination can be implemented with the transformation patterns.

5.4 The Matrix Property Propagation Problem

OoLaLa is able to propagate the properties of a matrix through matrix calculations. However, a library cannot efficiently propagate matrix properties that are a consequence of the history of previous matrix calculations. For example, the matrix multiplication AB where A and B are symmetric is known to generate a dense unsymmetric result matrix. Similarly, the matrix multiplication BA also generates a dense unsymmetric matrix. Applying the rules of addition, $AB+BA$ is the addition of two dense matrices and generates a dense unsymmetric matrix. However, for A and B symmetric, $AB + BA$ is also a symmetric matrix ($AB + (AB)^T = AB + BA$).

In order to address this problem, a library would have to keep a history for each matrix. This history would record the matrix calculations that have been carried out on each matrix and the parameters' matrix properties of those matrix operations. On the other hand, a compiler is able to identify these situations as long as they can be specified by a set of if-then rules. The implementation is similar to how a compiler checks the type of an expression; when it detects an incorrect type, it sends an error message. Similarly, the compiler is checking an expression of matrices and detects a special situation. Instead of sending an error message, the compiler changes the matrix properties of the expression. For a more technical approach to these compiler techniques consult [ASU85] Chapters 4 and 5.

Despite the fact that Marsolf's work ([Mar97], [MGG97]) in the Falcon environment and Bik and Wijshoff's Sparse Compiler ([Bik96], [BW96], [BBKW98], [BW99]) propagate matrix properties, they do not identify this problem or present any solution.

5.5 The Best Storage Format Problem

Matrices can be stored in different storage formats. Table 4.2 presents the advisable combinations of matrix properties and storage formats in the context of OoLaLa. The storage format influences the execution time of implementations of matrix operations and it determines the memory position where each element of a matrix is kept. An implementation of a matrix calculation determines a logical access pattern to the matrix elements, which is mapped to a physical access pattern to the memory. When the storage format is changed, the logical access pattern to the elements of a matrix remains unchanged, but the physical access pattern varies. Different physical access patterns have different rates of cache reuse. Consider, for example, the well-known case of arrays stored row-wise or column-wise. For this case, compiler optimisation techniques have been developed to modify the loops so that an array is traversed in the order it is stored in memory [BGS94].

OoLaLa enables users to abstract their programs from the storage formats and from how the matrix properties are propagated through matrix calculations. Hence, the structure of a linear algebra program is divided into two parts. The first part of the program declares the input matrices and their matrix properties (optionally their storage formats). In the second part, the matrices are operated and auxiliary matrices are created to hold intermediate or final results. Before each matrix calculation is performed, the storage format of the associated matrices can be changed. These storage format changes could be represented by invocations of mapping methods. These methods would map from a current storage format of a matrix to a specified new storage format. These mapping methods can be inserted at any point of the program and the semantics of the program remains unchanged. The program produces the same result (assuming perfect arithmetic) independently of the number and the location in the program of the mapping methods. The visible effects of mapping methods are the execution time and memory requirement. The execution time decreases when the time of executing the mapping methods added to the time of executing the matrix calculations with the new storage formats is less than the time of executing the matrix operations with the previous storage formats; otherwise the execution time increases (or remains unchanged).

The *best storage format problem* is defined as the search for the linear algebra program with the minimum execution time among those programs with equivalent functionality but with different storage formats.

In general, the solution of the best storage format problem is computationally infeasible [Mac87]. Bik and Wijshoff have proposed an heuristic to automatically select the storage format [BW96]; this heuristic is integrated with their Sparse Compiler and, since it requires knowledge of the instruction flow, it cannot be included in any library.

5.6 Overview of a Linear Algebra Problem Solving Environment

Previous sections have presented problems or limits associated with linear algebra libraries. Some of these, for example the best order and the best association problems, can be solved within a library, but this is unusual. The other problems, the maximum common factor, the matrix properties propagation and the best storage format, can only be approached at compile time. These problems motivate a move from linear algebra libraries to *problem solving environments*. A problem solving environment is software, often with graphical user interfaces, which enables users to develop programs using as the programming language the problem domain language. A problem solving environment integrates domain specific libraries, compiler techniques, artificial intelligence, visualisation and any other computer science discipline that may help users in developing their programs [GHR94].

A linear algebra problem solving environment should provide support for, and encapsulate, the different tasks that users have to perform when developing a linear algebra program, namely:

(a) describe the problem in terms of matrix calculations,

(b) analyse the matrices to determine their properties,

(c) select a library or libraries which support the calculations and properties,

(d) map the matrix calculations into the implementations provided by the library,

(e) analyse how the matrix properties are propagated through the matrix operations,

(f) declare the variables conforming to the storage format which is supported by the selected implementations,

(g) select the best combination of preconditioner and iterative solver for a given system of linear equations, and

(h) select the best ordering algorithm for a direct solver for a given sparse system of linear equations.

OoLaLa has encapsulated tasks (c) and (f), and, partially, tasks (d) and (e). This chapter has presented examples of how to help users to efficiently map their matrix calculations into matrix implementations provided by libraries, i.e. task (d). To this end, matrix operation properties have been presented as a way of describing different semantically equivalent programs but with different execution times. Solutions of the best association problem are proposed by Hu and Ching [HS82][HS84] and by Cohen [Coh99].

The basis for the solution of the maximum common factor problem is based on standard compiler optimisation techniques applied to variables of type matrix. Marsolf [Mar97] partially implements some of the solution of the maximum common factor problem together with solutions to other related problems' based on strength reduction.

The solution of propagating matrix properties through more than one operation at each time, i.e. task (e), is based on syntax directed translation [ASU85], a standard compiler technique to parse programming languages.

Automatic detection of nonzero structure, i.e. task (b), has been addressed by Bik and Wijshoff [BW99]. They have also proposed a heuristic for solving the best storage format problem [BW96].

The selection of the best combination of preconditioner and iterative solver, i.e. task (g), together with the best ordering algorithm, i.e. task (h), for sparse systems of linear equations, remain as open research problem.

Chapter 6

Conclusions

Object oriented linear algebra libraries are proposed as a way of improving the development process for linear algebra programs. Object oriented software construction offers linear algebra abstraction and encapsulation of implementation details. Designs for traditional linear algebra libraries are dominated by implementation details which are visible to the users. As a consequence, the intellectual distance between a linear algebra description of a problem and its description with traditional libraries is too large.

An object oriented analysis and design of a linear algebra library has been conducted, and, as a result, different object oriented models have been proposed. These models serve to classify a set of object oriented linear algebra libraries. The object oriented model accepted has features not found in other libraries, and it enables functionality previously reserved for compilers. Based on the reviewed object oriented libraries and on the conducted analysis and design, a library interface has been proposed for basic matrix operations and for the solution of matrix equations. The object oriented model, the increased functionality and the interface constitute the design of a new object oriented library.

Libraries offer limited help in developing a linear algebra program; they cannot identify a sequence of calls (or invocations) and match this with a different but semantically equivalent (assuming perfect floating point arithmetic) sequence of calls that could be less time consuming. This thesis has analysed and identified some of these limits.

The following section explains the above in more detail. The chapter ends with an evaluation of the limitations of the work presented (Section 6.2) and suggestions for future work (Section 6.3).

6.1 Summary

The numerical linear algebra community has analysed matrices and their calculations in order to find characteristics, i.e. matrix properties, which can be exploited by the implementations of the operations in order to reduce their execution times. Matrix properties are characteristics of matrix structures that arise repeatedly in linear algebra problems. Some of the matrix properties also enable matrices to be stored in compressed forms (e.g. for a sparse matrix that has 10% of nonzero elements). The algorithms that exploit the properties and use the data structures have been implemented in Fortran 77 as subroutines and these subroutines have been grouped into libraries, traditional libraries. For each algorithm there are as many different implementations as different combinations of advisable storage formats for the matrix parameters, and the number of algorithms is related to the number of combinations of properties for the matrix parameters. Thus, traditional libraries developers experience an explosion in the number of implementations and they have to choose which of the different possibilities are implemented.

The matrix calculations are divided into two groups: basic matrix operations and solution of matrix equations, some of which have rule based reasoning systems. These reasoning systems can be implemented as a set of "if-then" rules based on the properties and storage format of the matrices and decide the appropiate implementation for a matrix calculation.

When developing a linear algebra program with traditional libraries, the non-trivial tasks that have to be performed are:

- analysis of the properties of matrices,
- selection of the storage formats, and
- selection of the subroutines that deliver the minimum execution time.

Building on a review of existing object oriented linear algebra libraries a new class structure (see Figures 3.16 and 3.17) has been designed. This class structure enables a library to manage the storage formats and to propagate the matrix properties; a novel functionality for linear algebra libraries. In this way, matrices can transparently vary their properties and storage formats when they are operated on. This class structure and a proposed library interface constitute the design of a new library known as the Object Oriented Linear Algebra LibrAry (OoLaLa).

Developers of traditional libraries have benefited from two abstraction levels at which matrix calculations can be implemented. These abstraction levels reduce the number of implementations. The matrix abstraction level enables matrices to be represented and accessed independently of their storage formats and the iterator abstraction level provides an implicit way of traversing matrices.

A matrix calculation in OoLaLa is divided into checking the correctness of the parameters, propagating the properties, selecting an implementation and implementing the matrix calculation. The implementation of the matrix calculation can be at storage format, matrix or iterator abstraction levels.

Obviously, the selection algorithm varies depending on the abstraction level at which matrix calculations are implemented. OoLaLa can become an object oriented interface, or a wrapper, of traditional libraries if the selection algorithm selects always subroutines of these libraries. OoLaLa can also be a hybrid library where some matrix calculations are implemented at iterator abstraction level while others are implemented at storage format abstraction level.

The thesis concludes by identifying difficulties in developing a linear algebra program with minimum execution time that linear algebra libraries, both traditional and object oriented, cannot solve, and suggest that a problem solving environment [GHR94] might overcome the difficulties.

6.2 Critique

The main omission from the thesis is that it has not been possible to address the question of how much performance is lost by implementing matrix calculations at matrix and iterator abstraction levels compared with traditional libraries' implementations at storage format abstraction level. However, the main objective was to create an object oriented design of linear algebra. Due to time constraints, it has not been possible to implement fully this design.

Further, object oriented libraries have been justified because they are easier to use than traditional libraries, and this has been supported by clear arguments. However, a more scientific approach would have used *metrics* defined by the software engineering community to justify this claim.

6.3 Future Work

The previous section summarises the immediate future work: an evaluation of the performance lost when implementing matrix calculations at matrix and iterator abstraction levels compared with traditional libraries' implementations at storage format abstraction level. Hybrid libraries, where some matrix calculations are implemented at storage format abstraction level and others at iterator abstraction level, pose the further question – which matrix calculations should be implemented at which abstraction level in order to minimise execution time.

Another performance question is whether block algorithms and recursive algorithms ([WD98], [AGK$^+$99]) currently used in implementations at storage format abstraction level will reduce the execution time of implementations at iterator and matrix abstraction levels.

This thesis has concentrated on sequential linear algebra programs. A logical extension is to design and implement OoLaLa for parallel programs. Among others, the issues that need to be addressed are:

- threads sharing objects versus objects communicating and synchronising by remote method invocation,
- the way in which users take part in the parallelisation process of a linear algebra program,
- the performance comparison of parallel implementations at iterator and matrix abstraction levels with implementations at storage format abstraction level,
- the performance of compilers at parallelising implementations at iterator and matrix abstraction levels, ...

A long-term objective is the implementation of the outlined linear algebra problem solving environment for sequential and parallel programs is the objective. The implementation includes the improvement of current solutions to the tasks:

1. analysis of matrices to determine their properties ([Bik96], [BW99]);
2. mapping the matrix calculations into the implementations provided by libraries – ([Mar97], [MGG97]);

3. analysis of propagation of matrix properties through matrix operations, – ([Bik96], [BBKW98], [Mar97], [MGG97]);

4. selection of the best combination preconditioner and iterative solver for a given system of linear equations (open problem); and

5. selection of the best ordering algorithm for a direct solver for a given sparse system of linear equations (open problem).

Bibliography

[Abb83] Rossell J. Abbot. Program design by informal English descriptions. *Communications of the ACM*, 26(11):882–894, 1983.

[ABD$^+$95] E. Anderson, Z. Bai, C. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostouchov, and S. Sorensen. *LAPACK User's Guide*. SIAM Press, 2^{th} edition, 1995.

[Abr80] Jean-Raymond Abrial. The specification language Z: Syntax and "semantics". Technical report, Oxford University Computing Laboratory, Programming Research Group, 1980.

[ACMW99] Benjamin A. Allan, Robert L. Clay, Kyran D. Mish, and Alan B. Williams. *ISIS++ Reference Guide: Iterative Scalable Implicit Solver in C++ version 1.1.* Sandia National Laboratories Livermore, 1999.

[AF95] Niclas Andersson and Peter Fritzson. Generating parallel code from object oriented mathematical models. In *Proceedings of the* 5^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–57, 1995.

[AFM97] O. Agesin, S. Freund, and J. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the Symposium on Object Oriented Programming: Systems, Languages and Applications*, pages 49–65, 1997.

[AG99] Cleve Ashcraft and Roger Grimes. SPOOLES: An object-oriented sparse matrix library. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[AGK$^+$99] Bjarne Stig Andersen, Fred Gustavson, Alexander Karaivanov, Jerzy Wasniewski, and Plamen Y. Yalamov. Lawra – linear algebra with recursive algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, 1999.

[Åhl95] Krister Åhlander. An object-oriented approach to construct pde solvers. Technical Report 197, Department of Scientific Computing, Uppsala University, 1995.

[AL96] Cleve Ashcraft and Joseph W. H. Liu. *SMOOTH: A Software Package For Ordering Sparse Matrices*, November 1996.

[ANS83] ANSI (American National Standards Institute) and US Goverment Deparment of Defense. *Ada Joint Program Office: Military Standard – Ada Programming Language*, 1983.

[AR94] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Versions.* John Wiley & Sons, 7^{th} edition, 1994.

[Ara89] G. Arango. Domain analysis: From art to engineering discipline. *SISOFT Engineering Notes*, 14(3), 1989.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertran Meyer. A specification language. In R. McNaughten and R.C. Mckeag, editors, *On the Construction of Programs.* Cambridge University Press, 1980.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1985.

[Aus98] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison Wesley, 1998.

[Axe94] Owe Axelsson. *Iterative Solution Methods.* Cambridge University Press, 1994.

[BBC$^+$94] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack J. Dongarra, Voctor Eijkhout, Roldan Pozo, Charles Romine, and Hank van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[BBKW98] Aart J. C. Bik, Peter J. H. Brinkhaus, Peter M. W. Knijnenburd, and Harry A.G. Wijshoff. The automatic generation of sparse primitives. *ACM Transactions on Mathematical Software*, 24(2):190–225, June 1998.

[BBV$^+$99] Lubomir Birov, Yuri Bartenev, Anatoly Vargin, Avijit Purkayastha, Anthony Skjellum, Yoginder Dandass, and Purushotham Bangalore. The parallel mathematical libraries project (PMLP) – a next generation scalable sparse object oriented mathematical library suite. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[BC98] Brian Blount and Siddhartha Chatterjee. An evaluation of Java for numerical computing. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 35–46. Springer-Verlag, 1998.

[BC99] Brian Blount and Siddhartha Chatterjee. An evaluation of Java for numerical computing. *Scientific Programming*, 7(2):97–110, 1999. Special Issue: High Performance Java Compilation and Runtime Issues.

[BCHQ97] David L. Brown, Geoffrey S. Chesshire, William D. Henshaw, and Daniel J. Quinlan. OVERTURE: An object oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the Eigth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[BDD$^+$95] Zhaojun Bai, David Day, James Demmel, Jack Dongarra, Ming Gu, Axel Ruhe, and Henk van der Vorst. Templates for linear algebra problems. *Lecture Notes in Computer Science*, 1000:115–140, 1995.

[BDH$^+$98] David L. Brown, Kei Davis, William D. Henshaw, Daniel J. Quinlan, and Kristi Brislawn. OVERTURE: Object-oriented parallel adaptive mesh refinement for serial and parallel environments. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology –*

ECOOP'98 Workshop Reader, volume 1543 of *Lecture Notes in Computer Science*, pages 446–447. Springer-Verlag, 1998. Workshop on Parallel Object-Oriented Scientific Computing.

[BG97] Aart J. C. Bik and Dennis B. Gannon. A note on native level 1 BLAS in Java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997. Special Issue: Java for Computational Science and Engineering — Simulation and Modelling II.

[BGMS97] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[BGMS99] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, 1994.

[BHQ98] David L. Brown, William D. Henshaw, and Daniel J. Quinlan. OVERTURE: An object-oriented framework for solving partial differential equations. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments, First International Conference ISCOPE 97*, volume 1343 of *Lecture Notes in Computer Science*, pages 177–184. Springer-Verlag, 1998.

[BHQ99] David L. Brown, William D. Henshaw, and Daniel J. Quinlan. OVERTURE: An object-oriented framework for solving partial differential equations on overlapping grids. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, SIAM Proceedings in Applied Mathematics, 1999. Proceedings of SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, October 1998.

[Bik96] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, Department of Computer Science, Leiden University, 1996.

[BK99a] Zoran Budimlić and Ken Kennedy. The cost of being object-oriented: A preliminary study. *Scientific Programming*, 7(2):87–96, 1999. Special Issue: High Performance Java Compilation and Runtime Issues.

[BK99b] Zoran Budimlić and Ken Kennedy. Prospects for scientific computing in polymorphic object-oriented style. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[BKP98] Zoran Budimlić, Ken Kennedy, and Jeff Piper. The cost of being object-oriented: A preliminary study. In *Workshop for Java for High Performance Network Computing at EUROPAR'98*, 1998.

[BL97] Are Magnus Bruaset and Hans Petter Langatangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, 23(1):50–80, 1997.

[BLA99] BLAS Technical Forum. *Document for the Basic Linear Algebra Subprograms Standard*, August 1999. Draft.

[BML97] Josehp A. Bandk, Andrew C. Myers, and Barbara Liskov. Parametized types for Java. In *Proceeding of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, 1997.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications.* Benjamin Cummings, 1994.

[BPB$^+$99] Lubomir Birov, Arkady Prokofiev, Yuri Bartenev, Anatoly Vargin, Avijit Purkayastha, Yoginder Dandass, Vladimir Erzunov, Elena Shanikova, Anthony Skjellum, Purushotham Bangalore, Eugeny Shuvalov, Vitaly Ovechkin, Nataly Frolova, Sergey Orlov, and Sergey Egorov. The parallel mathematical libraries project (PMLP): Overview, innovations and design issues. In V. Malyshkin, editor, *Fifth International Conference on Parallel Computing Technologies*

– *PaCT'99*, number 1662 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[BPK] Computer Science Deparment, University of Minnesota and Minnesota Supercomputer Institute, and Mathematical Algorithms and Scalable Computing Group, SGI/Cray Research, Inc. *Block Preconditioning ToolKit (BPKIT) web page.* http://www.cs.umn.edu/ chow/bpkit.html.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[BW96] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, 1996.

[BW99] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal of Computing*, 28(5):1576–1587, 1999.

[CCH$^+$99] Julian C. Cummings, James A. Crotinger, Scott W. Haney, Willian F. Humphrey, Steve R. Karmesin, John V.W. Reynders, Stephen A. Simith, and Timothy J. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, SIAM Proceedings in Applied Mathematics, 1999. Proceedings of SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, October 1998.

[CH96] Edmond Chow and Michael A. Heroux. Block preconditioning toolkit reference manual. Technical Report UMSI 96/183, University of Minnesota Supercomputing Institute, September 1996.

[CH98] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, 1998.

[Cog] Software tools for High-Performance Computing, Department of Scientific Computing, Uppsala University. *Cogito project web page.* http://www.tdb.uu.se/research/swtools/cogito.html.

[Coh99] Edith Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1999.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM Press, 1979.

[DCHD90] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[DCHH88a] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[DCHH88b] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[DDS99] David M. Dooling, Jack Dongarra, and Keith Seymour. JLAPACK – compiling LAPACK FORTRAN to Java. *Scientific Programming*, 7(2):111–138, 1999. Special Issue: High Performance Java Compilation and Runtime Issues.

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

[Dif] Numerical Objects A.S. *Diffpack web page.* http://www.nobjects.com.

[Dij79] E. Dijkstra. Programming as a human activity. In *Classics in Software Engineering.* Yourdon Press, 1979.

[DKP98] Florin Dobrian, Gary Kumfert, and Alex Pothen. Object-oriented design for sparse direct solvers. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 207–214. Springer-Verlag, 1998.

[DKP99] Florin Dobrian, Gary Kumfert, and Alex Pothen. The design of sparse direct solvers using object-oriented techniques. In A. M. Bruaset, H. P. Langtangen, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, 1999.

[DLN+94] Jack Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozo, and Karin Remington. Sparse matrix libraries in C++ for high performance architectures. In *Proceedings of the Conference on Object Oriented Numerics OON-SKI*, pages 122–138, 1994.

[DLPR96] Jack Dongarra, Andrew Lumsdaine, Roldan Pozo, and Karin A. Remington. *IML++ v. 1.2: Iterative Methods Library Reference Guide*, 1996.

[DNS97a] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. Expressing object-oriented concepts in Fortran 90. *ACM Fortran Forum*, 16(1):13–18, 1997.

[DNS97b] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. How to express C++ concepts in Fortran 90. *Scientific Programming*, 6(4):363–390, 1997.

[DNS98] Viktor K. Decyk, Charles D. Norton, and Boleslaw K. Szymanski. How to support inheritance and run-time polymorphism in Fortran 90. *Computer Physics Communications*, 115:9–17, 1998.

[DPW93a] Jack J. Dongarra, Roldan Pozo, and David W. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, pages 162–171. IEEE Computer Society Press, 1993.

[DPW93b] Jack J. Dongarra, Roldan Pozo, and David W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proceedings of the Conference on Object Oriented Numerics OON-SKI*, 1993.

[DPW96] Jack Dongarra, Roldan Pozo, and David Walker. *LAPACK++ v. 1.1: High Performance Linear Algebra Users' Guide*, April 1996.

[Dub97] Paul F. Dubois. *Object Technology for Scientific Computing.* Prentice-Hall, 1997.

[Duf77] Iain S. Duff. MA28 : A set of FORTRAN subroutines for sparse unsymmetric linear equations. Technical Report R-8730, HMSO, AERE Harwell Laboratory, 1977.

[DW95] Jack J. Dongarra and David W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[EGSS82] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package. *International Journal of Numerical Methods for Engineering*, pages 1145–1151, 1982.

[FA93] Peter Fritzson and Niclas Andersson. Generating parallel code from equations in the objectmath programming environments. In Jens Volkert, editor, *Parallel Computation, Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 219–232. Springer-Verlag, 1993.

[FE98] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In Eric Jul, editor, *ECOOP'98 – Object-Oriented Programming* 12^{th} *European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer-Verlag, 1998.

[FEV93] Peter Fritzson, Vadim Engelson, and Lars Viklund. Variant handling, inheritance and composition in the ObjectMath computer algebra environment. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 145–160. Springer-Verlag, 1993.

[FVHF92] Peter Fritzson, Lars Viklund, Johan Herber, and Dag Fritzson. Industrial application of object-oriented mathematical modeling and computer algebra in mechanical analysis. In Georg Heeg, Boris Magnusson, and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems – TOOLS 7*, pages 167–181. Prentice Hall, 1992.

[FVHF95] Peter Fritzson, Lars Viklund, Johan Herber, and Dag Fritzon. High-level mathematical modeling and programming. *IEEE Software*, 12(4):77–87, 1995.

[Gan59a] F. R. Gantmacher. *The Theory of Matrices Vol. 1*. Chelsea, 1959.

[Gan59b] F. R. Gantmacher. *The Theory of Matrices Vol. 2*. Chelsea, 1959.

[GBDM77] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines: EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[GFHM98] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in Java: Exploiting native libraries. *Concurrency: Practice and Experience*, 10(11):863–872, 1998.

[GGMS99] Vladimir Getov, Paul Gray, Sava Mintcheva, and Vaidy Sunderam. Multi-language programming environments for high performance Java computing. *Scientific Programming*, 7(2):139–146, 1999. Special Issue: High Performance Java Compilation and Runtime Issues.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[GHR94] Estratis Gallopoulos, Elias N. Houstis, and John R. Rice. Computer as thinker/doer: Problem solving environments for computational science. *IEEE Computational Science Engineering Magazine*, 1(2):11–23, 1994.

[Gil94] John R. Gilbert. Predicting structure in sparse matrix computations. *SIAM Journal of Matrix Analysis and Applications*, 15(1):62–79, 1994.

[GJ95] F. Guidec and J. M. Jézéquel. Polymorphic matrices in paladin. In *Workshop on Object-based Parallel and Distributed Computation OBPDC*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[GJP96] F. Guidec, J. M. Jézéquel, and J. L. Pacherie. An object-oriented framework for supercomputing. *Systems and Software*, June 1996. Special issue on Software Engineering for Distributed Computing.

[GL79] Alan George and Joseph W. H. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, 5:134–162, 1979.

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

[GL99] Alan George and Joseph W. H. Liu. An object-oriented approach to the design of a user interface for a sparse matrix package. *SIAM Journal of Matrix Analisys and Applications*, 20(4):953–969, 1999.

[GMS92] John R. Gilbert, Cleve Moler, and Robert Schereiber. Sparse matrices in Matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[God73] Sadashiva S. Godbole. On efficient computation of matrix chain products. *IEEE Transactions on Computer*, C-22(9):864–866, 1973.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[GvL96] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* John Hopkins University Press, 3^{th} edition, 1996.

[HC99] Scott Haney and James Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4):66–72, 1999.

[Hig96] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM Publications, 1996.

[HKBR98] William Humphrey, Steve Karmesin, Federico Basseti, and John Reynders. Optimization of data-parallel field expressions in the POOMA framework. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments, First International Conference ISCOPE 97*, number 1343 in Lecture Notes in Computer Science, pages 184–194. Springer-Verlag, 1998.

[HRC$^+$98] William Humphrey, Robert Ryne, Timothy Cleand, Julian Cummings, Salman Habib, Graham Mark, and Ji Qiang. Particle beam dynamics simulations using the POOMA framework. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 25–34. Springer-Verlag, 1998.

[HS82] Te Chiang Hu and M. T. Shing. Computation of matrix chain products. Part I. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[HS84] Te Chiang Hu and M. T. Shing. Computation of matrix chain products. Part II. *SIAM Journal on Computing*, 13(2):228–251, 1984.

[IML] *Iterative Methods Library (IML++) library web page.* http://math.nist.gob/iml++/.

[ISI] Distributed Applications Research Deparment, Sandia National Laboratories. *Iterative Scalable Implicit Solver in C++ (ISIS++) web page.* http://z.ca.sandia.gov/isis.

[ITL] Laboratory for Scientific Computing, University of Notre Dame. *Iterative Template Library (ITL) web page.* http://www.lsc.nd.edu/research/itl.

[Jama] Department of Computer Science, University of Maryland and Mathematical and Computations Sciences Division, NIST. *Jampack library web page.* ftp://math.nist.gov/pub/Jampack/Jampack.html.

[JAMb] Mathematical and Computations Sciences Division, NIST and The MathWorks. *JAMA library web page*. http://math.nist.gov/jama/.

[Jav98] Java Grande Forum. *Making Java Work for High-End Computing*, November 1998. available at http://www.javagrande.org/reports.htm.

[Jav99] Java Grande Forum. *Interim Java Grande Forum Report*, June 1999. available at http://www.javagrande.org/reports.htm.

[JCJO92] Ivar Jacobson, Magnus Christenson, Patrik Johnson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[JLA] Department of Computer Science, University of North Carolina. *JLAPACK library web page*. http://www.cs.unc.edu/Research/HARPOON/jlapack.

[KCC+98] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy Williams. Array design and expression evaluation in POOMA II. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 231–238. Springer-Verlag, 1998.

[KP98] Gary Kumfert and Alex Pothen. An object-oriented collection of minimum degree algorithms. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 95–106. Springer-Verlag, 1998.

[Kru95] Philippe Kruchten. The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LAB+81] Barbara H. Liskov, Russel Atkinson, T. Bloom, E. Moss, J. Craig Schaffert, R. Scheiffer, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[Lan99] Hans Petter Langtangen. *Computational Partial Differential Equations, Numerical Methods and Diffpack Programming*, volume 2 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, 1999.

[LAP] *LAPACK++ library web page.* http://math.nist.gov/lapack++/.

[LHKK79] C. L. Lawson, R. J. Hanson, D. Kincais, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[Liu90] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal of Matrix Analysis and Applications*, 11(1):134–172, 1990.

[LS95] Meng Lee and Alexander Stepanov. The Standard Template Library. Technical report, Hewlet Packard Laboratories, 1995.

[Mac87] Mary E. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, 1987.

[Mar97] Bret Andrew Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation.* PhD thesis, University of Illinois At Urbana-Champaign, 1997.

[Mat] The MathWorks. *PRO-MATLAB User's Guide.*

[McD89] John Alan McDonald. Object-oriented programming for linear algebra. In *OOPSLA'89 Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 175–184, October 1989. Published in ACM SIGPLAN Notices, Vol. 24, No. 10.

[MEO98] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 2^{th} edition, 1997.

[MGG97] Bret Andrew Marsolf, K. A. Gallivan, and E. Gallopoulos. On the use of algebraic and structural information in a library prototyping and development environment. In *Proceedings* 15th *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, pages 565–570, 1997.

[MMG98] José E. Moreira, Sam P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. In *11th International Workshop on Languages and Compiler for Technical Computing*, 1998.

[MMG99] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. A standard java array package for technical computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[Mod] Modelica Design Group. *Modelica modeling language web page.* http://www.modelica.org/.

[MOT97] Eva Mossberg, Kurt Otto, and Michael Thuné. Object-oriented software tools for the construction of preconditioners. *Scientific Programming*, 6:285–295, 1997.

[MS95] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library.* Addison Wesley, 1995.

[MTL] Laboratory for Scientific Computing, University of Notre Dame. *Matrix Template Library (MTL) web page.* http://www.lsc.nd.edu/research/mtl.

[Mul97] Pierre-Alain Muller. *Instan UML.* Wrox, 1997.

[NE99] Eric Noulard and Nahid Emad. Object oriented design for reusable parallel linear algebra software. In Patrick Amestoy, Philippe Berger, Michael Daydé, Iain Duff, Valerie Fraysse Luc Giraud, and Daniel Ruiz, editors, *Proceedings Euro-Par'99 Parallel Processing* – 5th *International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[Nor96] Charles D. Norton. *Object-Oriented Programming Paradigms in Scientific Computing.* PhD thesis, Department of Computer Science, Rensselaer Polytechnic Institute, New York, 1996.

[Obj] Programming Environments Laboratory, Department of Computer and Information Science, Linköping University. *ObjectMath programming environment web page.* http://www.ida.liu.se/labs/pelab/omath/.

[OVE] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. *Overture framework web page.* http://www.llnl.gov/casc/Overture/.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proceeding of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, 1997.

[Owl] Department of Computer Science, University of Rice. *Objects within the Linear Algebra Package (OwlPack) web page.* http://www.cs.rice.edu/~budimlic/OwlPack.

[PB85] Paul W. Purdom and Cynthia A. Brown. *The Analysis of Algorithms.* Holt, Rinehart and Winston, 1985.

[PET] Mathematics and Computer Science Division at Argone National Laboratory. *Portable Extensible Toolkit for Scientific Computation (PETSc) web page.* http://www.mcs.anl.gov/petsc.

[PML] High Performance Computing Laboratory, Mississippi State University. *Parallel Mathematical library Project (PMLP) web page.* http://www.erc.msstate.edu/research/labs/hpcl/pmlp/.

[POO] Advanced Computing Laboratory, Los Alamos National Laboratory. *Parallel Object-Oriented Methods and Applications (POOMA) framework web page.* http://www.acl.lanl.gov/Pooma.

[Poz97] Roldan Pozo. Template numerical toolkit for linear algebra: High performance programming with C++ and the Standard Template

Library. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):251–263, 1997.

[Pre97] Roger S. Pressman. *Software Engineering: a Practioner's Approach.* McGraw Hill, 4th edition, 1997.

[PRL96] Roldan Pozo, Karin A. Remington, and Andrew Lumsdaine. *SparseLib++ v. 1.5: Sparse Matrix Class Library Reference Guide*, April 1996.

[Ran95] Jarmo Rantakokko. Object-oriented software tools for composite-grid methods on parallel computers. Technical Report 165, Department of Scientific Computing, Uppsala University, 1995.

[Rat97a] Rational Software Corporation. *Unified Modeling Language: Notation Guide*, version 1.1 edition, 1997. Available at http://www.rational.com/uml/1.1/.

[Rat97b] Rational Software Corporation. *Unified Modeling Language: Semantics*, version 1.1 edition, 1997. Available at http://www.rational.com/uml/1.1/.

[RB96] John R. Rice and Ronald F. Boisvert. From scientific software libraries to problem solving environments. *IEEE Computational Science and Engineering Magazine*, pages 44–53, 1996.

[Ric96] John R. Rice. Scalable scientific software libraries and problem solving environments. Technical report, Computer Science, Purdue University, 1996. TR-96-001.

[Rog93] Rogue Wave Software Inc. *First Annual Object Oriented Numerics Conference*, 1993.

[Saa96] Youcef Saad. *Iterative Methods for Sparse Linear Systems.* PWS, 1996.

[SBD$^+$76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix eigensystem routines: EISPACK guide.*, volume 5 of *Lecture Notes in Computer Science.* Springer-Verlag, 2nd edition, 1976.

[SL98a] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium ISCOPE 98*, number 1505 in Lecture Notes in Computer Science, pages 59–70. Springer-Verlag, 1998.

[SL98b] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology – ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 466–467. Springer-Verlag, 1998. Workshop on Parallel Object-Oriented Scientific Computing.

[SL98c] Jeremy G. Siek and Andrew Lumsdaine. A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology – ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 468–489. Springer-Verlag, 1998. Workshop on Parallel Object-Oriented Scientific Computing.

[SL99] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *IEEE Computing in Science and Engineering*, 1(6):70–78, 1999.

[SLL99] Jeremy G. Siek, Andrew Lumsdaine, and Lie-Quann Lee. Generic programming for high performance numerical linear algebra. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*, SIAM Proceedings in Applied Mathematics, 1999. Proceedings of SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, October 1998.

[Slo73] Neil J. A. Sloane. *A Handbook of Integer Sequences*. Academic Press, 1973.

[SM88] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, 1988.

[SMO] *Sparse Matrix Object-oriented Ordering methods (SMOOTH) web page.* http://www.cs.yorku.ca/ joseph/Smooth/SMOOTH.html.

[Spa] *SparseLib++ library web page.* http://math.nist.gov/sparselib++/.

[SPO] *Sparse Object Oriented Linear Equations Solver (SPOOLES) web page.* http://www.netlib.org/linalg/spooles/spooles.2.2.html.

[Ste73] George W. Stewart. *Introduction to Matrix Computations.* Academic Press, 1973.

[Ste99] George W. Stewart. *The Jampack Owner's Manual*, 1999. ftp://thales.cs.umd/pub/Jampack/AboutJampack.html.

[TI97] Lloyd N. Trefethen and D. Bau III. *Numerical Linear Algebra.* SIAM Press, 1997.

[TMO$^+$97] Michael Thuné, Eva Mossberg, Peter Olsson, Jarmo Rantakokko, Krister Åhlander, and Kurt Otto. Object-oriented construction of parallel PDE solvers. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 203–226. Birkhäuser, 1997.

[TNT] Mathematical and Computational Sciences Division, NIST. *Template Numerical Toolkit (TNT) web page.* http://math.nist.gov/tnt/.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98.* IEEE Press, 1998.

[Wie98] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, December 1998.